Software Security

Ethical Hacking

Alessandro Brighente
Eleonora Losiouk

Master Degree on Cybersecurity

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

Outline

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- Non-executable Stack countermeasure
- How to defeat the countermeasure
- Tasks involved in the attack
- Function Prologue and Epilogue
- Launching attack

Arbitrary Code Execution (ACE)

- The attacker's ability to execute arbitrary commands or code on a target machine or in a target process
- Can be induced by:

  Code injection

  Code reuse

Code Reuse Attacks

SPRITZ
Security & Privacy
Research Group

Università
degli Studi
di Padova

- Code-reuse attacks are software exploits in which an attacker directs control flow through existing code with a malicious result
- Examples of such attacks are:
  - Return-to-libc
  - Return Oriented Programming (ROP)
  - Jump Oriented Programming (JOP)

Return to libc...

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- A return-to-libc attack is a technique that, by using a buffer overflow, replaces a return address with the one of another function in the process memory
- The name is related to the fact that one of functions in the C Standard Library (libc) is used
- This attack was spotted in 1997 by Alexander Peslyak (aka Solar Designer)

Return to libc Challenges

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- Challenge 1: Find address of system()

  To overwrite return address with system()'s address
- Challenge 2 : Find address of the "/bin/sh" string

  To run command "/bin/sh" from system()
- Challenge 3 : Construct arguments for system()

  To find location in the stack to place "/bin/sh" address (argument for system())

Challenge 1

- To find address of *system()* we can use *gdb*:

```
(gdb) file retlibc
Reading symbols from retlibc...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x104b4
(gdb) run
Breakpoint 1, 0x000104b4 in main ()
(gdb) p system()
Too few arguments in function call.
(gdb) p system
$1 = {int (const char *)} 0xb6e909c8 <__libc_system>
```

Challenge 2

- Export an environment variable called “MYSHELL” with value “/bin/sh”
- MYSHELL is passed to the vulnerable program as an environment variable, which is stored on the stack
- We can find its address

Challenge 2


```
#include <stdio.h>

int main()
{
   char *shell = (char *)getenv("MYSHELL");

   if(shell){
      printf("  Value:   %s\n",   shell);
      printf("  Address: %x\n", (unsigned int)shell);
   }

   return 1;
}
```

Code to display address of environment variable

Challenge 2


```
$ gcc envaddr.c -o env55
$ export MYSHELL="/bin/sh"
$ ./env55
  Value:   /bin/sh
  Address: bffffe8c
```

- Export “MYSHELL” environment variable and execute the code

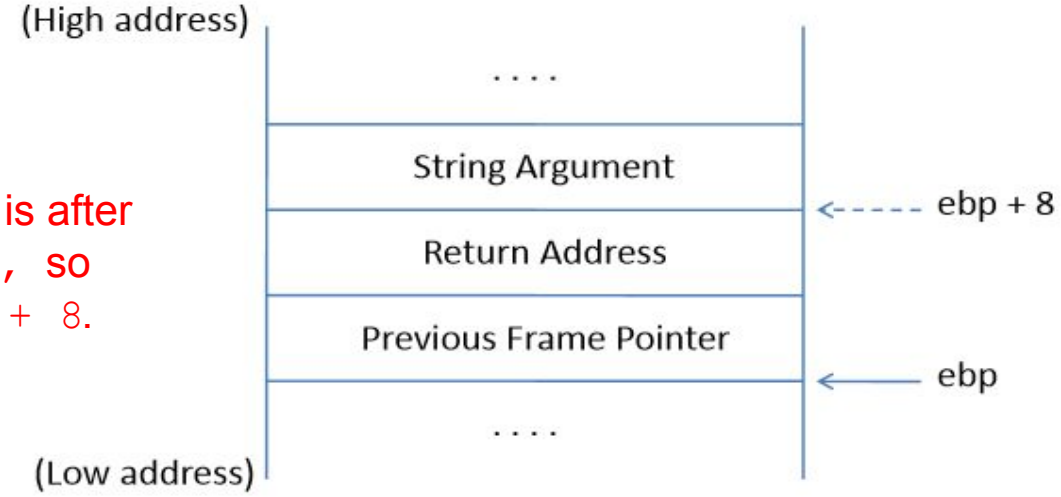
```
$ mv env55 env7777
$ ./env7777
  Value:   /bin/sh
  Address: bffffe88
```

