Lec 08: Attacks and Defenses (2)

CSED415: Computer Security

Spring 2025

Seulbae Kim

POSTECH

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Announcement

- No class meeting next Tuesday (March 18)
  - I will be traveling to present a funding proposal
  - The lecture video will be uploaded before Tuesday
    - We will begin a new section (Cryptography)
    - **Important**: Please watch the video before next Thursday's lecture to ensure you can follow the subsequent material

Recap

- NX is effective at preventing return-to-stack attacks
  - MMU raises page fault exception if fetched instruction belongs to a NX page
  - Stack is marked NX by default
- Return-to-libc attack bypasses NX protection
  - Return to a libc function useful for exploitations
- Return-Oriented Programming (ROP) generalizes code reuse attacks

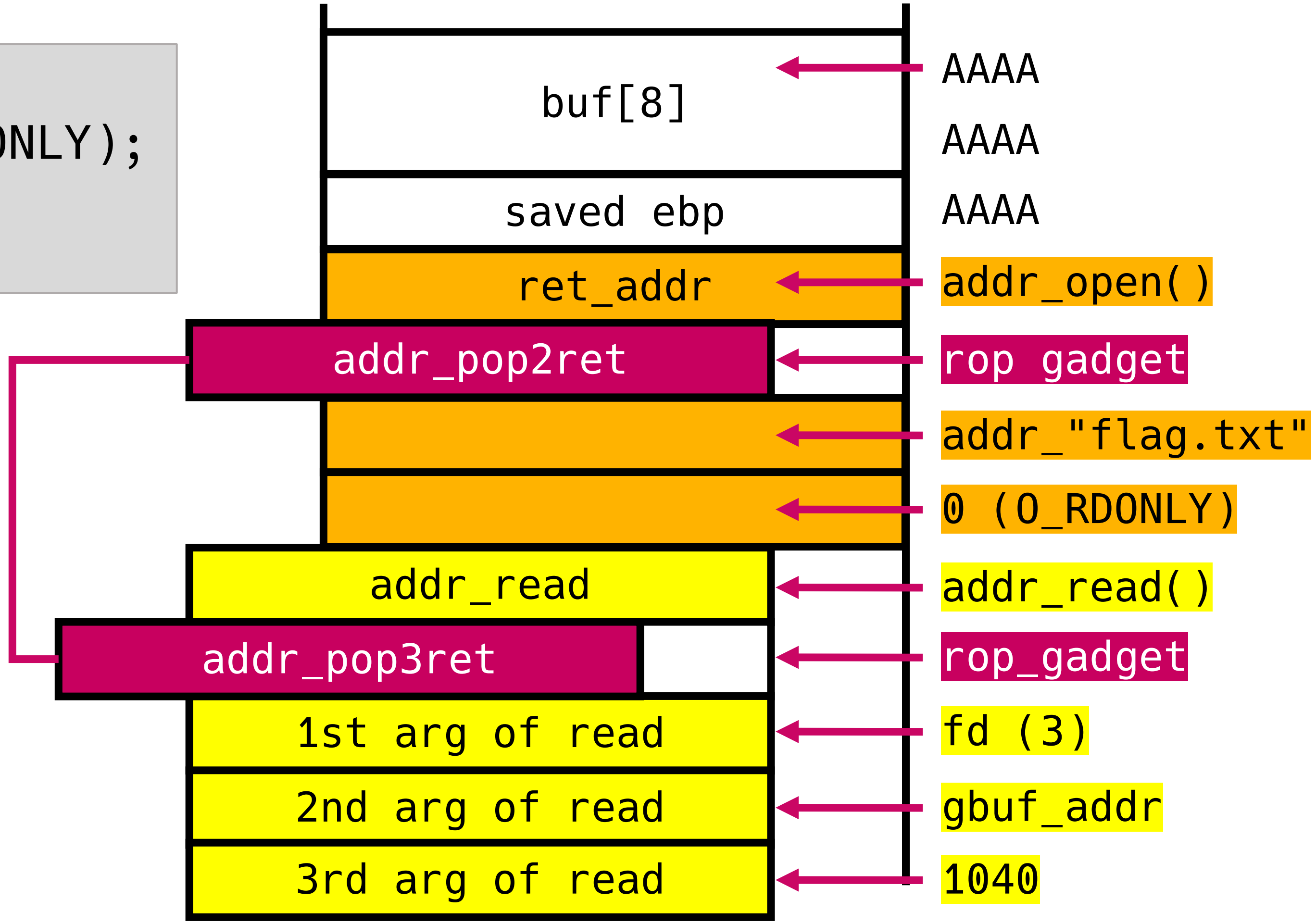
Recap

- Chaining multiple func calls

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

ROP gadgets are essential for chaining returns to multiple functions

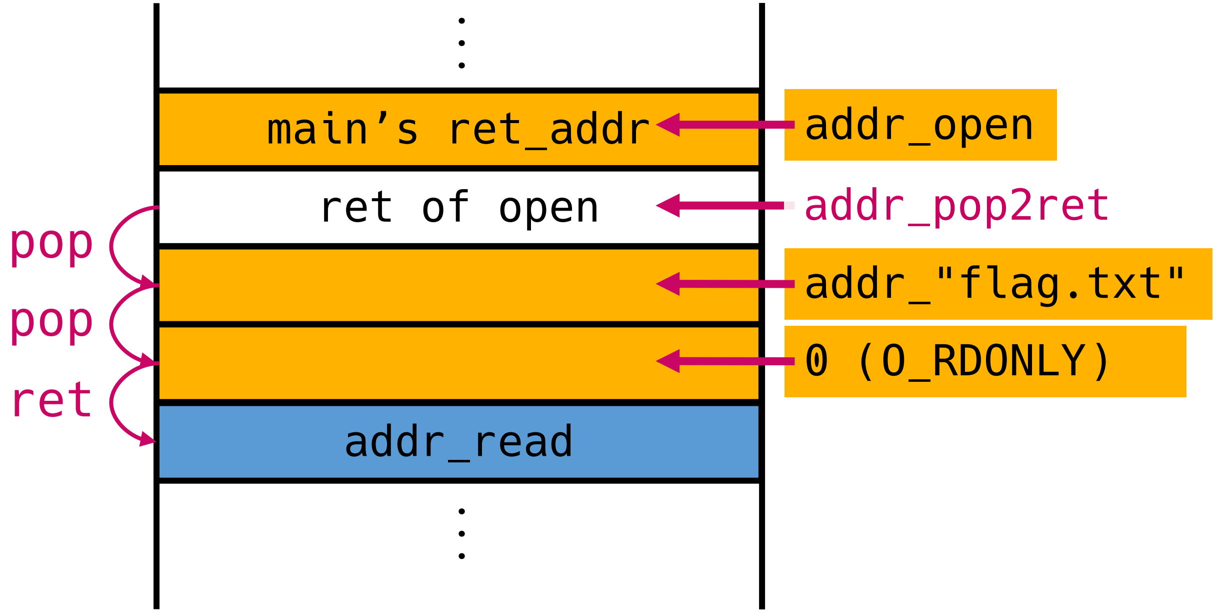
- ROP chain (x86, 32-bit)



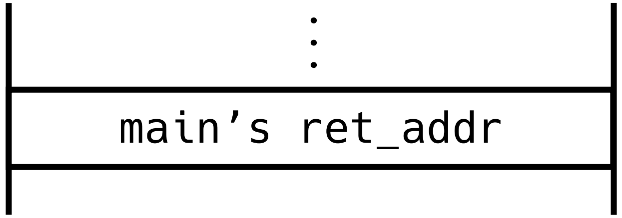
Attack #1-2: ROP (x86_64)

ROP for x86_64 ?

- x86 stores args on the stack
  - Any pop2ret gadget is useful



- x64 puts args in the registers
  - RDI, RSI, RDX, RCX, R8, R9
  - Only the extra args are stored on the stack



Before returning to `addr_open`

`addr_"flag.txt"` should be put in RDI

`0 (O_RDONLY)` should be put in RSI

How?

