Lec 07: Attacks and Defenses (1)

CSED415: Computer Security

Spring 2024

Seulbae Kim

POSTECH

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

POSTECH

- Lab 02 is out!
  - Due Mar 24
  - Presents more challenging tasks than Lab 01
  - Recommendations
    - Start early
    - Start early
    - Start early
    - Start early

Administrivia

- Project teams are ready!
  - Agustina & Megan
  - whysw
  - 구얏
  - h@ckerz
  - q1w2e3r4
  - Poulpy

Recap

POSTECH

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

How can we mitigate such 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

POSTECH

- Return-to-stack attack
  - Loads a shellcode on the **stack**
  - Jumps to the shellcode and execute it

But.. should the contents of the stack,
typically comprising data, be executable?

NX: No eXecute

- Hardware-based mitigation for arbitrary code execution
  - CPU’s MMU (memory management unit) is in charge
- Separate between memory regions (pages) that contain code to those containing data
  - Only grant eXecute permission to the code pages
  - Remove eXecute permission from the data pages
- Set NX flag for the stack pages (data region)
  - Applied by default

NX: No eXecute

POSTECH

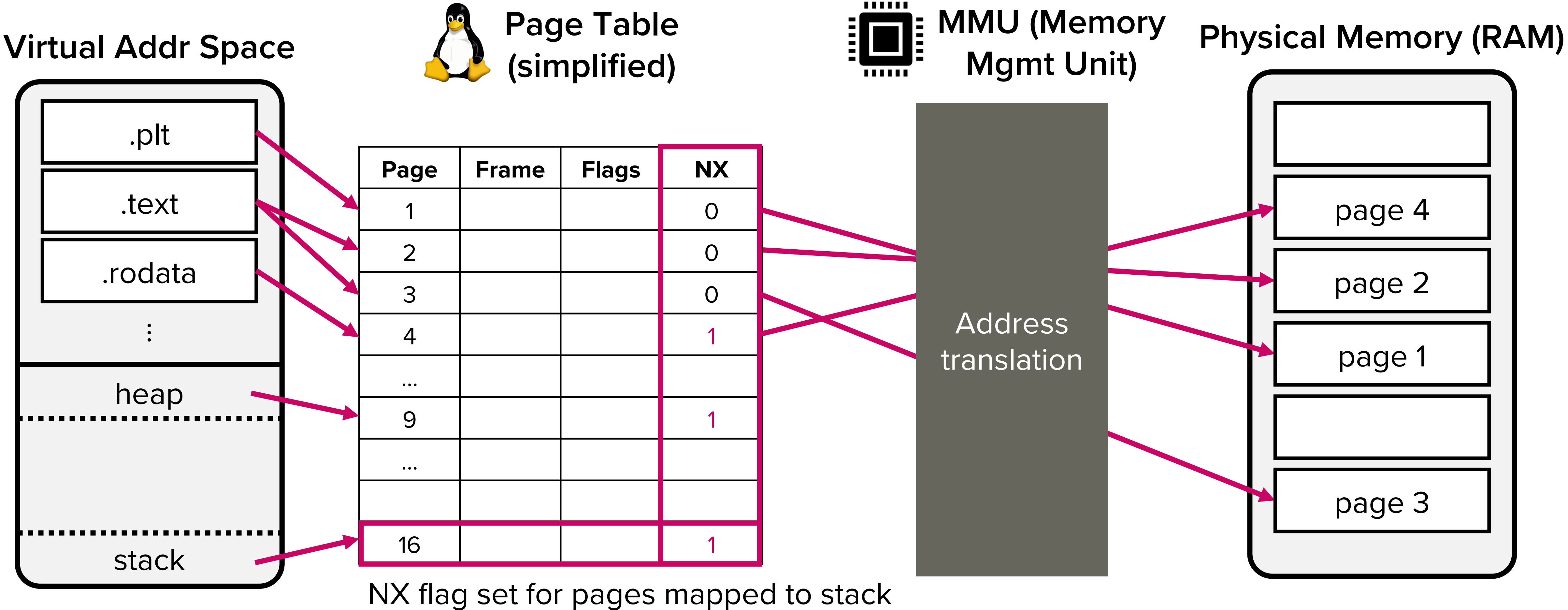
- Hardware-based mitigation for arbitrary code execution
  - CPU's MMU (memory management unit) is in charge
- Separate between memory regions (pages) that contain

Generalized policy utilizing NX: W^X (Write xor eXecute)
→ Every page in a process may be either writable or executable, but not both.

- Set NX flag for the stack pages (data region)
  - Applied by default

NX – low level implementation

POSTECH

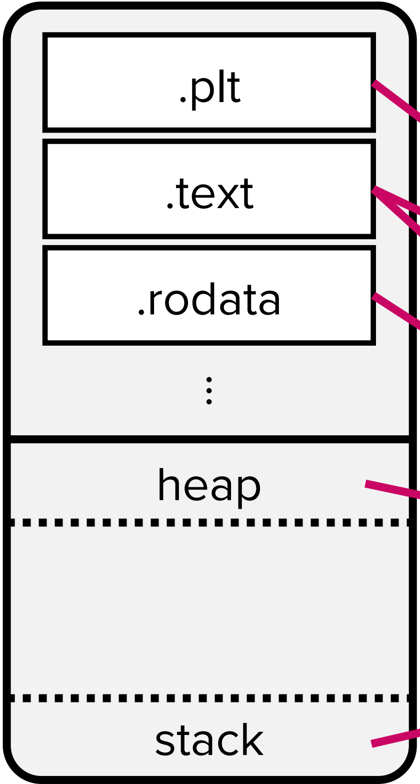


Page	Frame	Flags	NX
1			0
2			0
3			0
4			1
…			
9			1
…			
16			1

NX flag set for pages mapped to stack

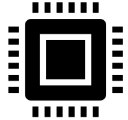
NX – low level implementation

Virtual Addr Space

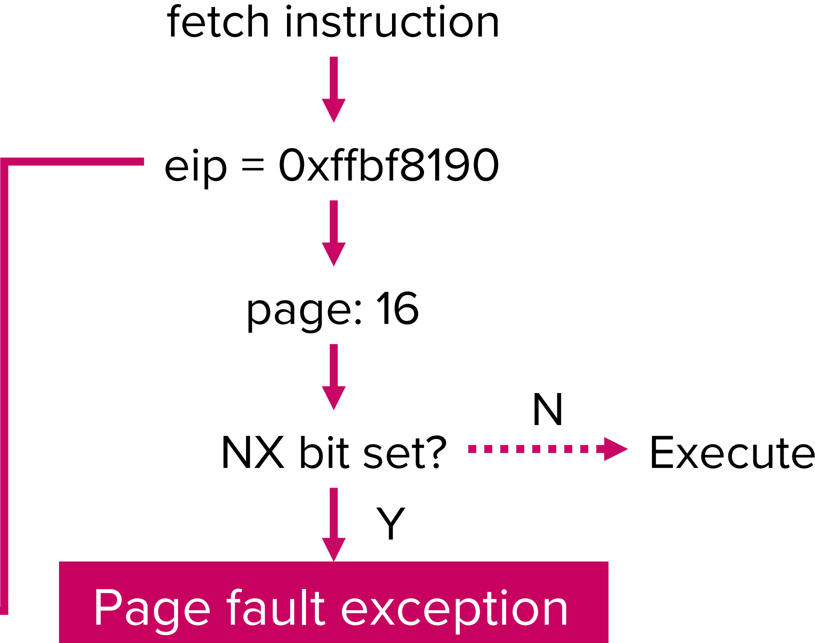


Page Table (simplified)

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



MMU (Memory Management Unit)



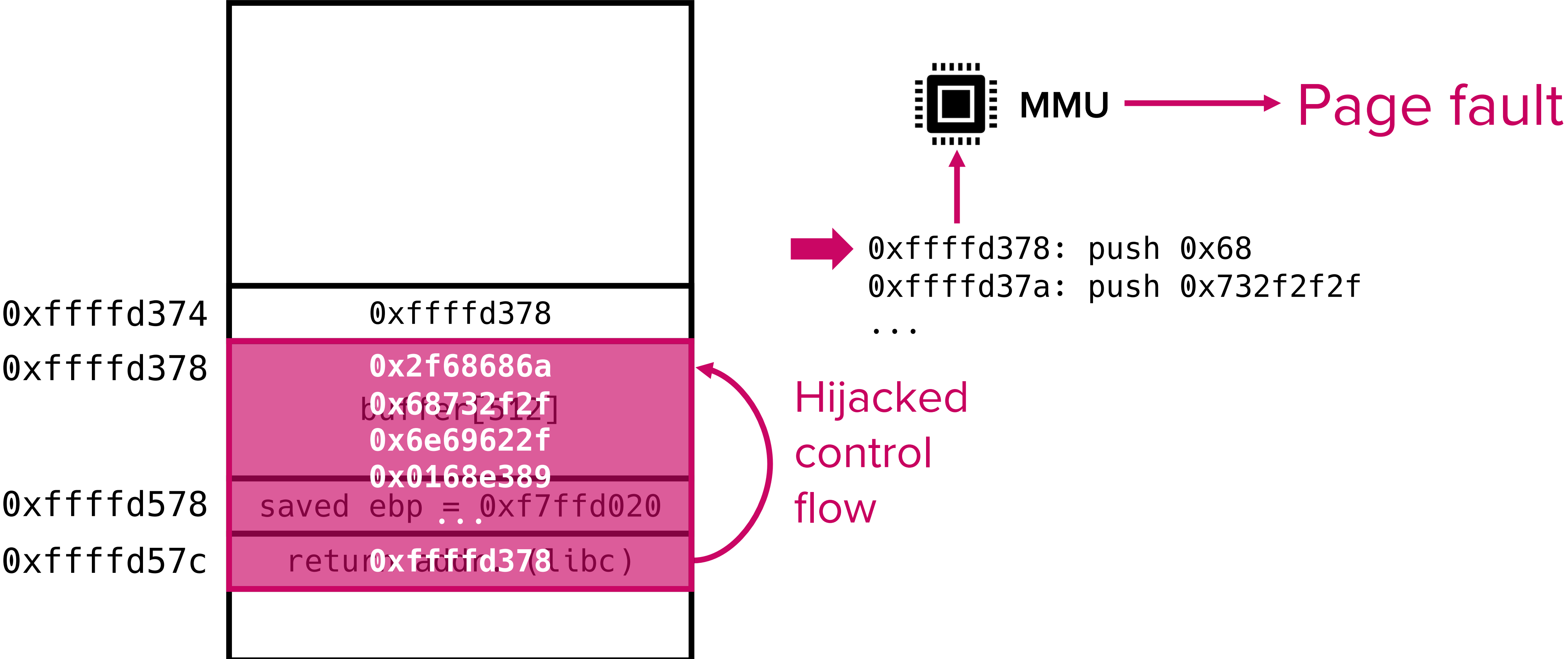
What if hardware (MMU) doesn't support NX?

POSTECH

- OS-level implementations exist
  - Linux PaX (PageeXec)
    - Emulates the NX bit on CPUs that do not support it
      - x86 (i386) CPUs did not initially support NX
    - **The kernel** checks if code can be executed from a page
      - Technical details: https://pax.grsecurity.net/docs/pageexec.txt

Defeating return-to-stack attacks

- Stack



execstack

POSTECH

- GCC compile option (passed directly to linker)
  - `$ gcc morris.c -z execstack -o morris`
  - Makes binary's stack executable by clearning 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`