- Address of “MYSHELL” environment variable is sensitive to the length of the program name
- If the program name is changed from env55 to env77, we get a different address

Challenge 3

- Arguments are accessed with respect to ebp
- Argument for system() needs to be on the stack

Need to know where exactly ebp is after we have "returned" to `system()`, so we can put the argument at `ebp + 8`.



Frame for the system() function

Function Prologue and Epilogue


```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
   1    movl %esp, %ebp
        subl $16, %esp
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
   2    leave
        ret
```

1 Function prologue

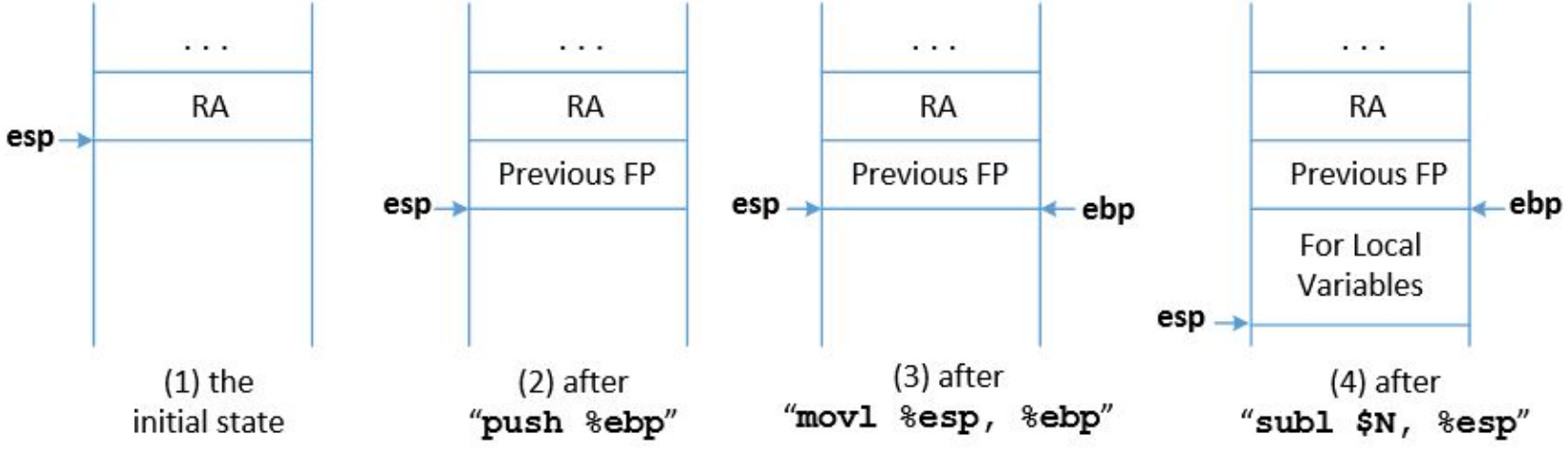
2 Function epilogue

8(%ebp) ⇒ %ebp + 8

Function Prologue

```
pushl   %ebp
movl    %esp, %ebp
subl    $N, %esp
```

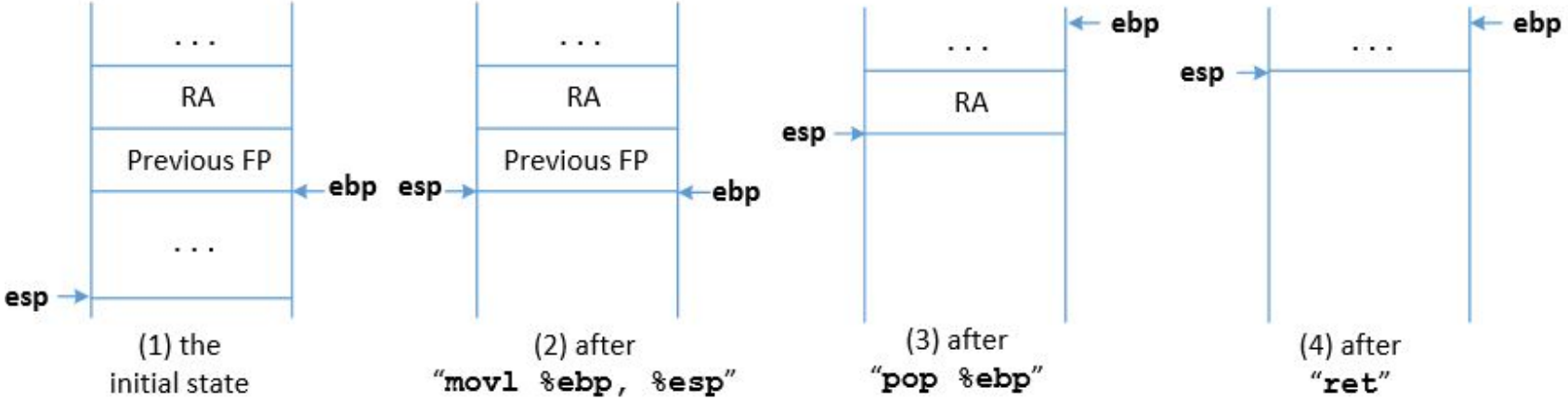
esp : Stack pointer
ebp : Frame Pointer



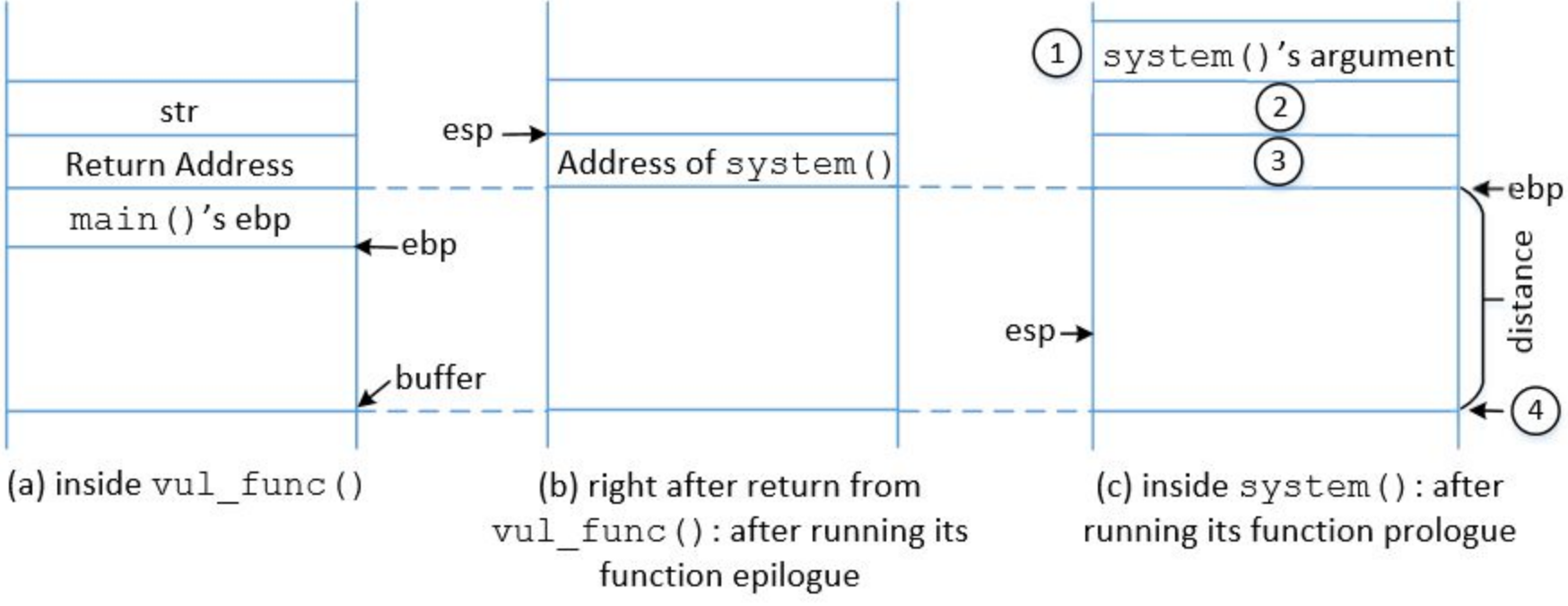
Function Epilogue


```
movl   %ebp, %esp
popl   %ebp
ret
```

