

# Exploiting Software Vulnerabilities

## Advanced Exploitation Techniques

### EXPLOIT PAYLOADS

© All wrongs reversed – under CC-BY-NC-SA 4.0 license



**Universidad**  
Zaragoza

Dept. of Computer Science and Systems Engineering  
University of Zaragoza, Spain

Course 2023/2024

**Master's Degree in Informatics Engineering**  
UNIVERSITY OF ZARAGOZA

*Room A.02, Ada Byron building*



# Outline

- 1 A Little Recap
- 2 Payload Types
- 3 Filters
- 4 Encoders/Decoders
- 5 Payload Components

# Outline

- 1** A Little Recap
- 2 Payload Types
- 3 Filters
- 4 Encoders/Decoders
- 5 Payload Components

# A Little Recap

What is an exploit payload?

## Shellcode?

- *Shellcode*: code that executes a shell
- *Exploit payload*: executable code in exploits

# A Little Recap

## What is a exploit payload?

### Shellcode?

- *Shellcode*: code that executes a shell
- *Exploit payload*: executable code in exploits

### Exploit payload

- Snippets of code that are injected into a running process and run from within that process
- It must keep the injected process running
  - Otherwise the process will terminate and thus the exploit will terminate as well

# A Little Recap

## What is a exploit payload?

### Requirements

- **Position-independent code**
  - Facilitates execution, regardless of the memory address or the segment in which they are injected
- **Size constraints:** as compact as possible
  - The smaller the payload, the more generically useful it will be
- **Avoid certain bytes that can be misinterpreted** (e.g., NULL bytes)
- **Cannot use library functions**
  - Unless they resolve the shared libraries themselves or they are located in the same fixed memory location

# A Little Recap

## System calls – syscalls

### Exploit payload manipulates the program to force it to make a syscall

- Functions that allow access to specific functions of the OS
- Interface between protected kernel mode and user mode

# A Little Recap

## Syscalls on Linux

- Through **software interrupts** (`int 0x80`)
- **Forces the switch to the kernel model and executes the syscall**
- Unlike other Unix syscall methods, Linux uses a fastcall convention (that is, it uses the CPU registers for higher performance)
  - The `eax` register contains the specific syscall number
  - The arguments of the syscall function are placed in other registers



# Outline

1 A Little Recap

**2** Payload Types

3 Filters

4 Encoders/Decoders

5 Payload Components

# Payload Types

## Byte content

### Null-free payloads

- Payloads that have NO null bytes
- Useful for string-based exploits
- What if we need, for instance, a null value for the execution of the shellcode?

■ Example: we need to insert a 0 value in the stack

#### ■ Solution: look for equally semantic instructions in the ISA

```
push 0 ; 0x6a00 → xor eax, eax ; 0x33c0  
push eax ; 0x50
```

```
mov eax, 0x00ddaa00 ; 0xb800aadd00 → mov eax, 0x88DDAA88 ; 0xb888aadd88  
xor eax, 0x77FFFF77 ; 0x3577ffff77
```

# Payload Types

## Byte content

### Alphanumeric

- **Only printable bits are valid**
  - For instance, ASCII bytes
- **Useful against certain filter functions**
- Further reading: *Writing IA32 alphanumeric shellcodes*  
(<http://phrack.org/issues/57/15.html>)

# Outline

1 A Little Recap

2 Payload Types

**3 Filters**

4 Encoders/Decoders

5 Payload Components

# Filters

- **Some applications may incorporate a **sanitized input filter****
  - Remove printable chars
  - Delete certain bytes
  - ASCII input → UNICODE input
- **A filter can modify the payload and then becomes useless**

**Payload can be prepared to bypass these filters**

# Filters

## Alphanumeric filters

- The filter only accepts printable ASCII characters of numbers and letters

- '0'...'9' (0x30...0x39)
- 'a'...'z', 'A'...'Z' (0x61...0x7a – 0x41...0x5a)

```
call eax ; 0xffd0 → push 0x50 ; 0x6a50
pop eax ; 0x58
xor al, 0x50 ; 0x3450
dec eax ; 0x48
xor eax, 0x47305757 ; 0x3557573047
xor eax, 0x68303838 ; 0x3538383068
push eax ; 0x50
```