NX is used in Lab target binaries

POSTECH

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

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
     Start        End Perm     Size Offset File
 0x8048000  0x8049000 r--p     1000      0 /home/lab01/target
 0x8049000  0x804a000 r-xp     1000   1000 /home/lab01/target
 0x804a000  0x804b000 r--p     1000   2000 /home/lab01/target
 0x804b000  0x804c000 r--p     1000   2000 /home/lab01/target
 0x804c000  0x804d000 rw-p     1000   3000 /home/lab01/target
0xf7d59000 0xf7d79000 r--p    20000      0 /lib/i386-linux-gnu/libc.so.6
0xf7d79000 0xf7efb000 r-xp   182000  20000 /lib/i386-linux-gnu/libc.so.6
0xf7efb000 0xf7f80000 r--p    85000 1a2000 /lib/i386-linux-gnu/libc.so.6
0xf7f80000 0xf7f81000 ---p     1000 227000 /lib/i386-linux-gnu/libc.so.6
0xf7f81000 0xf7f83000 r--p     2000 227000 /lib/i386-linux-gnu/libc.so.6
0xf7f83000 0xf7f84000 rw-p     1000 229000 /lib/i386-linux-gnu/libc.so.6
0xf7f84000 0xf7f8e000 rw-p     a000      0 [anon_f7f84]
0xf7f97000 0xf7f99000 rw-p     2000      0 [anon_f7f97]
0xf7f99000 0xf7f9d000 r--p     4000      0 [vvar]
0xf7f9d000 0xf7f9f000 r-xp     2000      0 [vdso]
0xf7f9f000 0xf7fa0000 r--p     1000      0 /lib/i386-linux-gnu/ld-linux.so.2
0xf7fa0000 0xf7fc5000 r-xp    25000   1000 /lib/i386-linux-gnu/ld-linux.so.2
0xf7fc5000 0xf7fd4000 r--p     f000  26000 /lib/i386-linux-gnu/ld-linux.so.2
0xf7fd4000 0xf7fd6000 r--p     2000  34000 /lib/i386-linux-gnu/ld-linux.so.2
0xf7fd6000 0xf7fd7000 rw-p     1000  36000 /lib/i386-linux-gnu/ld-linux.so.2
0xff7ee000 0xff80f000 rw-p    21000      0 [stack]
```

Rethinking the W^X policy

POSTECH

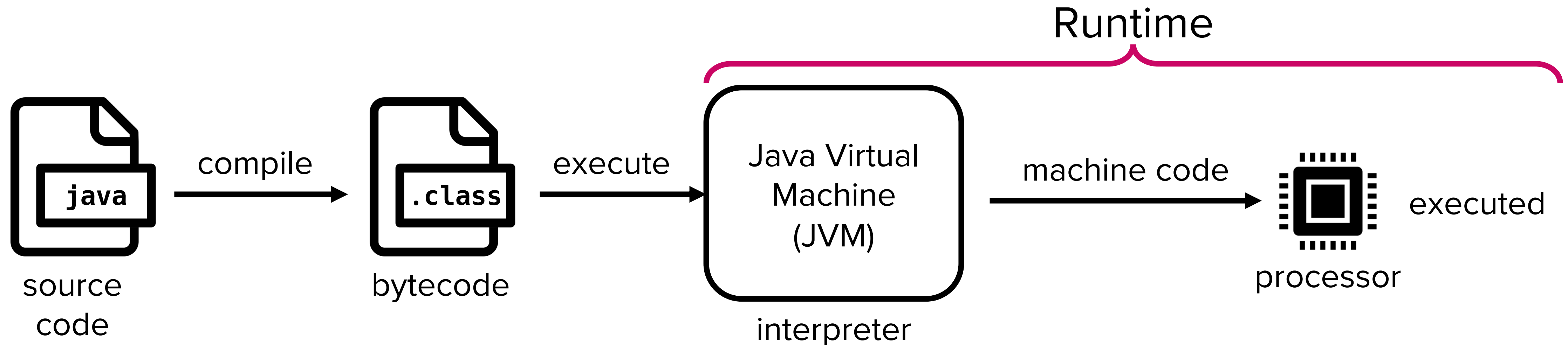
- NX is very effective against code injection attacks
  - Then, why is NX even an option?
  - Do we ever need to store code on stack and execute them?

Sometimes!

Just-in-time (JIT) compilation

POSTECH

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

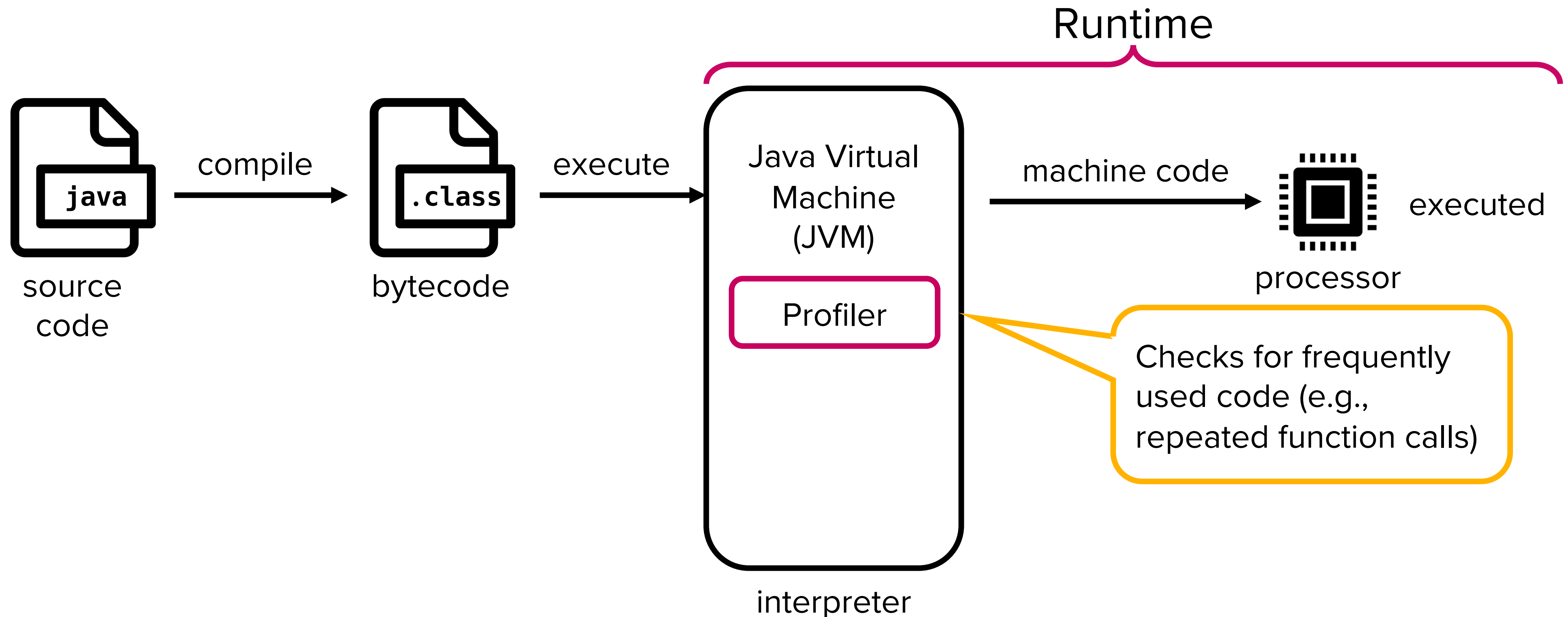


Machine code is generated at runtime → SLOW

Just-in-time (JIT) compilation

POSTECH

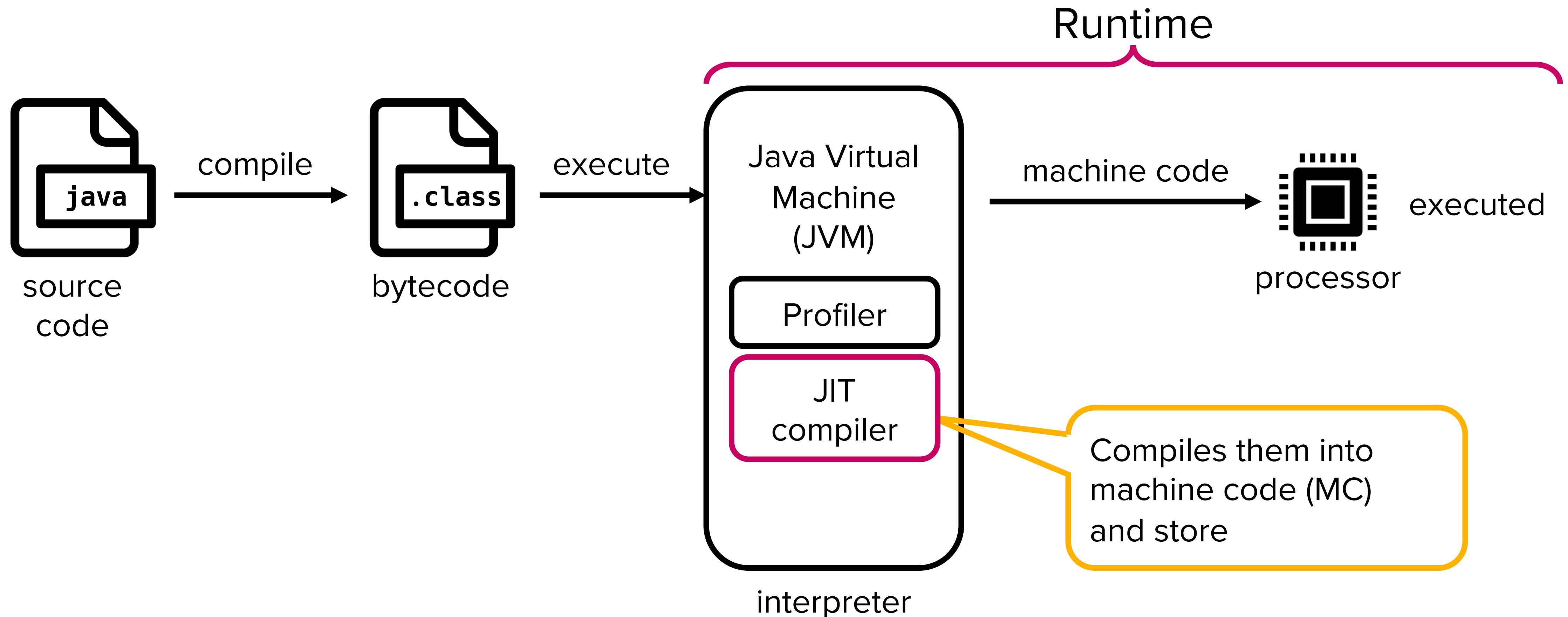
- Optimizing for better performance



Just-in-time (JIT) compilation

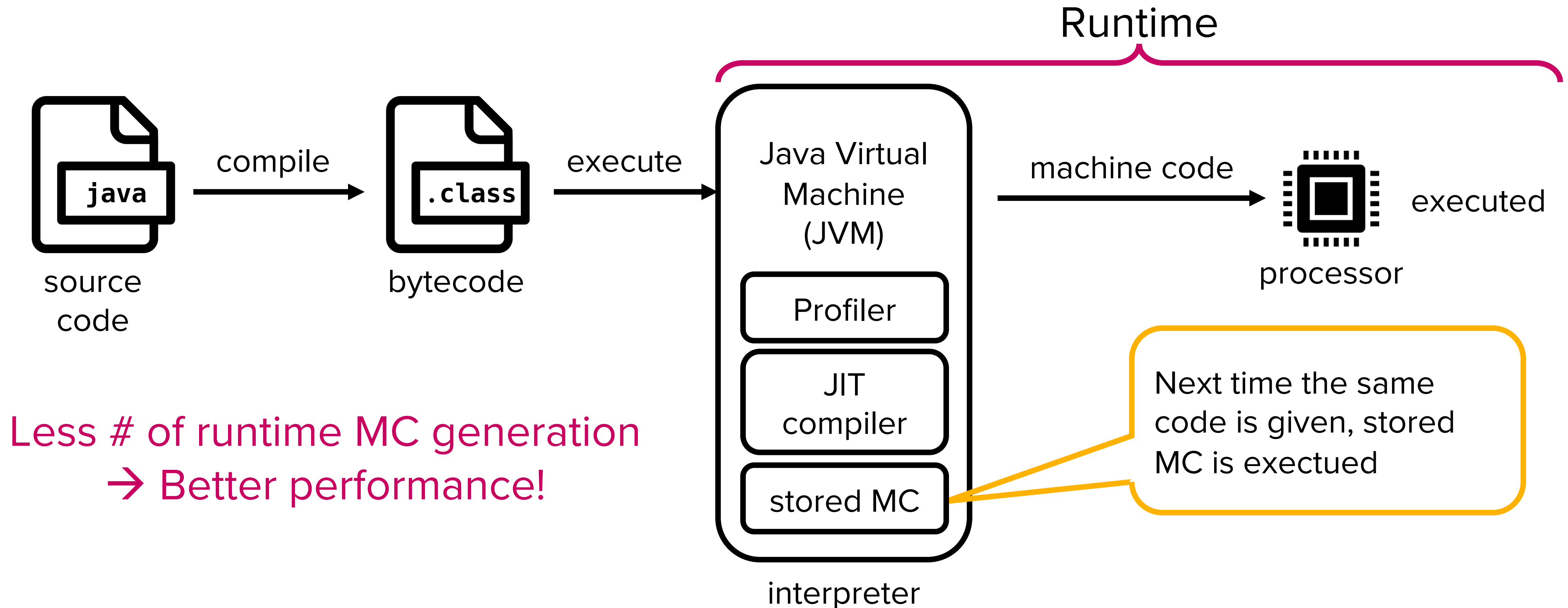
POSTECH

- Optimizing for better performance



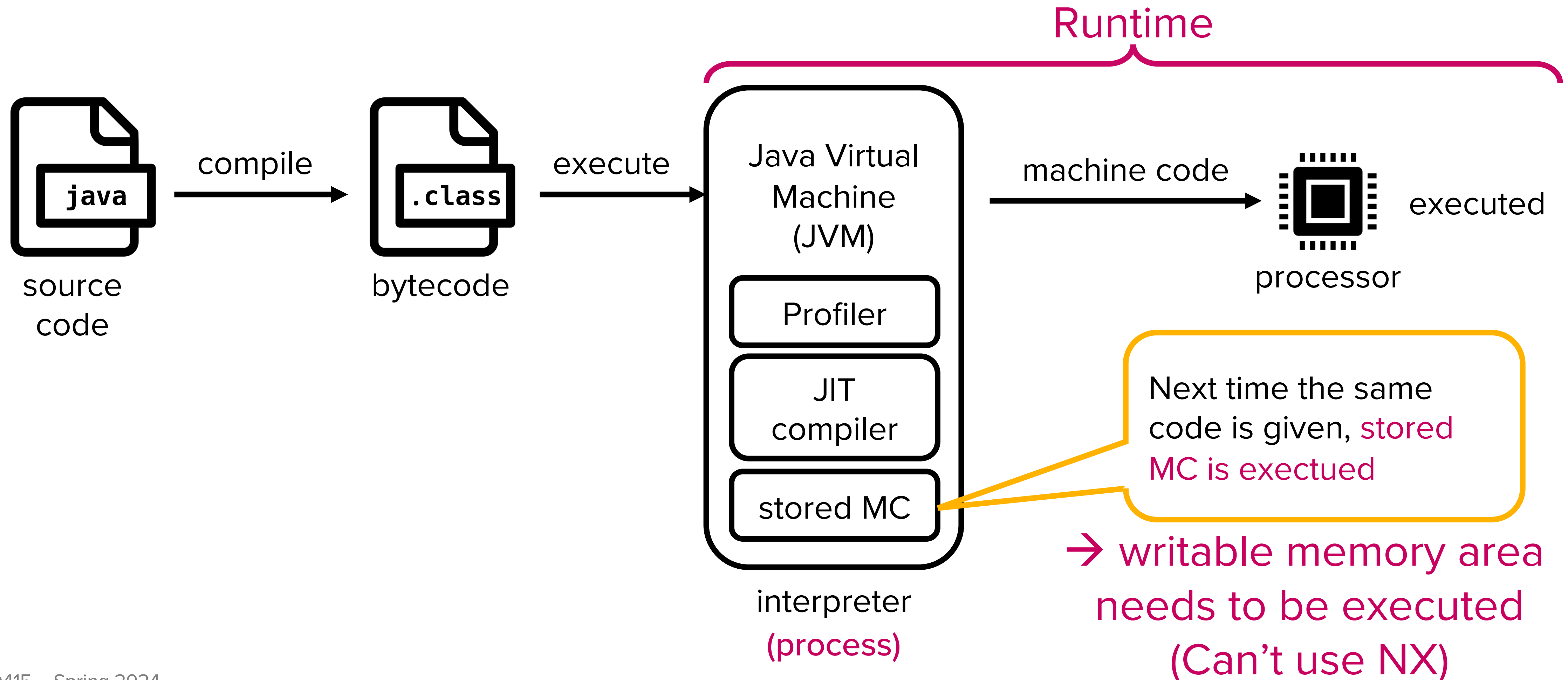
Just-in-time (JIT) compilation

- Optimizing for better performance



Just-in-time (JIT) compilation

- W^X policy cannot be enforced for JVM process



Attack #1-1: Return-to-libc

Bypassing NX

POSTECH

- Return-to-stack exploit is mitigated
  - Injected shellcode is not executable
- New attack idea: why don’t we return to an address of existing code?
  - Existing code segments are always executable

→ Called “Code reuse attack”

Libc (GNU C Library)

POSTECH

- A standard library that most C programs use
  - printf(), atoi(), getenv(), ...

- There are many useful functions in libc to return to
  - **Execution**: exec family (execl, execve, ...), system(), popen(), ...
  - **File I/O**: open(), read(), write(), fopen(), fread(), ...
  - **MMIO**: mmap()
  - **Memory protection**: mprotect()
  - **String operation**: strcpy(), memcpy(), memset(), ...

Return-to-libc attack

POSTECH

- Example: Typical invocation of system(“/bin/sh”);

```c
#include <stdlib.h>

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

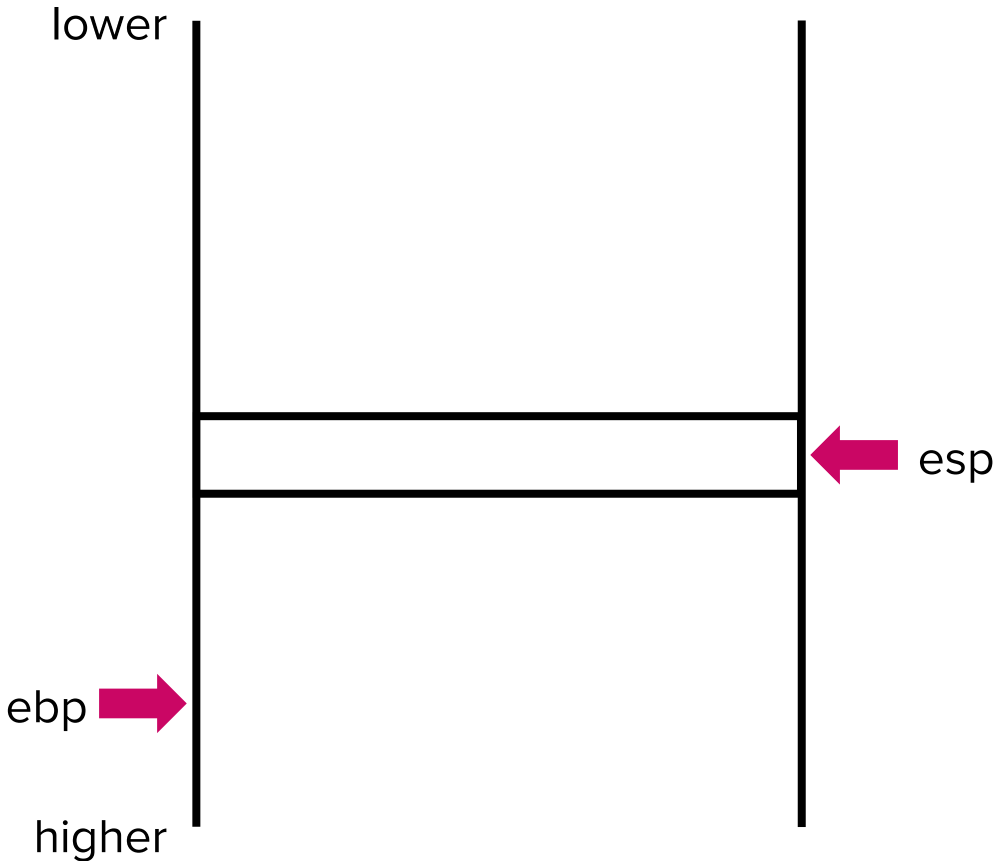
```
05 76 2e 00 00          add     eax,0x2e76
83 ec 0c                sub     esp,0xc
8d 90 08 e0 ff ff       lea     edx,[eax-0x1ff8]
52                      push    edx
89 c3                   mov     ebx,eax
e8 b0 fe ff ff          call    8049050 <system@plt>
```

(from Lec 06)

Background: Stack machine workflow

POSTECH

- Example: Typical invocation of system(“/bin/sh”);