esp : Stack pointer
ebp : Frame Pointer

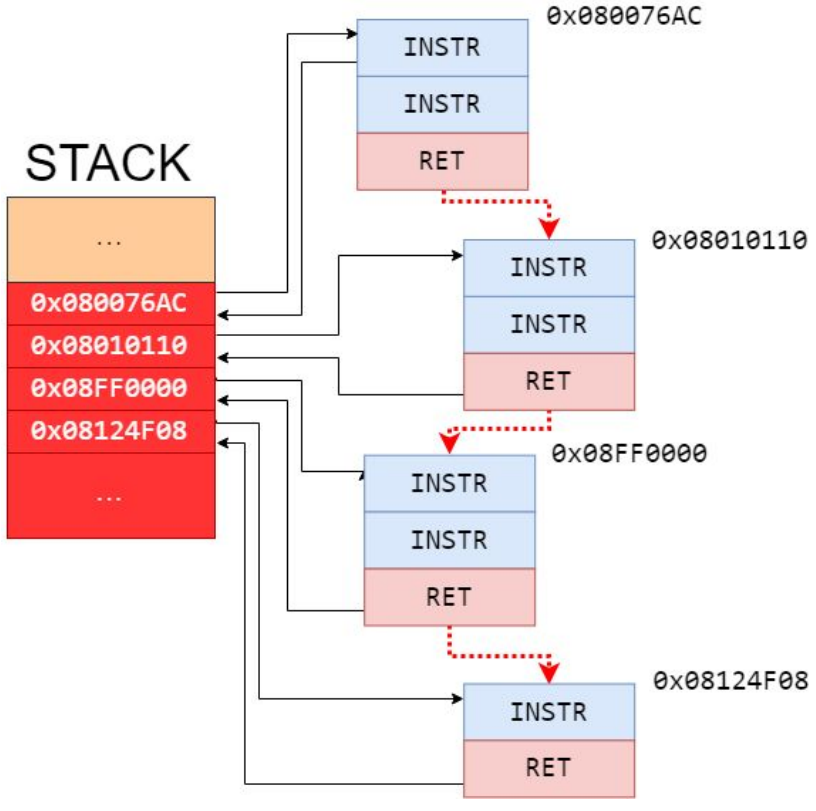


system() Arguments



Returned-Oriented Programming (ROP)

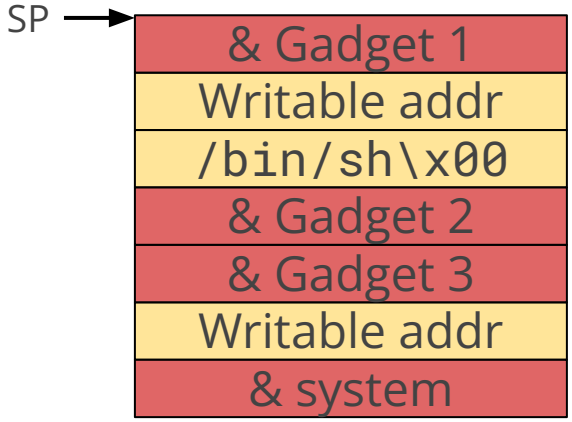
- Based on gadgets ending with a routine return instruction (RET)
- RET pops the return location from the stack and jumps there
- If the stack data are overflowed, a series of “ fake” return addresses can be stacked
- Every time a RET is executed, control is passed to the next gadget



