TU WIEN S&P Security & Privacy CYSEC CYBERSECURITYCENTER

Memory Corruption Attacks and Defenses

Introduction to Security (192.019)

Pedro **Bernardo**, Mauro **Tempesta**

Security & **Privacy** Research Unit (192-06)
https://**secpriv.wien**

Buffer Overflows

Computer Programs: Assumptions

- Programming languages (including C) offer several layers of abstraction
  - functions, control flow, variables, etc.
- Naturally, our execution model makes some assumptions:
  - Basic statements are atomic (e.g., assignments)
  - Functions start at the beginning, and execute until the end
  - When a function ends, execution returns to its call site
  - Only one branch of an `if` statement is taken at a time
  - Only program code can be executed
  - The set of executable instructions is limited to those output during the compilation of the program

Computer Programs: Assumptions vs Reality

- But, in reality (at the level of machine code)...
  - Basic statements are compiled into multiple instructions
  - Execution can start in the middle of functions
  - `return` instructions can go to any program location
  - There are no restrictions on branch targets
  - Dead code (e.g., unused library functions) can be executed
  - On x86-64 execution can also start in the middle of instructions!

Buffer Overflow - High-Level Overview

- Buffer overflows allow writing outside the bounds of a buffer
  - C/C++ expect the programmer to ensure this doesn't happen
  - But, humans make mistakes!
- Buffer overflows can cause program crashes (and usually do)
  - However, attackers can exploit buffer overflows to:
    - Steal confidential information
    - Corrupt or modify valuable information
    - Execute arbitrary code

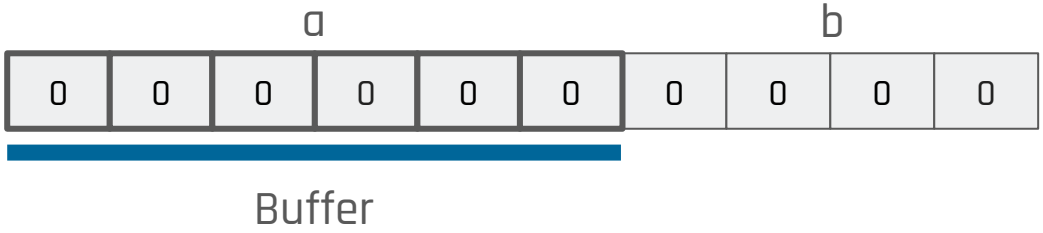
Buffer Overflow

Assumption: Buffer a is large enough to store the data.

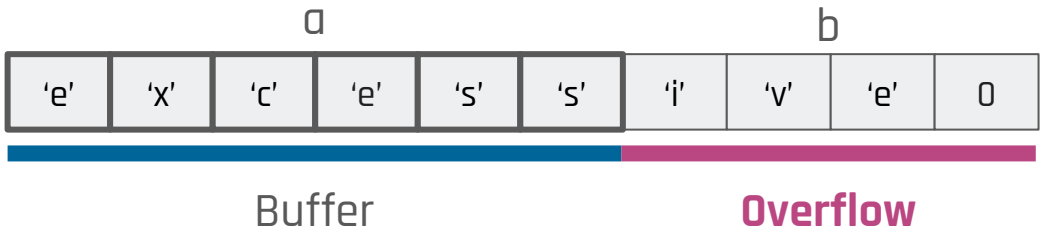
But what if the attacker provides larger data?

Buffer overflows enable a large amount of **stack corruption** attacks potentially leading to **control-flow hijacking.**

```
char a[6];
int b;
```

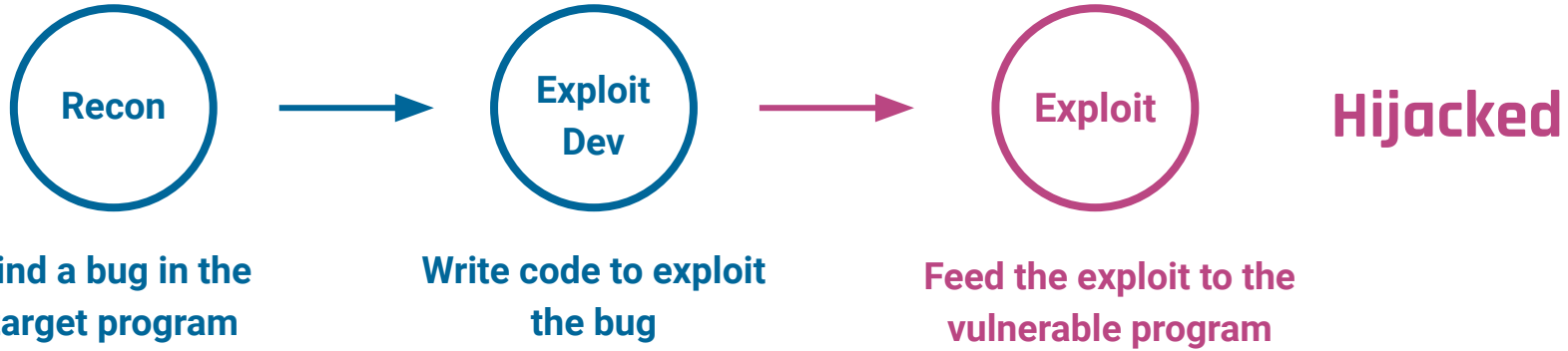



```
strcpy(a, "excessive");
```



Control-Flow Hijacking Attacks

- Goal: make the target program execute attacker-controlled code
  - There are many ways to achieve this goal, but a prominent one is by exploiting buffer overflows
- The attacker pattern is always similar:

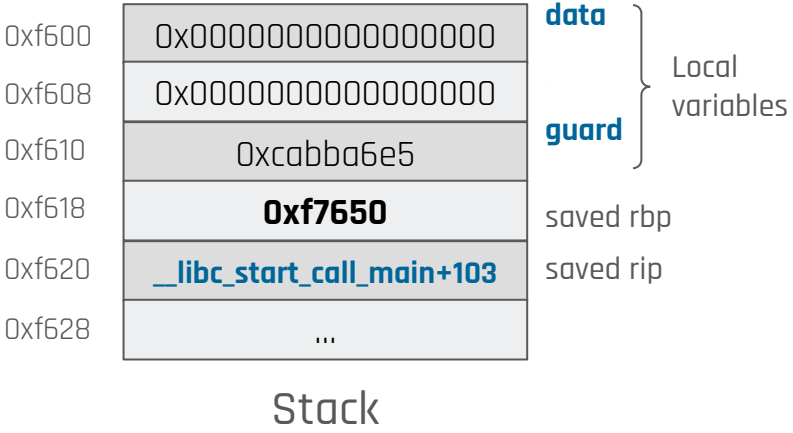


Stack Corruption - Overview

Buffer overflows on the stack allow for a range of stack corruption attacks with the main goal of hijacking the control-flow of a program:

- **Local variable clobbering**
  - Overwrite local variables on the stack to divert the execution flow
- **Function pointer clobbering**
  - Overwrite function pointers on the stack to completely control the execution flow
- **Instruction pointer hijacking**
  - Overwrite the saved instruction pointer on the stack to completely control the execution flow
- **Frame pointer hijacking**
  - Overwriting the saved frame/base pointer on the stack to move the caller's stack frame (potentially to attacker-controlled memory)

Local Variable Clobbering

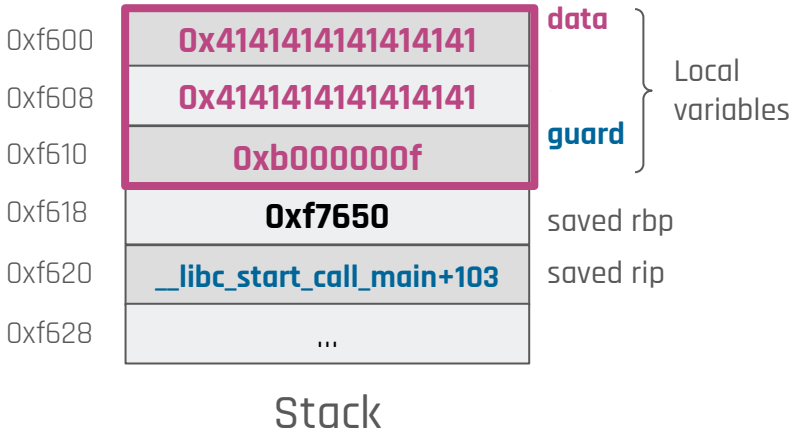



```c
int main(void) {
    long guard = 0xcabba6e5;
    char data[0x10] = {0};

    gets(data);

    if(guard == 0xb000000f) {
        printf("Win \\o/\n");
    } else {
        printf("N00b :(\n");
    }

    return 0;
}
```

Under normal circumstances, “Win \o/” should never be printed

Local Variable Clobbering

By overflowing into the **guard** variable, an attacker is able to bypass the check and reach the **"Win \0/"** message

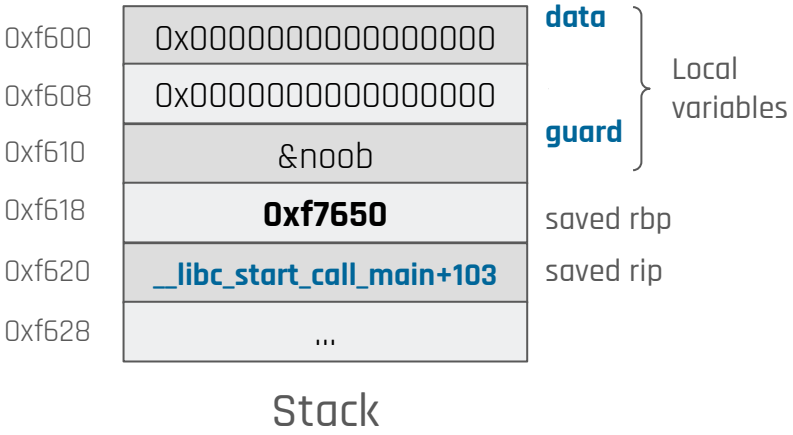



```c
int main() {
    long guard = 0xcabba6e5;
    char data[0x10] = {0};

    gets(data);

    if(guard == 0xb000000f) {
        printf("Win \\o/\n");
    } else {
        printf("N00b :(\n");
    }

    return 0;
}
```

gets does not limit the number of characters read

Function Pointer Clobbering

Clobbering function pointers gives attackers complete control over the program's execution flow.



