Lec 08: Attacks and Defenses (2)

CSED415: Computer Security

Spring 2024

Seulbae Kim

POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Recap

- NX (No eXecute) is effective at preventing return-to-stack attacks
  - MMU aborts if NX flag is set for fetched instruction's page
  - Stack is flagged as not executable
- Return-to-libc attack bypasses NX protection
  - Basic code reuse attack: return to libc functions useful for exploitations
- Return-Oriented Programming (ROP) generalizes code reuse attacks

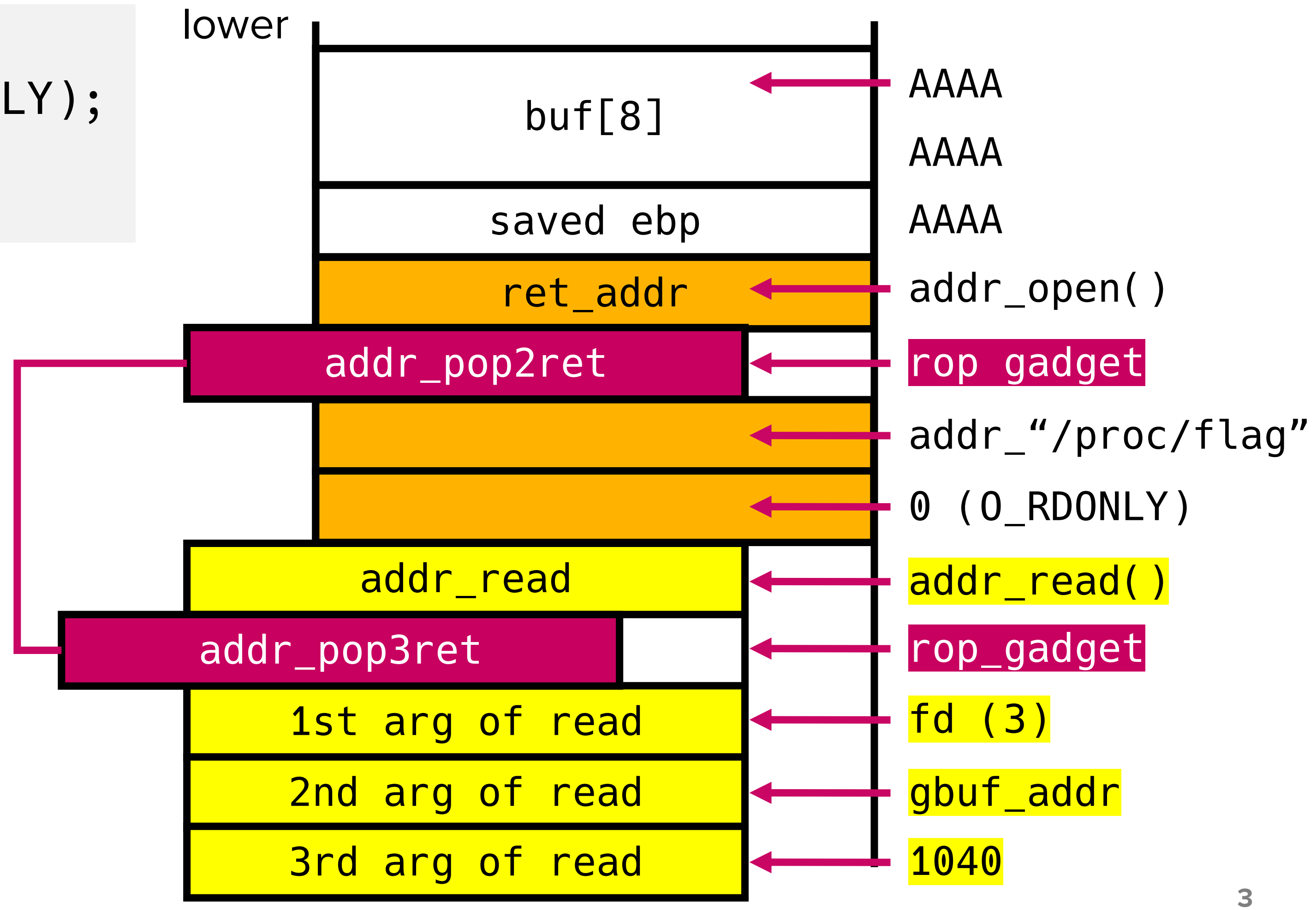
Recap

- Chaining three func calls

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

ROP gadgets are essential for chaining returns to multiple functions

- ROP chain



Attack #1-2: ROP (cont'd)

ROP gadgets

- Small sequences of instructions found within the existing code of a program or its libraries
- Key property
  - ROP gadget ends with an instruction that affects the eip register
    - ret   ; == pop eip
    - call <label>; == push next_eip + jmp <label>
    - jmp <label>;

ROP gadgets in Lab 02's target binary

- Finding gadgets through objdump

```
lab02@csed415:~$ objdump -D ./target -M intel | grep -B3 ret

 8049587:   5b                                   pop    ebx
 8049588:   5e                                   pop    esi
 8049589:   5d                                   pop    ebp
 804958a:   c3                                   ret
--

 80495a2:   5b                                   pop    ebx
 80495a3:   c3                                   ret
--
```

Lucky!

ROP gadgets in Lab 02’s target binary

- Finding gadgets through objdump

```
lab02@csed415:~$ objdump -D ./target -M intel | grep -B3 ret

 8049587:    5b                         pop    ebx
 8049588:    5e                         pop    esi
 8049589:    5d                         pop    ebp
 804958a:    c3                         ret
--

 80495a2:    5b                         pop    ebx
 80495a3:    c3                         ret
--
```

esp += 4
then ret to the address at the top of the stack

ROP gadgets in Lab 02's target binary

- Finding gadgets through objdump

```
lab02@csed415:~$ objdump -D ./target -M intel | grep -B3 ret

 8049587:	5b                   	pop    ebx
 8049588:	5e                   	pop    esi
 8049589:	5d                   	pop    ebp
 804958a:	c3                   	ret
--

 80495a2:	5b                   	pop    ebx
 80495a3:	c3                   	ret
--
```

esp += 8
then ret to the address
at the top of the stack

ROP gadgets in Lab 02's target binary

- Finding gadgets through objdump

```
lab02@csed415:~$ objdump -D ./target -M intel | grep -B3 ret

 8049587:   5b                      pop    ebx
 8049588:   5e                      pop    esi
 8049589:   5d                      pop    ebp
 804958a:   c3                      ret
--

 80495a2:   5b                      pop    ebx
 80495a3:   c3                      ret
--
```

esp += 12
then ret to the address
at the top of the stack

ROP gadgets in Lab 02's target binary

POSTECH

- Finding gadgets through objdump

```
lab02@csed415:~$ objdump -D ./target -M intel | grep -B3 ret

 8049587:   5b                                      pop    ebx
 8049588:   5e                                      pop    esi
 8049589:   5d                                      pop    ebp
 804958a:   c3                                      ret
--

 80495a2:   5b                                      pop    ebx
 80495a3:   c3                                      ret
--
```

esp += 12
then ret to the address at the top of the stack

Q) Given pop3ret, can we do esp += 16 or more?

ROP gadgets in Lab 02's target binary

- Finding gadgets through objdump

```
lab02@csed415:~$ objdump -D ./target -M intel | grep -B3 ret

 8049587:     5b                              pop     ebx
```

Can we find whether more gadgets exist?

```
 80495a2:     5b                              pop     ebx
 80495a3:     c3                              ret
--
```

Background: Variable length instructions

- Length of encoded instructions vary
  - e.g., x86

Asm		**Binary**
`push ebp`	→	`0x55`
`mov ebp, esp`	→	`0x89 0xe5`
`add esp, 0x10`	→	`0x83 0xc4 0x10`
`endbr32`	→	`0xf3 0x0f 0x1e 0xfb`
`call (near call)`	→	`0xe8 0xca 0xfd 0xff 0xff`
`...`		`...`

Q) Advantage?

Smaller binary size
- Assign small length to frequent instructions

Q) Disadvantage?
- Following slides

Disassembling x86

x86 instructions are variable-length!

```
08049266 <setup_rules>:
 8049266:       55                      push   ebp
 8049267:       89 e5                   mov    ebp,esp
 8049269:       53                      push   ebx
 804926a:       83 ec 14                sub    esp,0x14
 804926d:       e8 2e ff ff ff          call   80491a0 <__x86.get_pc_thunk.bx>
 8049272:       81 c3 8e 2d 00 00       add    ebx,0x2d8e
 8049278:       c7 45 e8 73 65 63 63    mov    DWORD PTR [ebp-0x18],0x63636573
 804927f:       c7 45 ec 6f 6d 70 00    mov    DWORD PTR [ebp-0x14],0x706d6f
 8049286:       8d 45 e8                lea    eax,[ebp-0x18]
 8049289:       50                      push   eax
 804928a:       8d 45 f0                lea    eax,[ebp-0x10]
 804928d:       50                      push   eax
 804928e:       e8 2d fe ff ff          call   80490c0 <strcpy@plt>
 8049293:       83 c4 08                add    esp,0x8
 8049296:       8d 45 f0                lea    eax,[ebp-0x10]
```

Disassembling x86

```
c7 45 ec 6f 6d 70 00 8d 45 e8 ...
```

↑

```
mov    DWORD PTR [ebp-0x18],0x63636573
mov    DWORD PTR [ebp-0x14],0x706d6f
...
```

Q) What if we disassemble from the second byte (0x45)?