After the last `xor` instruction, `eax` will contain the value `0xD0FF9090`

## How to use `eax`?

- From 2 bytes to 17 bytes (+ extras, as the required value is in a register!)
- Very tedious and error prone task
- There are automatic tools to create alphanumeric payloads
  - Or algorithms, such as base64 encoding (if supported)

# Filters

## Skipping alphanumeric filters

### Base16 data encoding

- **Standard, case-insensitive hex encoding**
- **The 16-characters subset of US-ASCII is used**
  - 4 bits to represent a printable character
- **Encoding process:**
  - Represents input bit octets as 2-character encoded output strings
  - Each octet is divided into two parts (nibble)
  - Each nibble is translated to a single character in the base16 alphabet

**Further reading:** <https://tools.ietf.org/html/rfc4648>

# Filters

## Skipping alphanumeric filters

### Encoding algorithm

- For each input byte, divided it into its nibble parts
- For each nibble, add the value 'A' (0x41)
  - The result will be in the range 0x41...0x50 ('A'...'P')
- Mark the end of the payloads with some character greater than 'P'

### Decoding algorithm

- For each input byte, subtract the value 'A' (0x41)
- Shift the result to the left
- Add the next input byte, after subtracting the value 'A' (0x41)
- For each nibble, add the value 'A' (0x41)



# Filters

## Skipping alphanumeric filters

```
INIT:
8A07          MOV AL, BYTE PTR DS:[EDI]
2C 41        SUB AL, 0x41
C0E0 04      SHL AL, 0x4
47           INC EDI
0207        ADD AL, BYTE PTR DS:[EDI]
2C 41        SUB AL, 0x41
8806        MOV BYTE PTR DS:[ESI], AL
46           INC ESI
47           INC EDI
803F 51      CMP BYTE PTR DS:[EDI], 0x51
72 EB       JB @INIT
```

- **edi**: encoded shellcode buffer
- **esi**: decoded shellcode buffer
  - Can they both be the same buffer?

*Note that these bytecodes **are not** alphanumeric. Some initial conversion is needed, as discussed before*

# Filters

## Skipping alphanumeric filters

- How to achieve **execution of the decoded payload**?
- Can be **located just after the conditional jump of the previous code**
- **Question: how to configure edi/esi values properly?**

```
A: EB 02          JMP B
   EB 05          JMP C
B:  E8 F9FFFFFF   CALL A
C:  5F             POP EDI
   83C7 1C        ADD EDI,0x1C
   57             PUSH EDI
   5E             POP ESI
```

## UNICODE filters

### ■ UNICODE character set

- 16 bits (instead of 8) to represent characters
- UNICODE characters equivalent to ASCII character are named *wide chars*

### ■ A wide character is its ASCII code plus the null byte

- In particular, from 0x01 to 0x7F

### ■ This null byte is used for other alphabetic encodings, such as Chinese, Russian, etc.

```

nop      ; 0x90
nop      ; 0x90
nop      ; 0x90
nop      ; 0x90
→  nop      ; 0x90
   add byte ptr ds:[eax + 90009000], dl ; 0x009000900090
```

# Filters

## Skipping UNICODE filters

### Valid instructions

- Single opcode
- `0xNN 0x00 0xNN` opcodes
- `0x00 0xNN 0x00` opcodes
- `0xNN 0x00 0xNN 0x00 0xNN` opcodes

```
push eax      ; 0x50      →  push eax      ; 0x50
pop ecx       ; 0x59      →  add byte ptr [ebp], ch ; 0x006d00
               ;          →  pop ecx       ; 0x59
               ;          →  add byte ptr [ebp], ch ; 0x006d00
```

**NOTE:** `ebp` must point to a writable memory address (otherwise, it will crash)

# Filters

## Skipping UNICODE filters

### How to jump to the payload?

- Find the payload in ASCII mode in memory
- Write a UNICODE-compliant payload manually
- Use a encoder
  - alpha2
  - vense: Perl script