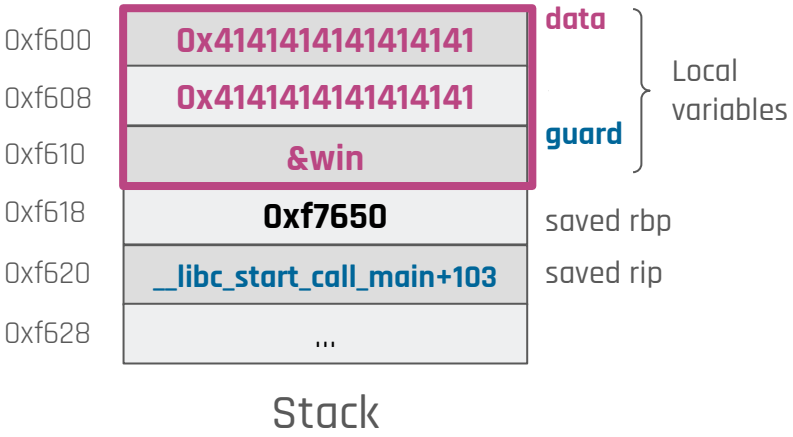

```c
void noob(){
    printf("N00b :(\n");
}

void win(){
    printf("Win \\o/\n");
}

int main() {
    void (*fptr)(void) = &noob;
    char data[0x10] = {0};

    gets(data);
    (*fptr)();

    return 0;
}
```

Function pointer is declared

Function pointer is called

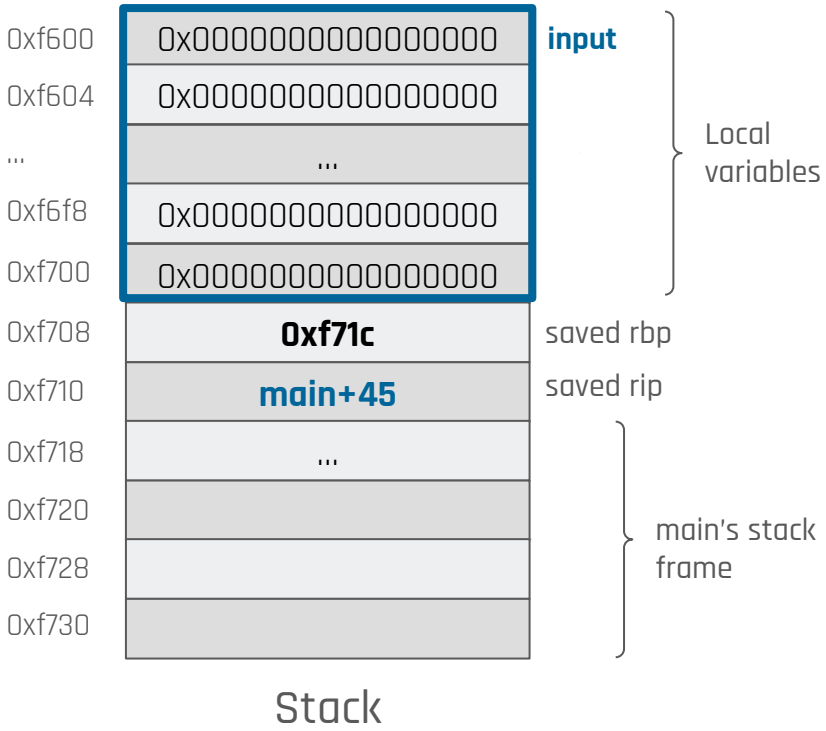
Function Pointer Clobbering

The attacker overwrites the local function pointer `fptr` to point to the `win` function.



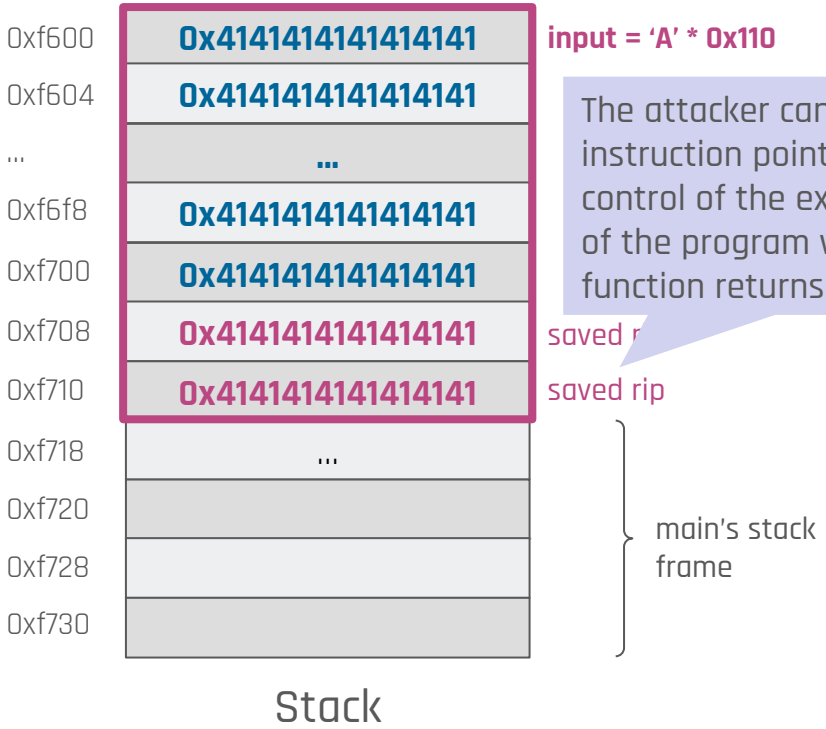

```c
void noob(){
    printf("N00b :(\n");
}

void win(){
    printf("Win \\o/\n");
}

int main() {
    void (*fptr)(void) = &noob;
    char data[0x10] = {0};

    gets(data);
    (*fptr)();

    return 0;
}
```

Smashing the Stack (For Fun and Profit?)




```c
void vuln(){
    char input[0x100]= {0};
    gets(input);
    return;
}

int main(){
    vuln();
    return 0;
}
```

Smashing the Stack (For Fun and Profit?)

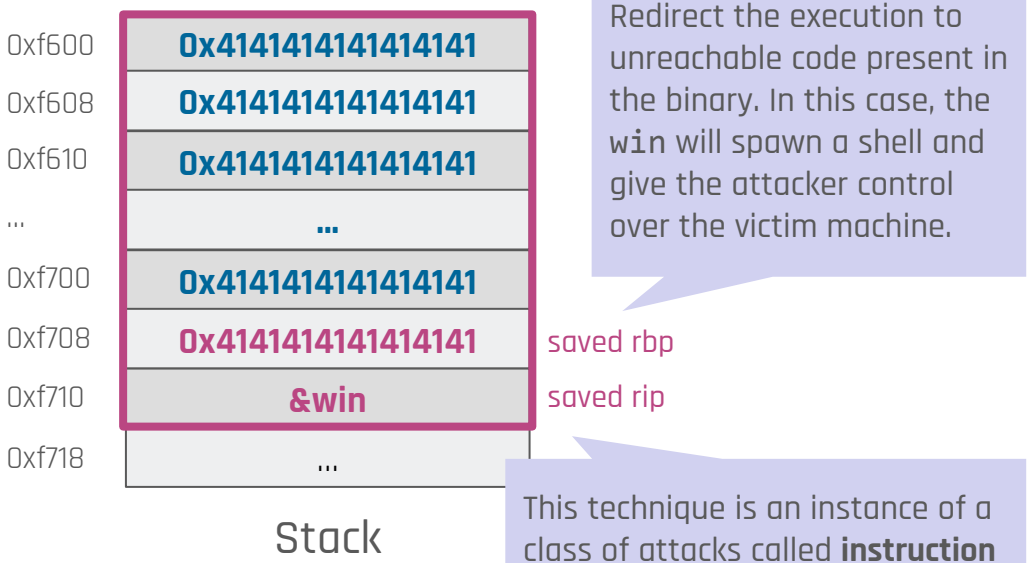


The attacker can overwrite the instruction pointer, gaining control of the execution flow of the program when the **vuln** function returns!

```
void vuln(){
    char input[0x100]= {0};
    gets(input);
    return;
}

int main(){
    vuln();
    return 0;
}
```

Instruction Pointer Hijacking

And what can an attacker do with control of the instruction pointer?



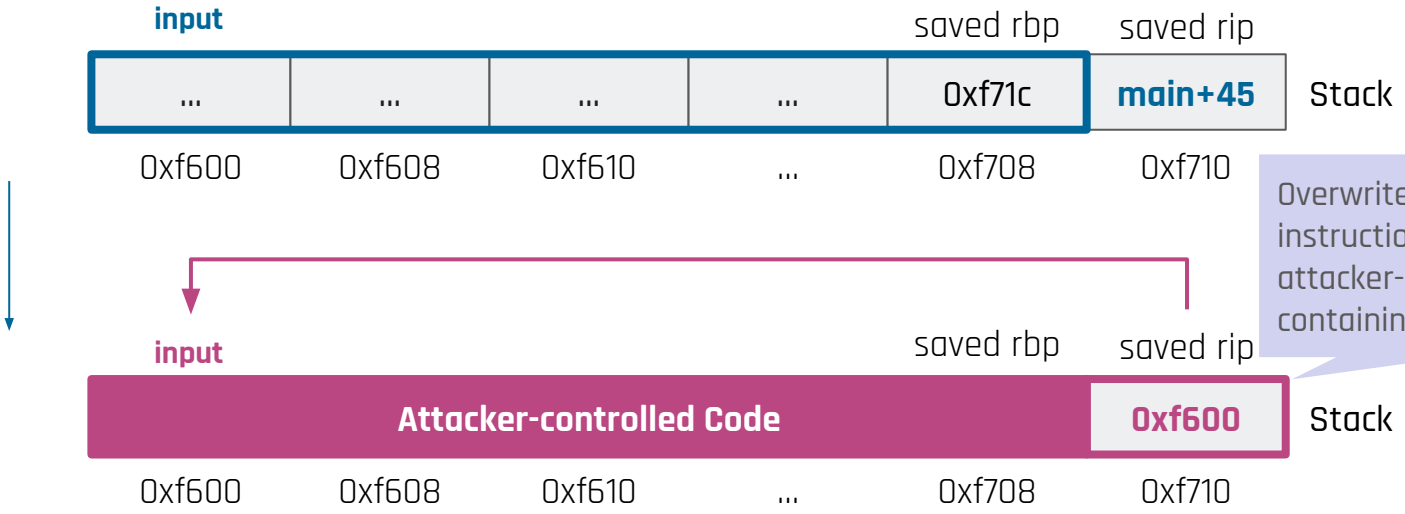
Redirect the execution to unreachable code present in the binary. In this case, the `win` will spawn a shell and give the attacker control over the victim machine.

This technique is an instance of a class of attacks called **instruction pointer hijacking**

```c
void win(){
    system("/bin/sh");
}

void vuln(){
    char input[0x100] = {0};
    gets(input);
    return;
}

int main(){
    vuln();
    return 0;
}
```

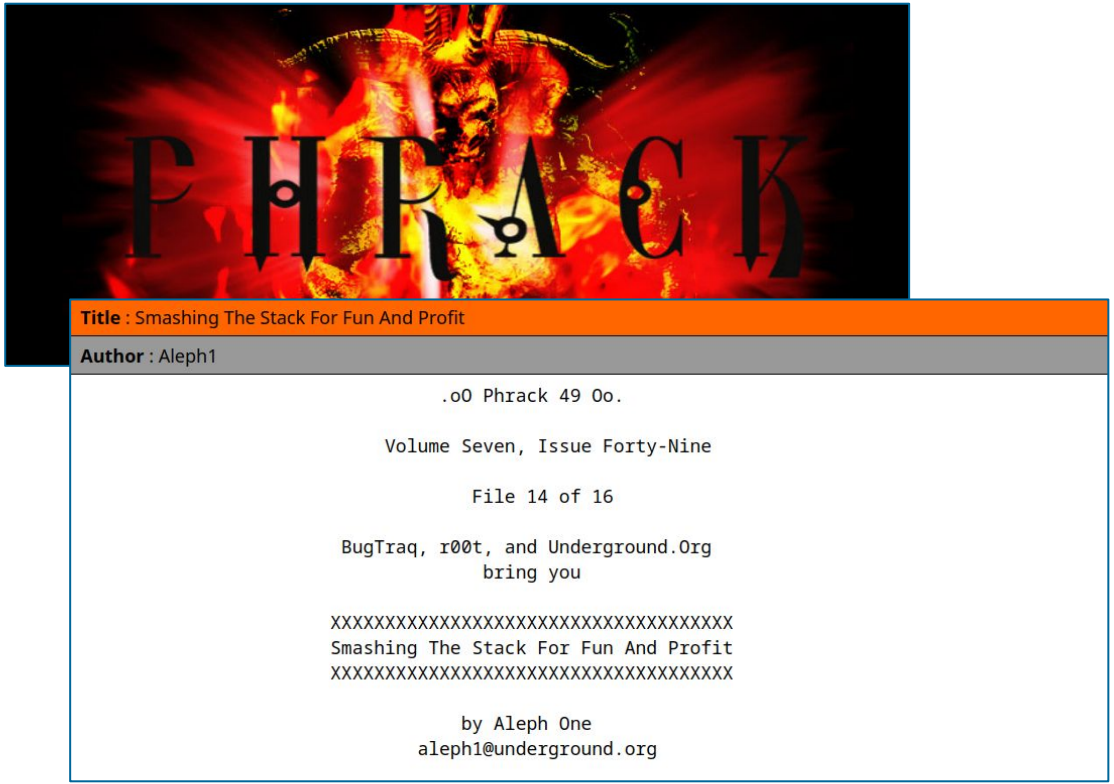
Instruction Pointer Hijacking

But, what if there are no convenient functions or instructions present in the binary?