Disassembling x86

```
c7 45 ec 6f 6d 70 00 8d 45 e8 ...
```

```
inc     ebp
call    0x6b67f7f2
...
```

Completely different results, but instructions are still valid

Disassembling x86

```
c7 45 ec 6f 6d 70 00 8d 45 e8 ...
```

```
jae     0x80492e2 <setup_rules+124>
arpl    WORD PTR [ebx-0x39],sp
...
```

Completely different results, but instructions are still valid

Var-len instructions can be disassembled from any addr

- Is it legal to use (e.g., jump to) the middle of variable-length instructions?
  - Perfectly legal in terms of program execution
  - “The program doesn’t care about the semantics of execution”- Lec 07

- Security problem:
  - We can find many **unintended** ret instructions

Unintended ret instructions

- Aligned instructions (i.e., what objdump sees):

```
e8 05 ff ff ff                  call   8048330
81 c3 59 12 00 00               add    ebx,0x1259
```

- Disassembled from the 2nd byte:

```
05 ff ff ff 81                  add    eax,0x81ffffff
c3                              ret
```

Unintended ret instruction appears

ropper: A tool to find ROP gadgets

```
lab02@csed415:~$ ropper -f ./target --search "pop %; ret"
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop %; ret

[INFO] File: ./target
0x080491f1: pop eax; rol byte ptr [eax + ecx], 0x2d; pop eax; rol byte ptr
[eax + ecx], 0x89; ret 0xe8c1;
0x080491f6: pop eax; rol byte ptr [eax + ecx], 0x89; ret 0xe8c1;
0x0804924a: pop eax; rol byte ptr [eax + ecx], 1; leave; ret;
0x0804941d: pop ebp; cld; leave; ret;
0x08049589: pop ebp; ret;
0x08049587: pop ebx; pop esi; pop ebp; ret;
0x08049022: pop ebx; ret;
0x08049588: pop esi; pop ebp; ret;
```

Defense #2: ASLR

Address Space Layout Randomization

- Pre-ASLR world
  - An executable was loaded to the same virtual address space
  - All sections consistently mapped

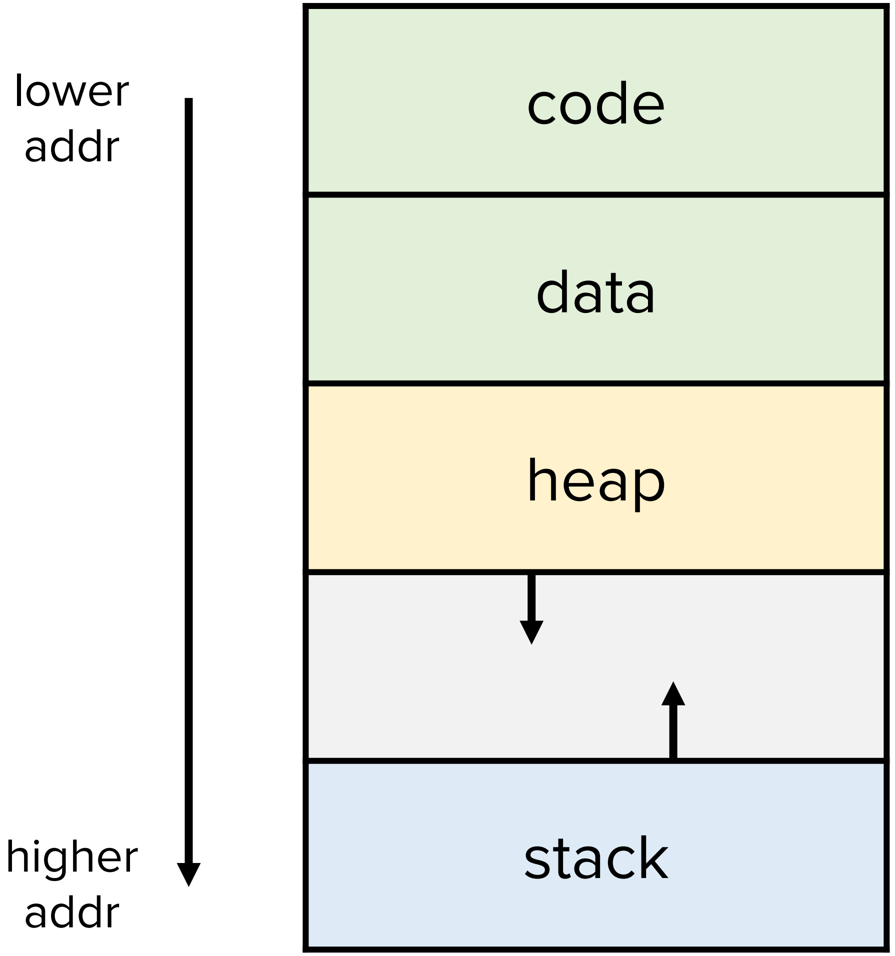
- Naturally, all addresses of code and data were “invariants”
  - All function addresses, data (e.g., string) addresses, and ROP gadget addresses were known
  - Attackers could easily land ret-to-libc or ROP attacks

Idea

- Can we put each segment of memory in a different location each time the program is loaded?