ROP for x64

- We should utilize gadgets that POP stack values into registers
  - e.g., `pop {rdi, rsi, rdx, rcx, r8, r9}; ret;`
  - Example gadgets:

```
0x401303: pop rdi
0x401304: ret
```

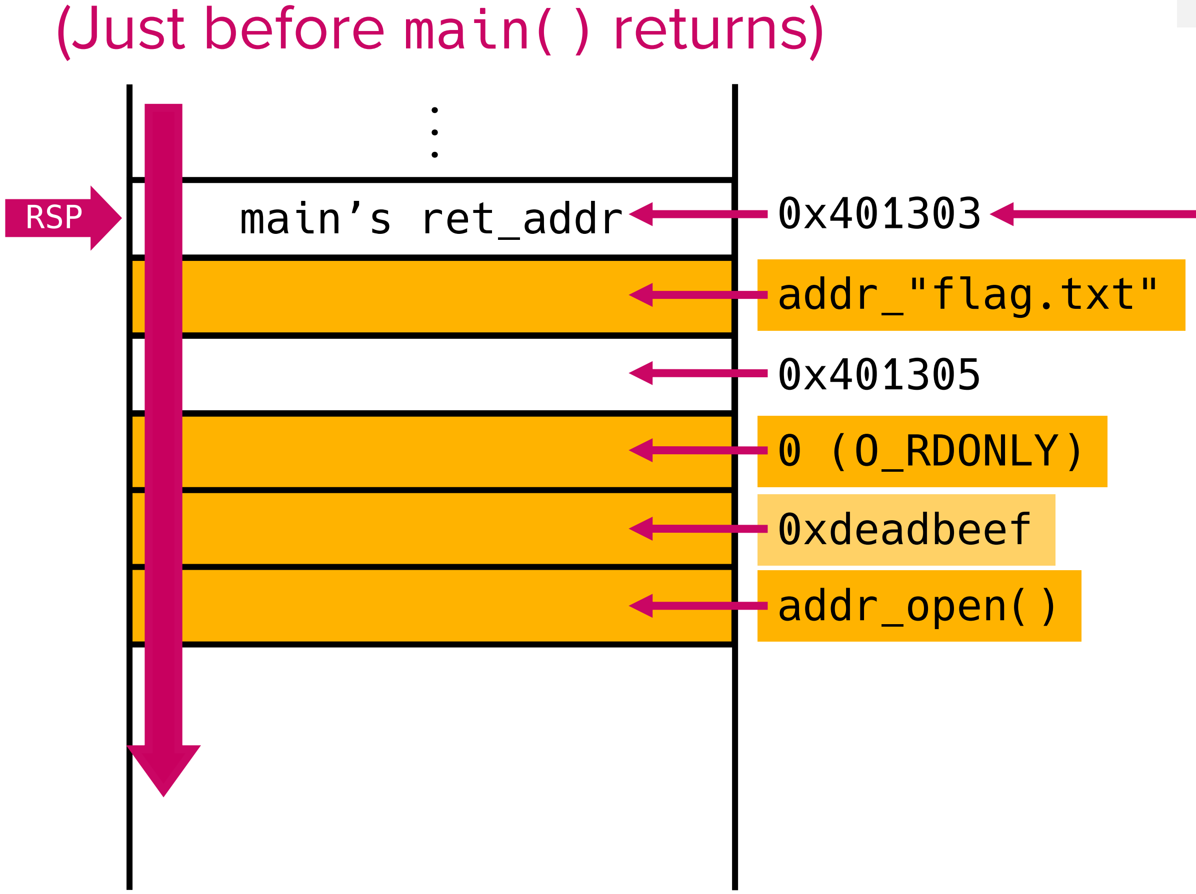
```
0x401305: pop rsi
0x401306: pop r15
0x401307: ret
```

```
0x401309: pop rdx
0x40130a: ret
```

ROP for x64

- Setting up args using gadgets

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




```
0x401303: pop rdi
0x401304: ret
```

```
0x401305: pop rsi
0x401306: pop r15
0x401307: ret
```

```
0x401309: pop rdx
0x40130a: ret
```