Also known as **shellcode**, since historically, its main purpose has been to spawn a shell – `execve("/bin/sh")`

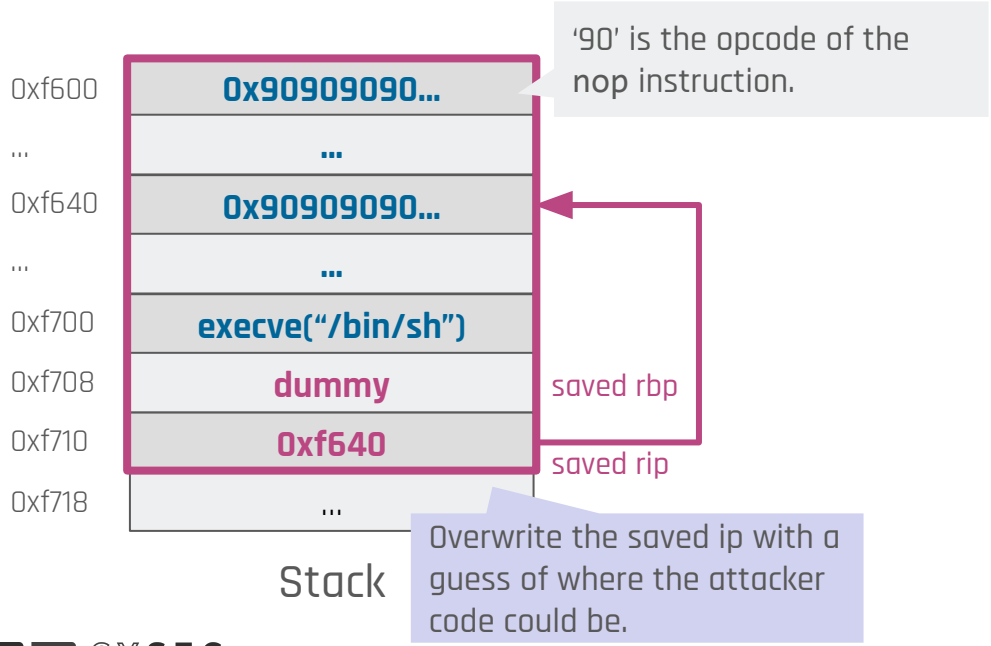
Smashing the Stack For Fun and Profit



Shellcode was originally introduced by Aleph One in the classic Phrack article "Smashing The Stack For Fun And Profit" from 1996

Instruction Pointer Hijacking - Shellcode

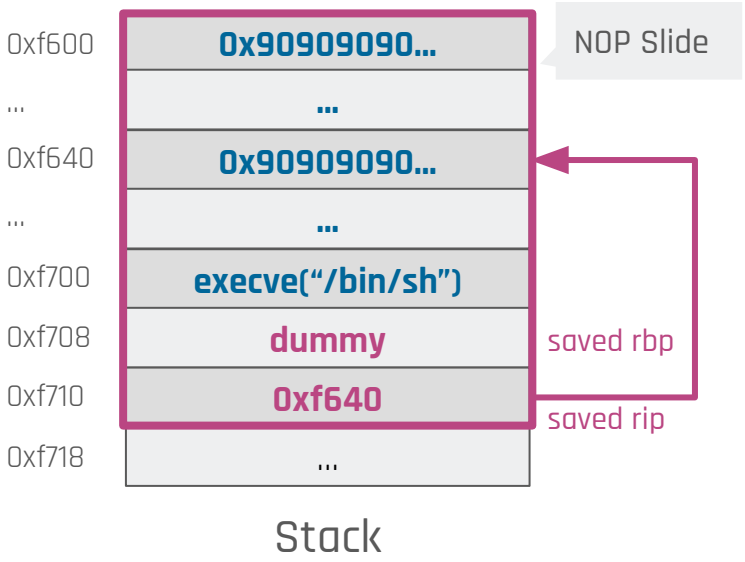
However, stack addresses are not always predictable.
How does the attacker know where to jump to?




```c
void vuln(){
    char input[0x100]= {0};
    gets(input);
    return;
}

int main(){
    vuln();
    return 0;
}
```

Instruction Pointer Hijacking - Shellcode

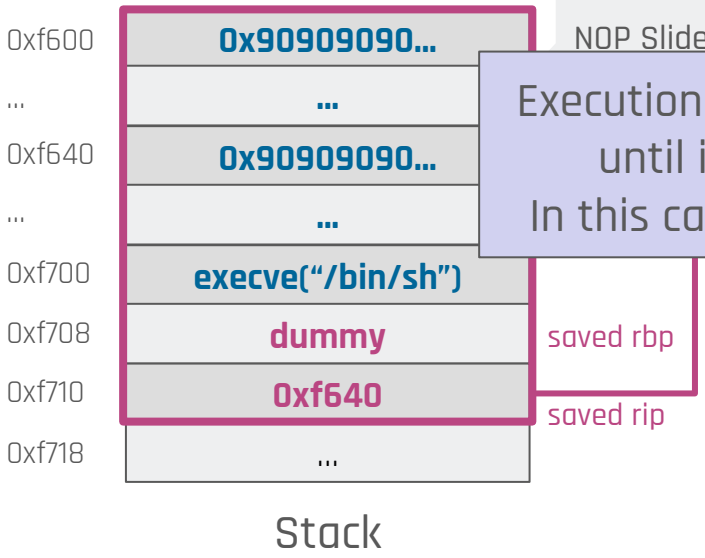
If the guess was correct, execution will be directed to attacker code and land in the middle of a nop slide.




```
void vuln(){
    char input[0x100]= {0};
    gets(input);
    return;
}

int main(){
    vuln();
    return 0;
}
```

Instruction Pointer Hijacking - Shellcode

If the guess was correct, execution will be directed to attacker code and land in the middle of a nop slide.



Execution will go down the nop slide until it reaches the payload.
In this case `execve("/bin/sh")`

```
void vuln(){
    char input[0x100]= {0};
    gets(input);
    return;
}

main(){
    vuln();
    return 0;
}
```

Shellcode Payload Example

This is a compact (26 bytes) "execve(/bin/sh)" shellcode.

Offset	Opcode	Assembly
0:	48 31 f6	xor rsi,rsi
3:	56	push rsi
4:	48 bf 2f 62 69 6e 2f	movabs rdi,0x68732f2f6e69622f
b:	2f 73 68	
e:	57	push rdi
f:	54	push rsp
10:	5f	pop rdi
11:	b0 3b	mov al,0x3b[1]
13:	48 98	cdqe
15:	0f 05	syscall

This shellcode has no **null bytes**, making it useful for when the input is read through string functions that stop at the string terminator character!

set `rsi` to 0 and move it to the top of the stack

move the bytes corresponding to the string "/bin//sh" to the top of the stack

put the address of the top of the stack (pointing to "/bin//sh") on the stack and pop it to `rdi`

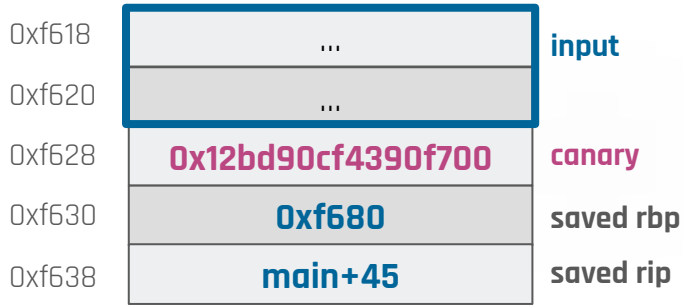
set `rax` to 0x3b (the execve syscall number) and perform the `syscall`

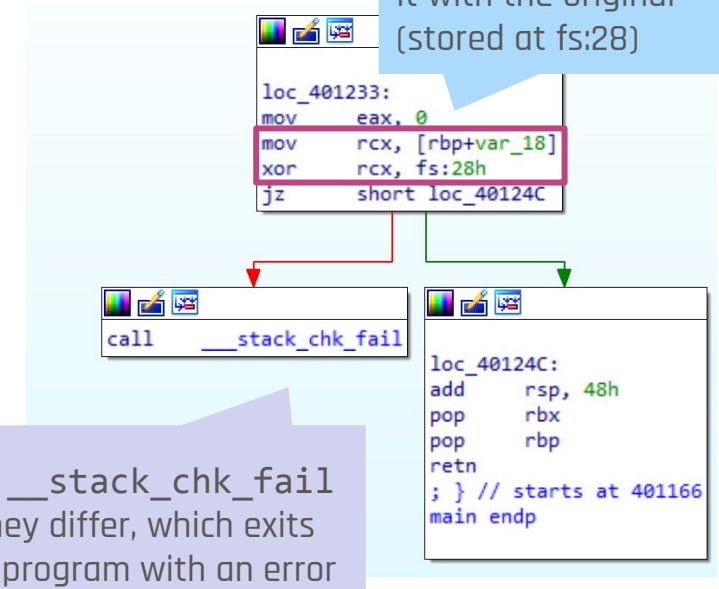
[1] see https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

Instruction Pointer Hijacking - Mitigations

Stack Canaries/Cookies place a value in the stack, before the **saved rip** and **rbp**, and check it before returning from the function

- Canary is **randomly** determined **at runtime** and remains constant for that specific execution
- Its least-significant byte is always **0x00** to stop buffer over-reads
- Program must be **recompiled** to enable canary support



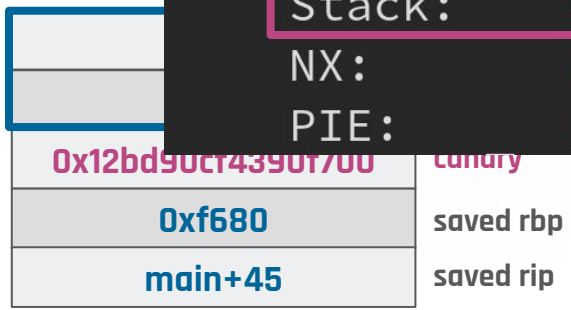


Instruction Pointer Hijacking - Mitigations

Stack Canaries/Cookies place a value in the stack, before the **saved rip** and **rbp**, and check it before returning from the function

- Canary is **randomly**
  remains constant f
- Its least-significant
  over-reads
- Program must be r

```
λ juno ~ → checksec binary
[*] '/home/pedro/binary'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```



Take the canary from the stack and compare it with the original (stored at fs:28)

```
loc_401233:
mov     eax, 0
mov     rcx, [rbp+var_18]
xor     rcx, fs:28h
jz      short loc_40124C
```