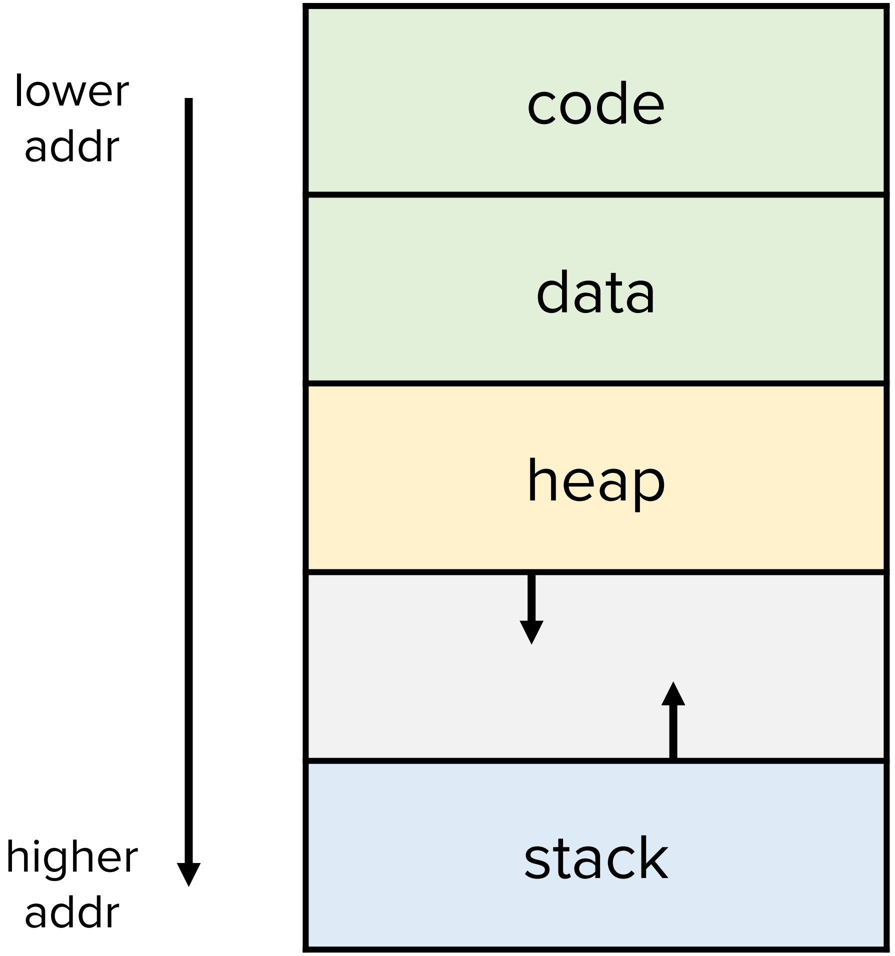
Recall: x86 memory layout

- In theory: packed

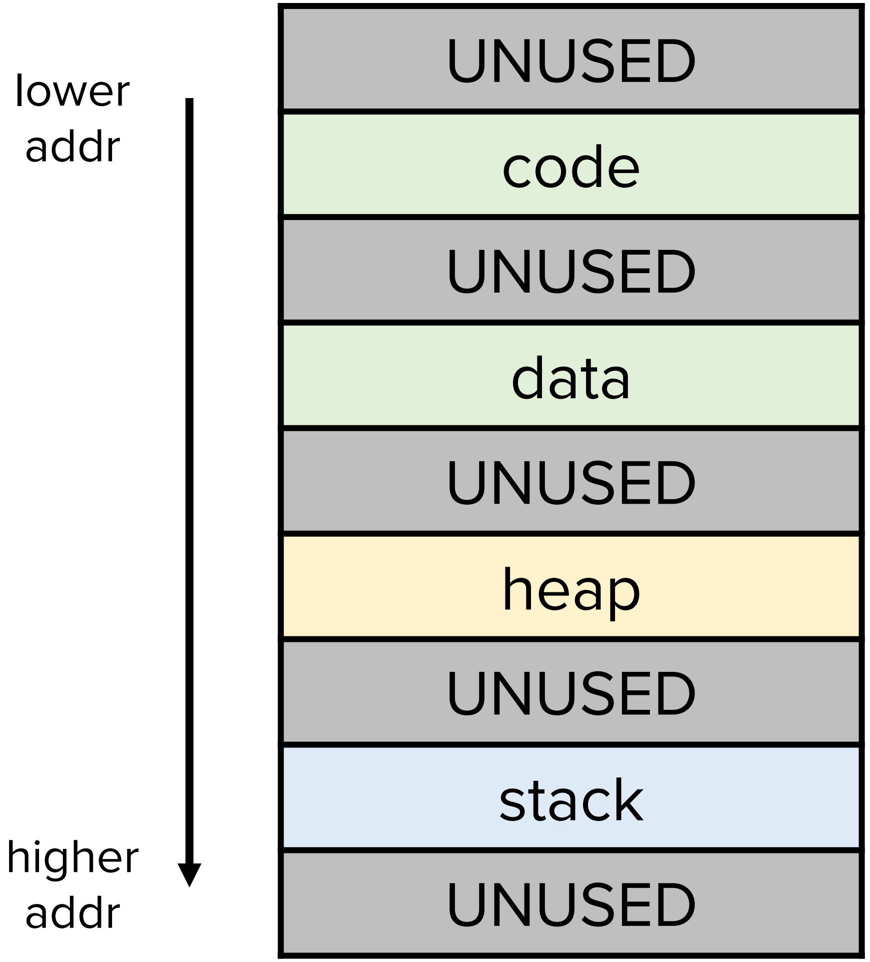


Recall: x86 memory layout

- In theory: packed

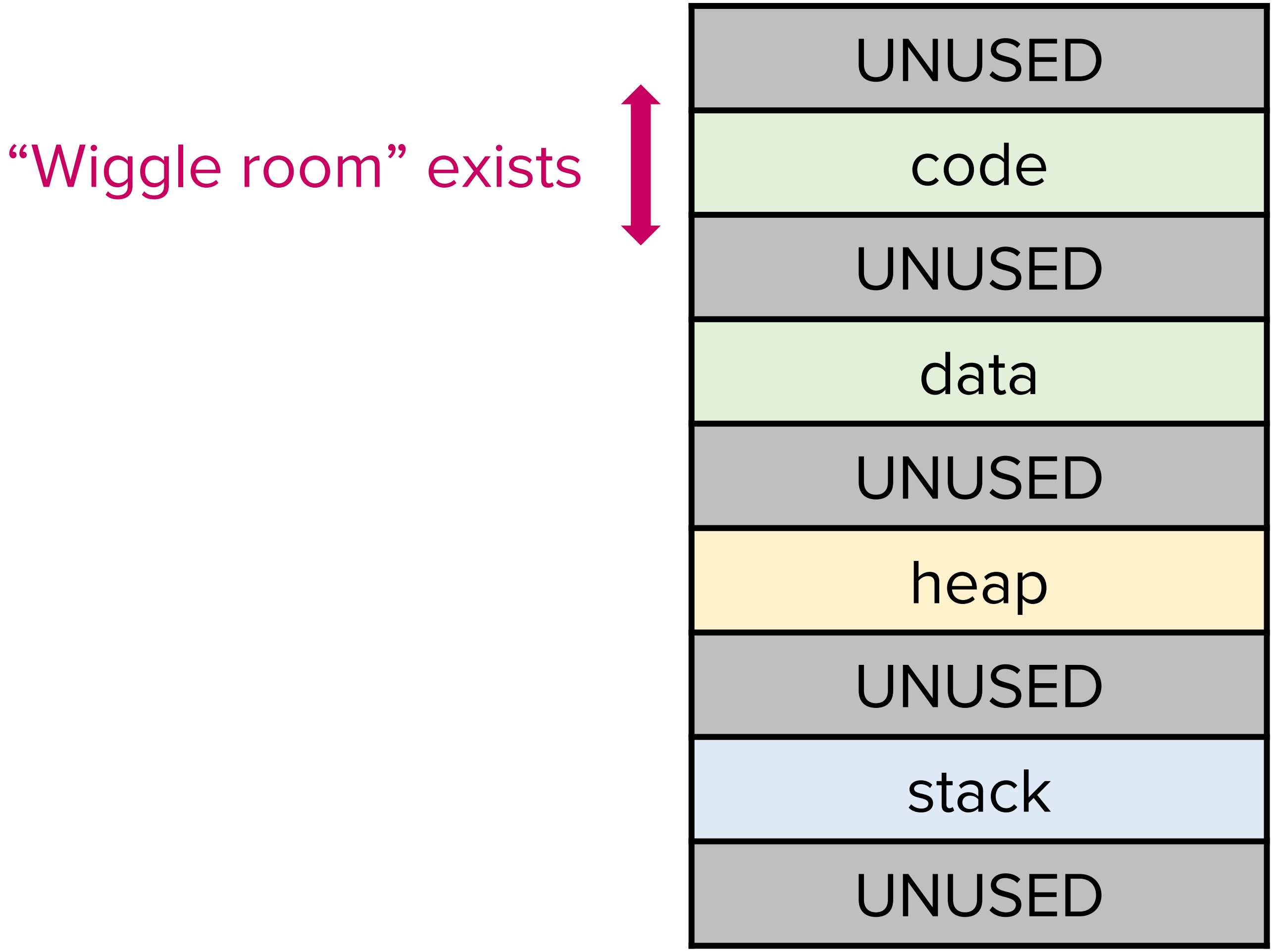


- In practice: mostly empty

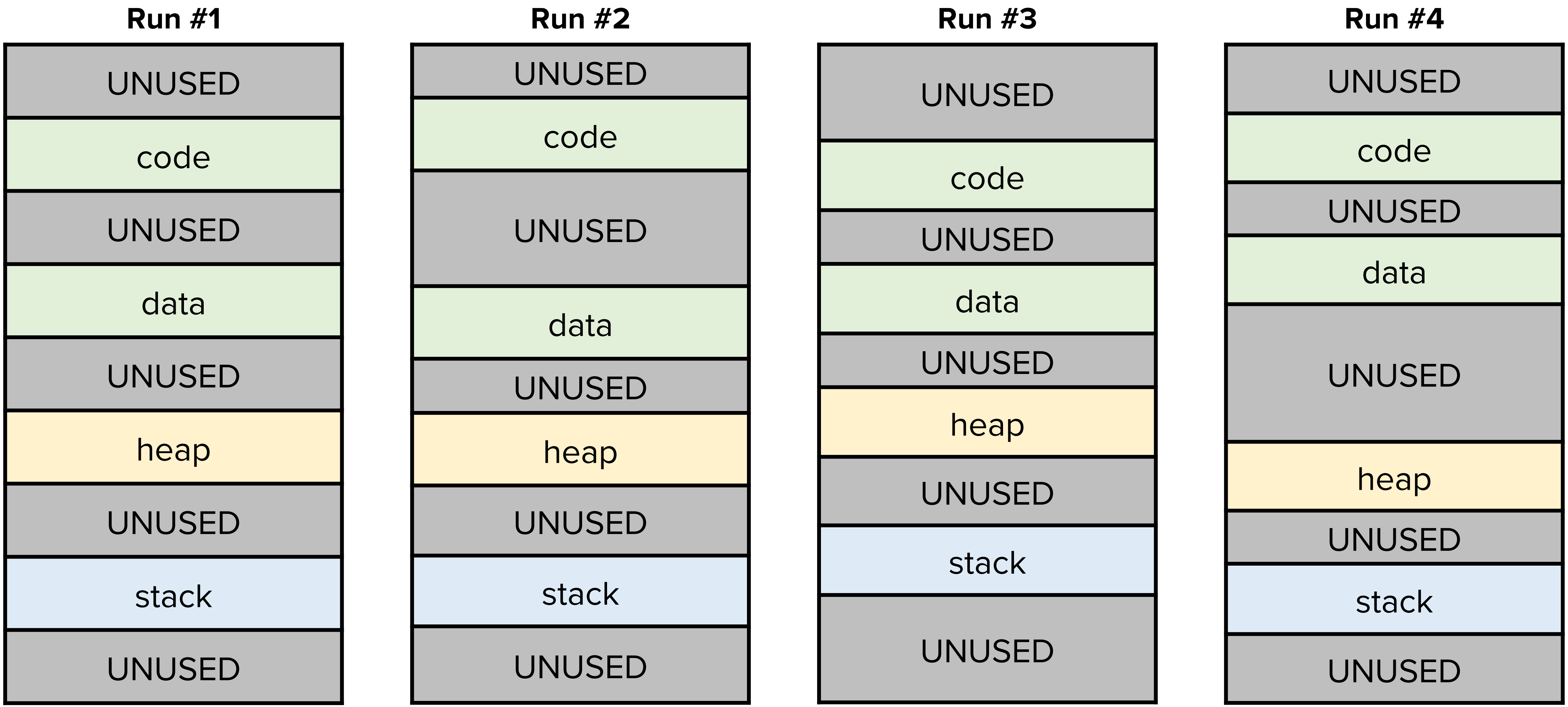


Recall: x86 memory layout

- In practice: mostly empty



Idea: Load each segment at different address



ASLR can shuffle all four segments

- Randomized stack: Can't put shellcode on stack without knowing the address of the stack
- Randomized heap: Can't put shellcode on the heap
- Randomized code: Can't construct a ROP chain or ret-to-libc without knowing the address of code
- Randomized data: Can't reuse existing data