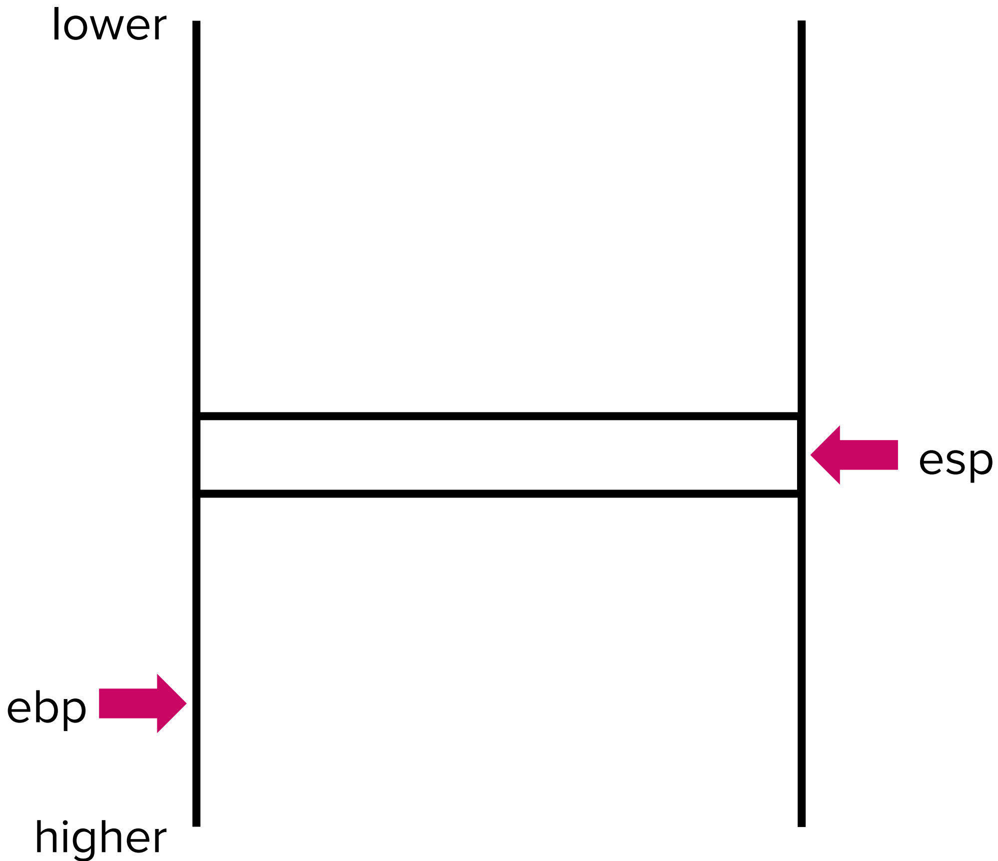
eip →

```
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:
Load the address of “/bin/sh” in edx

Background: Stack machine workflow

- Example: Typical invocation of system(“/bin/sh”);



eip →

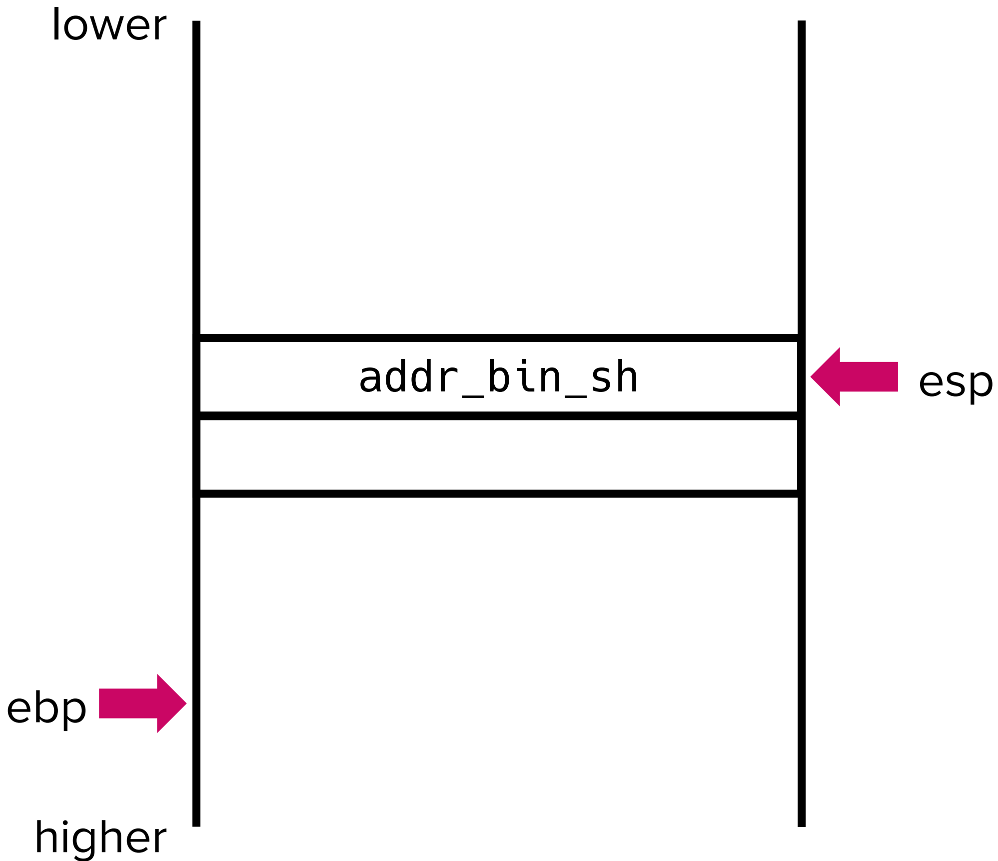
```
05 76 2e 00 00          add    eax,0x2e76
83 ec 0c                sub    esp,0xc
8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
52                      push   edx
89 c3                   mov    ebx,eax
e8 b0 fe ff ff          call   8049050 <system@plt>
```

Next instruction:
Push the address of “/bin/sh”

Background: Stack machine workflow

POSTECH

- Example: Typical invocation of system("/bin/sh");



eip

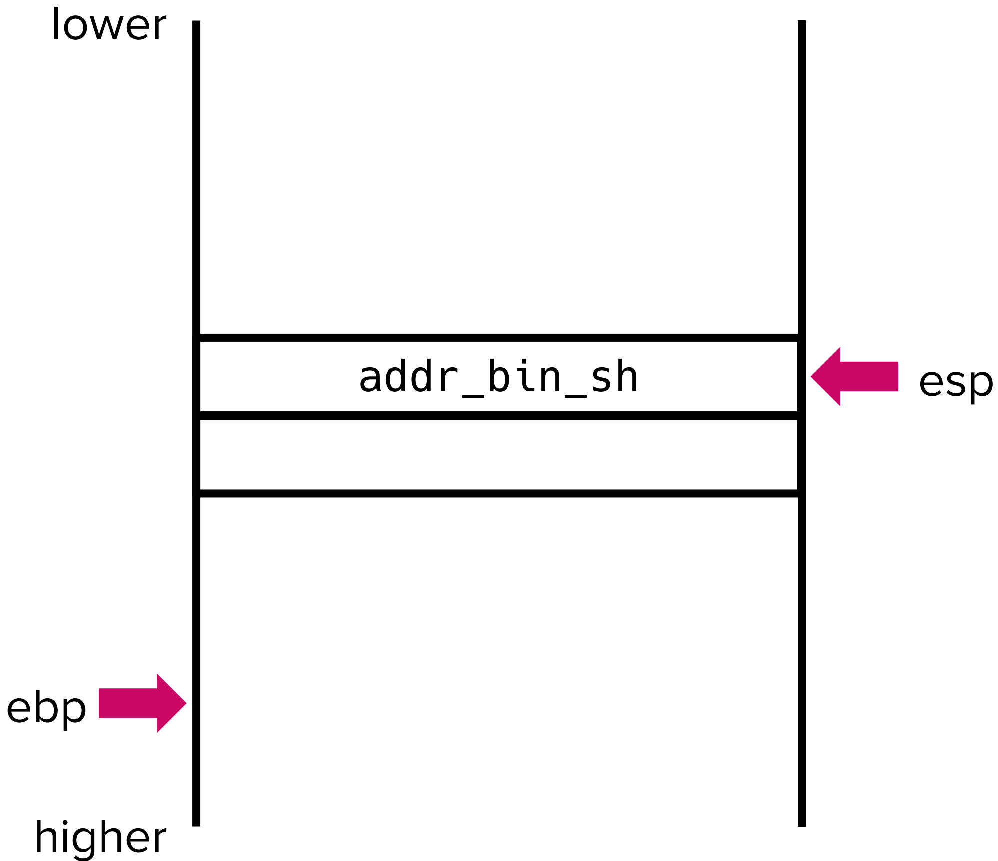
```
05 76 2e 00 00          add    eax,0x2e76
83 ec 0c                sub    esp,0xc
8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
52                      push   edx
89 c3                   mov    ebx,eax
e8 b0 fe ff ff          call   8049050 <system@plt>
```

Next instruction:
(not important)

Background: Stack machine workflow

POSTECH

- Example: Typical invocation of system("/bin/sh");



eip

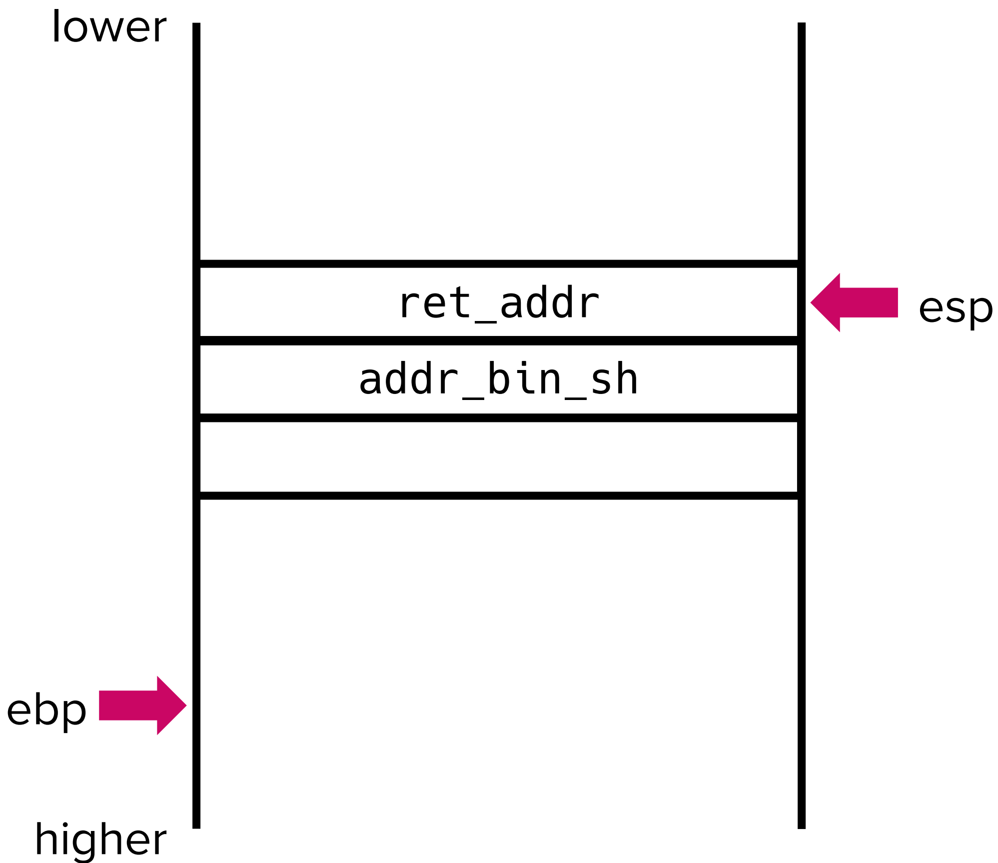
```
05 76 2e 00 00          add    eax,0x2e76
83 ec 0c                sub    esp,0xc
8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
52                      push   edx
89 c3                   mov    ebx,eax
e8 b0 fe ff ff          call   8049050 <system@plt>
```

Next instruction:
Call == Push return addr (next eip) and jump to system
(ref: Lec 05)

Background: Stack machine workflow

POSTECH

- Example: Typical invocation of system("/bin/sh");



eip

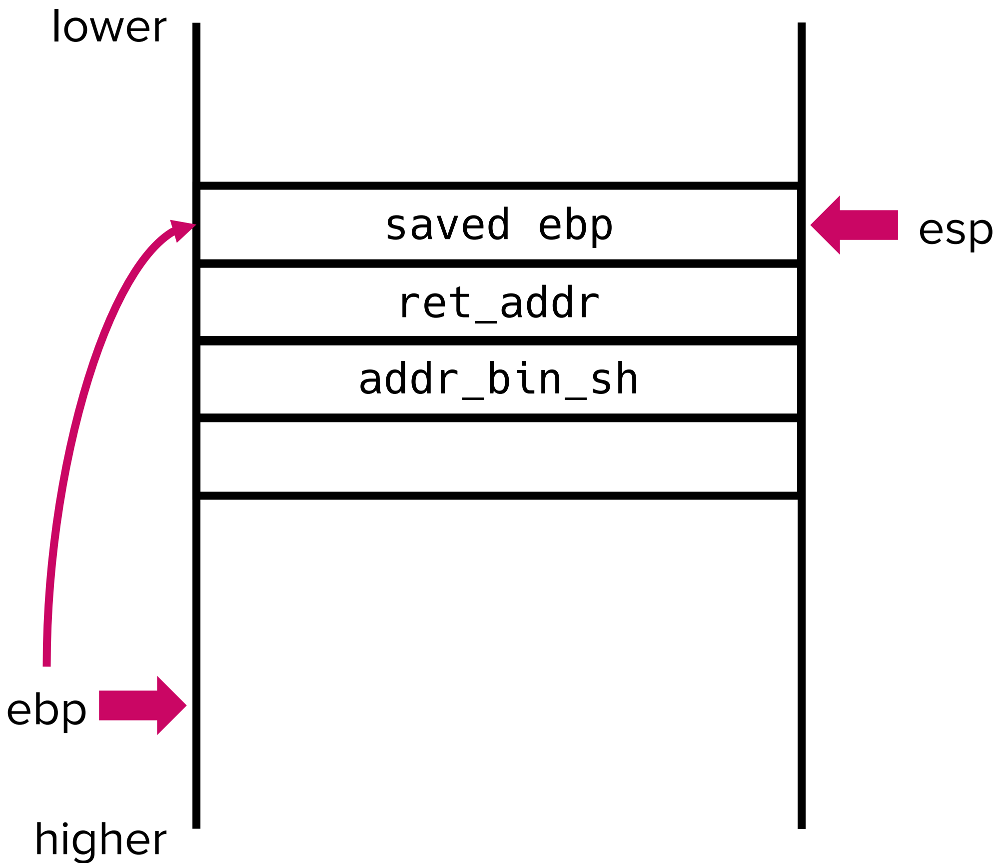
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
function prologue (1): save ebp

Background: Stack machine workflow

POSTECH

- Example: Typical invocation of system(“/bin/sh”);



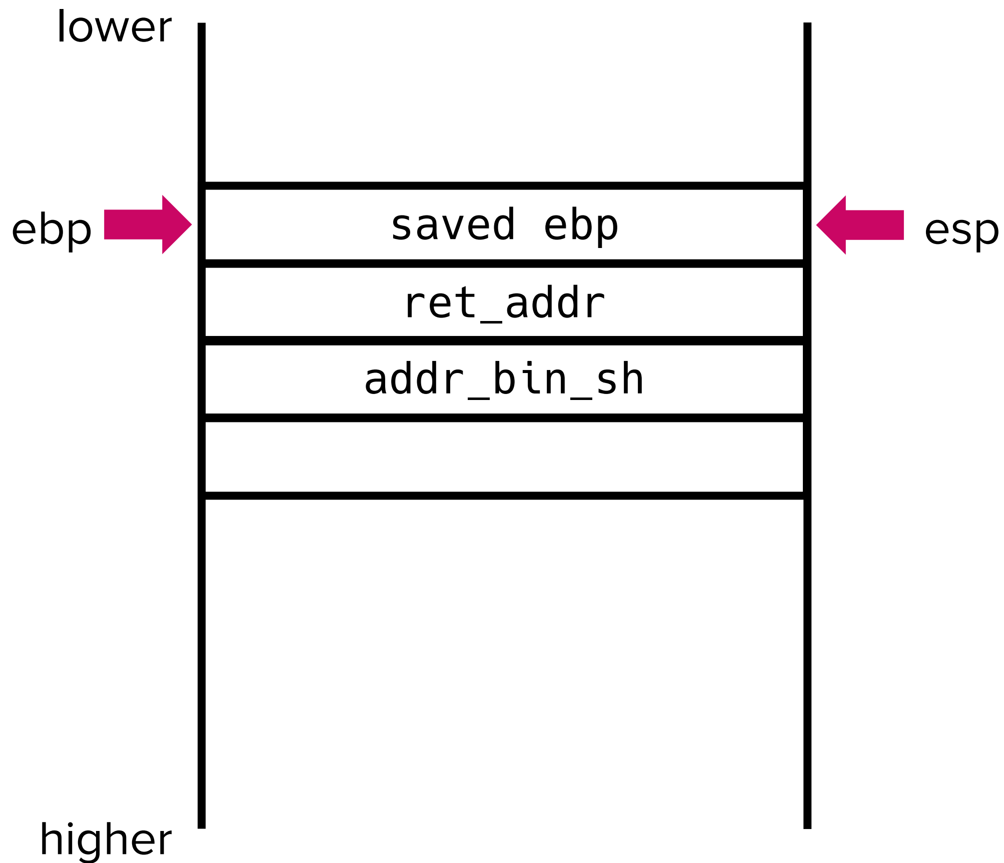
eip →

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

Next instruction:
function prologue (2): copy esp into ebp

Background: Stack machine workflow

- Example: Typical invocation of system("/bin/sh");



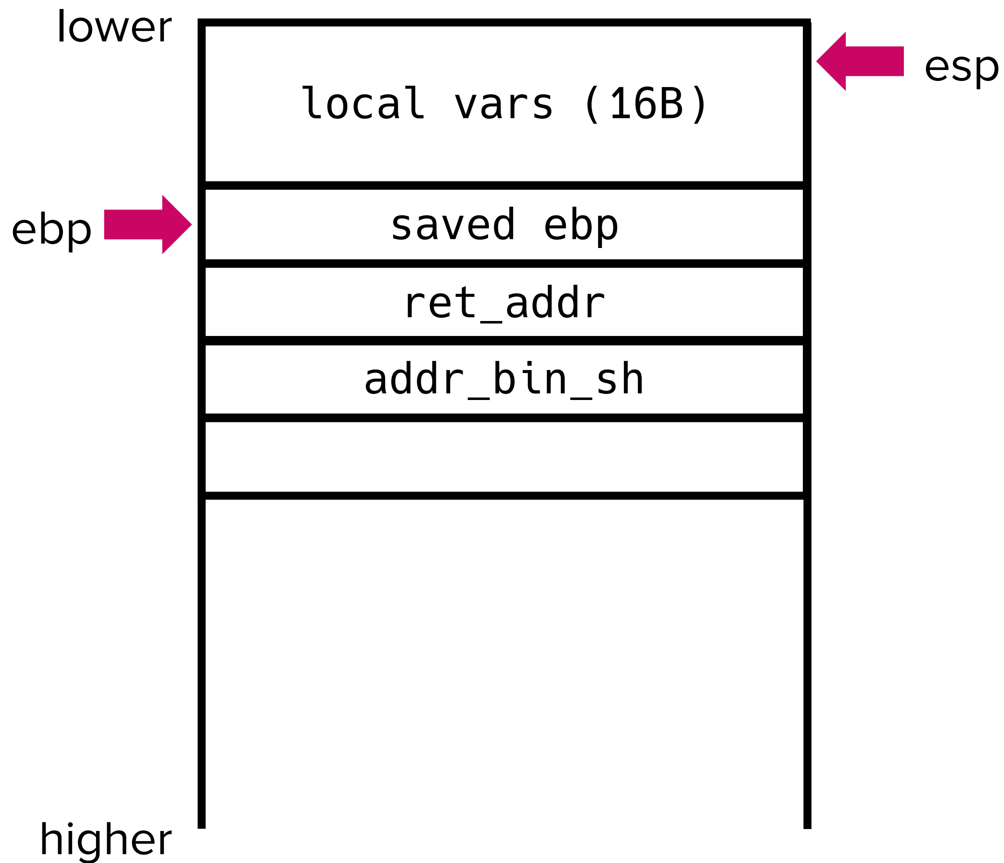
eip

```
<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: Typical invocation of system("/bin/sh");



eip

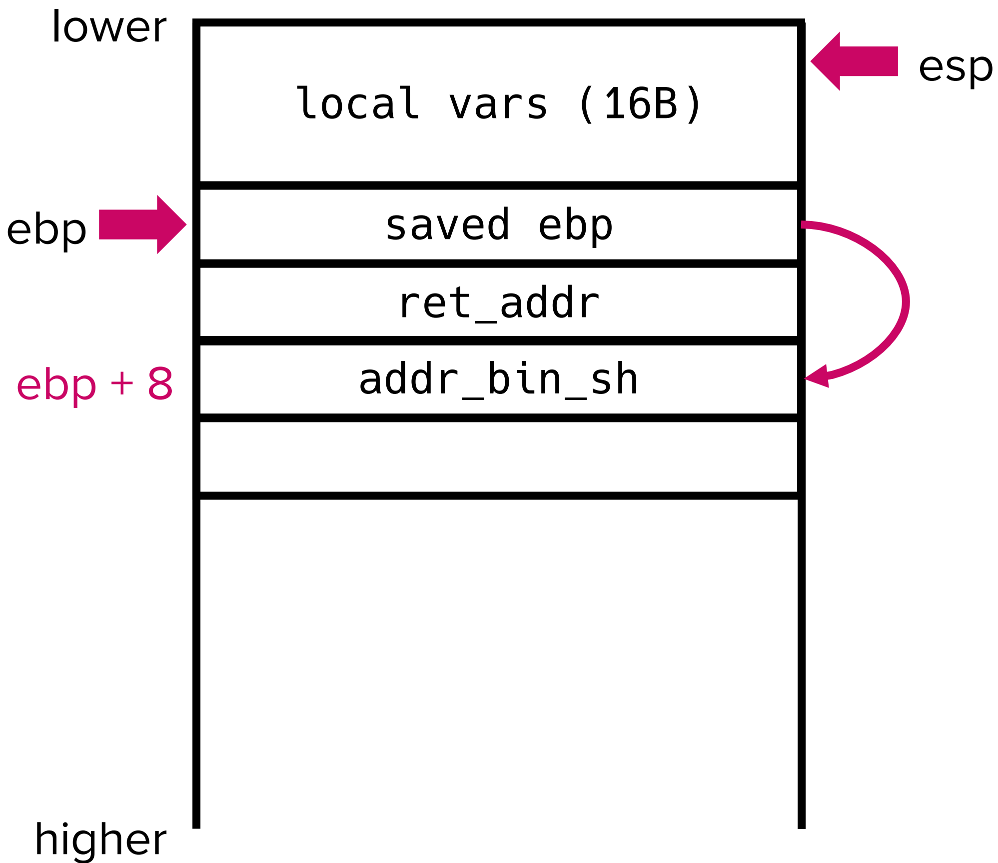
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
Args are accessed using ebp
(e.g., 1st arg is at ebp+8)

Background: Stack machine workflow

POSTECH

- Example: Typical invocation of system(“/bin/sh”);



eip

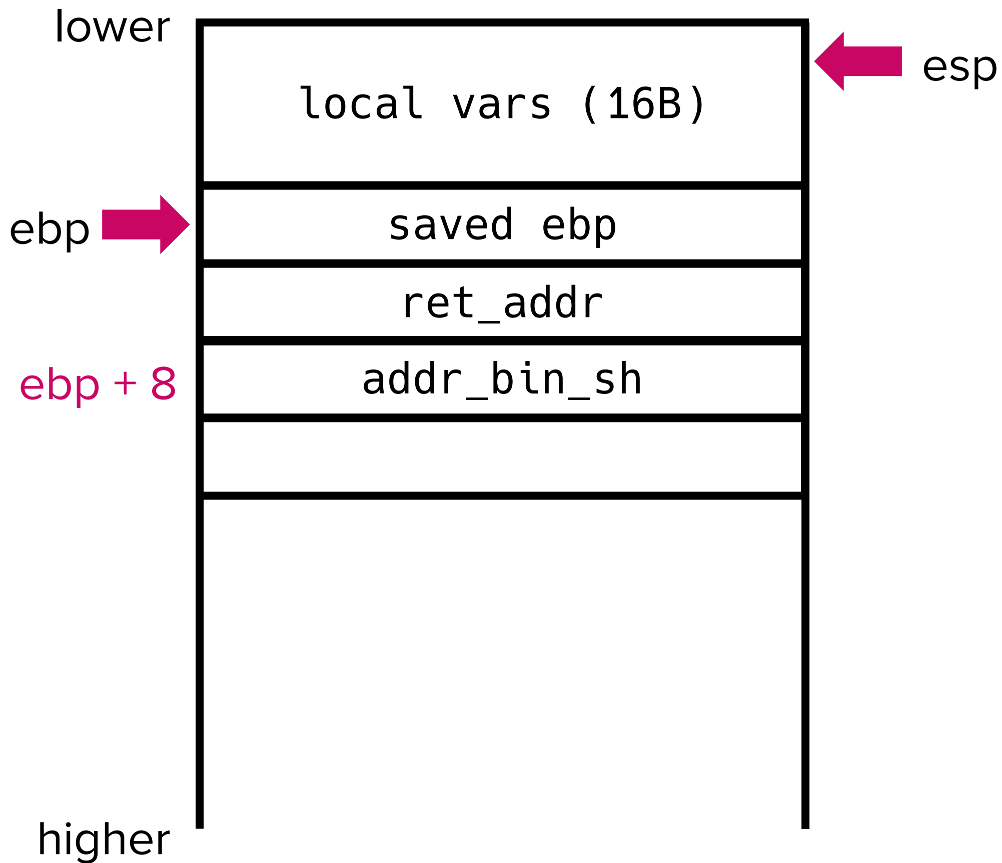
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

addr_bin_sh is saved in edx for internal use

Background: Stack machine workflow

POSTECH

- Example: Typical invocation of system(“/bin/sh”);




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

eip → leave

Next instruction:

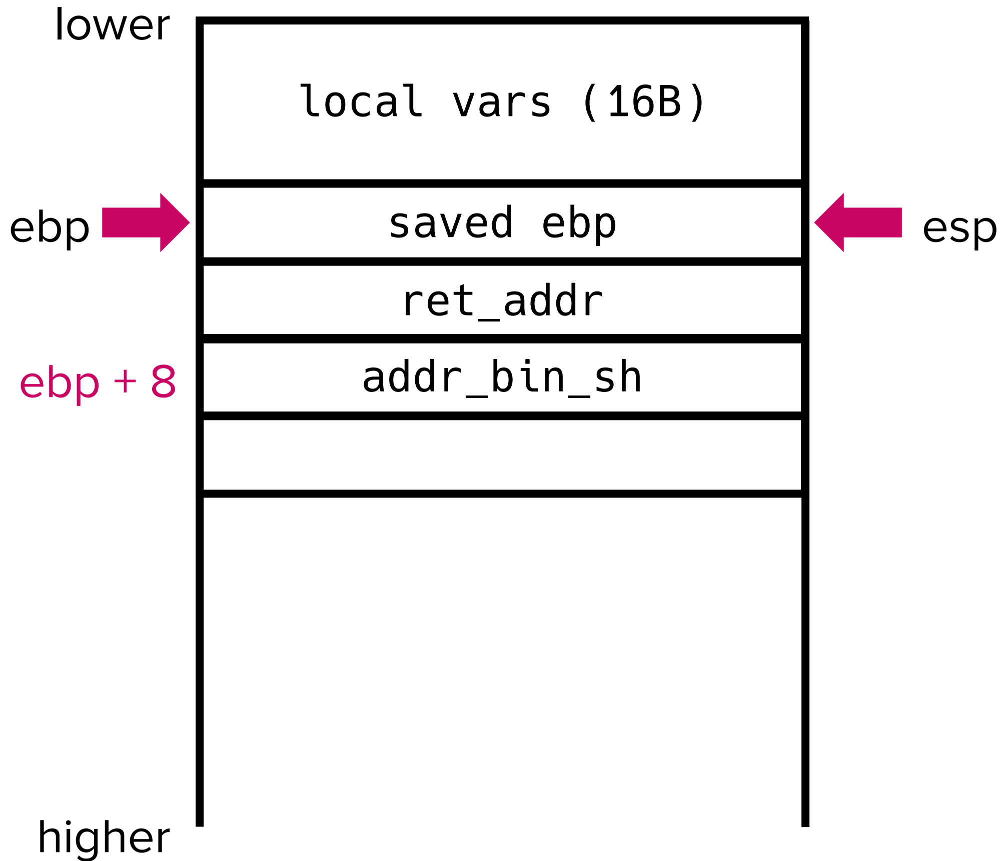
leave == mov esp,ebp;
pop ebp;