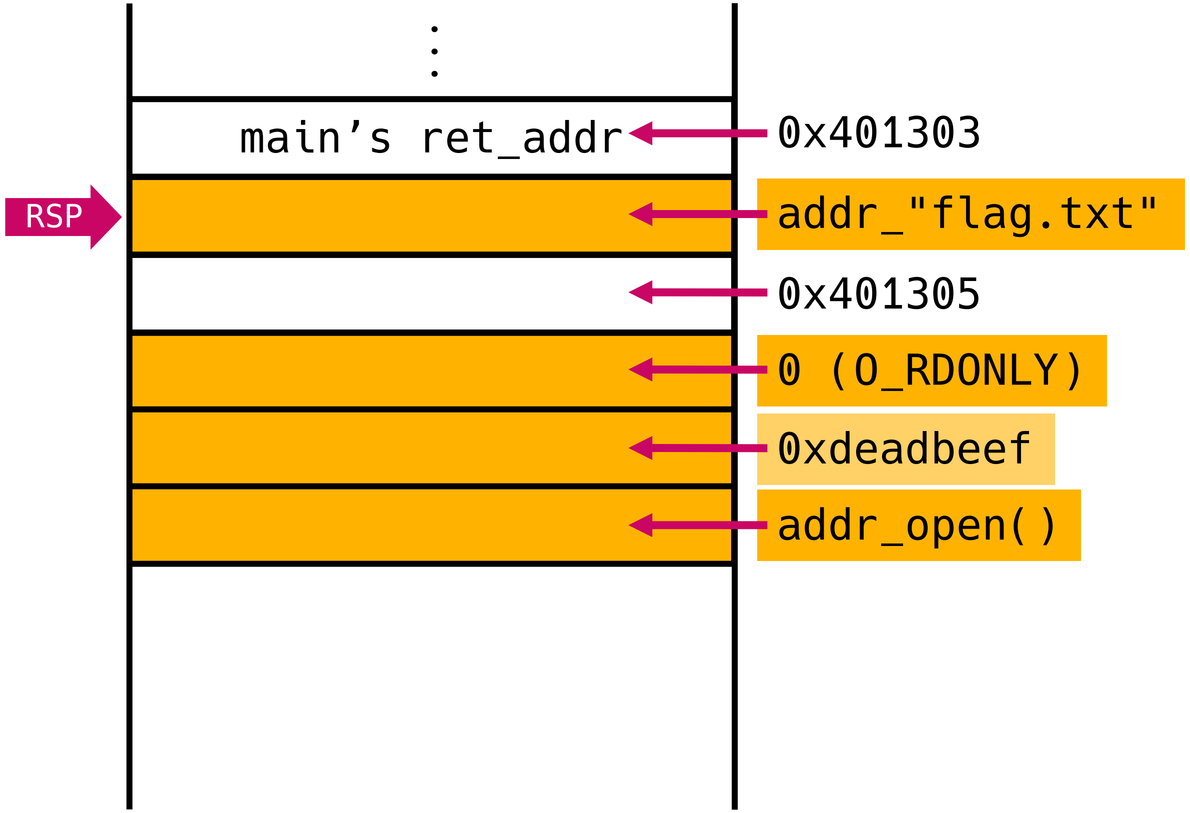
REG	VALUE
RDI	?
RSI	?
RDX	?

ROP for x64

- Setting up args using gadgets

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

(Return to the "pop rdi" gadget)



RIP

```
0x401303: pop rdi
0x401304: ret

0x401305: pop rsi
0x401306: pop r15
0x401307: ret

0x401309: pop rdx
0x40130a: ret
```