Checking ASLR – try it yourself

```
#include <stdio.h>

int main(void) {
  int x = 0xdeadbeef;
  return printf("%08x\n", &x);
}
```

Code to print stack address

```
$ gcc -m32 aslr.c -o aslr
```

Compilation

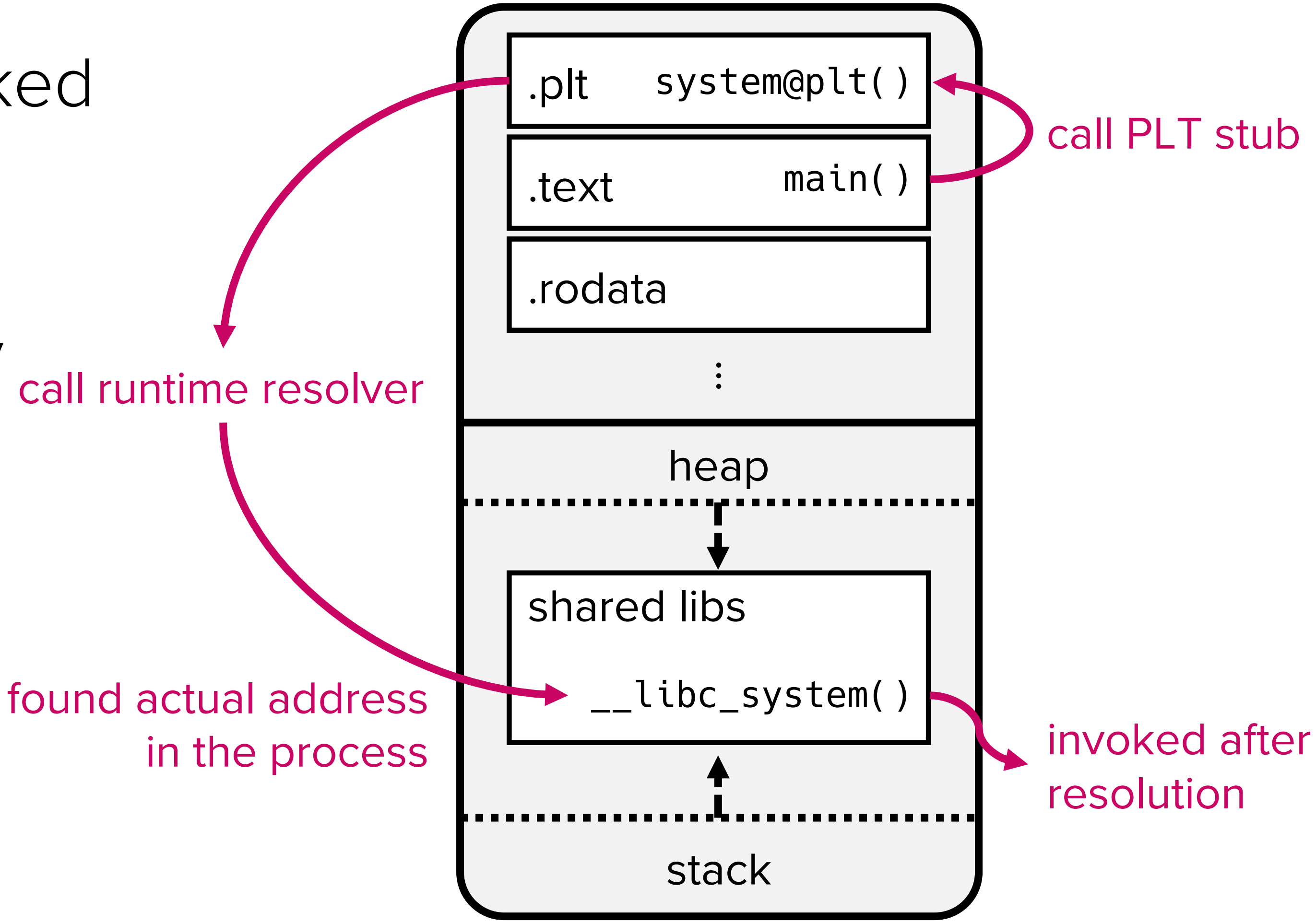
```
$ ./aslr
ffd45c68
$ ./aslr
ffed76c8
$ ./aslr
ffc832c8
```

Stack address randomized for each run

Is ASLR efficient?

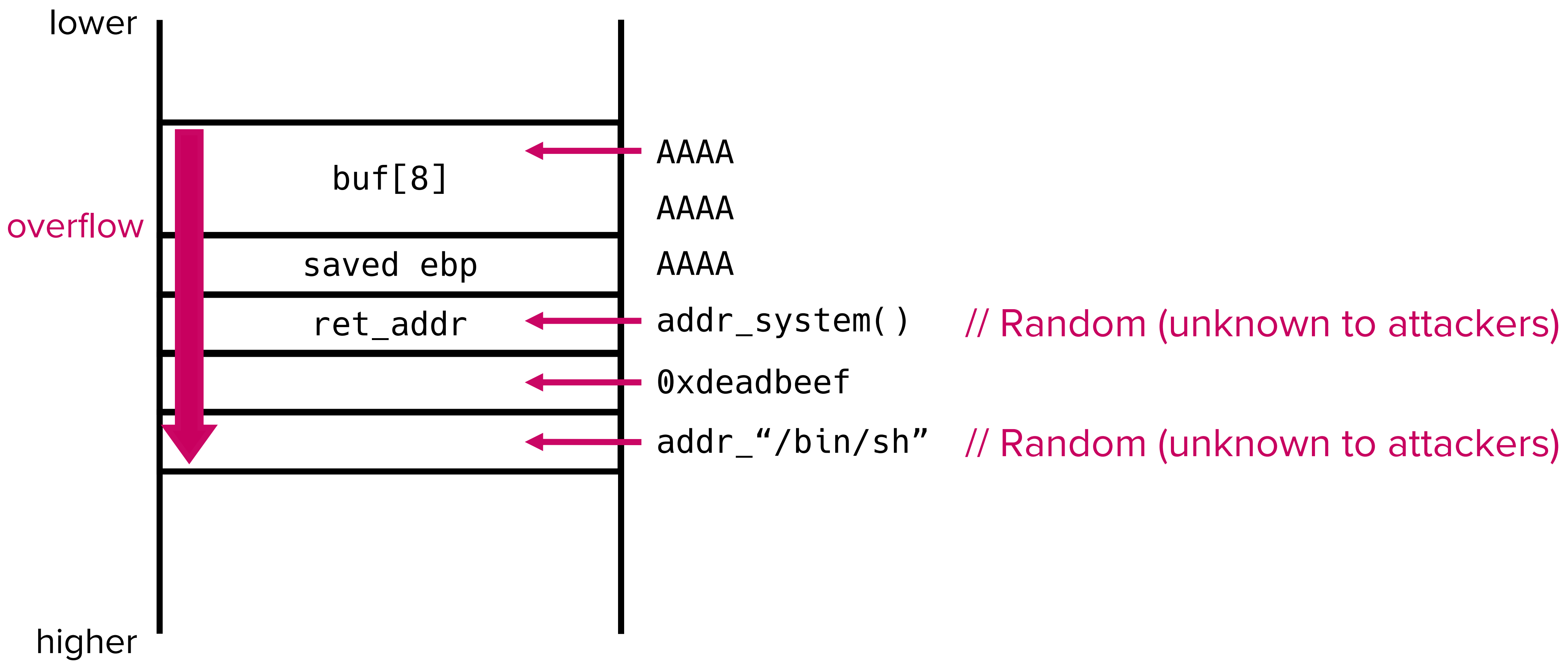
- Recall
  - Programs are dynamically linked at runtime
  - Dynamically linked programs have to do relocation anyway
    → ASLR causes no additional overhead