(clean up stack and restore saved ebp)

Background: Stack machine workflow

POSTECH

- Example: Typical invocation of system(“/bin/sh”);



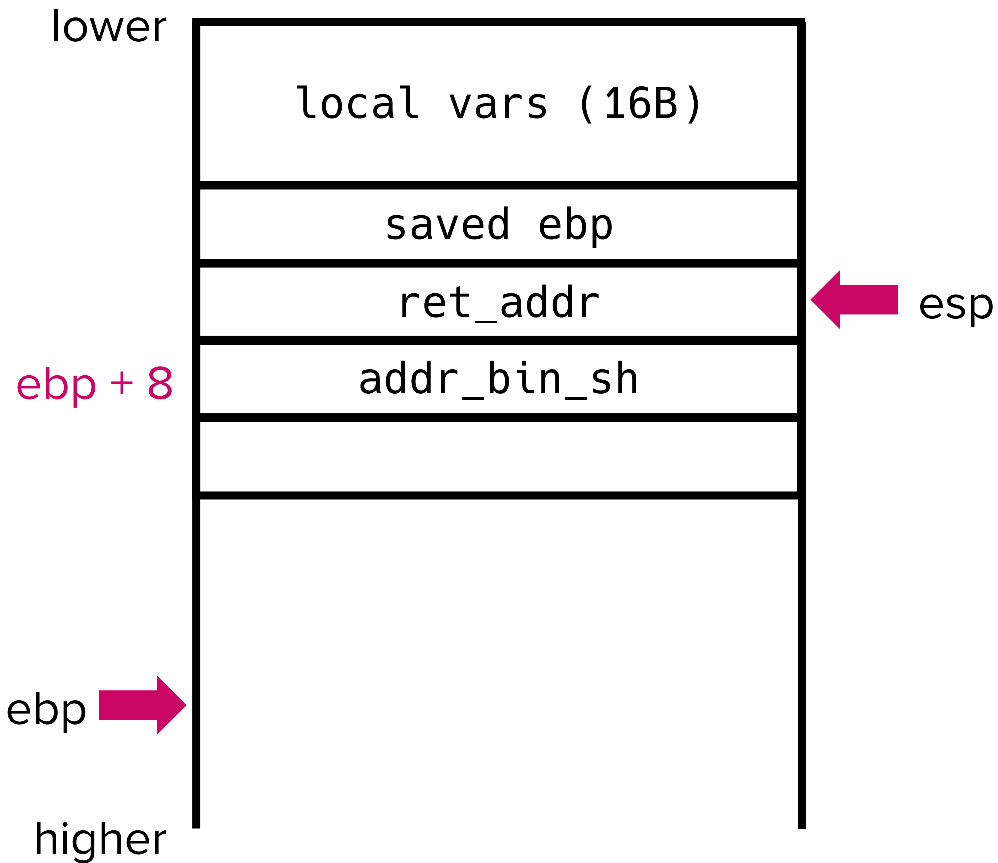

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

eip → ret

leave == mov esp,ebp;
pop ebp;
(clean up stack and restore saved ebp)

Background: Stack machine workflow

- Example: Typical invocation of system("/bin/sh");




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

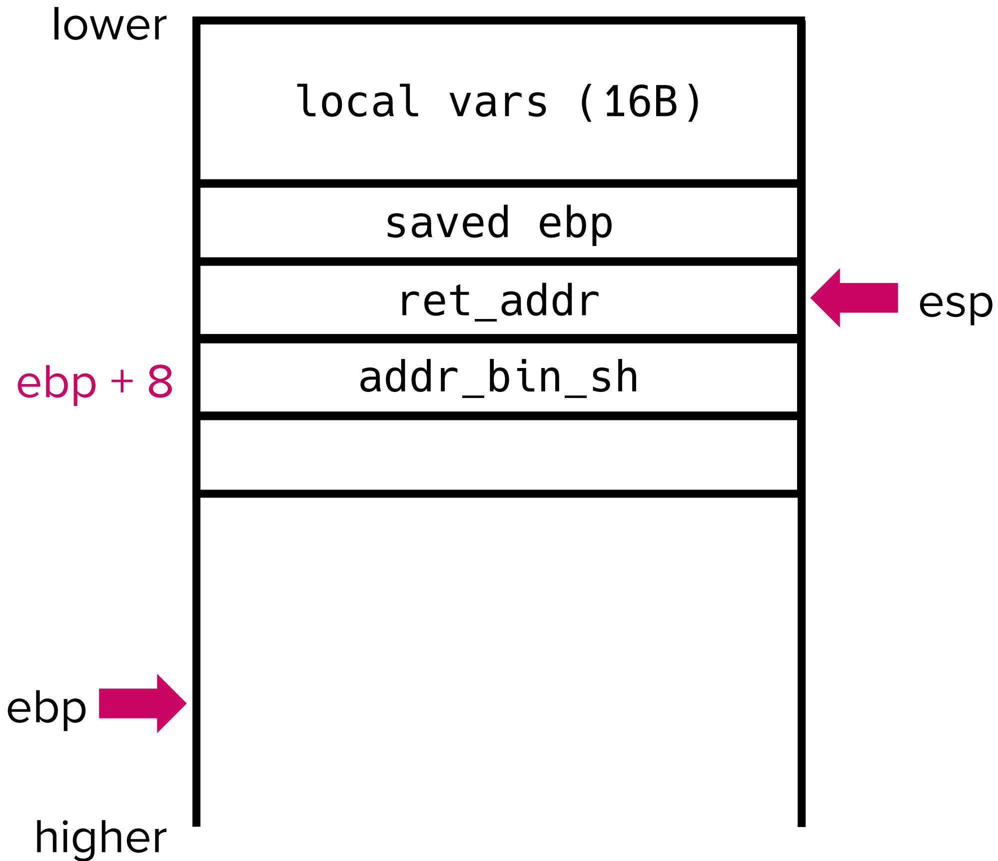
eip

leave == mov esp,ebp;
pop ebp;
(clean up stack and restore saved ebp)

Background: Stack machine workflow

POSTECH

- Example: Typical invocation of system(“/bin/sh”);



eip →

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

Next instruction:
ret == pop eip
(return to saved address)

Background: Stack machine workflow

- Example: Typical invocation of system("/bin/sh");

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

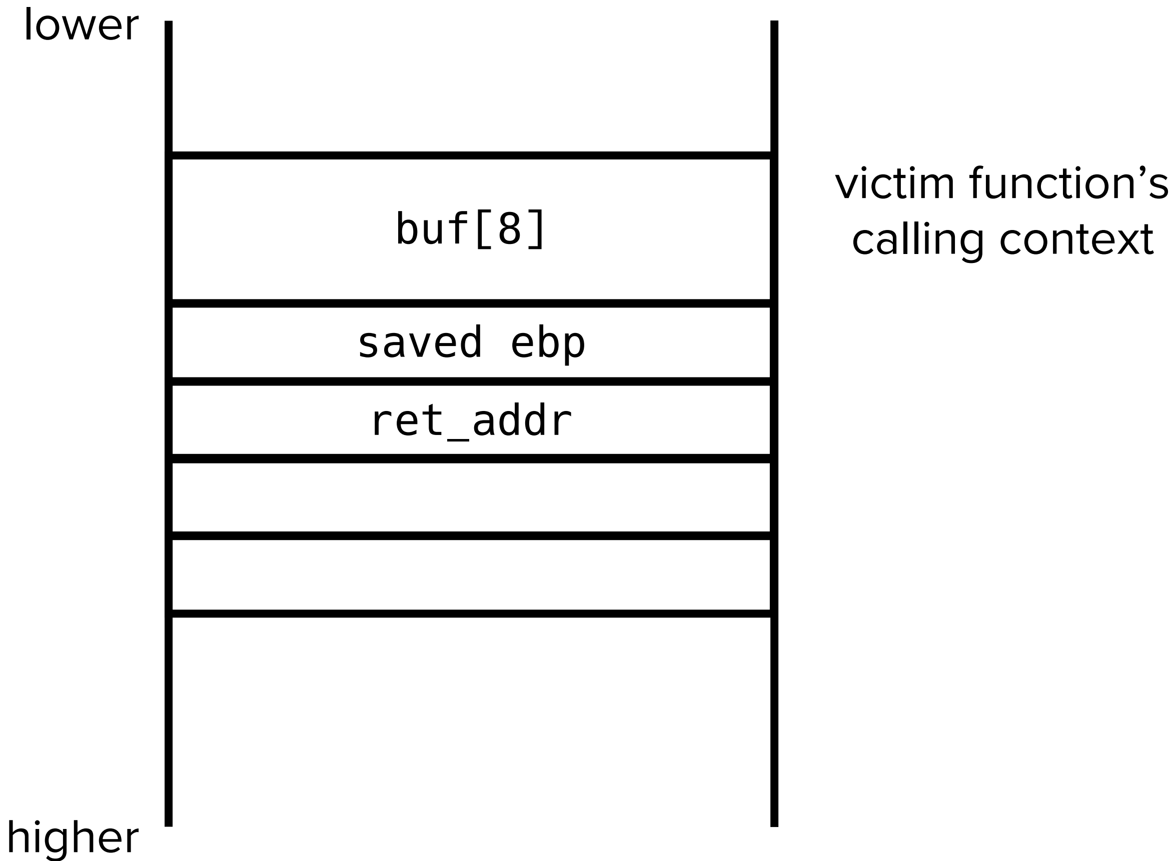
Next instruction:
ret == pop eip

The program doesn't know (and doesn't care about) the semantics of execution. It just accesses args utilizing ebp and returns to the saved address by utilizing esp.

Return-to-libc attack

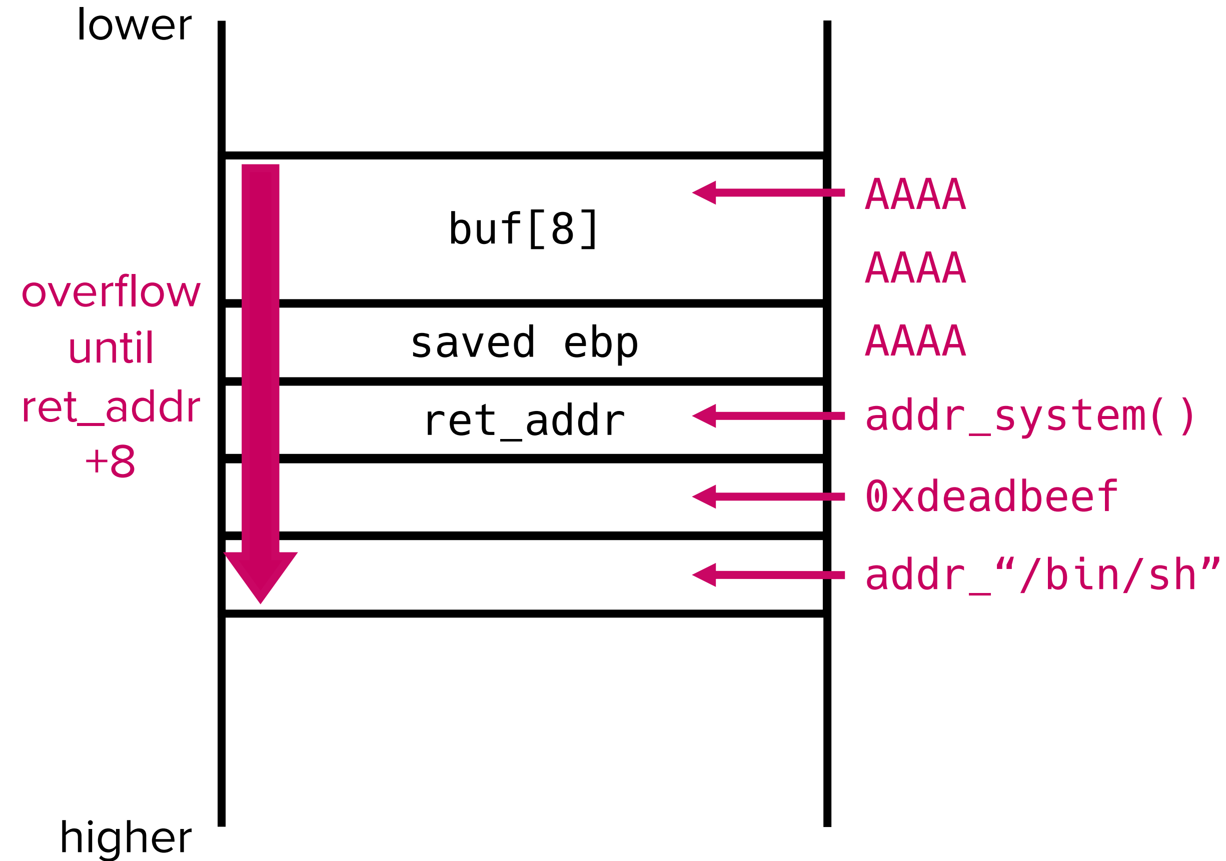
POSTECH

- Stack layout of victim function



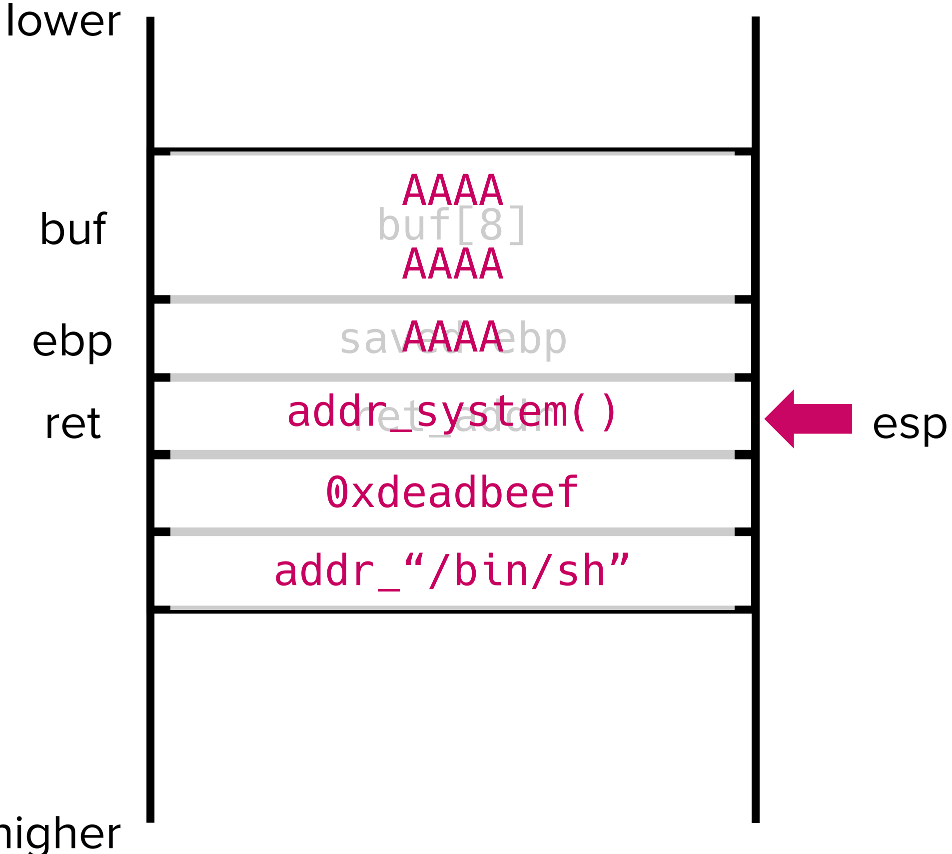
Return-to-libc attack

- Attack payload



Return-to-libc attack

- Exploit



eip →

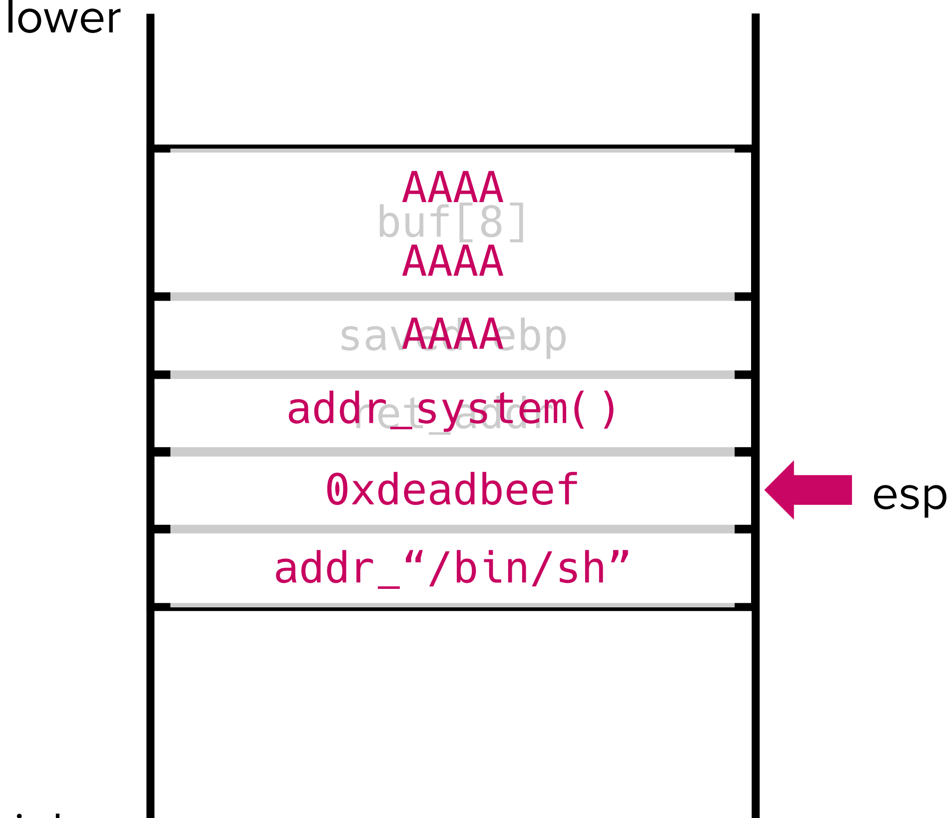
```
<victim_function>:
...
leave
ret
```