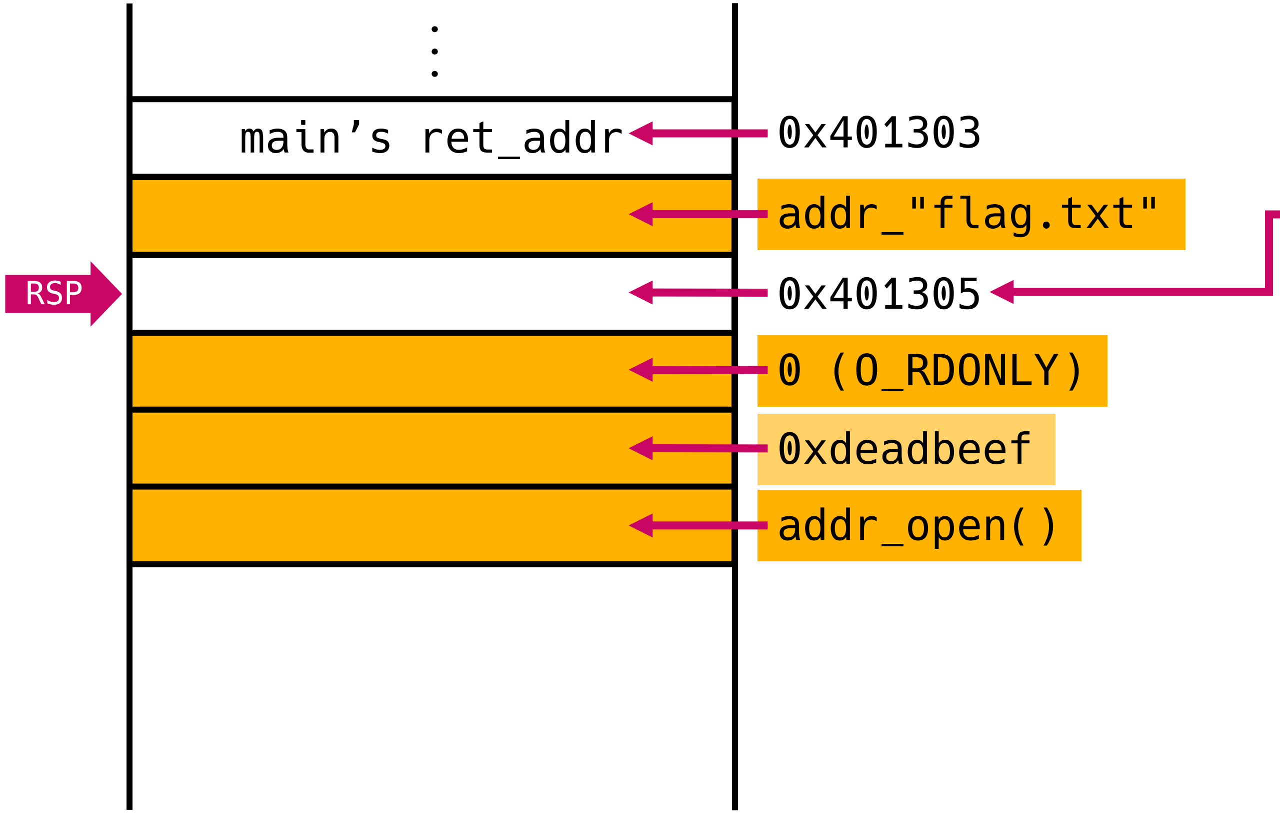
REG	VALUE
RDI	?
RSI	?
RDX	?

ROP for x64

- Setting up args using gadgets

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

(addr_"flag.txt" is popped into rdi)



RIP

```
0x401303: pop rdi
0x401304: ret
```

```
0x401305: pop rsi
0x401306: pop r15
0x401307: ret
```

```
0x401309: pop rdx
0x40130a: ret
```

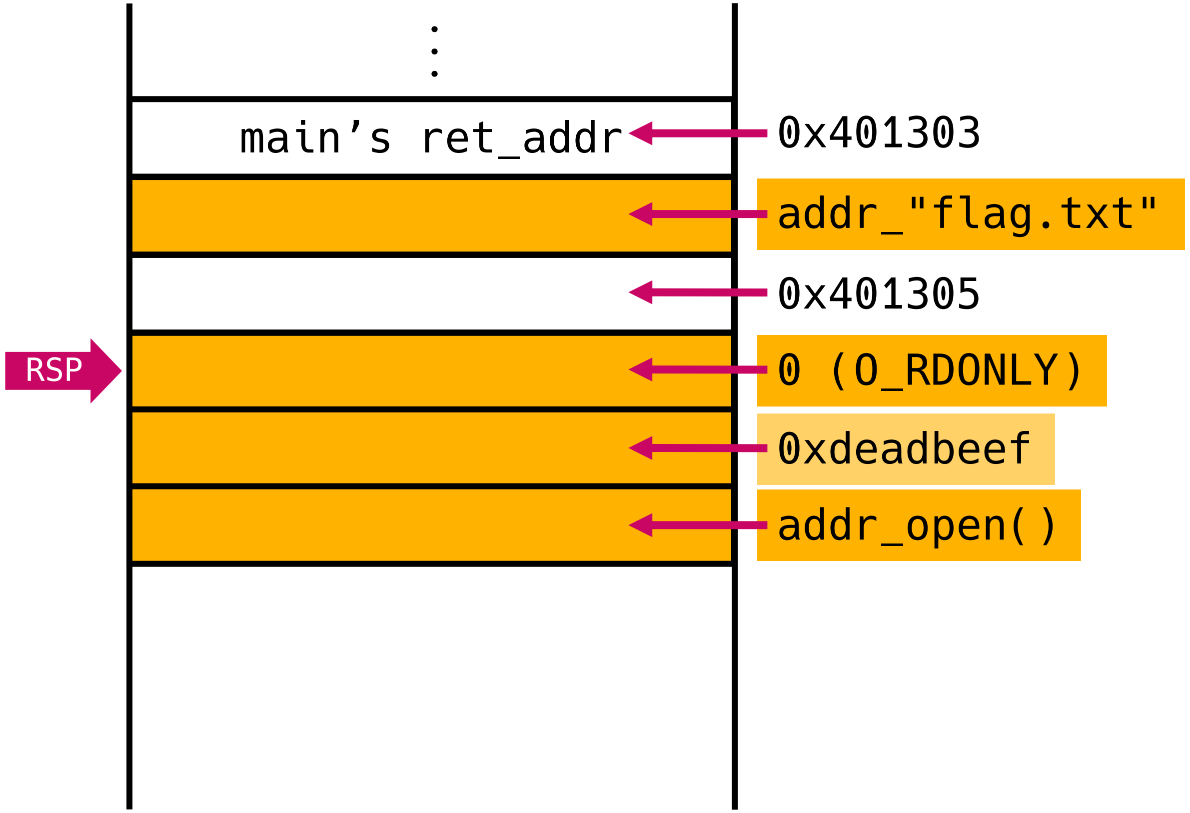
REG	VALUE
RDI	addr_"flag.txt"
RSI	?
RDX	?