**Remember: you must first configure the EIP with a valid address**

**Further reading:** *Unicode – from 0x00410041 to calc,*

<https://www.corelan.be/index.php/2009/11/06/exploit-writing-tutorial-part-7-unicode-from-0x00410041-to-calc>

# Outline

- 1 A Little Recap
- 2 Payload Types
- 3 Filters
- 4 Encoders/Decoders**
- 5 Payload Components

# Encoders/Decoders

## XOR encoders

- Take advantage of XOR properties
  - $a \otimes b = c; c \otimes b = a; c \otimes a = b$
- XOR-based code obfuscation: generally used by malware
- Useful to get shellcodes without null bytes
- Example: XOR 1-byte cipher

```
int encode(unsigned char xorKey, unsigned char *buf, int shellcodelen)
{
    for(int i = 0; i < shellcodelen; i++)
        if(xorKey != (unsigned char)shellcode[i])
            buf[i] = ((unsigned char)shellcode[i])^xorKey;
}
```

# Encoders/Decoders

## Assembler code for XOR decoder

```
EB 02          JMP B
A:
EB 05          JMP C
B:
E8 F9FFFFFF    CALL A
C:
5F            POP EDI
83C7 1A       ADD EDI,1A
57            PUSH EDI
5E            POP ESI
33C0          XOR EAX,EAX
33C9          XOR ECX,ECX
B1 NN        MOV CL, NNh # shellcode size
DEC:
8A07          MOV AL,BYTE PTR DS:[EDI]
3C 41         CMP AL,41 # cipher key
74 02         JE G
34 41         XOR AL,41 # cipher key
G:
8806          MOV BYTE PTR DS:[ESI],AL
47            INC EDI
46            INC ESI
E2 F2        LOOPD DEC
```



# Encoders/Decoders

## ■ Addition/subtraction encoder

- Uses **add/sub** instructions, instead of **xor**
- *Example:* <https://github.com/h0mbre/Myth>

## ■ Shikata Ga Nai polymorphic XOR additive feedback encoder

- **Rotating key:** it changes the key in each round!
- Helps prevent detection based on signatures (e.g., byte patterns)

## ■ Other variants:

- XOR-ROR additive feedback (<https://github.com/Re4son/slae-4>)
- ...

# Encoders/Decoders

## ■ Addition/subtraction encoder

- Uses **add/sub** instructions, instead of **xor**
- *Example:* <https://github.com/h0mbre/Myth>

## ■ Shikata Ga Nai polymorphic XOR additive feedback encoder

- **Rotating key:** it changes the key in each round!
- Helps prevent detection based on signatures (e.g., byte patterns)

## ■ Other variants:

- XOR-ROR additive feedback (<https://github.com/Re4son/slae-4>)
- ...

## Custom encoders/decoders

## ■ Customize your encoder/decoder!

## ■ **Always following these steps:**

- 1 Choose an encoding mechanism
- 2 Develop an encoder
- 3 Develop a decoder
- 4 Decoder must be located before the modified payload

## ■ **Tedious manual work, but (almost) all filters can be skipped!**

# Encoders/Decoders

## *Encoders available in Metasploit*

Name	Rank	Description
x86/add_sub	manual	Add/Sub Encoder
x86/alpha_mixed	low	Alpha2 Alphanumeric Mixedcase Encoder
x86/alpha_upper	low	Alpha2 Alphanumeric Uppercase Encoder
x86/avoid_underscore_tolower	manual	Avoid underscore/tolower
x86/avoid_utf8_tolower	manual	Avoid UTF8/tolower
x86/bloxor	manual	BloXor – A Metamorphic Block Based XOR Encoder
x86/bmp_polyglot	manual	BMP Polyglot
x86/call4_dword_xor	normal	Call+4 Dword XOR Encoder
x86/context_cpuid	manual	CPUID-based Context Keyed Payload Encoder
x86/context_stat	manual	stat(2)-based Context Keyed Payload Encoder
x86/context_time	manual	time(2)-based Context Keyed Payload Encoder
x86/countdown	normal	Single-byte XOR Countdown Encoder
x86/fnstenv_mov	normal	Variable-length Fnstenv/mov Dword XOR Encoder
x86/jmp_call_additive	normal	Jump/Call XOR Additive Feedback Encoder
x86/nonalpha	low	Non-Alpha Encoder
x86/nonupper	low	Non-Upper Encoder
x86/opt_sub	manual	Sub Encoder (optimised)
x86/service	manual	Register Service
x86/shikata_ga_nai	excellent	Polymorphic XOR Additive Feedback Encoder
x86/single_static_bit	manual	Single Static Bit
x86/unicode_mixed	manual	Alpha2 Alphanumeric Unicode Mixedcase Encoder
x86/unicode_upper	manual	Alpha2 Alphanumeric Unicode Uppercase Encoder
x86/xor_dynamic	normal	Dynamic key XOR Encoder