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

Return-to-libc attack

POSTECH

- Exploit



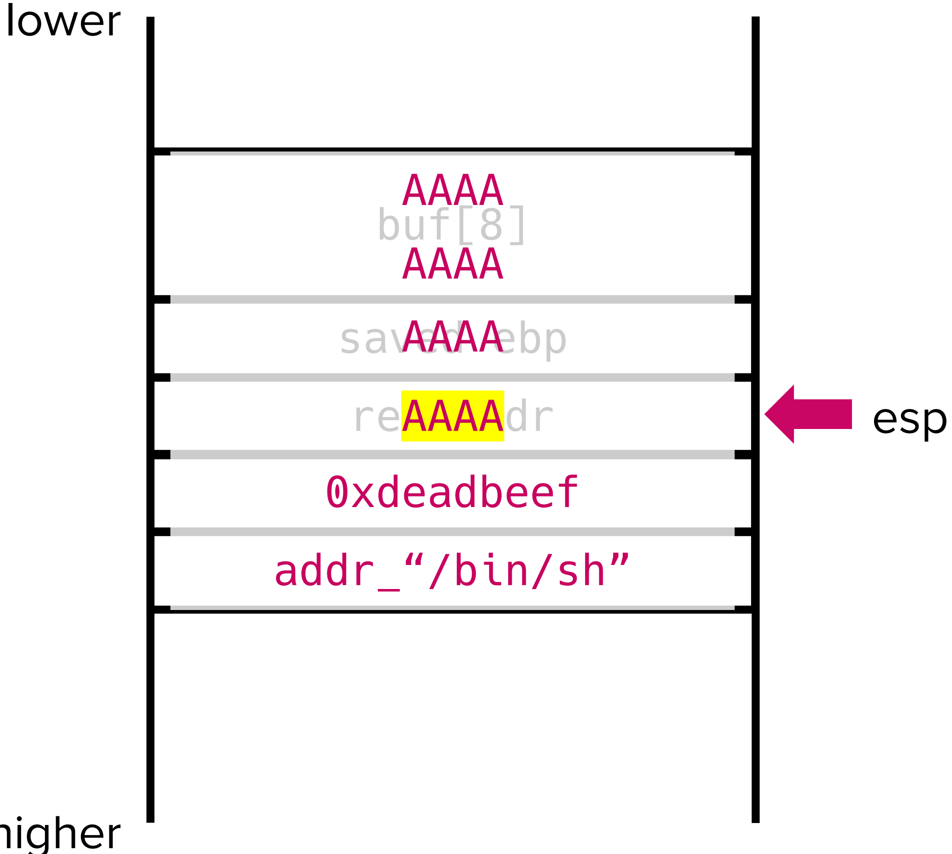
eip →

```
<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

- Exploit



eip →

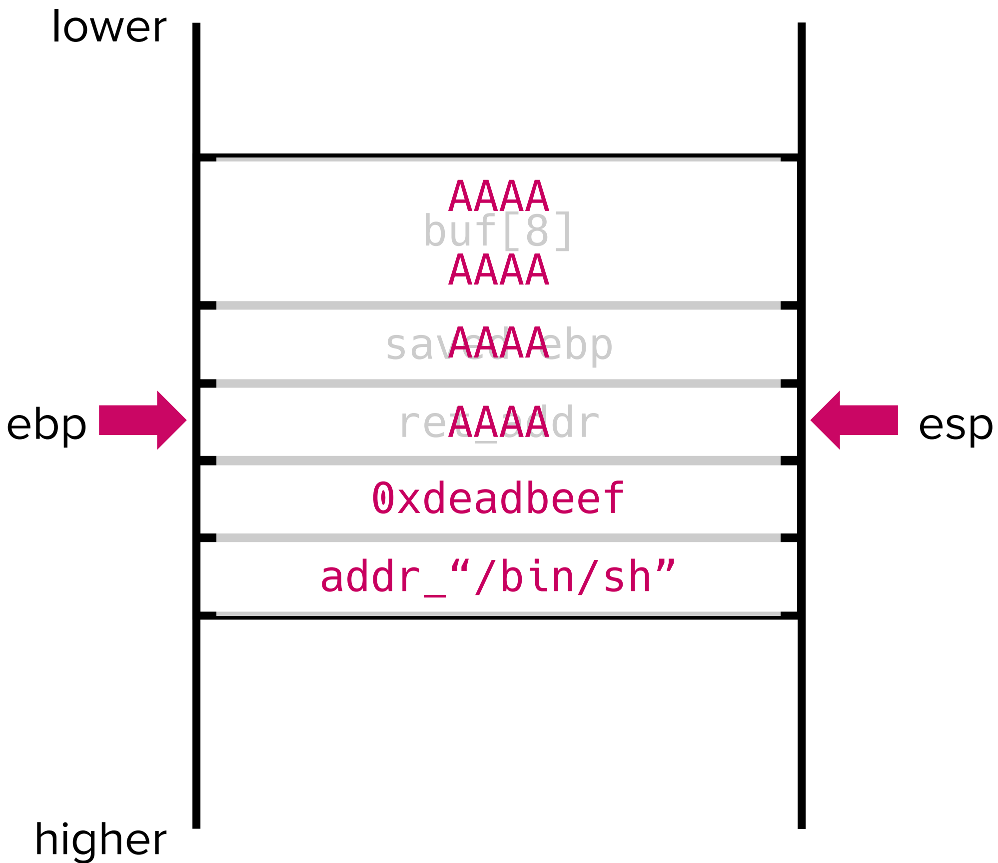
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
function prologue (2): copy esp into ebp

Return-to-libc attack

POSTECH

- Exploit



eip

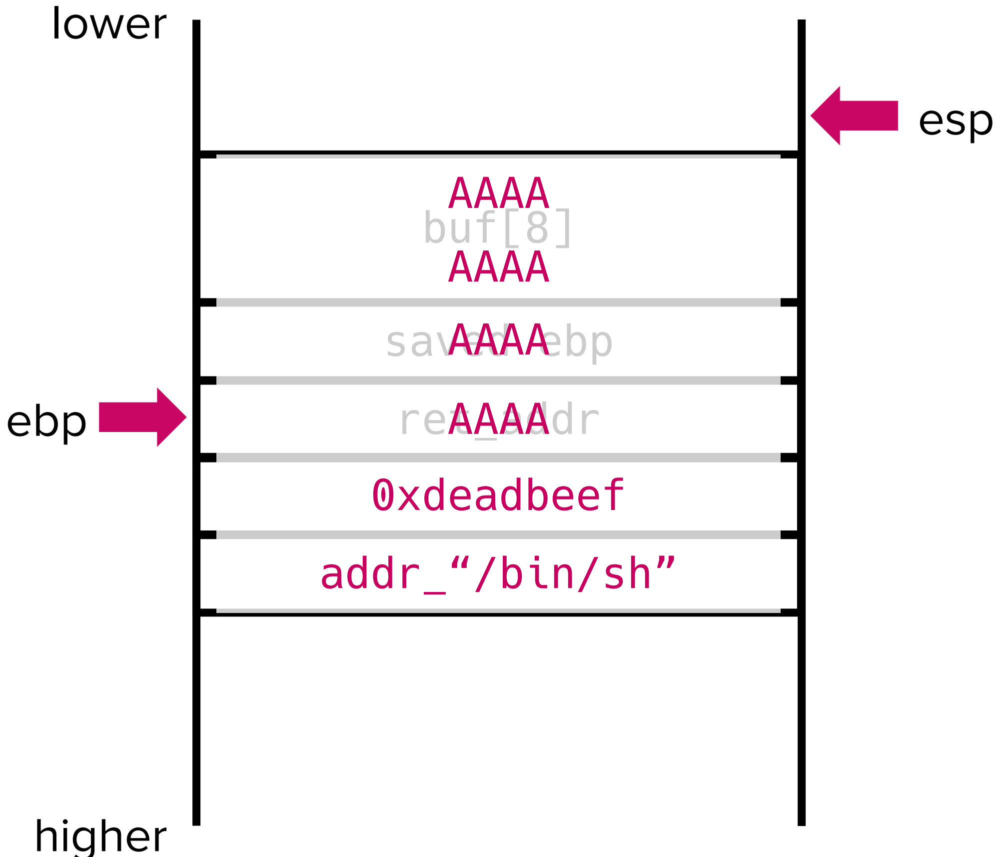
```
<system>:
push   ebp
mov    ebp,esp
sub    esp, 0x10
...
mov    edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
Reserve stack space

Return-to-libc attack

POSTECH

- Exploit



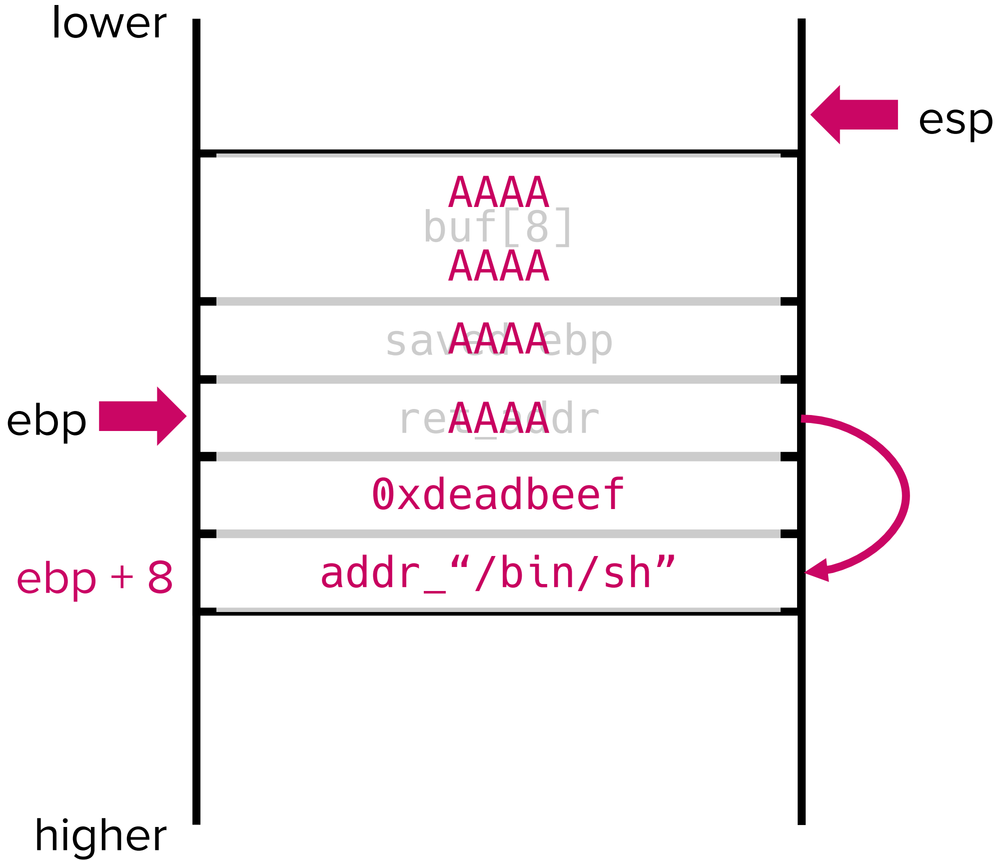
eip

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

Next instruction:
Args are accessed using ebp
(e.g., 1st arg is at ebp+8)

Return-to-libc attack

- Exploit



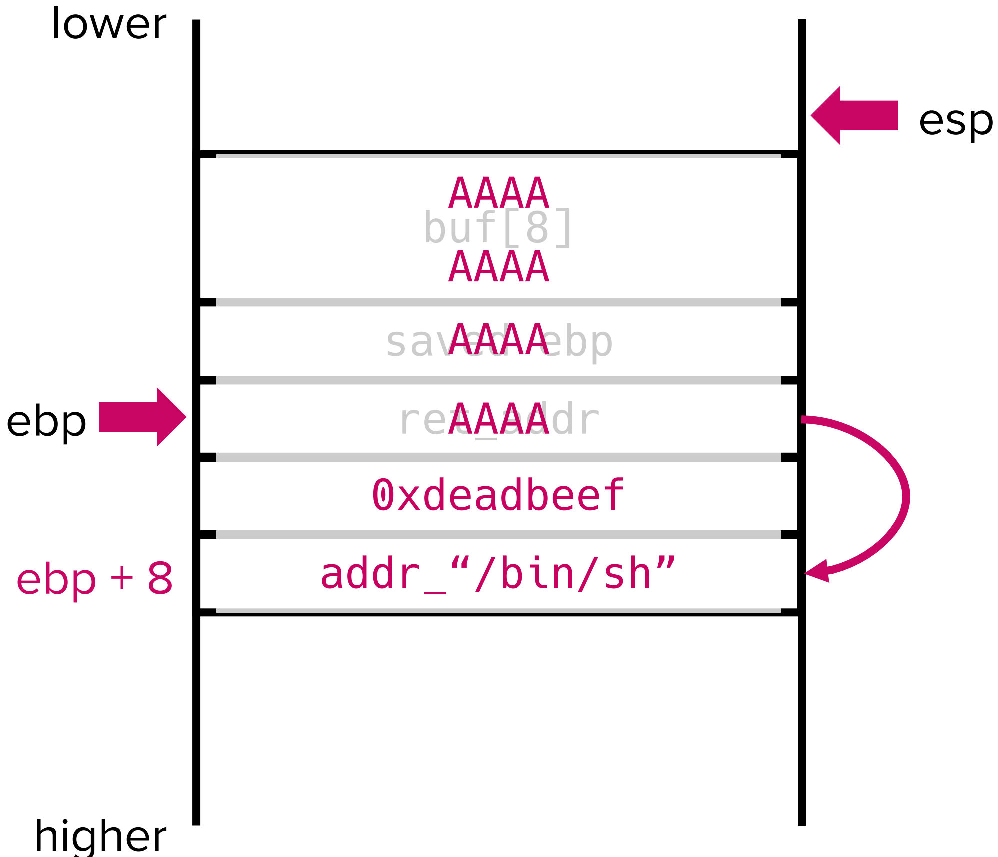
eip

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

addr_bin_sh is saved in edx for internal use

Return-to-libc attack

- Exploit



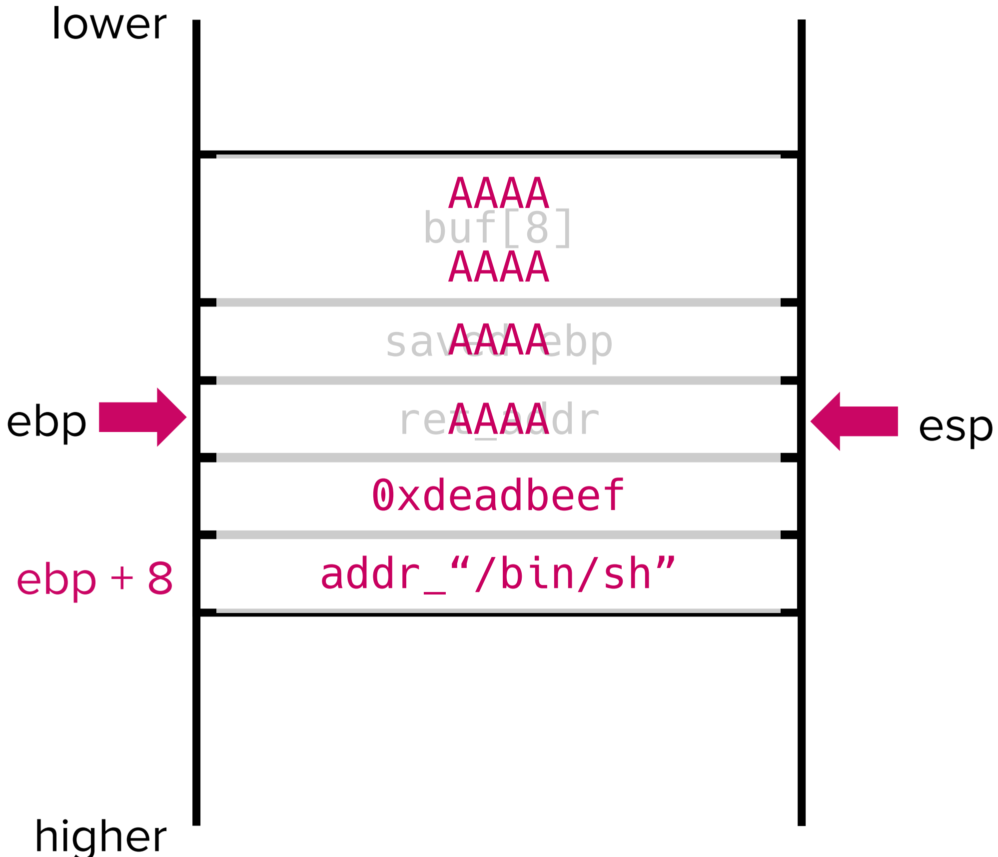
eip

```
<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

- Exploit



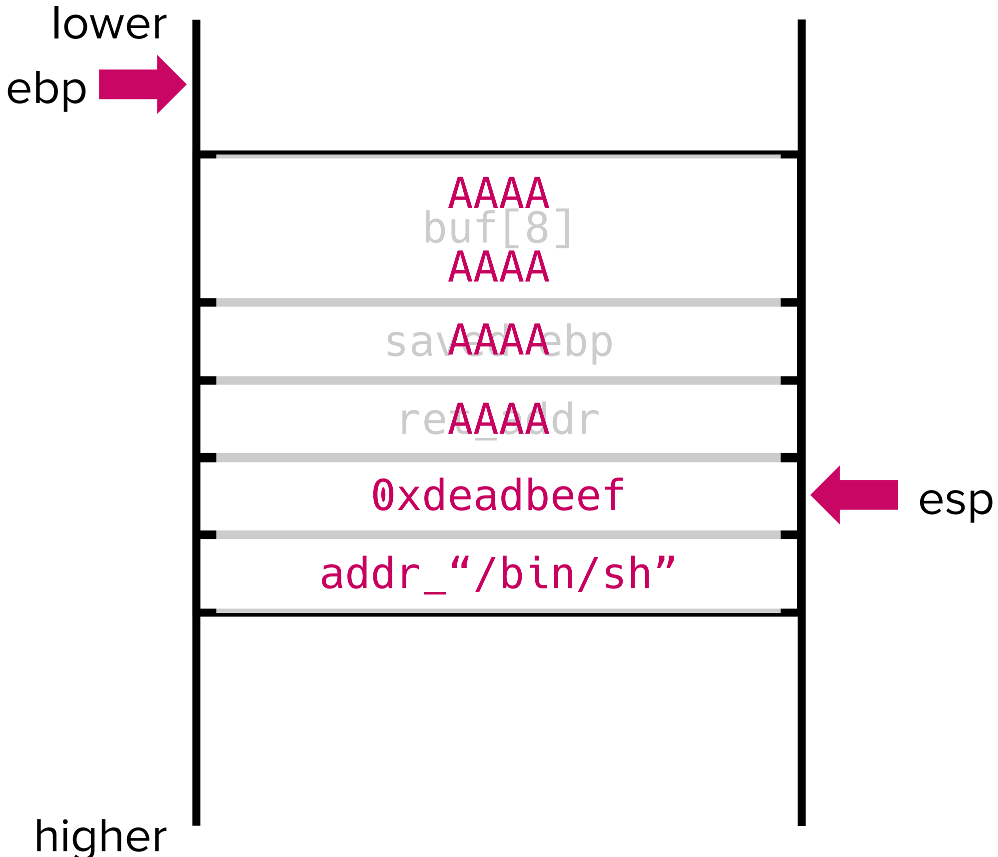

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

eip

leave == mov esp,ebp;
pop ebp;
(clean up stack and restore saved ebp)

Return-to-libc attack

- Exploit



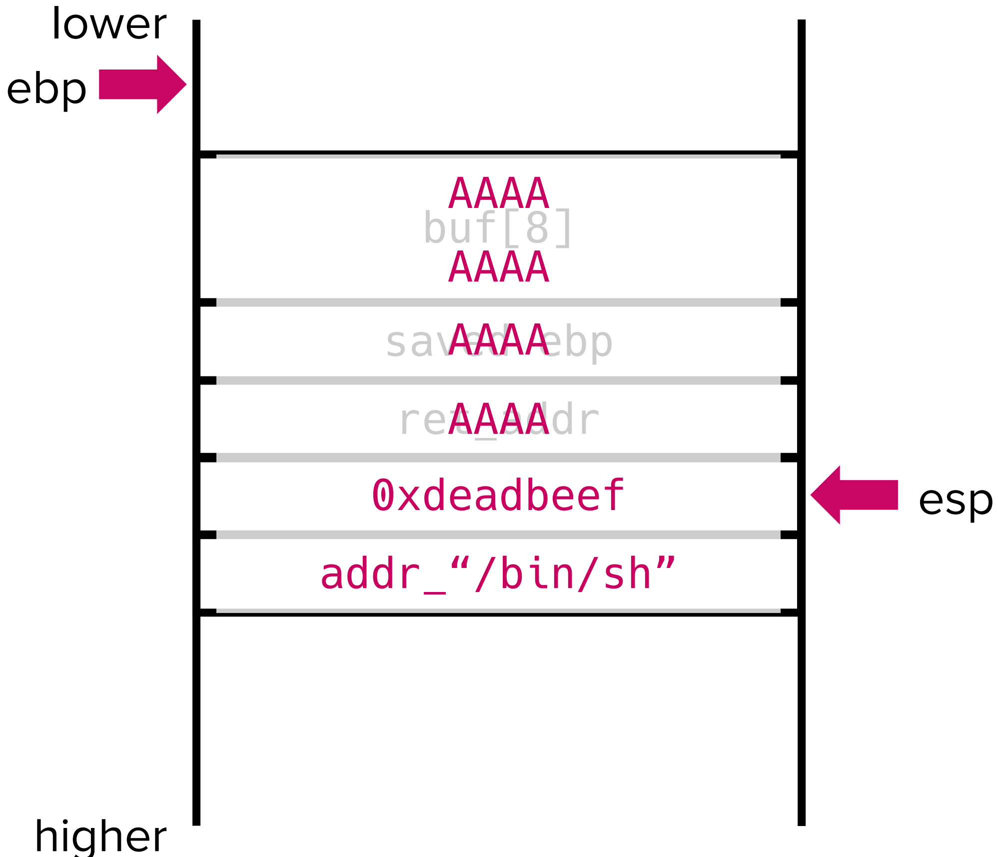
eip →

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

leave == mov esp,ebp;
pop ebp;
(clean up stack and restore saved ebp)

Return-to-libc attack

- Exploit



eip →

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

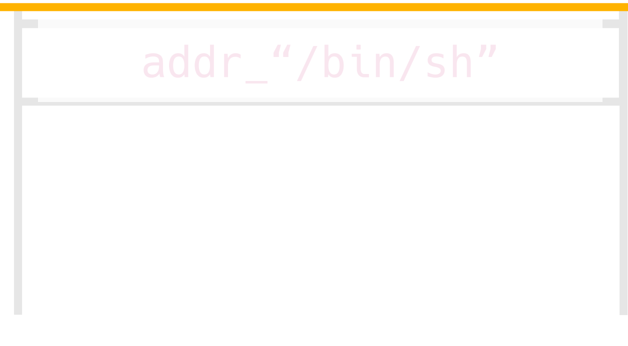
Next instruction:
return to 0xdeadbeef (and crash)

Return-to-libc attack

- Exploit

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

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 crash

Return-to-libc summary

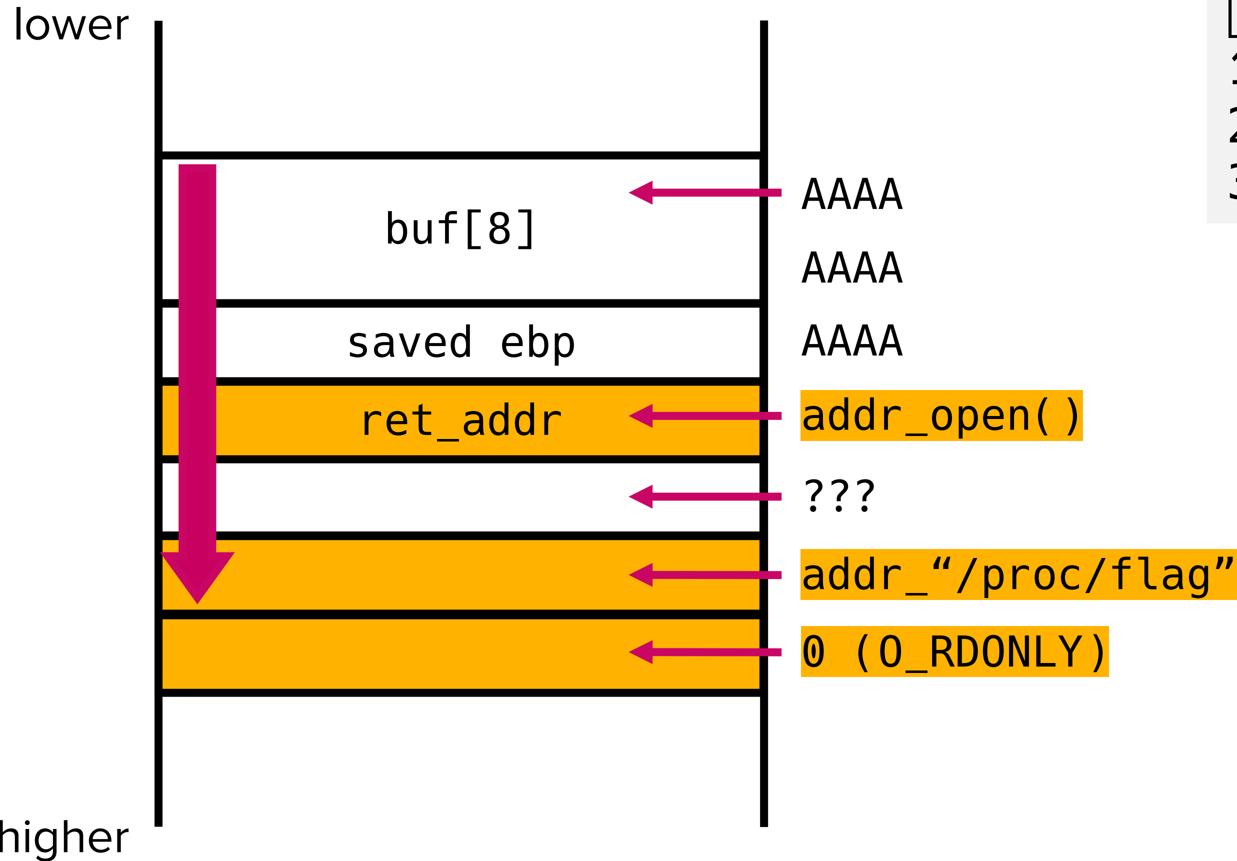
POSTECH

- Returning to the exising code, we can bypass NX
  - libc functions are good targets and they are executable
- Question: Can we chain multiple function calls?
  - Instead of letting the program crash at `0xdeadbeef`, can we have it keep executing multiple libc functions?
  - e.g., a sequence of functions to print "/proc/flag"