ROP for x64

- Setting up args using gadgets

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

(return to the "pop rsi" gadget)




```
0x401303: pop rdi
0x401304: ret
```

RIP

```
0x401305: pop rsi
0x401306: pop r15
0x401307: ret
```

```
0x401309: pop rdx
0x40130a: ret
```

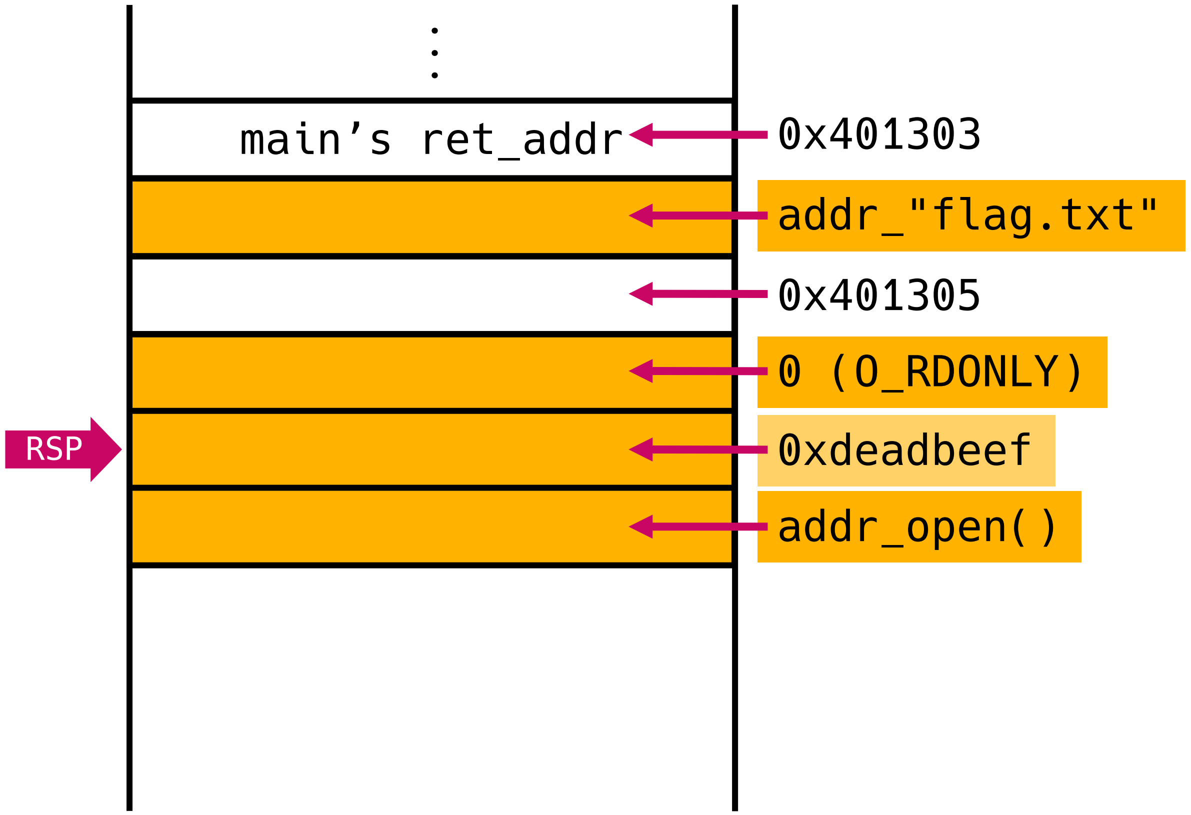
REG	VALUE
RDI	addr_"flag.txt"
RSI	?
RDX	?