- Question: how can we find such gadgets?
  - By hand, e.g., by inspecting objdump (this is the old school approach)
  - By using one of the available tools:
    - Ropper (https://github.com/sashs/Ropper)
    - ROPGadget (https://github.com/JonathanSalwan/ROPgadget)
    - Pwntools

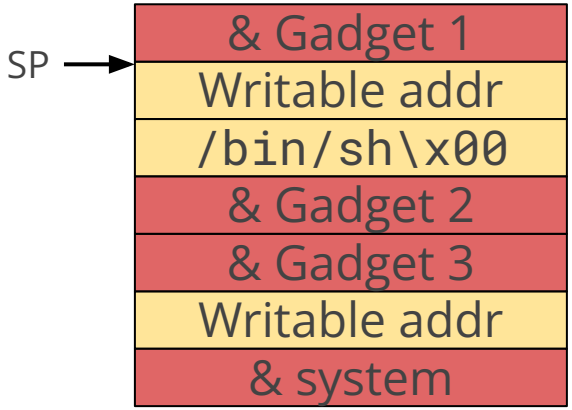
ROP Chain Example

Goal: call `system("/bin/sh")`



ROP Chain Example

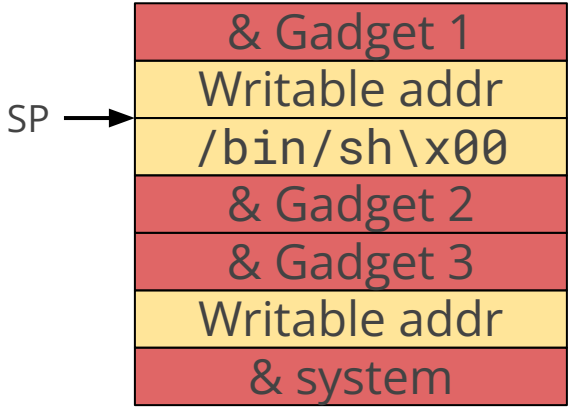
Goal: call `system("/bin/sh")`



① `pop rax; pop rbx; ret`

ROP Chain Example

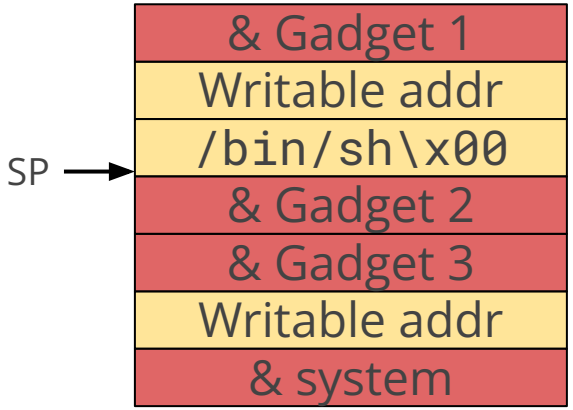
Goal: call `system("/bin/sh")`



① `pop rax; pop rbx; ret`

ROP Chain Example

Goal: call `system("/bin/sh")`

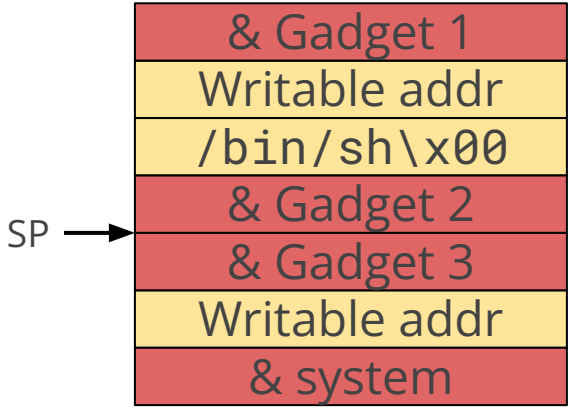


① `pop rax; pop rbx; ret`

```
rax = WADDR
rbx = '/bin/sh\x00'
```

ROP Chain Example

Goal: call `system("/bin/sh")`