```
1. int fd = open("/proc/flag", O_RDONLY); // open a file (fd=3)
2. read(fd, gbuf_addr, 1040); // read into gbuf
3. write(1, gbuf_addr, 1040); // write gbuf to stdout (fd=1)
```

Extensibility of return-to-libc

- Trying to chain three libc function calls

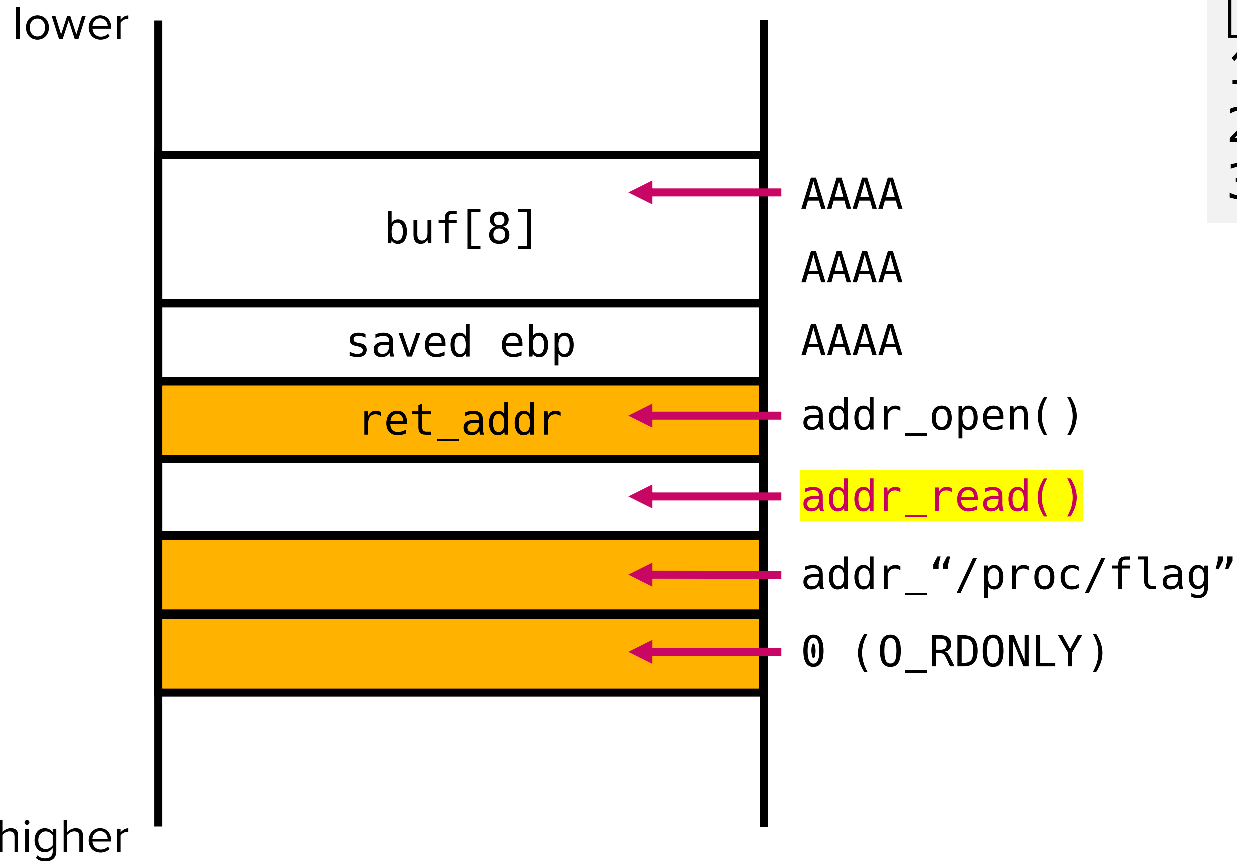



```
[Goal]
1. int fd = open("/proc/flag", O_RDONLY);
2. read(fd, gbuf_addr, 1040);
3. write(stdout, gbuf_addr, 1040);
```

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

Extensibility of return-to-libc

- Trying to chain three libc function calls



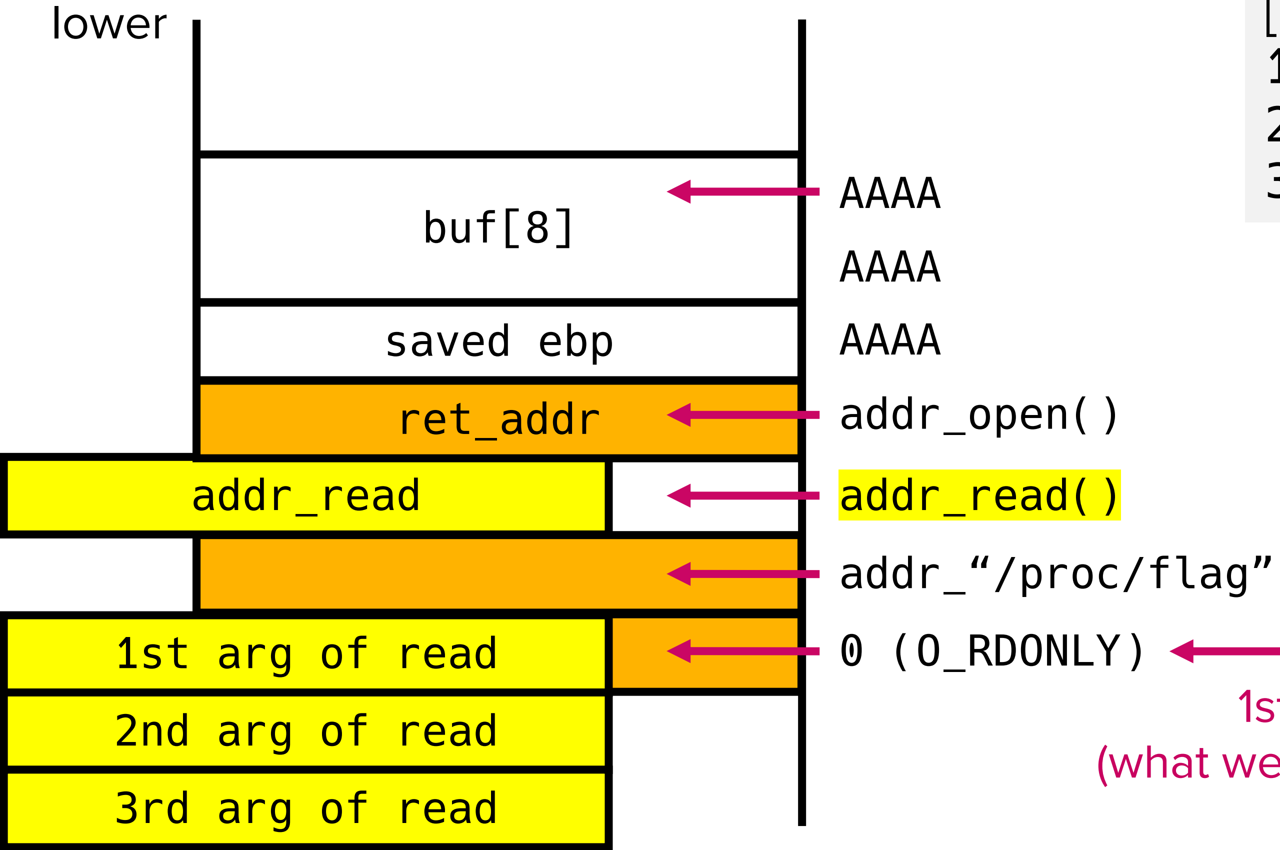

```
[Goal]
1. int fd = open(“/proc/flag”, O_RDONLY);
2. read(fd, gbuf_addr, 1040);
3. write(stdout, gbuf_addr, 1040);
```

1. `open(“/proc/flag”, O_RDONLY);` is invoked
2. return to `read();`

args??

Extensibility of return-to-libc

- Trying to chain three libc function calls




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

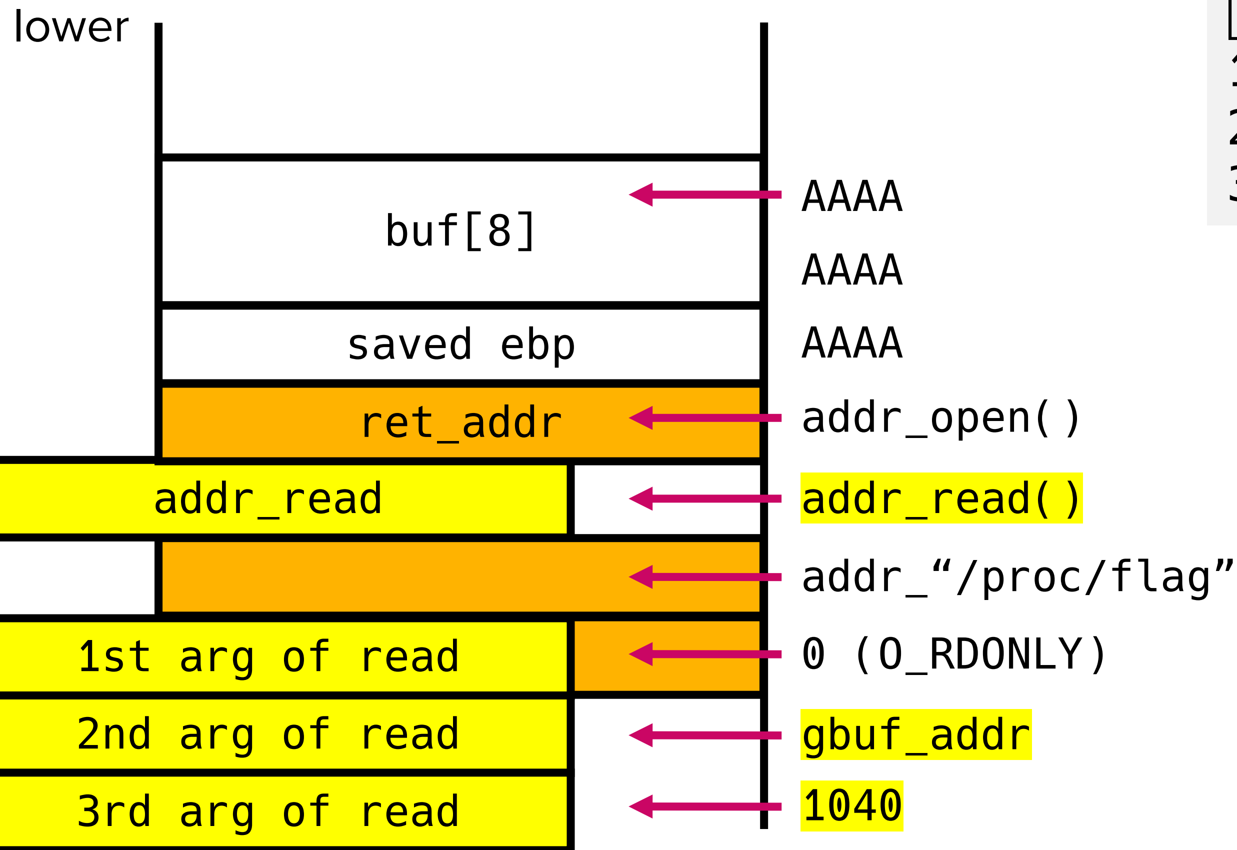
1. `open("/proc/flag", O_RDONLY);` is invoked
2. return to `read();`

args??

1st arg already set to 0
(what we need: fd returned by open)

Extensibility of return-to-libc

- Trying to chain three libc function calls




```
[Goal]
1. int fd = open(“/proc/flag”, O_RDONLY);
2. read(fd, gbuf_addr, 1040);
3. write(stdout, gbuf_addr, 1040);
```

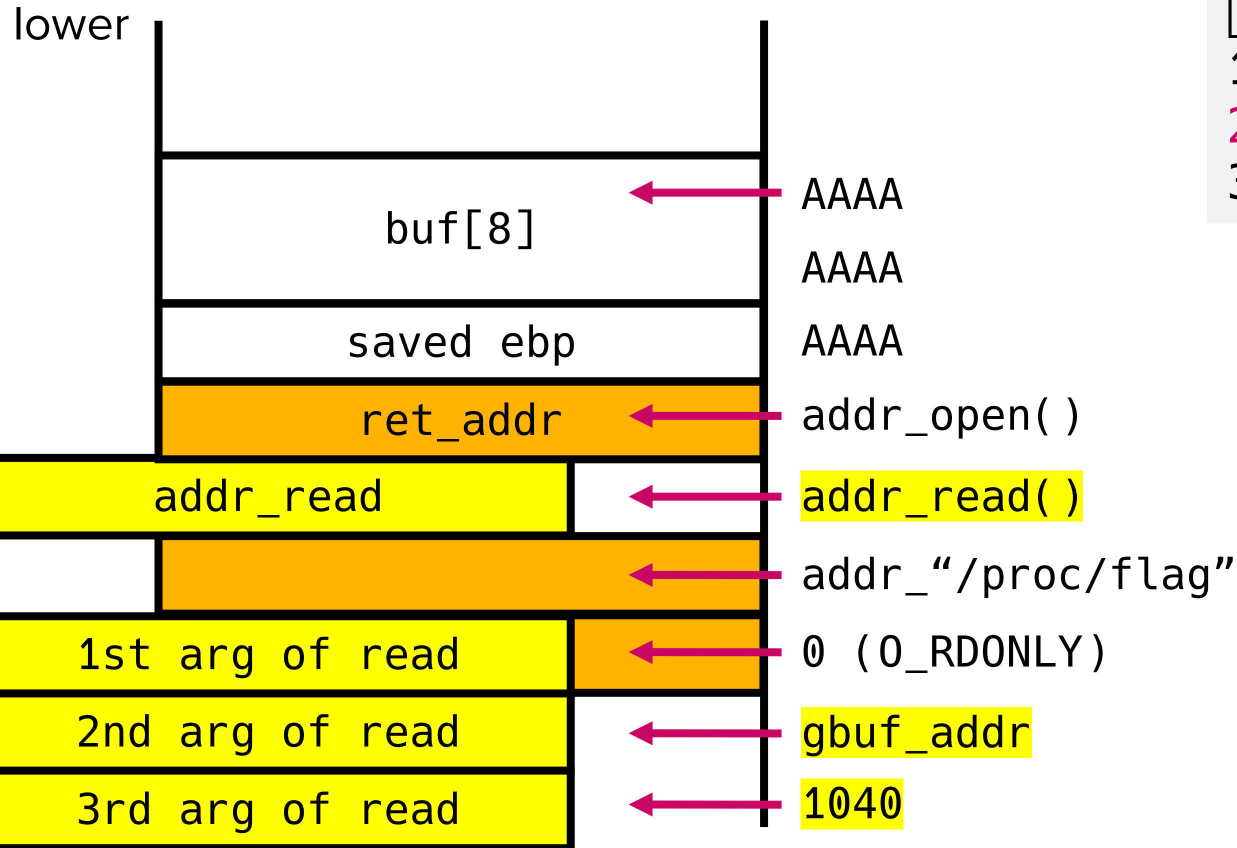
1. `open(“/proc/flag”, O_RDONLY);` is invoked
2. return to `read(0, gbuf_addr, 1040);`

Q) Can you identify two issues?

Extensibility of return-to-libc

POSTECH

- Trying to chain three libc function calls




