Module: Shellcoding

Common Challenges

Yan Shoshitaishvili
Arizona State University

Common Issues: Memory Access Width

Be careful about sizes of memory accesses:

```
 single byte: mov [rax], bl
 2-byte word: mov [rax], bx
4-byte dword: mov [rax], ebx
8-byte qword: mov [rax], rbx
```

Sometimes, you might have to explicitly specify the size to avoid ambiguity:

```
 single byte: mov BYTE PTR [rax], 5
 2-byte word: mov WORD PTR [rax], 5
4-byte dword: mov DWORD PTR [rax], 5
8-byte qword: mov QWORD PTR [rax], 5
```

Common Issues: Forbidden Bytes

Depending on the injection method, certain bytes might not be allowed. Some common issues:

Byte (Hex Value)	Problematic Methods
Null byte \0 (0x00)	strcpy
Newline \n (0x0a)	scanf gets getline fgets
Carriage return \r (0x0d)	scanf
Space (0x20)	scanf
Tab \t (0x09)	scanf
DEL (0x7f)	protocol-specific (telnet, VT100, etc)

Many other situations arise. Be ready for anything!

Forbidden Bytes: More than One Way to Pwn Noobs

Convey your values creatively!

Bad	Good
`mov rax, 0 (48c7c0`**`00000000`**`)`	`xor rax, rax (4831C0)`
`mov rax, 5 (48c7c005`**`000000`**`)`	`xor rax, rax; mov al, 5 (4831C0B005)`
`mov rax, 10 (48c7c0`**`0a`**`000000)`	`mov rax, 9; inc rax (48C7C009`**`0`**`00000048FFC0)`
`mov rbx, 0x67616c662f "/flag" (48BB2F666C6167`**`000000`**`)`	`mov ebx, 0x67616c66; shl rbx, 8; mov bl, 0x2f (BB666C616748C1E308B32F)`

There are tools that will do this automatically, but they *do not always work*, and when they fail, your broken shellcode can waste hours of your life!

Forbidden Bytes: Meaning Within Meaning...

If the constraints on your shellcode are too hard to get around with clever synonyms, but the page where your shellcode is mapped is writable...

Remember: `code == data`!

Bypassing a restriction on `int3`:

```
inc BYTE PTR [rip]
.byte 0xcb
```

When testing this, you'll need to make sure .text is writable:

```
gcc -Wl,-N --static -nostdlib -o test test.s
```

Forbidden Bytes: Try and Try Again!

Sometimes, there are very complex constraints on your shellcode, which might make it hard to do anything useful!

Potential solution: multi-stage shellcode

Stage 1: `read(0, rip, 1000)`.

- getting your current instruction pointer might be hard, depending on the architecture
- on amd64, you can do it with `lea rax, [rip]`
- a read like this will overwrite the rest of your shellcode with unfiltered data!

Stage 2: whatever you want!

A good stage-1 shellcode is very short and simple, letting you get around complex shellcode requirements.

Downside: you don't always have access to inject more shellcode...

Shellcode Mangling

Your shellcode might be mangled beyond recognition.

Example situations:

- your shellcode might be *sorted*!
- your shellcode might be *compressed* or *uncompressed*.
- your shellcode might be *encrypted* or *decrypted*.

Start from what you want your shellcode to look like when it's executed, and *work backwards*.

Parts of your shellcode might be uncontrollable! You can jump over these parts to avoid them.

What good is shellcode when you are unable to speak?

Normally, your shellcode will just give you a shell (or the flag).

What if there is no way to output data (i.e., `close(1); close(2);`)?

What other ways can you use to communicate the flag?

Useful Tools

pwntools (https://github.com/Gallopsled/pwntools), a library for writing exploits (and shellcode).

rappel (https://github.com/yrp604/rappel) lets you explore the effects of instructions.

- easily installable via https://github.com/zardus/ctf-tools

amd64 opcode listing: http://ref.x86asm.net/coder64.html

Several gdb plugins exist to make exploit debugging easier!

- https://github.com/scwuaptx/Pwngdb
- https://github.com/pwndbg/pwndbg
- https://github.com/longld/peda