```
ck_chk_fail
```

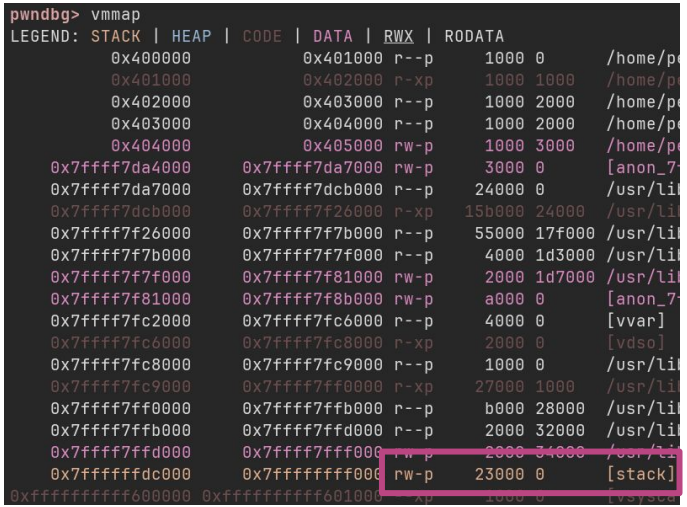
```
loc_40124C:
add     rsp, 48h
pop     rbx
pop     rbp
retn
; } // starts at 401166
main endp
```

Call __stack_chk_fail if they differ, which exits the program with an error

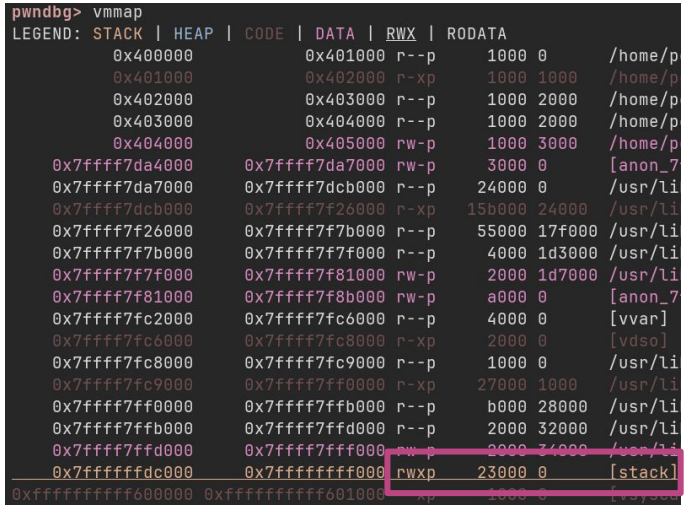
Data Execution Prevention / NX

- **DEP** (also known as **NX**) is a security feature that makes the stack **N**ot e**X**ecutable
- Ensures that **only code segments are marked as executable**
- **Writable and executable permissions** on memory segments **are mutually exclusive**.

NX enabled



NX disabled



Data Execution Prevention / NX

- **DEP** (also known as **NX**) is a security feature that makes the stack **N**ot e**X**ecutable
- Ensures that **only code segments are marked as executable**
- **Writable and execu lly exclusive.**

NX enabled

```
λ juno ~ → checksec binary
[*] '/home/pedro/binary'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

```
pwndbg> vmmap
LEGEND: STACK | HEAP | C
          0x400000
          0x401000
          0x402000
          0x403000
          0x404000
    0x7ffff7da4000     0
    0x7ffff7da7000     0
    0x7ffff7dcb000     0
    0x7ffff7f26000     0
    0x7ffff7f7b000     0
    0x7ffff7f7f000     0
    0x7ffff7f81000     0
    0x7ffff7fc2000     0
    0x7ffff7fc6000     0x7ffff7fc8000 r-xp     2000 0      [vdso]
    0x7ffff7fc8000     0x7ffff7fc9000 r--p     1000 0      /usr/li
    0x7ffff7fc9000     0x7ffff7ff0000 r-xp    27000 1000   /usr/li
    0x7ffff7ff0000     0x7ffff7ffb000 r--p     b000 28000  /usr/li
    0x7ffff7ffb000     0x7ffff7ffd000 r--p     2000 32000  /usr/li
    0x7ffff7ffd000     0x7ffff7fff000 rw-p     2000 34000  /usr/li
    0x7ffffffdc000     0x7ffffffff000 rw-p    23000 0      [stack]
0xffffffffff600000 0xffffffffff601000 --xp     1000 0      [vsysca
```

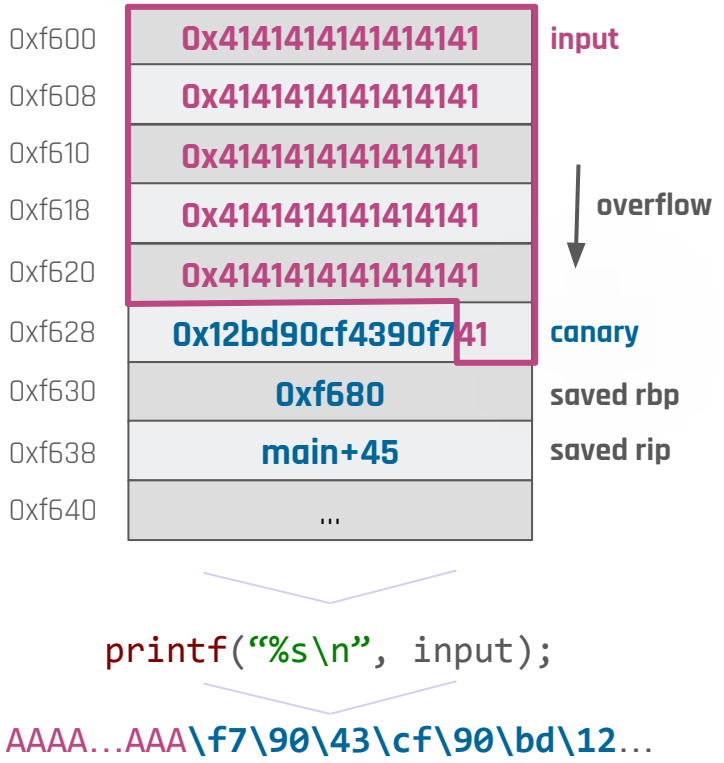
NX disabled

```
RODATA
    1000 0      /home/p
    1000 1000   /home/p
    1000 2000   /home/p
    1000 2000   /home/p
    1000 3000   /home/p
    3000 0      [anon_7
   24000 0      /usr/li
   15b000 24000 /usr/li
   55000 17f000 /usr/li
    4000 1d3000 /usr/li
    2000 1d7000 /usr/li
    a000 0      [anon_7
    4000 0      [vvar]
    0x7ffff7fc6000     0x7ffff7fc8000 r-xp     2000 0      [vdso]
    0x7ffff7fc8000     0x7ffff7fc9000 r--p     1000 0      /usr/li
    0x7ffff7fc9000     0x7ffff7ff0000 r-xp    27000 1000   /usr/li
    0x7ffff7ff0000     0x7ffff7ffb000 r--p     b000 28000  /usr/li
    0x7ffff7ffb000     0x7ffff7ffd000 r--p     2000 32000  /usr/li
    0x7ffff7ffd000     0x7ffff7fff000 rw-p     2000 34000  /usr/li
    0x7ffffffdc000     0x7ffffffff000 rwxp    23000 0      [stack]
0xffffffffff600000 0xffffffffff601000 --xp     1000 0      [vsysca
```

Stack Canary Bypasses

Stack canaries can be bypassed in a few ways:

- **Leaking the canary**
  - Achieved by abusing a buffer overflow to partially overwrite the least significant byte of the canary. When paired with a string printing function can lead to a buffer over-read, printing also the canary value to stdout
- **Override the call to `__stack_chk_fail`**, which is called when the stack canary is corrupted
  - This requires other primitives we will see later…
- In general, it depends on the binary
  - If the binary contains the instruction `*a = b`, where the values of `a` and `b` are controlled by the attacker, it is possible to overwrite the saved address without touching the canary

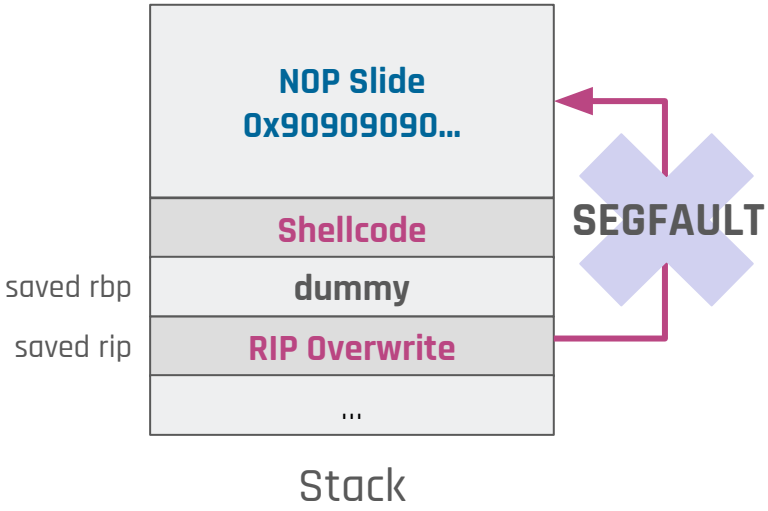


NX/DEP Bypasses

NX makes the stack not executable

- In general, data should never be executable
- Trying to execute data leads to a **SEGFAULT**
- So, attackers can no longer execute shellcode

But, what if an attacker redirects execution to code that is already present in the binary?



Return Oriented Programming

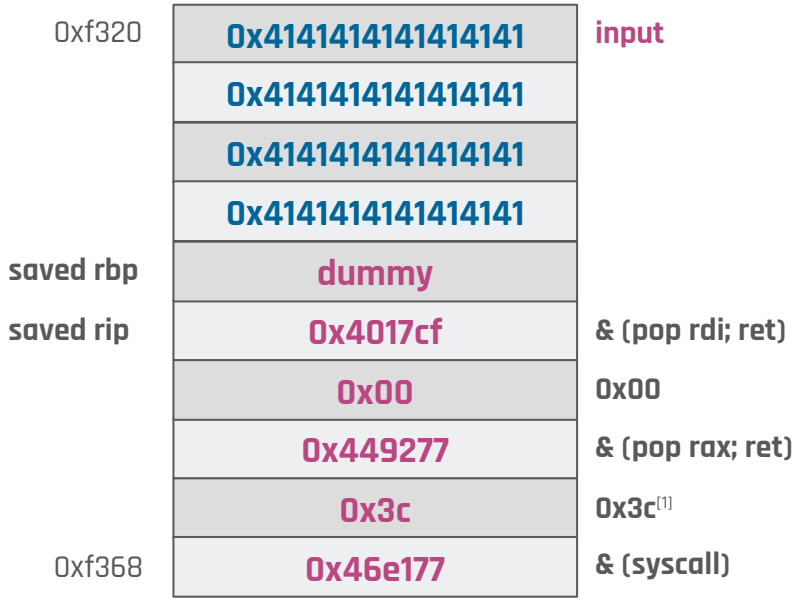
Return-Oriented Programming - Terminology

- Return-Oriented Programming (ROP)
  - Exploitation technique that consists in reusing existing code snippets in the target binary
- ROP Gadget
  - Sequence of instructions, usually followed by a `ret` instruction
  - Examples:
    - `pop rdi ; ret`
    - `pop rax ; pop rbx ; ret`
    - `xor rax, rax ; ret`
    - `mov qword [rsi], rax ; ret`
- ROP Chain
  - A sequence of ROP gadgets chained together to perform a given task
  - Usually, this task involves spawning a shell, similar to a shellcode payload

ROP - How does it work?