```
[Goal]
1. int fd = open(“/proc/flag”, O_RDONLY);
2. read(fd, gbuf_addr, 1040);
3. write(stdout, gbuf_addr, 1040);
```

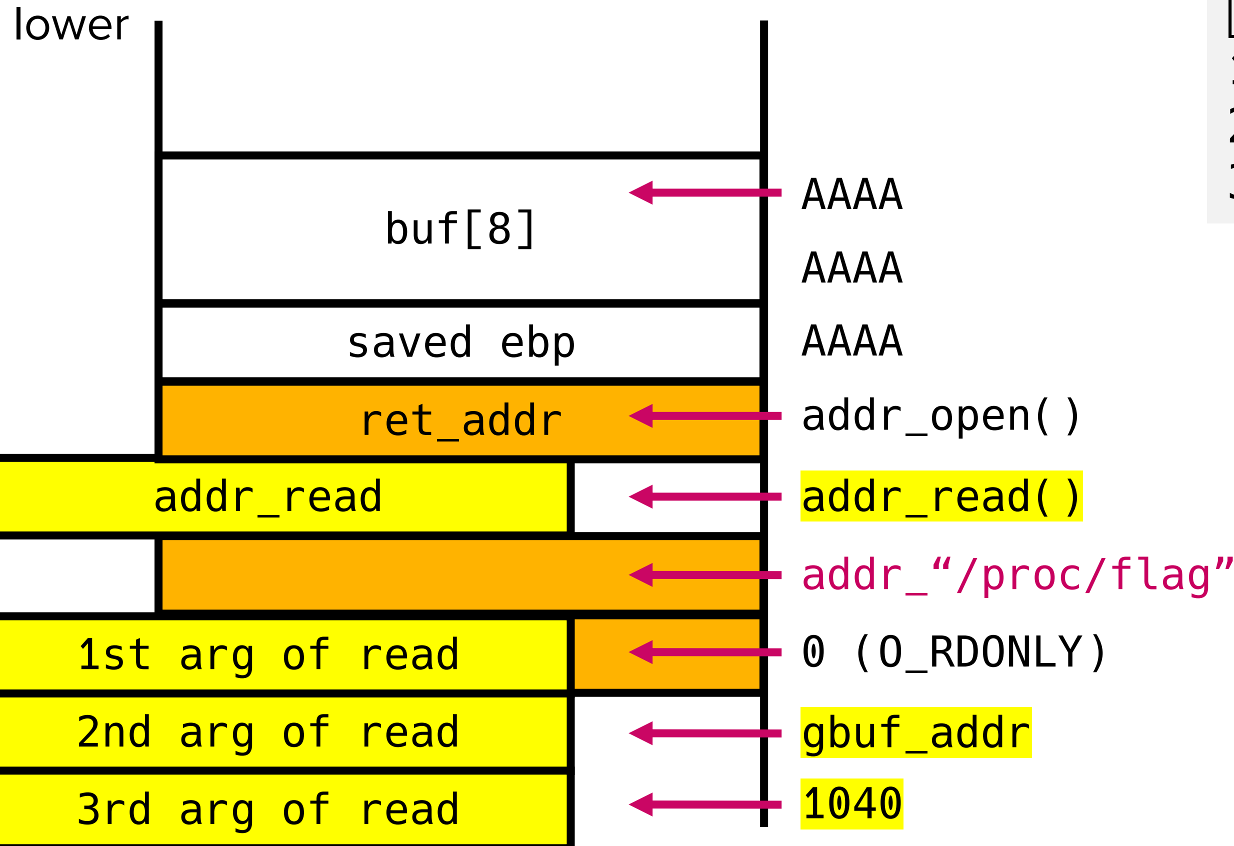
1. `open(“/proc/flag”, O_RDONLY);` is invoked
2. return to `read(0, gbuf_addr, 1040);`

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

Extensibility of return-to-libc

POSTECH

- Trying to chain three libc function calls



[Goal]

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

1. `open("/proc/flag", 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_"/proc/flag"
→ Call chain breaks here :(

Problems of naïve chaining

POSTECH

- To chain multiple functions, the payload must include:

ret: 1st func addr
ret addr 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:

1st func		2nd func
ret: 1st func addr		
ret addr after 1st func		2nd func addr
1st func arg 1	**conflict**	ret addr 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

POSTECH

- Returning to code that changes `esp` and ends with `ret`
  - e.g., Target binary of Lab 02 contains a "`pop; pop; ret;`" 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 addr `esp` points to

Attack #1-2: ROP

Return-Oriented Programming (ROP)

POSTECH

- Generalized version of 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

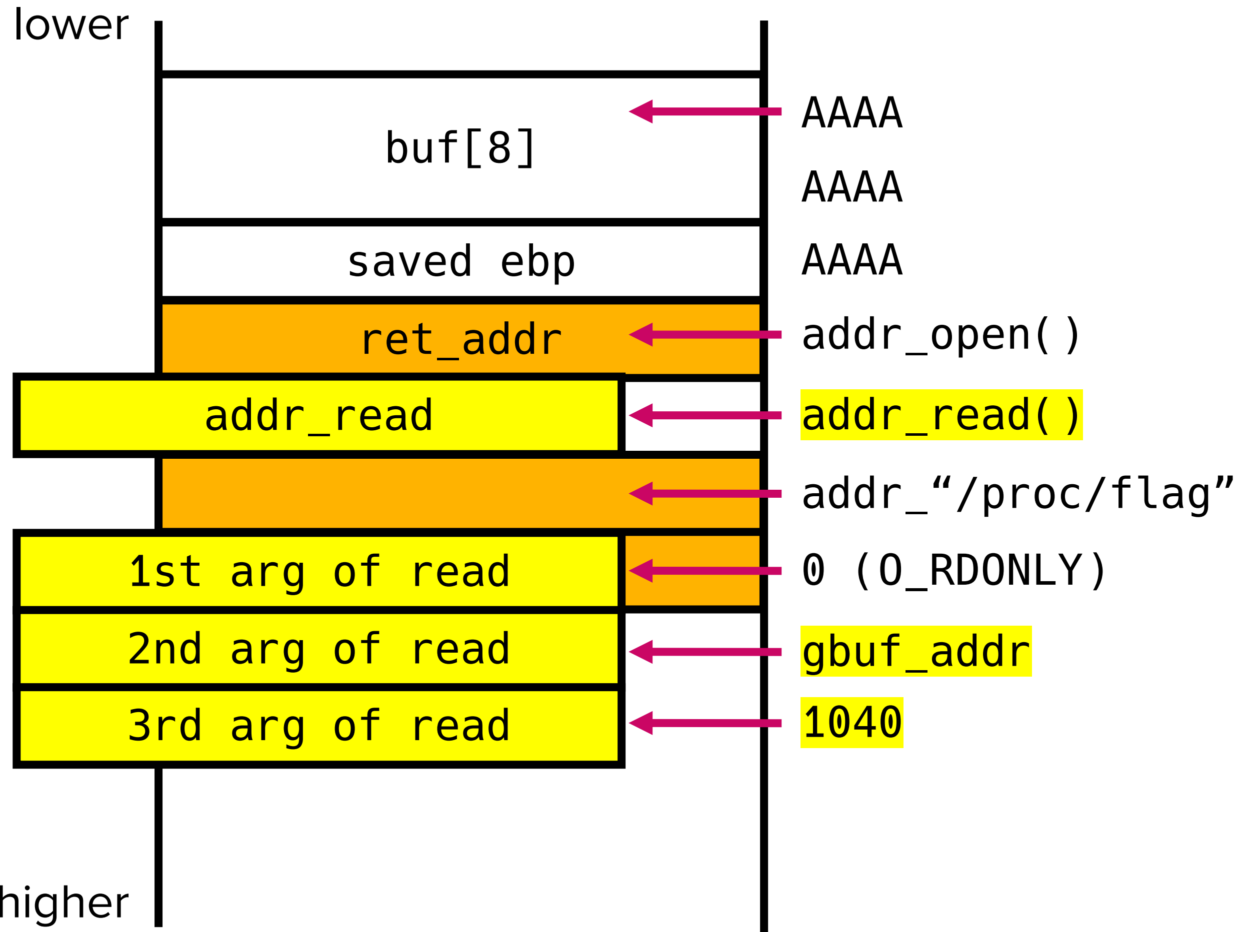
Abstract

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 through ROP gadgets

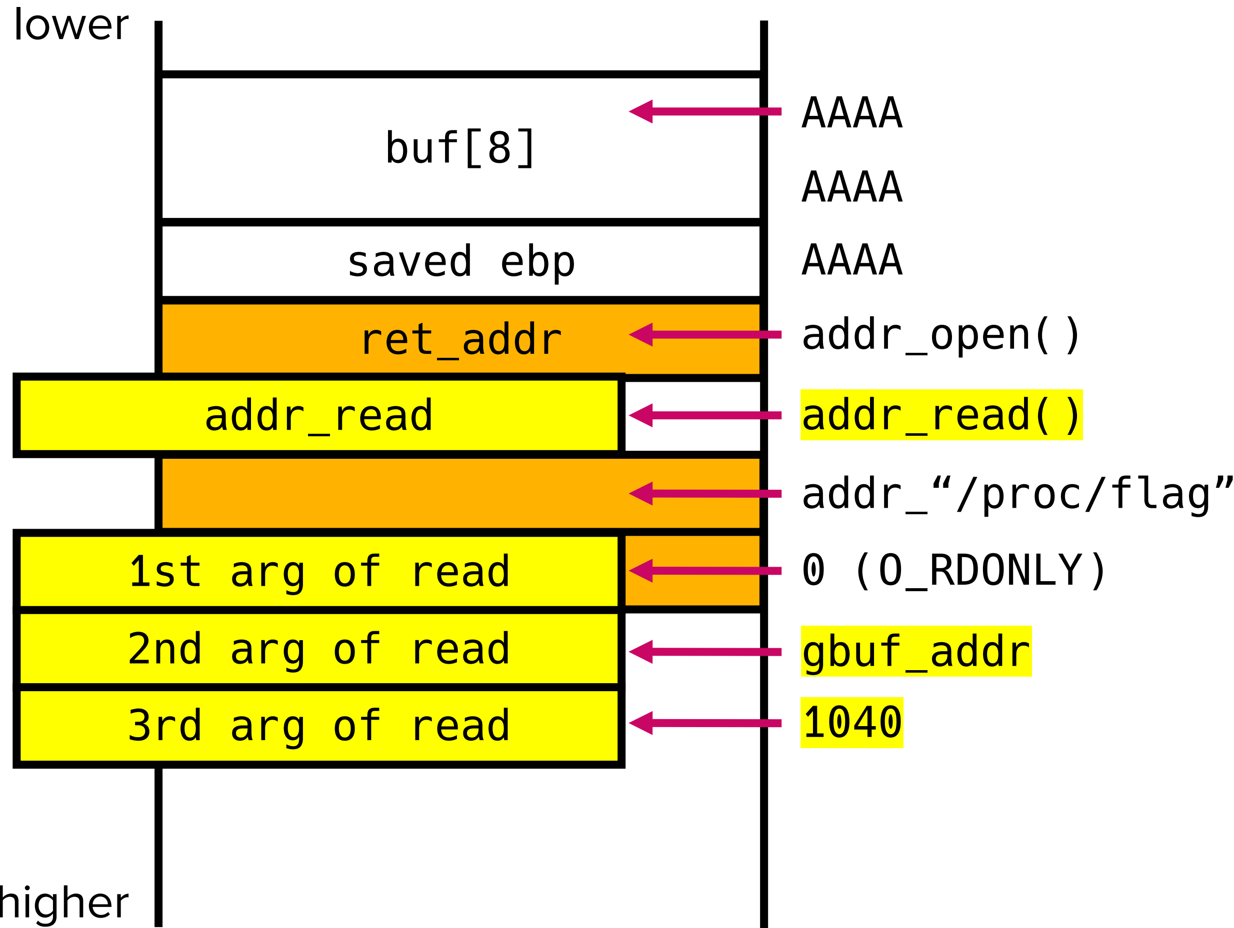
POSTECH

- Naïve chain

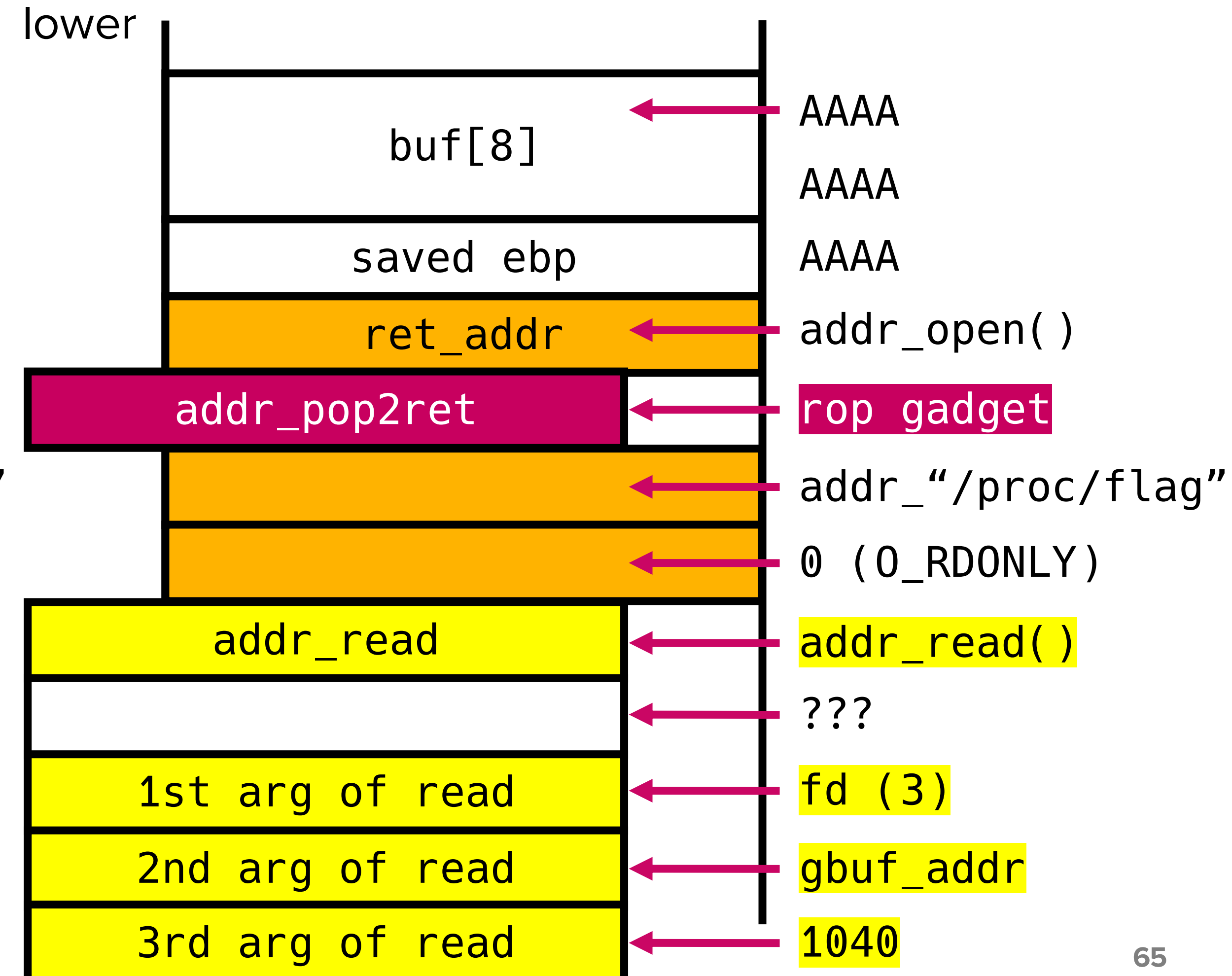


Chaining functions through ROP gadgets