Dynamically linked process



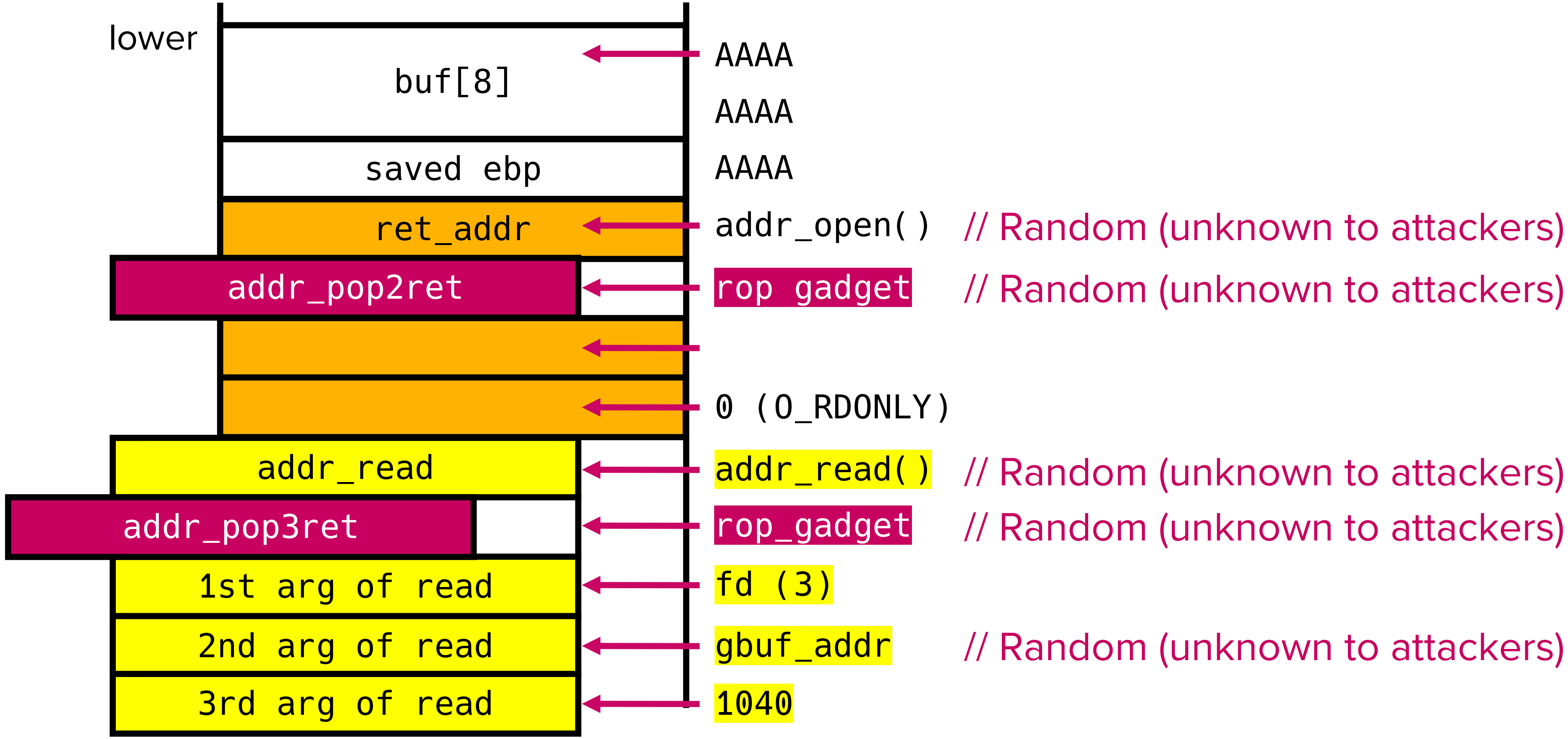
Mitigating return-to-libc attacks

- Cannot return to libc without knowing function and data addresses



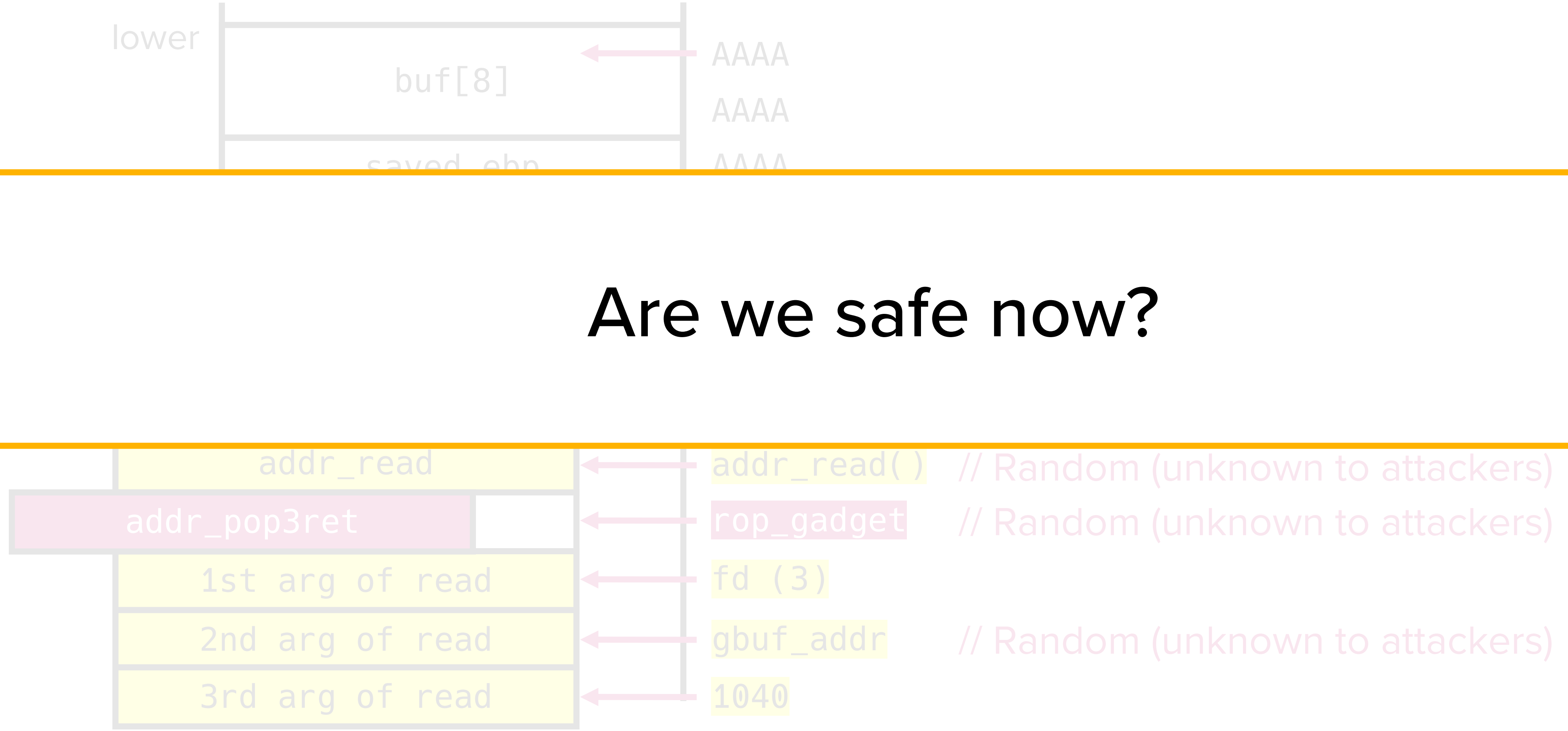
Mitigating ROP

- Similarly, cannot do ROP without knowing addresses



Mitigating ROP

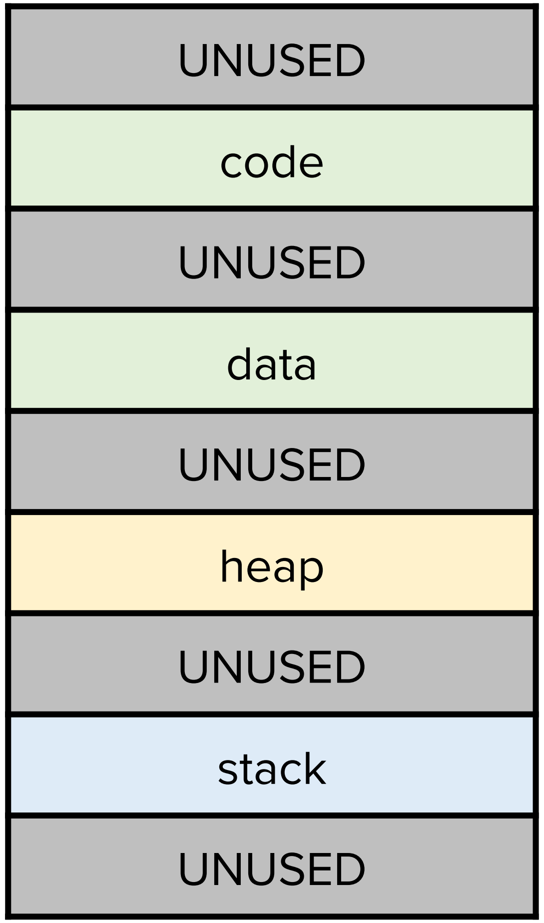
- Similarly, cannot do ROP without knowing addresses

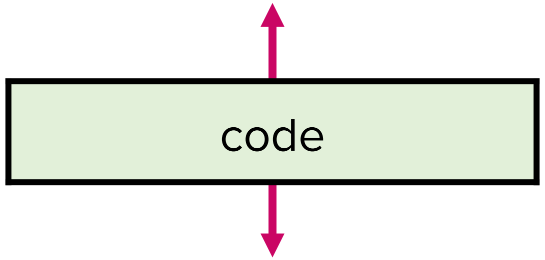


Are we safe now?