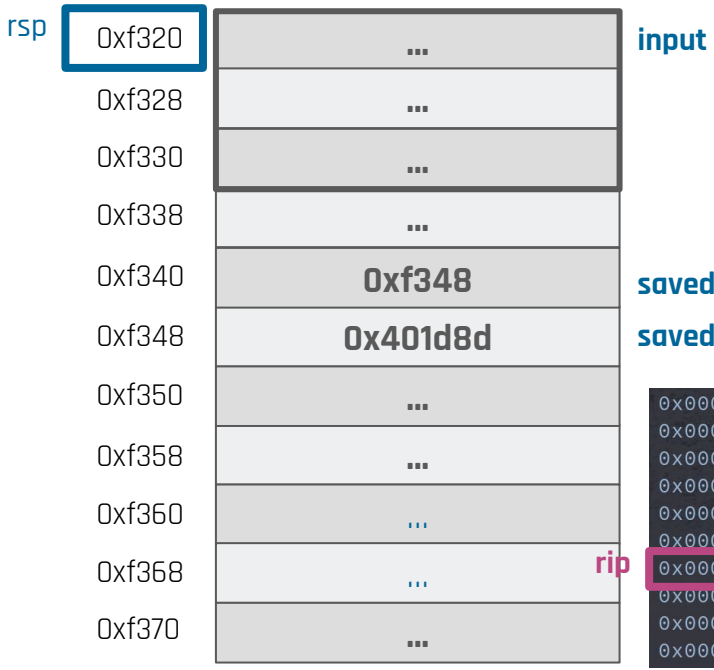
- The **ret** instruction copies the contents of the top of the stack to the instruction pointer
  - equivalent to:
    `mov rip, [rsp] ; add rsp, 8`
- The **ret** instruction of every gadget will consume the next gadget's address by popping off the stack into **rip**
  - Note that changes to the stack made by gadgets must be taken into account

`exit(0)` ROP chain



[1] see https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

ROP - Walkthrough



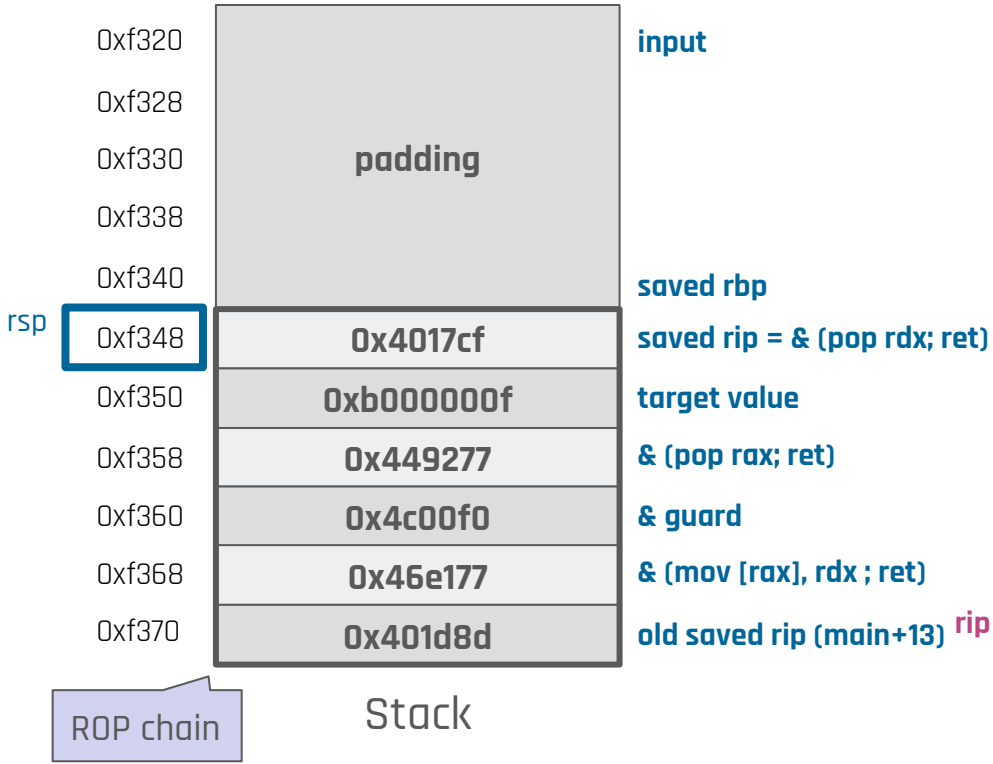
rax: ????????
rdx: ????????
rsp: 0xf320
rbp: 0xf340
rip: 0x401d78 <readstuff+19>

```
0x0000000000401d65 <+0>:     endbr64
0x0000000000401d69 <+4>:     push   rbp
0x0000000000401d6a <+5>:     mov    rbp,rsp
0x0000000000401d6d <+8>:     sub    rsp,0x20
0x0000000000401d71 <+12>:    lea    rax,[rbp-0x20]
0x0000000000401d75 <+16>:    mov    rdi,rax
0x0000000000401d78 <+19>:    call   0x411700 <gets>
0x0000000000401d7d <+24>:    nop
0x0000000000401d7e <+25>:    leave
0x0000000000401d7f <+26>:    ret
```

```
int guard = 0xcabba6e5;

void readstuff(){
    char data[20];
    gets(data);
}

int
main(){
    readstuff();

    if(guard == 0xb000000f)
        printf("Win \\o/\n");
    else
        printf("N00b :(\n");

    return 0;
}
```

The goal is to modify the **guard** variable to reach the “win” printf

ROP - Walkthrough

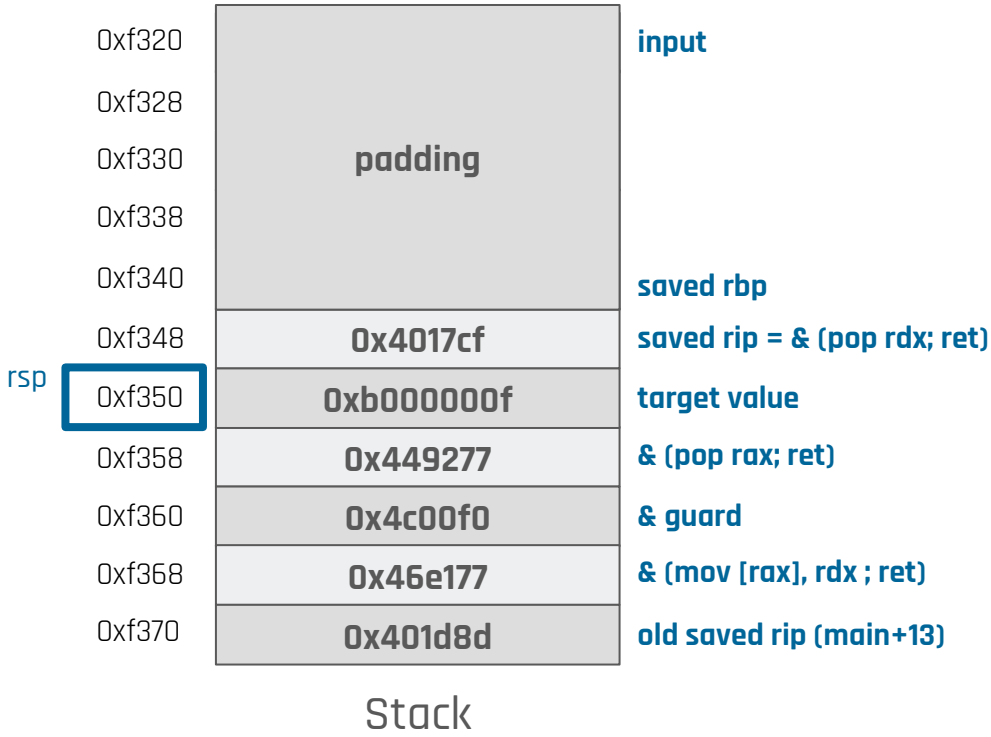


rax: ????????
rdx: ????????
rsp: 0xf348
rbp: AAAAAAAA
rip: 0x401d7f <readstuff+26>

```
0x0000000000401d65 <+0>:     endbr64
0x0000000000401d69 <+4>:     push   rbp
0x0000000000401d6a <+5>:     mov    rbp,rsp
0x0000000000401d6d <+8>:     sub    rsp,0x20
0x0000000000401d71 <+12>:    lea    rax,[rbp-0x20]
0x0000000000401d75 <+16>:    mov    rdi,rax
0x0000000000401d78 <+19>:    call   0x411700 <gets>
0x0000000000401d7d <+24>:    nop
0x0000000000401d7e <+25>:    leave
0x0000000000401d7f <+26>:    ret
```

rip

ROP - Walkthrough

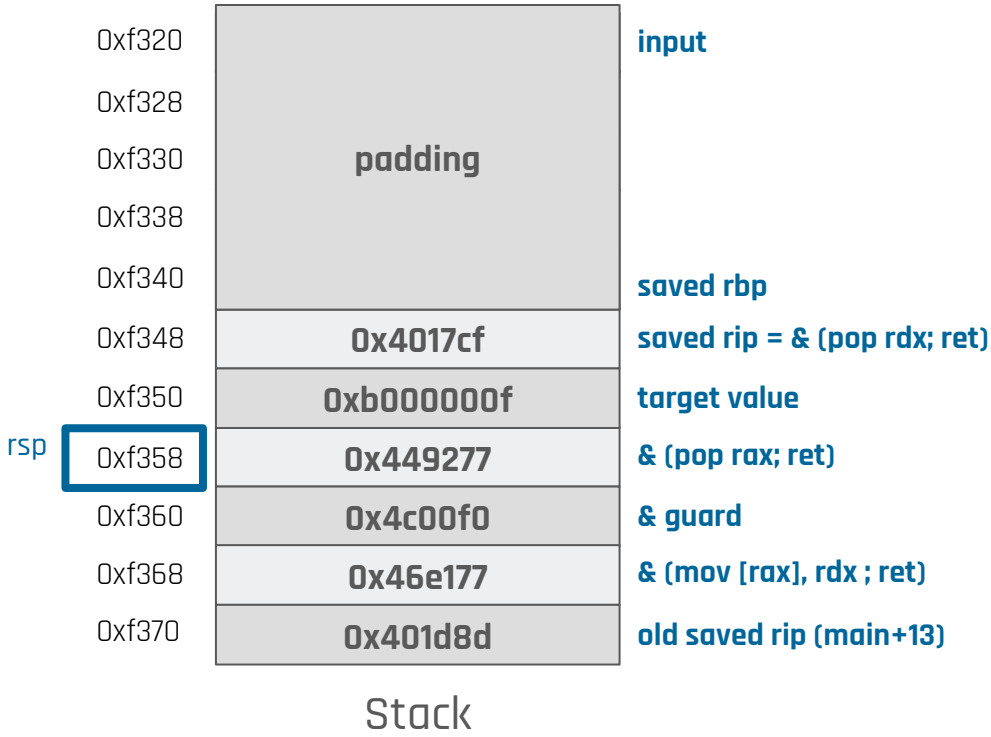


rax: ????????
rdx: ????????
rsp: 0xf350
rbp: AAAAAAAA
rip: 0x4017cf <fini+63>

rip

```
0x4017cf <fini+63>:  pop    rdx
0x4017d0 <fini+64>:  ret
```

ROP - Walkthrough

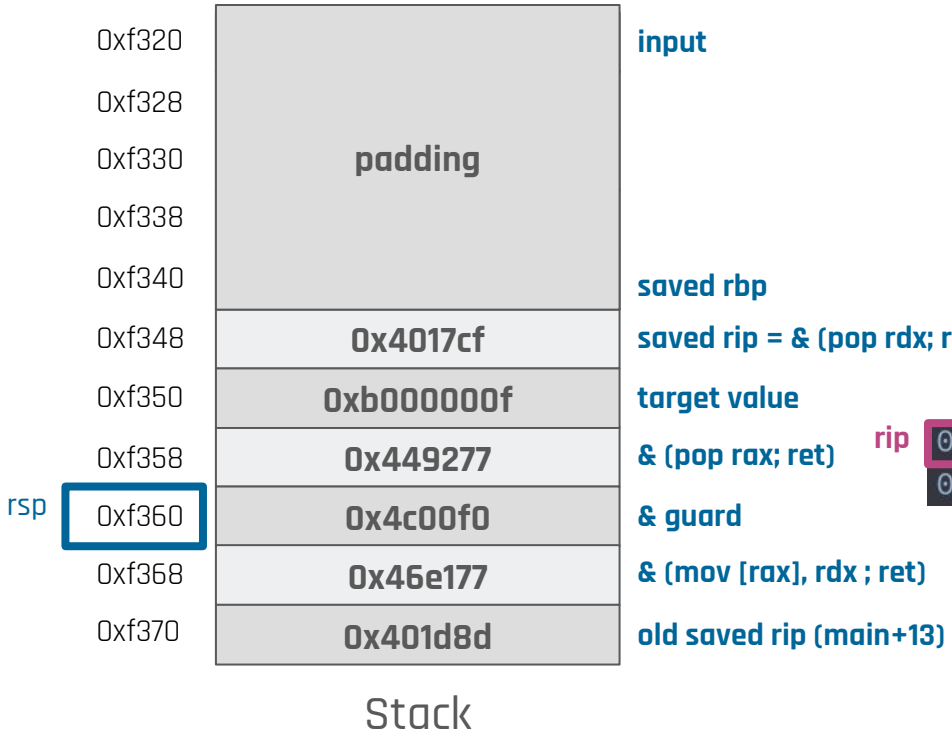


rax: ????????
rdx: 0xb000000f
rsp: 0xf358
rbp: AAAAAAAA
rip: 0x4017d0 <fini+64>

```
0x4017cf <fini+63>:  pop    rdx
0x4017d0 <fini+64>:  ret
```