- Naïve chain



- ROP chain

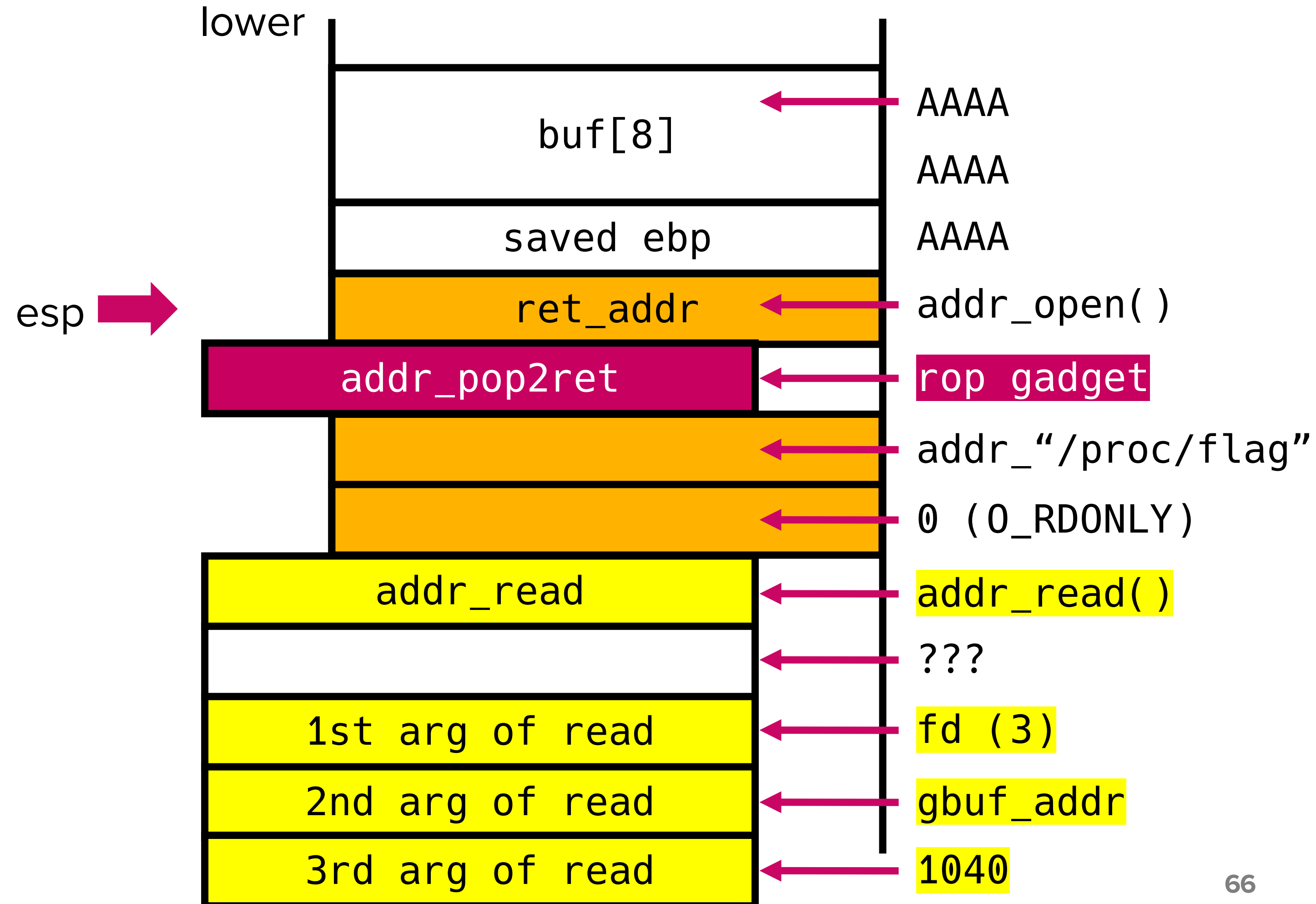


Chaining functions through ROP gadgets

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

eip →

- ROP chain



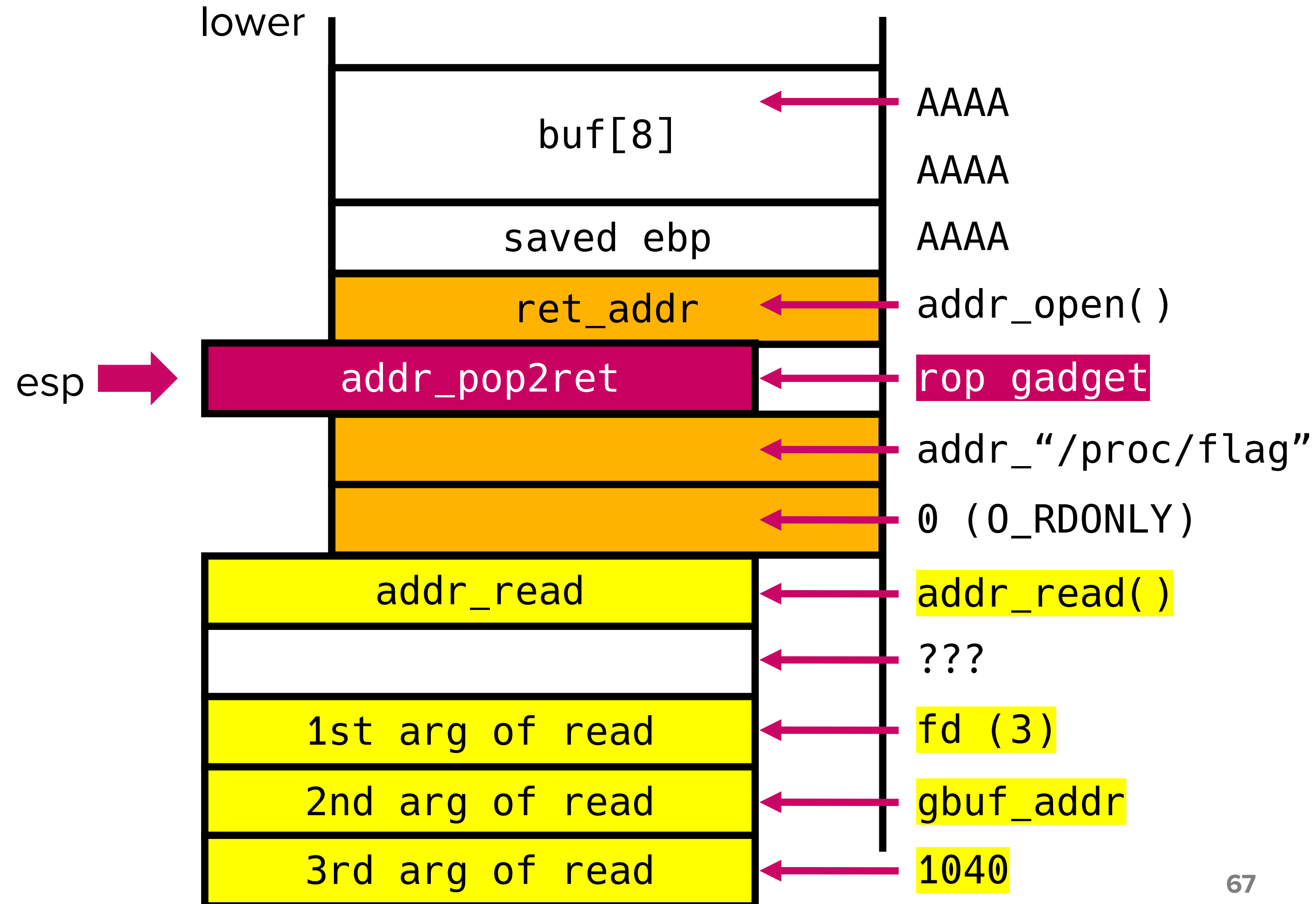
Chaining functions through ROP gadgets

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

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

eip →

- ROP chain



Chaining functions through ROP gadgets

POSTECH

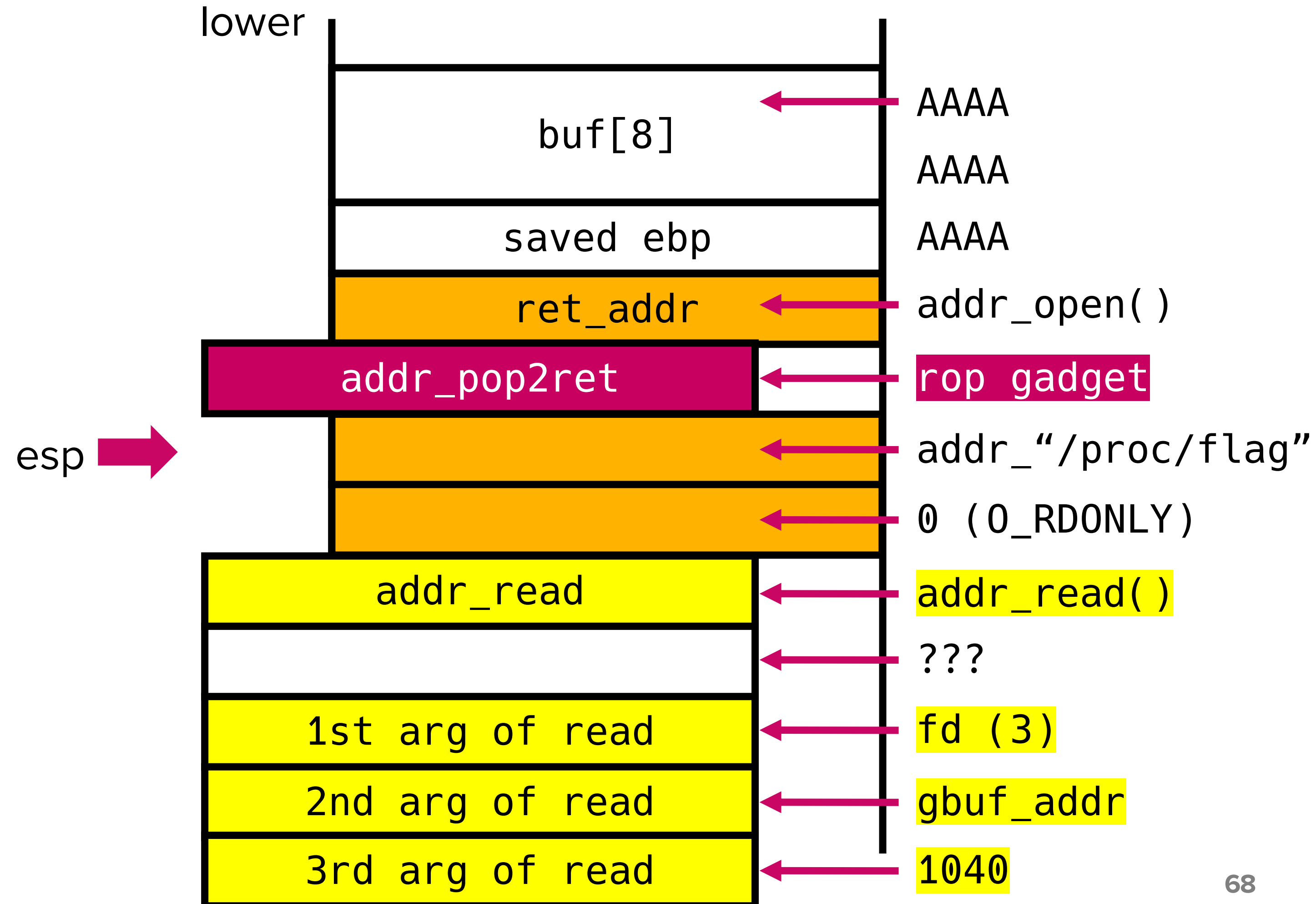
```
<victim_function>:
...
leave
ret
```

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

eip →

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

- ROP chain



Chaining functions through ROP gadgets

POSTECH

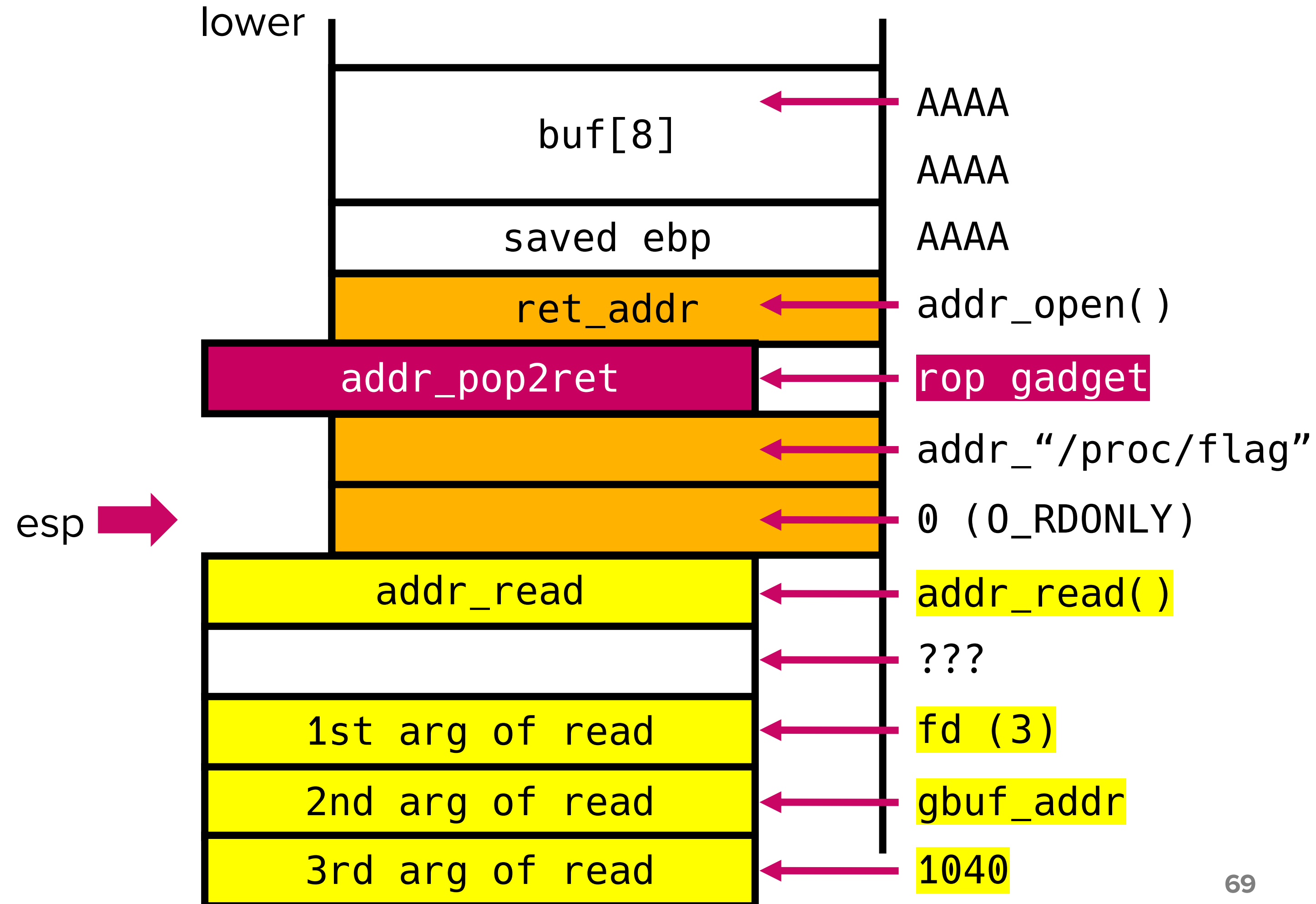
```
<victim_function>:
...
leave
ret
```

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

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

eip

- ROP chain



Chaining functions through ROP gadgets

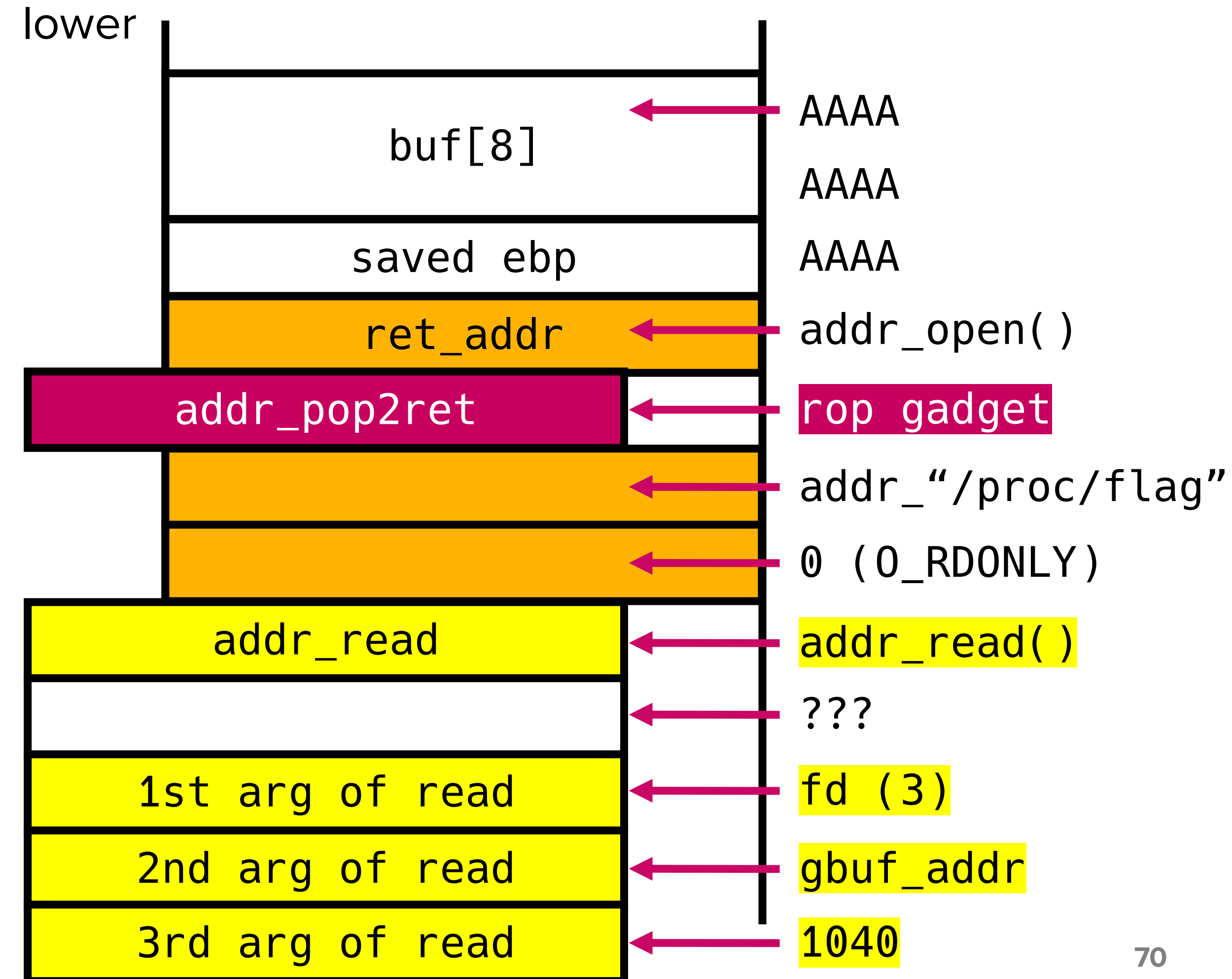
POSTECH

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

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

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

- ROP chain



Adjusted esp points to the addr of the 2nd function!

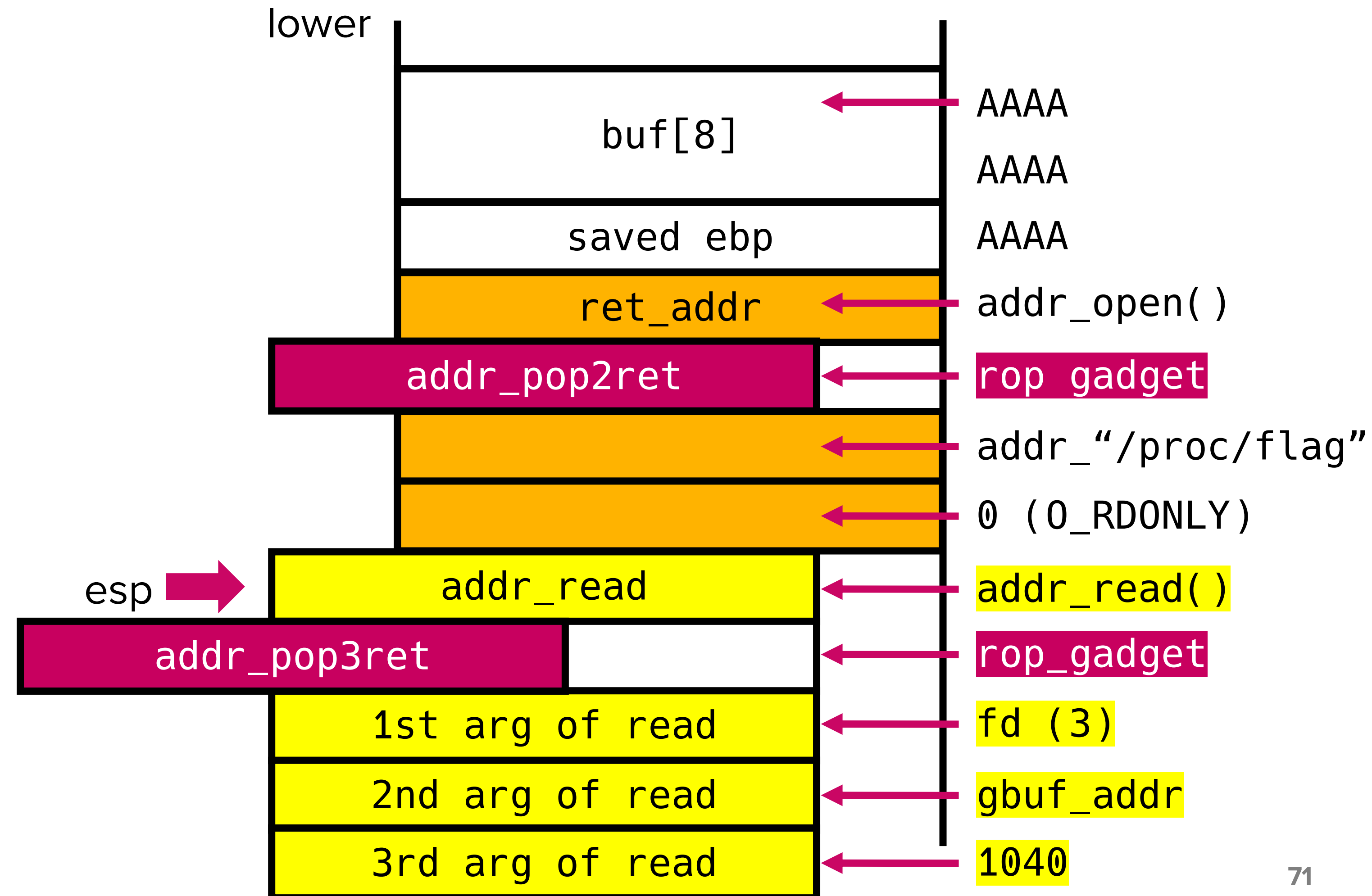
esp

eip

Chaining functions through ROP gadgets

POSTECH

- ROP chain



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

(Three pops move esp down by 12 bytes)

Questions

POSTECH

- Where are 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?