## Steps to prepare an encoder/decoder that works

- Recognize the filter in the vulnerable program
- Know (in detail) the underlying ISA

# Outline

- 1 A Little Recap
- 2 Payload Types
- 3 Filters
- 4 Encoders/Decoders
- 5 Payload Components**

# Payload Components

## Restore privileges

- Useful on Unix-like systems: effective user ID vs real user ID
  - `eid` governs what access the process has
  - `uid` determines who the user really is
- Some programs may drop privileges before execution (e.g., `/etc/sh` in the latest versions of GNU/Linux and macOS)
- You can run `seteuid(0)` before the shellcode payload to get an elevated shell (in a `+s` program)

```
xor eax, eax
mov al, 70
xor ebx, ebx
xor ecx, ecx
int 0x80
```

# Payload Components

## Creation of new processes

- Some systems (like macOS) may require your program to call `vfork()` beforehand to run a new process
  - Otherwise, `execve()` will return the error `ENOTSUP`
- `vfork()` is like `fork()`, except that the parent process is suspended until the child process executes the `execve()` system call or exits

# Payload Components

## Shell execution

- Minimal payload to run a shell
- You have worked with this payload before, see the previous topic slides (or lab workbooks)!
- Note that on some systems, a drop of privileges may occur by default as a good practice of security principles
- On remote, variants: *bind shell* and *reverse shell*

```
xor    eax, eax
push   eax
push   0x68732f2f
push   0x6e69622f
mov    ebx, esp
push   eax
push   ebx
mov    ecx, esp
mov    al, 0xb
int    0x80
```

# Payload Components

## Bind shell

- Payload that opens a listening port
- When the attacker connects, it automatically launches a shell
- Think of a **client/server architecture**:
  - The attacker acts as a client, the target acts as a server





# Payload Components

## Reverse shell

- Payload that connects to a specific address
- When connecting to the address, it automatically launches a shell
- Think of a **client/server architecture**:
  - **The attacker acts as a server, the target acts as a client**
- **Useful to bypass firewalls or other port blocking procedures**



# Payload Components

## Redirection of std to fds

- Duplicate a socket file descriptor (std) into standard input, standard output, and standard error file descriptors (fds)
- Useful to remotely interact with the target system through the socket

# Payload Components

## Redirection of std to fds

- Duplicate a socket file descriptor (std) into standard input, standard output, and standard error file descriptors (fds)
- Useful to remotely interact with the target system through the socket

## Staged payload

- Useful to avoid payload size constraints
- Each stage prepares the runtime environment for the next stage, allowing the next stage to run with fewer constraints
  - For instance, the first stage can search for the subsequent stage somewhere else in memory and decode it, or download it over the network, and then run it (or inject it into a running process)

# Exploiting Software Vulnerabilities

## Advanced Exploitation Techniques

### EXPLOIT PAYLOADS

© All wrongs reversed – under CC-BY-NC-SA 4.0 license



**Universidad**  
Zaragoza

Dept. of Computer Science and Systems Engineering  
University of Zaragoza, Spain

Course 2023/2024

**Master's Degree in Informatics Engineering**  
UNIVERSITY OF ZARAGOZA

*Room A.02, Ada Byron building*