Subverting ASLR (1)

- ASLR only randomizes the **base address** of segments



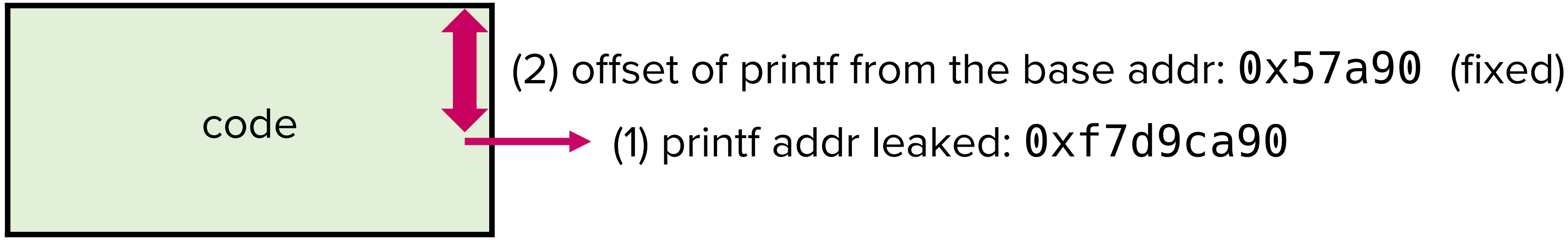



```
<main>:
base + 11ad: lea  ecx,[esp+0x4]
base + 11b1: and  esp,0xfffffff0
base + 11b4: push DWORD PTR [ecx-0x4]
base + 11b7: push ebp
base + 11b8: mov  ebp,esp
...
```

Relative addresses are fixed!

Subverting ASLR (1)

- If the address of a pointer within a segment is leaked, the base address can be inferred



- Then, the absolute address of all other pointers can be inferred

Subverting ASLR (1)

- Verification with Lab 02's target binary

```
lab02@csed415:~$ gdb ./target
pwndbg> b main
pwndbg> r
pwndbg> print system
$1 = {int (const char *)} 0xf7d8d170 <__libc_system>
pwndbg> print printf
$2 = {int (const char *, ...)} 0xf7d9ca90 <__printf>
pwndbg> p/x $2-$1
$3 = 0xf920
```

```
lab02@csed415:~$ gdb ./target
pwndbg> b main
pwndbg> r
pwndbg> print system
$1 = {int (const char *)} 0xf7d5d170 <__libc_system>
pwndbg> print printf
$2 = {int (const char *, ...)} 0xf7d6ca90 <__printf>
pwndbg> p/x $2-$1
$3 = 0xf920
```

Subverting ASLR (1)

- Verification with Lab 02's target binary (continued)

```
lab02@csed415:~$ ldd ./target
        linux-gate.so.1 (0xf7f4b000)
        libseccomp.so.2 => /lib/i386-linux-gnu/libseccomp.so.2 (0xf7f1a000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7ce5000)
        /lib/ld-linux.so.2 (0xf7f4d000)

lab02@csed415:~$ objdump -t /lib/i386-linux-gnu/libc.so.6 | grep __libc_system
00048170 g     F .text 0000003f __libc_system

lab02@csed415:~$ objdump -t /lib/i386-linux-gnu/libc.so.6 | grep "__printf\b"
00057a90 l     F .text 0000002d __printf

lab02@csed415:~$ python3 -c "print(hex(0x57a90-0x48170))"
0xf920
```

offset of system() from libc's base addr: 0x48170

offset of printf() from libc's base addr: 0x57a90

diff: 0xf920 (confirmed)

Subverting ASLR (1)

- Verification with Lab 02's target binary (continued)

```
lab02@csed415:~$ ldd ./target
        linux-gate.so.1 (0xf7f4b000)
        libseccomp.so.2 => /lib/i386-linux-gnu/libseccomp.so.2 (0xf7f1a000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7ce5000)
```

Q) Why does ASLR only randomize the base address of segments?

```
lab02@csed415:~$ python3 -c "print(hex(0x57a90-0x48170))"
0xf920
```

offset of system() from libc's base addr: 0x48170
offset of printf() from libc's base addr: 0x57a90

diff: 0xf920 (confirmed)

Subverting ASLR (2)

- Entropy (randomness) of ASLR on x86 Linux is small
  - x86 only randomizes 16 bits of base address (code, heap segments)
  - 2^16 addresses are possible for a function
    - Feasibly brute-forced
  - Let’s try:

```
$ for i in {1..40}; do echo "X" | ./target | grep system; done
```

Defense #3: Stack Canary

Canary

img: Rio Wiki

Canary in coal mines (Late 1800's ~ 1986)

- Canaries are sensitive to toxic gas (e.g., CO, CH4)
- Coal miners brought canaries into coal mines
- Canaries die if toxic gases build up
- Miners bail out if a canary dies
  - Sacrificed canaries for miners' lives