rip

ROP - Walkthrough

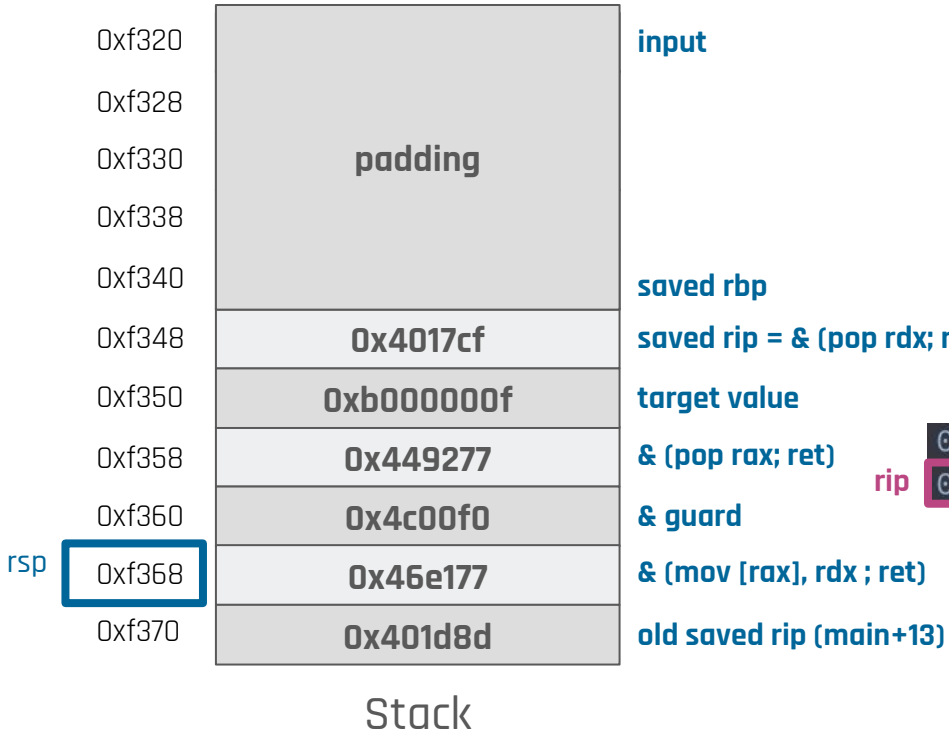


rax: ????????
rdx: 0xb000000f
rsp: 0xf350
rbp: AAAAAAAA
rip: 0x449277 <__open...+103>

rip

```
0x449277 <__open_nocancel+103>:      pop    rax
0x449278 <__open_nocancel+104>:      ret
```

ROP - Walkthrough

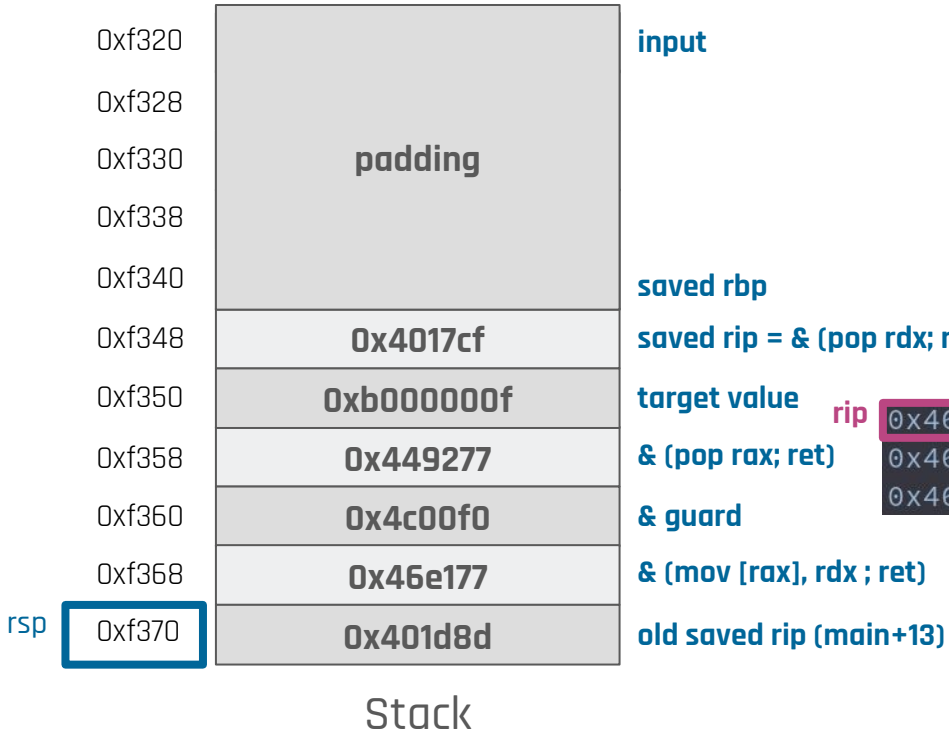


rax: 0x4c00f0
rdx: 0xb000000f
rsp: 0xf350
rbp: AAAAAAAA
rip: 0x449278 <__open...+104>

rip

```
0x449277 <__open_nocancel+103>:      pop    rax
0x449278 <__open_nocancel+104>:      ret
```

ROP - Walkthrough

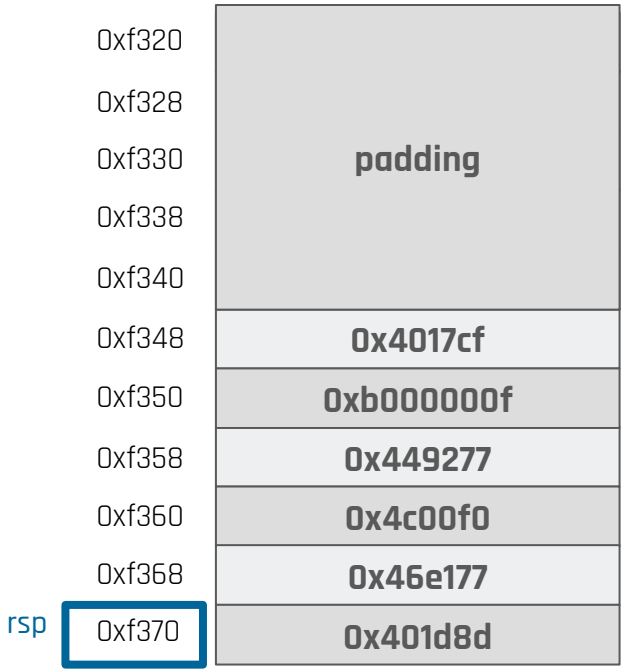


rax: 0x4c00f0
rdx: 0xb000000f
rsp: 0xf350
rbp: AAAAAAAA
rip: 0x46e177 <_IO_see...+87>

rip

```
0x46e177 <_IO_seekwmark+87>: mov    QWORD PTR [rax],rdx
0x46e17a <_IO_seekwmark+90>: xor    eax,eax
0x46e17c <_IO_seekwmark+92>: ret
```

ROP - Walkthrough



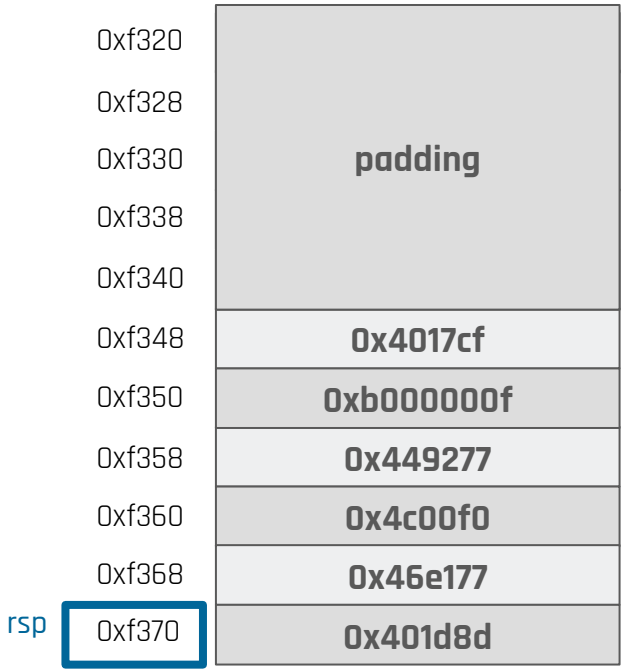
rax: 0x4c00f0
rdx: 0xb000000f
rsp: 0xf350
rbp: AAAAAAAA
rip: 0x46e17a <_IO_see...+90>

guard = 0xb000000f

rip

```
0x46e177 <_IO_seekwmark+87>: mov    QWORD PTR [rax],rdx
0x46e17a <_IO_seekwmark+90>: xor    eax,eax
0x46e17c <_IO_seekwmark+92>: ret
```

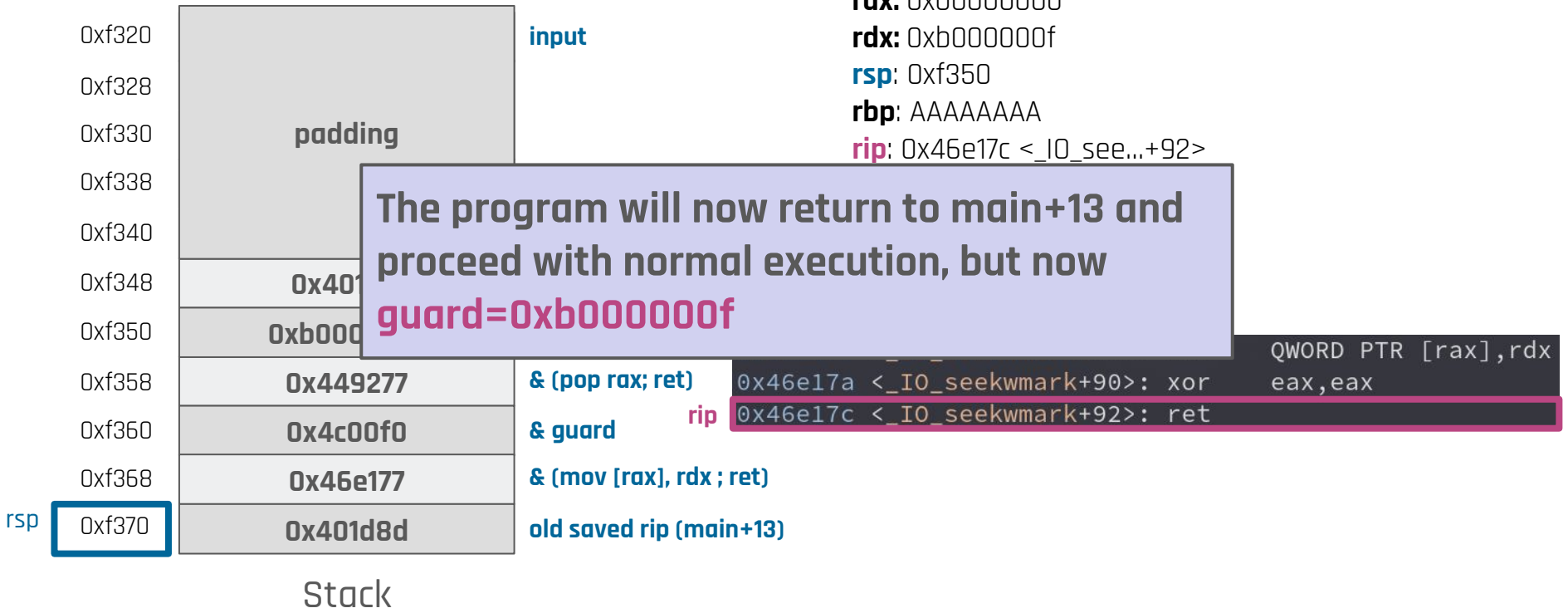
ROP - Walkthrough



rax: 0x00000000
rdx: 0xb000000f
rsp: 0xf350
rbp: AAAAAAAA
rip: 0x46e17c <_IO_see...+92>