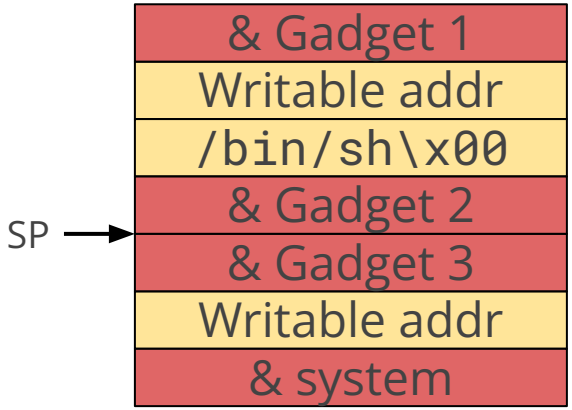


② `mov [rax], rbx; ret`

```
rax = WADDR
rbx = '/bin/sh\x00'
```

ROP Chain Example

Goal: call `system("/bin/sh")`

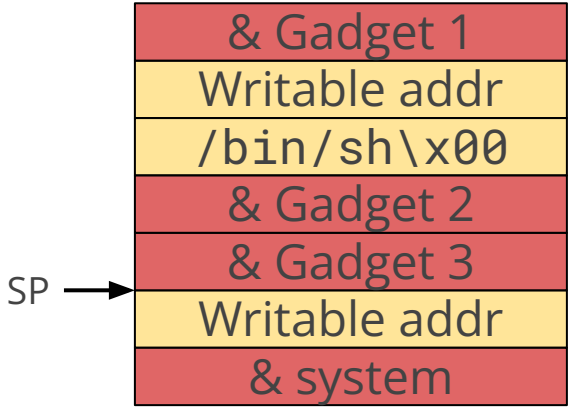


② `mov [rax], rbx; ret`

```
rax = WADDR
rbx = '/bin/sh\x00'
*WADDR = '/bin/sh\x00'
```

ROP Chain Example

Goal: call `system("/bin/sh")`

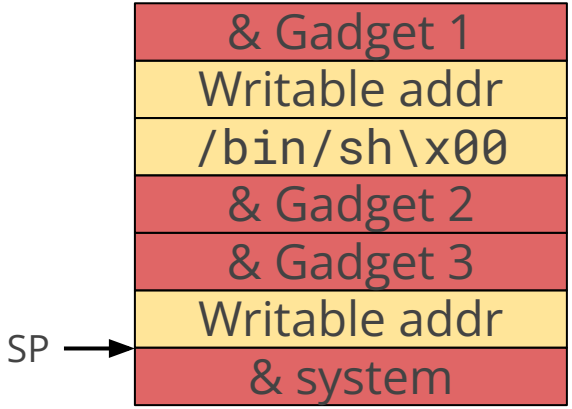


③ `pop rdi; ret`

```
rax = WADDR
rbx = '/bin/sh\x00'
*WADDR = '/bin/sh\x00'
```

ROP Chain Example

Goal: call `system("/bin/sh")`



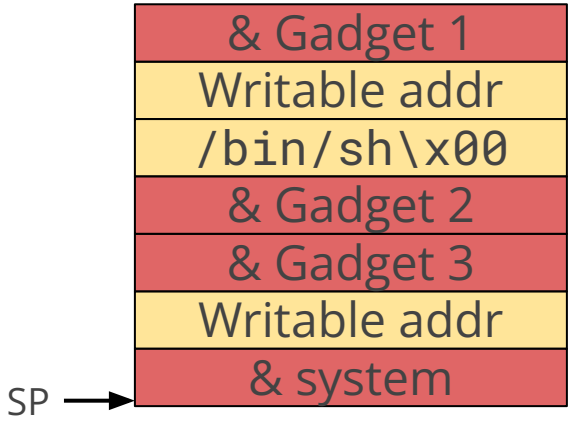
③ `pop rdi; ret`

```
rax = WADDR
rbx = '/bin/sh\x00'
*WADDR = '/bin/sh\x00'
```

`rdi = WADDR`

ROP Chain Example

Goal: call `system("/bin/sh")`



④ `system("/bin/sh")`

```
rax = WADDR
rbx = '/bin/sh\x00'
*WADDR = '/bin/sh\x00'
rdi = WADDR
```