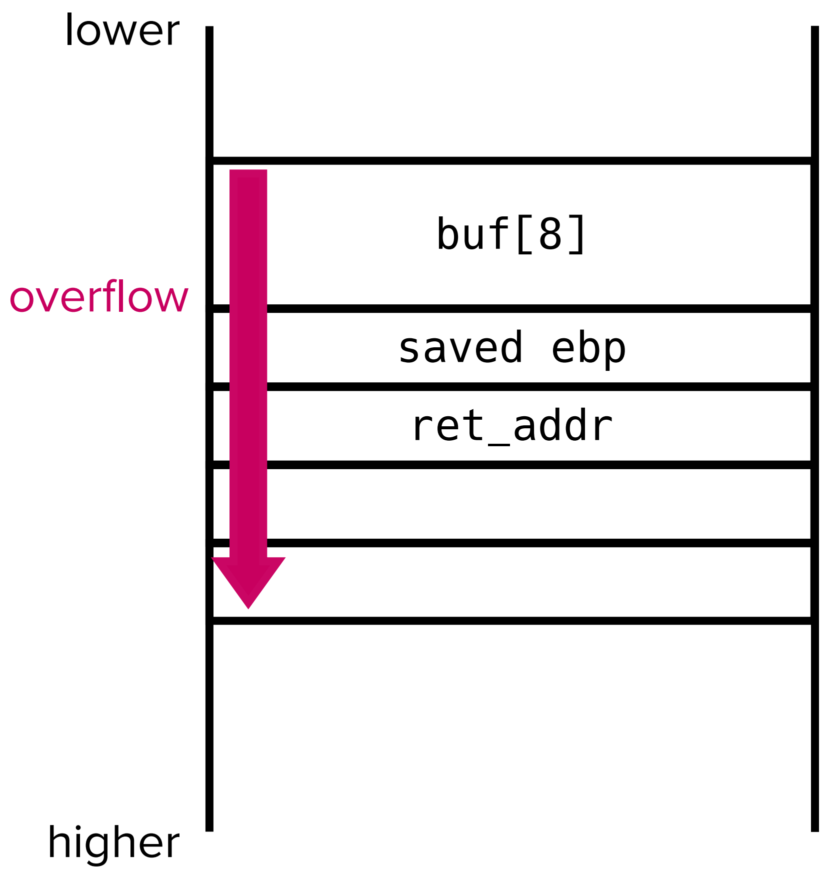
img: Times Higher Education

Canaries for binaries

- Idea
  - Add a sacrificial value on the stack and check if it has been changed

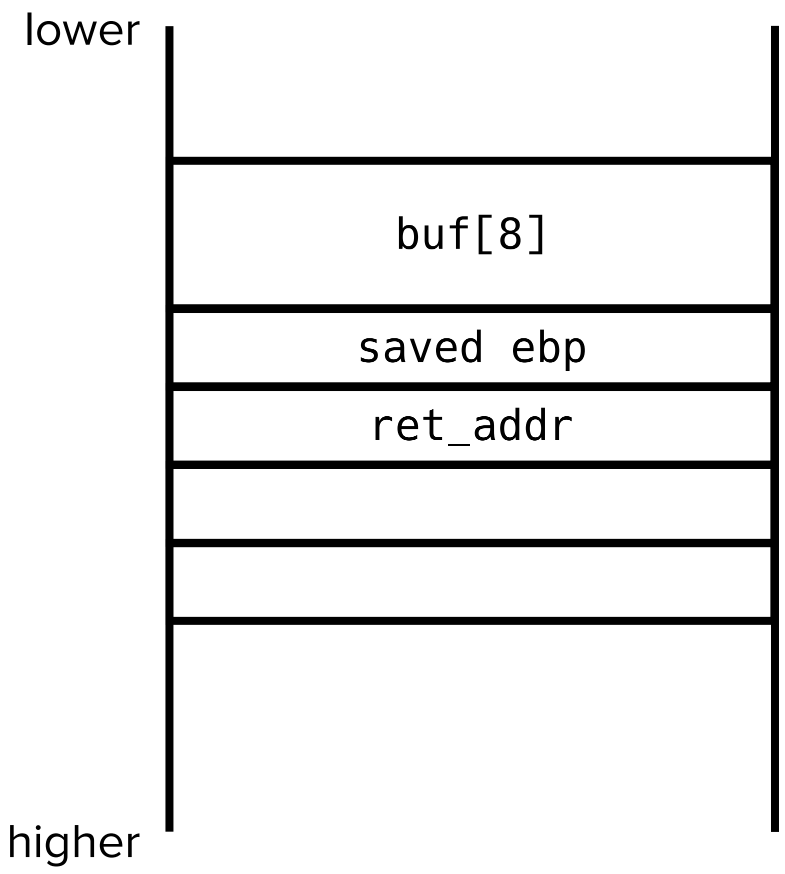
Stack diagram

- Without a canary

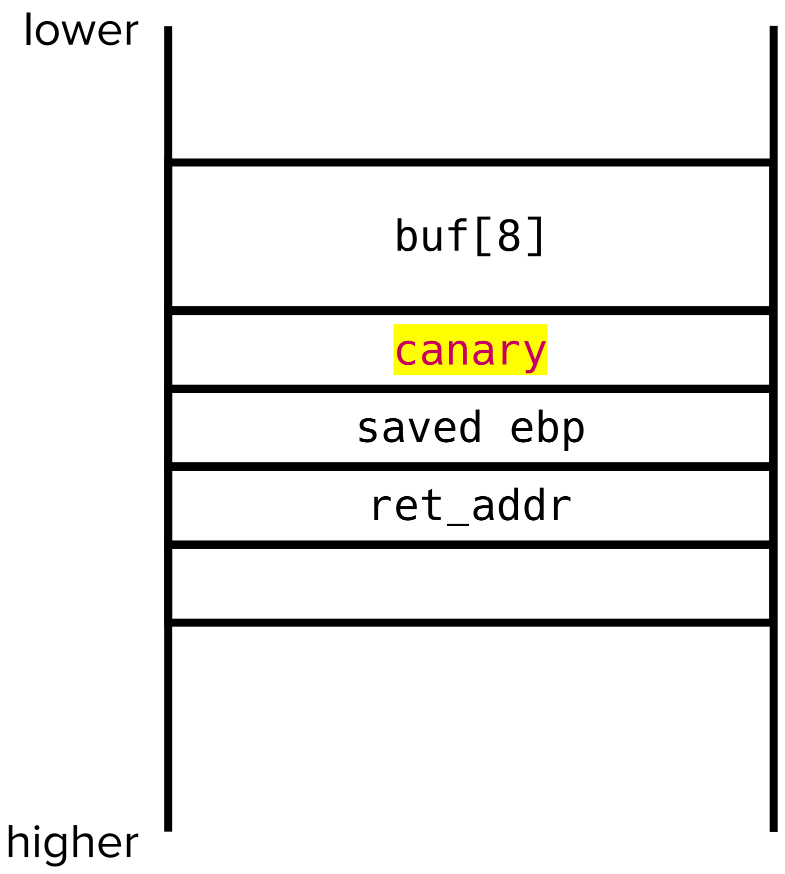


Stack diagram

- Without a canary



- With a canary

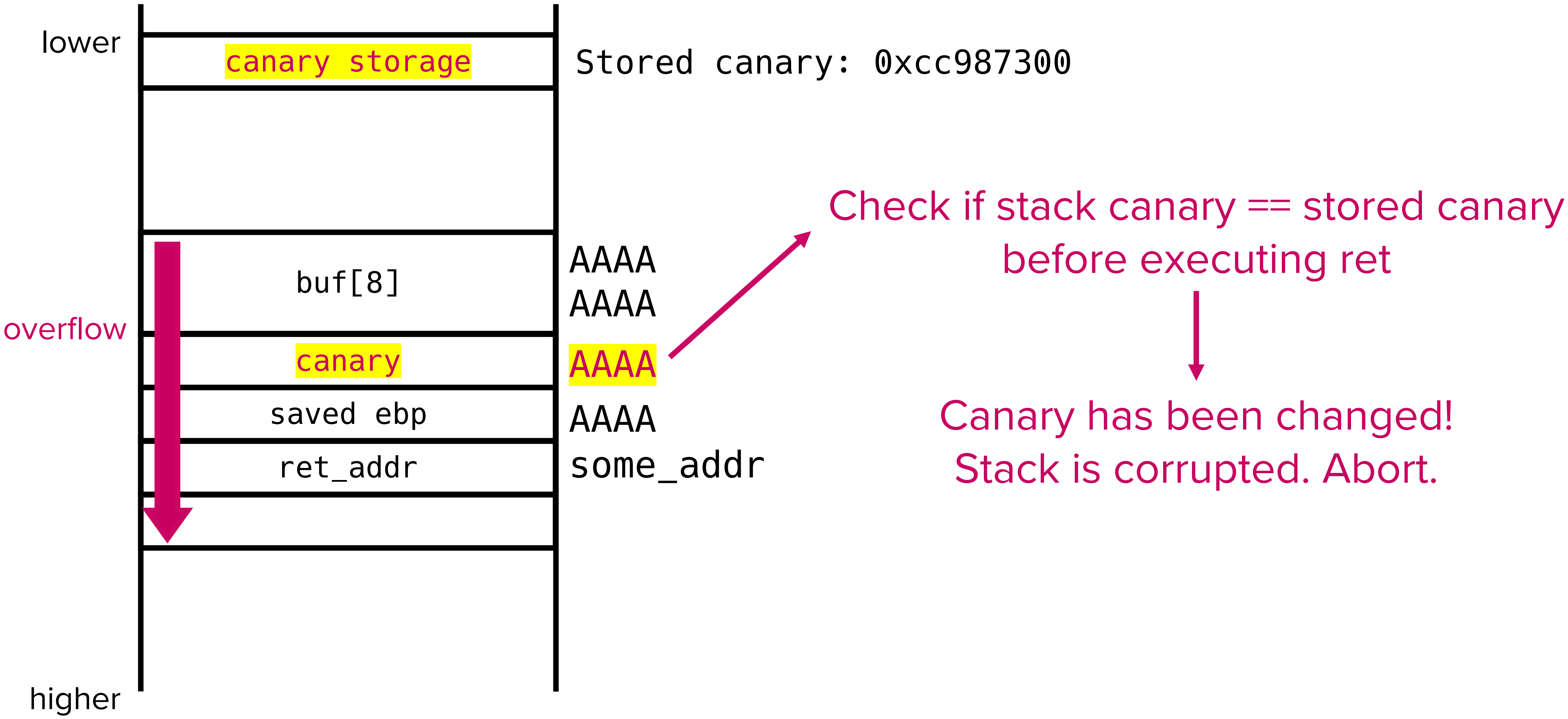


Canary workflow

- Binary is executed
- Generate a random secret value and store it in the canary storage
- In the function prologue, place the canary right before the saved ebp and return address
- In the function epilogue, check the value on the stack and compare it aginst the value in canary storage

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

Canary workflow



Stack canary in practice

```
<vuln>:
push   ebp
mov    ebp,esp
push   ebx
sub    esp,0x14
call   0x80491df
add    eax,0x2e7e
sub    esp,0x8
push   DWORD PTR [ebp+0x8]
lea    edx,[ebp-0x10]
push   edx
mov    ebx,eax
call   0x8049050 <strcpy@plt>
add    esp,0x10
mov    eax,0x0
mov    ebx,DWORD PTR [ebp-0x4]
leave
ret
```

- Left: $ gcc -fno-stack-protector
- Right: $ gcc -fstack-protector

```
<vuln>:
push   ebp
mov    ebp,esp
push   ebx
sub    esp,0x10
call   0x80491f8
add    eax,0x2e6e
mov    edx,DWORD PTR [ebp+0x8]
mov    DWORD PTR [ebp-0x14],edx
mov    edx,DWORD PTR gs:0x14
mov    DWORD PTR [ebp-0x8],edx
xor    edx,edx
push   DWORD PTR [ebp-0x14]
lea    edx,[ebp-0x10]
push   edx
mov    ebx,eax
call   0x8049060 <strcpy@plt>
add    esp,0x8
mov    eax,0x0
mov    edx,DWORD PTR [ebp-0x8]
sub    edx,DWORD PTR gs:0x14
je     0x80491d0 <vuln+74>
call   0x8049200 <__stack_chk_fail_local>
mov    ebx,DWORD PTR [ebp-0x4]
leave
ret
```

Stack canary in practice

```
<vuln>:
push   ebp
mov    ebp,esp
push   ebx
sub    esp,0x14
call   0x80491df
add    eax,0x2e7e
sub    esp,0x8
push   DWORD PTR [ebp+0x8]
lea    edx,[ebp-0x10]
push   edx
mov    ebx,eax
call   0x8049050 <strcpy@plt>
add    esp,0x10
mov    eax,0x0
mov    ebx,DWORD PTR [ebp-0x4]
leave
ret
```

```
<vuln>:
push   ebp
mov    ebp,esp
push   ebx
sub    esp,0x10
call   0x80491f8
add    eax,0x2e6e
mov    edx,DWORD PTR [ebp+0x8]
mov    DWORD PTR [ebp-0x14],edx
mov    edx,DWORD PTR gs:0x14
mov    DWORD PTR [ebp-0x8],edx
xor    edx,edx
push   DWORD PTR [ebp-0x14]
lea    edx,[ebp-0x10]
push   edx
mov    ebx,eax
call   0x8049060 <strcpy@plt>
add    esp,0x8
mov    eax,0x0
mov    edx,DWORD PTR [ebp-0x8]
sub    edx,DWORD PTR gs:0x14
je     0x80491d0 <vuln+74>
call   0x8049200 <__stack_chk_fail_local>
mov    ebx,DWORD PTR [ebp-0x4]
leave
ret
```

Stack canary in practice

(canary storage)