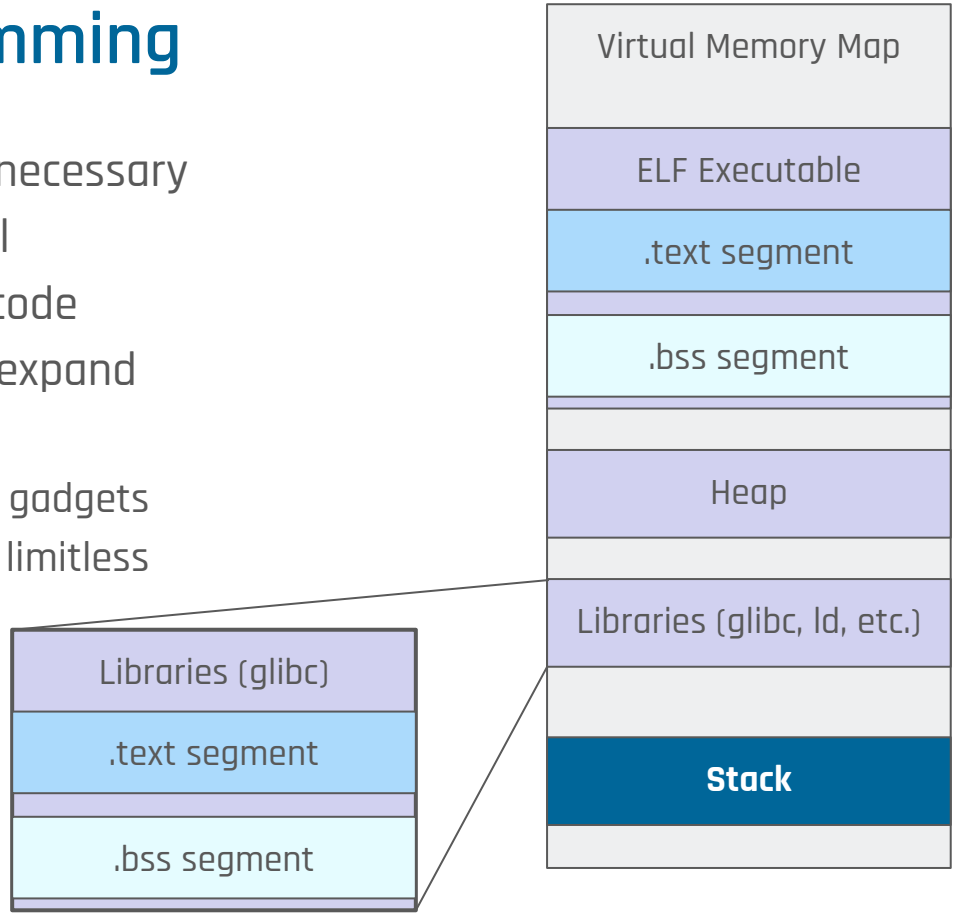
```
0x46e177 <_IO_seekwmark+87>: mov    QWORD PTR [rax],rdx
0x46e17a <_IO_seekwmark+90>: xor    eax,eax
rip 0x46e17c <_IO_seekwmark+92>: ret
```

ROP - Walkthrough



Return Oriented Programming

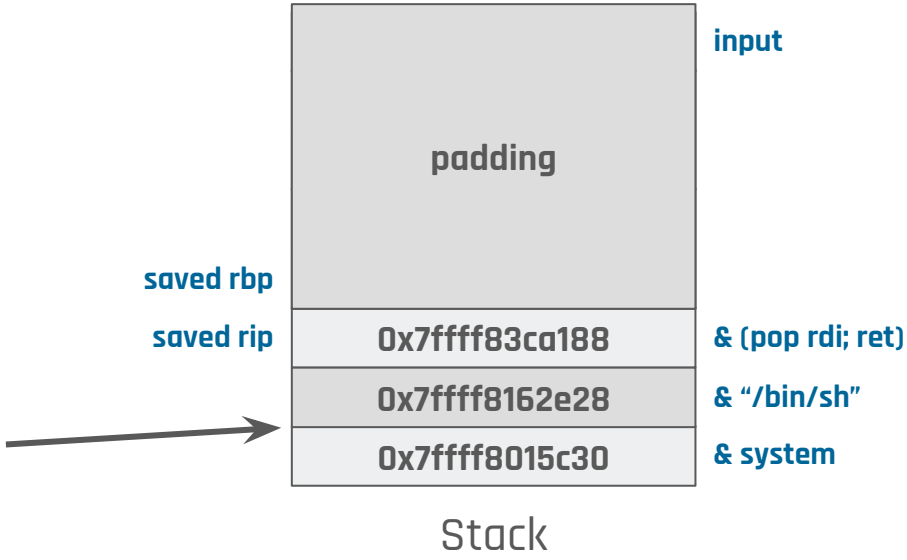
- Binaries may not contain all the necessary gadgets to achieve a certain goal
- However, attackers can jump to code present in the linked libraries to expand the set of gadgets available
  - Due to the size of the glibc, the gadgets available give attackers almost limitless options!



Return-to-libc

- Return-to-libc (ret2libc) attacks are subset of ROP attacks where an attacker jumps to libc functions to simplify the ROP chain.
- Common target functions are **system** and **I/O-related functions** like **open**, **read** and **write**

Pitfall: GCC assumes the stack is 16-byte aligned on function calls. If the stack is not aligned, the program would crash inside the **system** (specifically, the **movabs** instruction used inside **system** requires the stack to be aligned).
Fix: Add an additional instruction (like **ret**) to shift the stack by 8 bytes before jumping to **system**.



Return Oriented Programming - Constraints

- What if the attacker does not have enough space for a ROP chain?
- This is a very typical constraint in real-world binaries
- Possible solution: **Stack Pivoting**
  - i.e., modify the **rsp** to a location where the attacker has more control over the memory (stack, .bss buffer, heap chunk)
  - The ROP chain equivalent of a **NOP slide** (with **ret**) can be used to increase the accuracy of the pivot

```
add rsp, <imm>
ret
```

```
sub rsp, <imm>
ret
```

```
ret <imm>
```

```
leave ; (mov rsp, rbp)
      ; (pop rbp)
ret
```

```
xchg <reg>, rsp
ret
```

Address Space Layout Randomization (ASLR)

- **ASLR** is mechanism implemented by the kernel to mitigate exploits relying on hardcoded stack, heap and library addresses
- With ASLR, the base address of memory segments is randomized in every execution

```
λ juno ~ → ldd /bin/ls
        linux-vdso.so.1 (0x00007ffcb0c8a000)
        libcap.so.2 => /usr/lib/libcap.so.2 (0x00007f030dda4000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007f030dbbd000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007f030ddfa000)
λ juno ~ → ldd /bin/ls
        linux-vdso.so.1 (0x00007ffc7d364000)
        libcap.so.2 => /usr/lib/libcap.so.2 (0x00007fc5c333f000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007fc5c3158000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007fc5c3395000)
λ juno ~ → ldd /bin/ls
        linux-vdso.so.1 (0x00007ffeb2671000)
        libcap.so.2 => /usr/lib/libcap.so.2 (0x00007f72445a1000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007f72443ba000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007f72445f7000)
```

Address Space Layout Randomization (ASLR)

Run 1

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
             Start                End Perm     Size Offset
           0x400000           0x401000 r--p     1000      0
           0x401000           0x402000 r-xp     1000   1000
           0x402000           0x403000 r--p     1000   2000
           0x403000           0x404000 r--p     1000   2000
           0x404000           0x405000 rw-p     1000   3000
    0x7d86d9d78000     0x7d86d9d7b000 rw-p     3000      0
    0x7d86d9d7b000     0x7d86d9d9f000 r--p    24000      0
    0x7d86d9d9f000     0x7d86d9efa000 r-xp   15b000  24000
    0x7d86d9efa000     0x7d86d9f4f000 r--p    55000 17f000
    0x7d86d9f4f000     0x7d86d9f53000 r--p     4000 1d3000
    0x7d86d9f53000     0x7d86d9f55000 rw-p     2000 1d7000
    0x7d86d9f55000     0x7d86d9f5f000 rw-p     a000      0
    0x7d86d9f81000     0x7d86d9f85000 r--p     4000      0
    0x7d86d9f85000     0x7d86d9f87000 r-xp     2000      0
    0x7d86d9f87000     0x7d86d9f88000 r--p     1000      0
    0x7d86d9f88000     0x7d86d9faf000 r-xp    27000   1000
    0x7d86d9faf000     0x7d86d9fba000 r--p     b000  28000
    0x7d86d9fba000     0x7d86d9fbc000 r--p     2000  32000
    0x7d86d9fbc000     0x7d86d9fbe000 rw-p     2000  34000
    0x7fff2d9c9000     0x7fff2d9eb000 rw-p    22000      0
0xffffffffff600000 0xffffffffff601000 --xp     1000      0
```

Run 2

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
             Start                End Perm     Size Offset
           0x400000           0x401000 r--p     1000      0
           0x401000           0x402000 r-xp     1000   1000
           0x402000           0x403000 r--p     1000   2000
           0x403000           0x404000 r--p     1000   2000
           0x404000           0x405000 rw-p     1000   3000
    0x7ba3cb2b5000     0x7ba3cb2b8000 rw-p     3000      0
    0x7ba3cb2b8000     0x7ba3cb2dc000 r--p    24000      0
    0x7ba3cb2dc000     0x7ba3cb437000 r-xp   15b000  24000
    0x7ba3cb437000     0x7ba3cb48c000 r--p    55000 17f000
    0x7ba3cb48c000     0x7ba3cb490000 r--p     4000 1d3000
    0x7ba3cb490000     0x7ba3cb492000 rw-p     2000 1d7000
    0x7ba3cb492000     0x7ba3cb49c000 rw-p     a000      0
    0x7ba3cb4be000     0x7ba3cb4bf000 r--p     1000      0
    0x7ba3cb4bf000     0x7ba3cb4e6000 r-xp    27000   1000
    0x7ba3cb4e6000     0x7ba3cb4f1000 r--p     b000  28000
    0x7ba3cb4f1000     0x7ba3cb4f3000 r--p     2000  32000
    0x7ba3cb4f3000     0x7ba3cb4f5000 rw-p     2000  34000
    0x7ffd0bce9000     0x7ffd0bd0b000 rw-p    22000      0
    0x7ffd0bd76000     0x7ffd0bd7a000 r--p     4000      0
    0x7ffd0bd7a000     0x7ffd0bd7c000 r-xp     2000      0
0xffffffffff600000 0xffffffffff601000 --xp     1000      0
```