Fetches a canary value from **gs:0x14**
Stores the value at ebp-8
(between local vars and return addr)

```
<vuln>:
push   ebp
mov    ebp,esp
push   ebx
sub    esp,0x10
call   0x80491f8
add    eax,0x2e6e
mov    edx,DWORD PTR [ebp+0x8]
mov    DWORD PTR [ebp-0x14],edx
mov    edx,DWORD PTR gs:0x14
mov    DWORD PTR [ebp-0x8],edx
xor    edx,edx
push   DWORD PTR [ebp-0x14]
lea    edx,[ebp-0x10]
push   edx
mov    ebx,eax
call   0x8049060 <strcpy@plt>
add    esp,0x8
mov    eax,0x0
mov    edx,DWORD PTR [ebp-0x8]
sub    edx,DWORD PTR gs:0x14
je     0x80491d0 <vuln+74>
call   0x8049200 <__stack_chk_fail_local>
mov    ebx,DWORD PTR [ebp-0x4]
leave
ret
```

Stack canary in practice

(canary storage)

Fetches a canary value from gs:0x14
Stores the value at ebp-8
(between local vars and return addr)

Compares the stack canary at ebp-8
with canary storage at gs:0x14

```
<vuln>:
push   ebp
mov    ebp,esp
push   ebx
sub    esp,0x10
call   0x80491f8
add    eax,0x2e6e
mov    edx,DWORD PTR [ebp+0x8]
mov    DWORD PTR [ebp-0x14],edx
mov    edx,DWORD PTR gs:0x14
mov    DWORD PTR [ebp-0x8],edx
xor    edx,edx
push   DWORD PTR [ebp-0x14]
lea    edx,[ebp-0x10]
push   edx
mov    ebx,eax
call   0x8049060 <strcpy@plt>
add    esp,0x8
mov    eax,0x0
mov    edx,DWORD PTR [ebp-0x8]
sub    edx,DWORD PTR gs:0x14
je     0x80491d0 <vuln+74>
call   0x8049200 <__stack_chk_fail_local>
mov    ebx,DWORD PTR [ebp-0x4]
leave
ret
```

Stack canary in practice

(canary storage)

Fetches a canary value from gs:0x14
Stores the value at ebp-8
(between local vars and return addr)

Compares the stack canary at ebp-8
with canary storage at gs:0x14
Calls __stack_chk_fail if they are different

```
<vuln>:
push   ebp
mov    ebp,esp
push   ebx
sub    esp,0x10
call   0x80491f8
add    eax,0x2e6e
mov    edx,DWORD PTR [ebp+0x8]
mov    DWORD PTR [ebp-0x14],edx
mov    edx,DWORD PTR gs:0x14
mov    DWORD PTR [ebp-0x8],edx
xor    edx,edx
push   DWORD PTR [ebp-0x14]
lea    edx,[ebp-0x10]
push   edx
mov    ebx,eax
call   0x8049060 <strcpy@plt>
add    esp,0x8
mov    eax,0x0
mov    edx,DWORD PTR [ebp-0x8]
sub    edx,DWORD PTR gs:0x14
je     0x80491d0 <vuln+74>
call   0x8049200 <__stack_chk_fail_local>
mov    ebx,DWORD PTR [ebp-0x4]
leave
ret
```

gs:0x14?

POSTECH

- x86 maintains a Local Descriptor Table (LDT) in memory
- On Linux, GS segment register points to the Thread Control Block (TCB) entry of LDT

```
typedef struct {
  void *tcb;                 /* gs:0x00 Pointer to the TCB. */
  dtv_t *dtv;                /* gs:0x04 */
  void *self;                /* gs:0x08 Pointer to the thread descriptor.  */
  int multiple_threads;      /* gs:0x0c */
  uintptr_t sysinfo;         /* gs:0x10 Syscallinterface */
  uintptr_t stack_guard;     /* gs:0x14 Random value used for stack protection */
  uintptr_t pointer_guard;   /* gs:0x18 Random value used for pointer protection */
  int gscope_flag;           /* gs:0x1c */
  int private_futex;         /* gs:0x20 */
  void *__private_tm[4];     /* gs:0x24 Reservation of some values for the TM ABI.  */
  void *__private_ss;        /* gs:0x34 GCC split stack support.  */
} tcbhead_t;
```

Who initializes [gs:0x14]?

- Runtime dynamic linker initializes the canary every time it launches a process
  - Pseudocode:

```
uintptr_t ret;
int fd = open(“/dev/urandom”, O_RDONLY);
if (fd > 0) {
  ssize_t len = read(fd, &ret, sizeof(ret));
  if (len == (ssize_t) sizeof(ret)) {
    asm(“mov DWORD PTR gs:0x14, ret”);
  }
}
```

Subverting stack canary (1)

- Leak the value of the canary
  - Any vulnerability that leaks stack memory can be used
  - e.g., format string vulnerabilities let you print out stack values

→ Once leaked, overwrite the canary with the leaked canary

→ Q) A concrete attack scenario?

Subverting stack canary (2)

- Guess the value of canary
  - LSB of canary is always 0x00 // why?
  - On 32-bit systems, there are only 24 bits to guess
  - 2^24 (~16 million) can be feasibly brute-forced

→ Once successfully guessed, overwrite the canary with the guessed canary

- Q) Doesn't canary value change every time a process is run?
- A) Think about server application, which forks per request

Subverting stack canary (3)

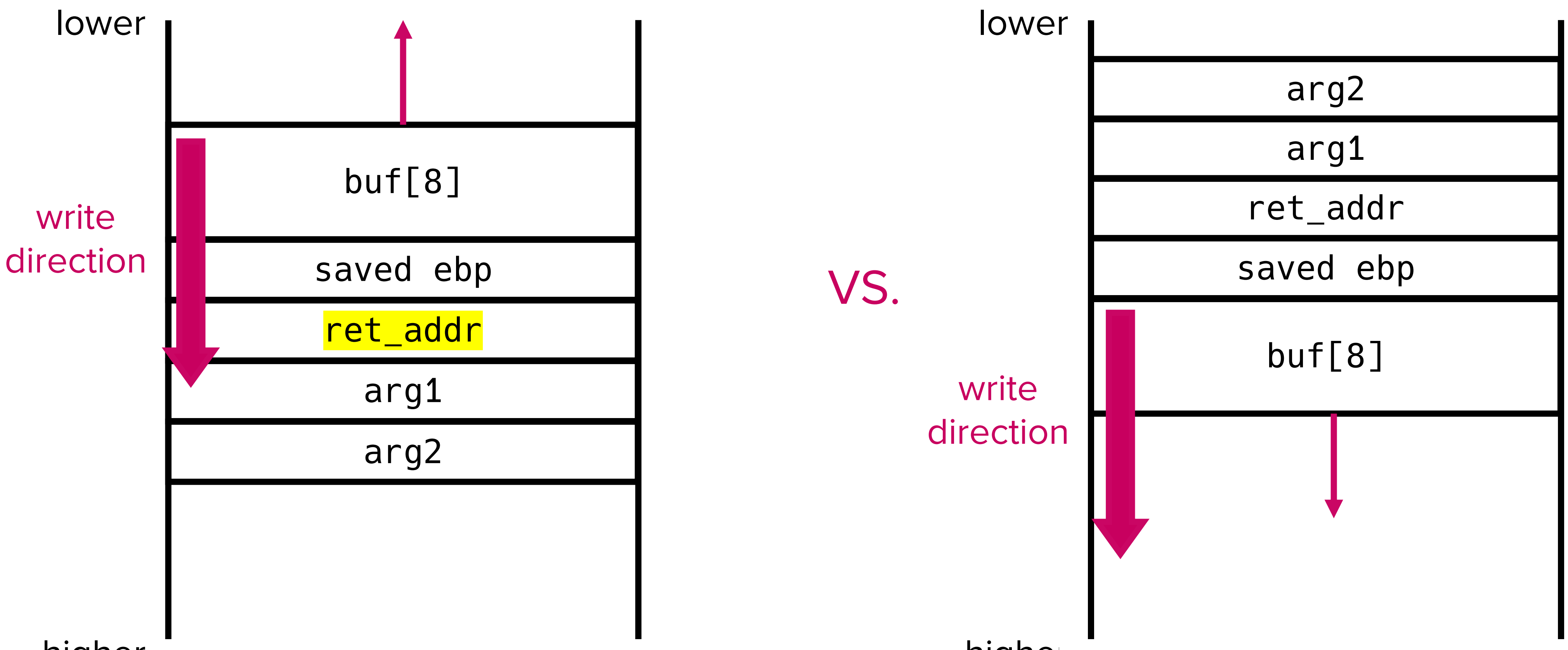
- Bypass the value of canary
  - Some vulnerabilities let you do “arbitrary write”
  - That is, writing an arbitrary vaule at an arbitrary address
  - Stack canary only mitigates sequential writes (strcpy, …)

→ Overwrite the return address without touching the stack canary

Other defense

Advanced topic: Flipped stack

- What if the stack grows down?



ret_addr can be overwritten

ret_addr cannot be overwritten

Discussion: Lab 02's target

POSTECH

- Mitigations applied
  - NX
  - ASLR
  - Canary

- How can we attack this binary?
- What if the binary does not leak a code pointer?

Section 1 of CSED415 is over

- Attacks start from a BoF vulnerability (Attack surface - Lec 02)
  - Buggy code is the root of evil (Lec 03)
- Mitigations exist, but they are not perfect
  - Designed in a way a method compromises C/I/A as little as possible
  - Cost vs Effectiveness

Primary focus has been System Integrity (ref: Lec 02)
“A system performs its intended function in an unimpaired manner.”

→ How do we preserve Confidentiality?

Coming up next

- Cryptographic primitives and their applications

Questions?