Run 3

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
             Start                End Perm     Size Offset
           0x400000           0x401000 r--p     1000      0
           0x401000           0x402000 r-xp     1000   1000
           0x402000           0x403000 r--p     1000   2000
           0x403000           0x404000 r--p     1000   2000
           0x404000           0x405000 rw-p     1000   3000
    0x79f104274000     0x79f104277000 rw-p     3000      0
    0x79f104277000     0x79f10429b000 r--p    24000      0
    0x79f10429b000     0x79f1043f6000 r-xp   15b000  24000
    0x79f1043f6000     0x79f10444b000 r--p    55000 17f000
    0x79f10444b000     0x79f10444f000 r--p     4000 1d3000
    0x79f10444f000     0x79f104451000 rw-p     2000 1d7000
    0x79f104451000     0x79f10445b000 rw-p     a000      0
    0x79f10447d000     0x79f10447e000 r--p     1000      0
    0x79f10447e000     0x79f1044a5000 r-xp    27000   1000
    0x79f1044a5000     0x79f1044b0000 r--p     b000  28000
    0x79f1044b0000     0x79f1044b2000 r--p     2000  32000
    0x79f1044b2000     0x79f1044b4000 rw-p     2000  34000
    0x7ffd439dd000     0x7ffd439ff000 rw-p    22000      0
    0x7ffd43add000     0x7ffd43ae1000 r--p     4000      0
    0x7ffd43ae1000     0x7ffd43ae3000 r-xp     2000      0
0xffffffffff600000 0xffffffffff601000 --xp     1000      0
```

Address Space Layout Randomization (ASLR)

Run 1

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
              Start                End Perm     Size Offset
          0x400000           0x401000 r--p     1000      0
          0x401000           0x402000 r-xp     1000   1000
          0x402000           0x403000 r--p     1000   2000
          0x403000           0x404000 r--p     1000   2000
          0x404000           0x405000 rw-p     1000   3000
    0x7d86d9d78000     0x7d86d9d7b000 rw-p     3000      0
    0x7d86d9d7b000     0x7d86d9d9f000 r--p    24000      0
    0x7d86d9d9f000     0x7d86d9efa000 r-xp   15b000  24000
    0x7d86d9efa000     0x7d86d9f4f000 r--p    55000 17f000
    0x7d86d9f4f000     0x7d86d9f53000 r--p     4000 1d3000
    0x7d86d9f53000     0x7d86d9f55000 rw-p     2000 1d7000
    0x7d86d9f55000     0x7d86d9f5f000 rw-p     a000      0
    0x7d86d9f81000     0x7d86d9f85000 r--p     4000      0
    0x7d86d9f85000     0x7d86d9f87000 r-xp     2000      0
    0x7d86d9f87000     0x7d86d9f88000 r--p     1000      0
    0x7d86d9f88000     0x7d86d9faf000 r-xp    27000   1000
    0x7d86d9faf000     0x7d86d9fba000 r--p     b000  28000
    0x7d86d9fba000     0x7d86d9fbc000 r--p     2000  32000
    0x7d86d9fbc000     0x7d86d9fbe000 rw-p     2000  34000
    0x7fff2d9c9000     0x7fff2d9eb000 rw-p    22000      0
0xffffffffff600000 0xffffffffff601000 --xp     1000      0
```

Notice they are always page-aligned (4Kb)

Run 2

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
              Start                End Perm     Size Offset
          0x400000           0x401000 r--p     1000      0
          0x401000           0x402000 r-xp     1000   1000
          0x402000           0x403000 r--p     1000   2000
          0x403000           0x404000 r--p     1000   2000
          0x404000           0x405000 rw-p     1000   3000
    0x7ba3cb2b5000     0x7ba3cb2b8000 rw-p     3000      0
    0x7ba3cb490000     0x7ba3cb492000 rw-p     2000 1d7000
    0x7ba3cb492000     0x7ba3cb49c000 rw-p     a000      0
    0x7ba3cb4be000     0x7ba3cb4bf000 r--p     1000      0
    0x7ba3cb4bf000     0x7ba3cb4e6000 r-xp    27000   1000
    0x7ba3cb4e6000     0x7ba3cb4f1000 r--p     b000  28000
    0x7ba3cb4f1000     0x7ba3cb4f3000 r--p     2000  32000
    0x7ba3cb4f3000     0x7ba3cb4f5000 rw-p     2000  34000
    0x7ffd0bce9000     0x7ffd0bd0b000 rw-p    22000      0
    0x7ffd0bd76000     0x7ffd0bd7a000 r--p     4000      0
```

Heap, library and stack addresses change

Run 3

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
              Start                End Perm     Size Offset
          0x400000           0x401000 r--p     1000      0
          0x401000           0x402000 r-xp     1000   1000
          0x402000           0x403000 r--p     1000   2000
          0x403000           0x404000 r--p     1000   2000
          0x404000           0x405000 rw-p     1000   3000
    0x79f104274000     0x79f104277000 rw-p     3000      0
    0x79f104277000     0x79f10429b000 r--p    24000      0
    0x79f10429b000     0x79f1043f6000 r-xp   15b000  24000
    0x79f1043f6000     0x79f10444b000 r--p    55000 17f000
    0x79f10444b000     0x79f10444f000 r--p     4000 1d3000
    0x79f10444f000     0x79f104451000 rw-p     2000 1d7000
    0x79f104451000     0x79f10445b000 rw-p     a000      0
    0x79f10447d000     0x79f10447e000 r--p     1000      0
    0x79f10447e000     0x79f1044a5000 r-xp    27000   1000
    0x79f1044a5000     0x79f1044b0000 r--p     b000  28000
    0x79f1044b0000     0x79f1044b2000 r--p     2000  32000
    0x79f1044b2000     0x79f1044b4000 rw-p     2000  34000
    0x7ffd439dd000     0x7ffd439ff000 rw-p    22000      0
    0x7ffd43add000     0x7ffd43ae1000 r--p     4000      0
    0x7ffd43ae1000     0x7ffd43ae3000 r-xp     2000      0
0xffffffffff600000 0xffffffffff601000 --xp     1000      0
```

Address Space Layout Randomization (ASLR)

- But, the main ELF base address remains constant throughout different runs!
- This means that it is still possible to find some ROP gadgets (usually very limited) within the binary code segments (.text, .plt, etc.) which will not be randomized

Takeaway: ASLR is not applied to the ELF base address

Run 1

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
                Start                End Perm  Size Offset
          0x400000           0x401000 r--p  1000      0
          0x401000           0x402000 r-xp  1000   1000
          0x402000           0x403000 r--p  1000   2000
          0x403000           0x404000 r--p  1000   2000
          0x404000           0x405000 rw-p  1000   3000
    0x7d86d9d78000     0x7d86d9d7b000 rw-p  3000      0
    0x7d86d9d7b000     0x7d86d9d9f000 r--p 24000      0
```

Run 2

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
                Start                End Perm  Size Offset
          0x400000           0x401000 r--p  1000      0
          0x401000           0x402000 r-xp  1000   1000
          0x402000           0x403000 r--p  1000   2000
          0x403000           0x404000 r--p  1000   2000
          0x404000           0x405000 rw-p  1000   3000
    0x7ba3cb2b5000     0x7ba3cb2b8000 rw-p  3000      0
    0x7ba3cb2b8000     0x7ba3cb2dc000 r--p 24000      0
```

Run 3

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
                Start                End Perm  Size Offset
          0x400000           0x401000 r--p  1000      0
          0x401000           0x402000 r-xp  1000   1000
          0x402000           0x403000 r--p  1000   2000
          0x403000           0x404000 r--p  1000   2000
          0x404000           0x405000 rw-p  1000   3000
    0x79f104274000     0x79f104277000 rw-p  3000      0
    0x79f104277000     0x79f10429b000 r--p 24000      0
```

Bypassing ASLR

There are a few main ways to bypass randomization:

- Information leaks:
  - Leaking one address **de-randomizes the entire library,** since only the base address is randomized (provided the attacker knows the libc version being used) but within the library, functions offsets are constant
  - How to obtain leaks depends heavily on the target binary, but buffer over-reads are the most common ways to obtain leaks
- Partial overwrite
  - Clobbering only the least significant bytes of a function pointer, or the saved rip on the stack may be enough for an attacker to successfully hijack the control flow
  - Bruteforcing could also be used if the amount of bits to bruteforce is low enough

Bypassing ASLR - ret2plt

- The **Procedure Linkage Table** (PLT) and the **Global Offset Table** (GOT) are used to resolve library addresses at runtime
  - This is the mechanism that enables dynamically linking libraries
- The GOT contains the addresses of libc functions used in the binary
  - It is populated at runtime by a name resolution mechanism
- The PLT is contained in the **.plt** section of an ELF file
  - So, the PLT is not affected by ASLR
  - The PLT is a jump table that dereferences the GOT to jump to the correct address in the given library

PLT and GOT - Example

The PLT consists of a **jump table** that uses the contents of the GOT as the jump targets

Calls to `puts` are compiled to calls to `puts@plt`

```
lea    rdi,[rip+0xe00]
call   0x401030 <puts@plt>
```

PLT

```
0x401030 (puts@plt) <- jmp qword ptr [rip + 0x2fe2]
```

= 0x404018

Procedure Linkage Table

```
Section .plt 0x401020-0x401080:
0x401030: puts@plt
0x401040: strlen@plt
0x401050: __stack_chk_fail@plt
0x401060: strcmp@plt
0x401070: __isoc99_scanf@plt
```

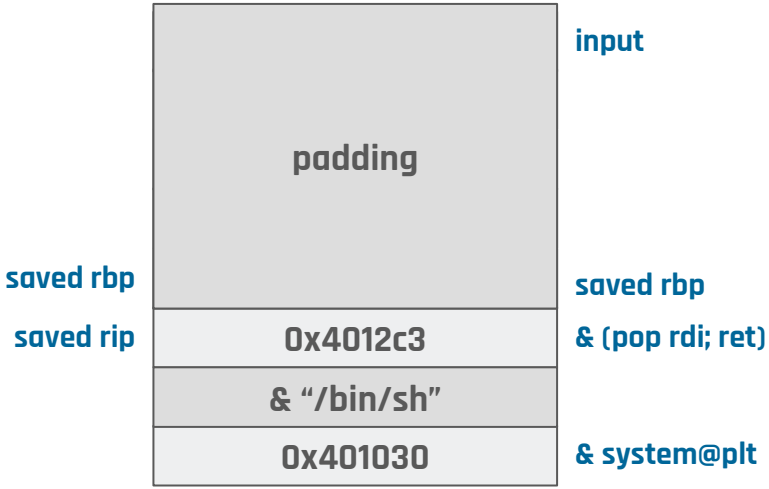
Global Offset Table

```
GOT protection: Partial RELRO | GOT functions: 5
[0x404018] puts@GLIBC_2.2.5 -> 0x7ffff7e55420 (pu
[0x404020] strlen@GLIBC_2.2.5 -> 0x7ffff7f596d0
[0x404028] __stack_chk_fail@GLIBC_2.4 -> 0x40105
[0x404030] strcmp@GLIBC_2.2.5 -> 0x7ffff7f54bd0
[0x404038] __isoc99_scanf@GLIBC_2.7 -> 0x7ffff7e
```

GOT contains the addresses of libc functions used in the binary

ret2plt

- **Idea:** Hijack the control flow of the program to call libc functions through the PLT
  - Example: **system@plt("/bin/sh")**
- This technique removes the need for a libc leak, since both the GOT and the PLT are part of the binary and not randomized every execution
  - PLT entries are **only** available for functions actually invoked in the original binary



Position-Independent Executable

- Binaries can be compiled with the **P**osition-**I**ndependent **E**xecutable (**PIE**) flag
- PIE is essentially ASLR for ELF binaries
- Now, all addresses are randomized and attackers cannot assume where anything is in memory
- Exploits require more advanced techniques like leaks or partial overwrites to work

image: Flaticon.com

Resources

- **Common pitfalls**: https://ropemporium.com/guide.html#Common%20pitfalls
- **Pwntools documentation**: https://github.com/Gallopsled/pwntools
- **GDB stack offsets**: https://stackoverflow.com/questions/17775186/buffer-overflow-works-in-gdb-but-not-without-it/17775966#17775966
- **Calling conventions:** https://en.wikipedia.org/wiki/X86_calling_conventions
- **Ropchain:** https://en.wikipedia.org/wiki/Return-oriented_programming
- **Phrack Article**: http://phrack.org/issues/49/14.html
- **LiveOverflow - Binary Exploitation/Memory Corruption Playlist**: https://www.youtube.com/watch?v=iyAyN3GFM7A&list=PLhixgUqwRTjxglIswKp9mpkfPNfHkzyeN