ROP for x64

- Setting up args using gadgets

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

(0 is popped into RSI)



RIP

```
0x401303: pop rdi
0x401304: ret

0x401305: pop rsi
0x401306: pop r15
0x401307: ret

0x401309: pop rdx
0x40130a: ret
```

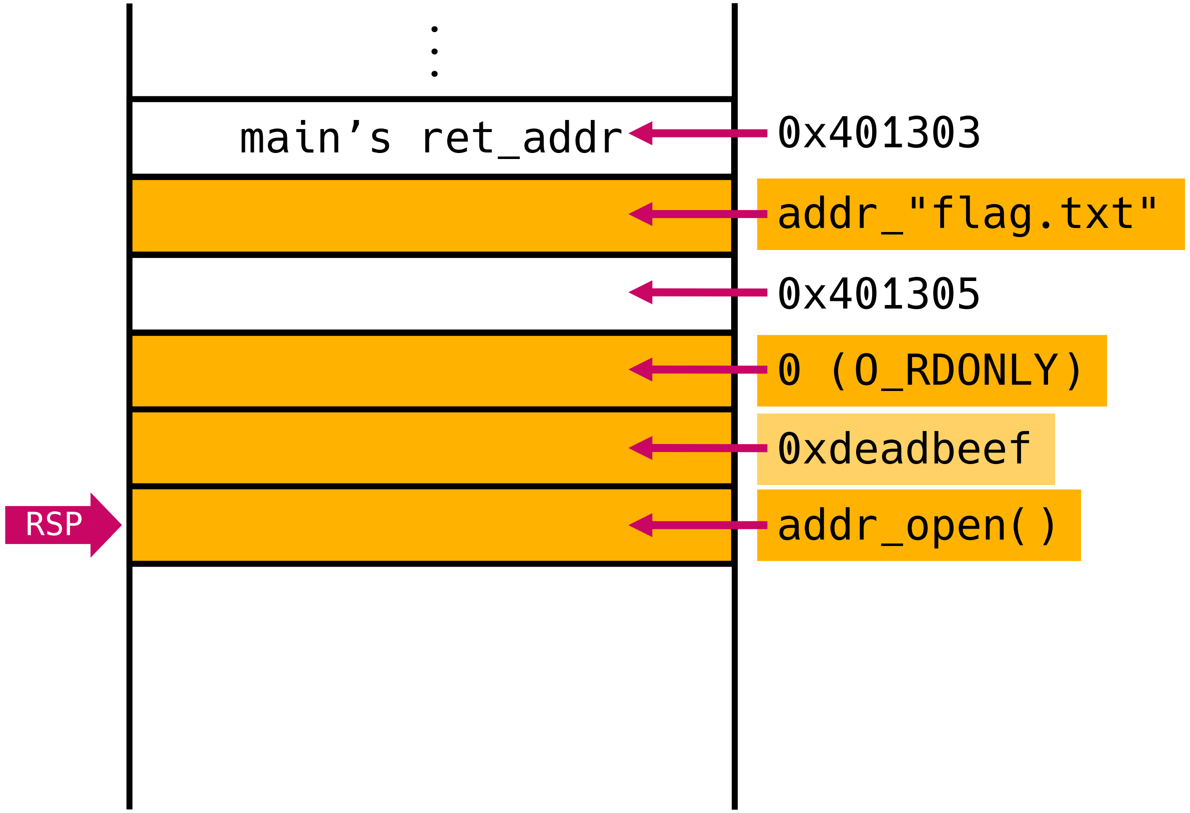
REG	VALUE
RDI	addr_"flag.txt"
RSI	0
RDX	?

ROP for x64

- Setting up args using gadgets

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

Pop a dummy into r15 (irrelevant)




```
0x401303: pop rdi
0x401304: ret

0x401305: pop rsi
0x401306: pop r15
0x401307: ret        <- RIP

0x401309: pop rdx
0x40130a: ret
```

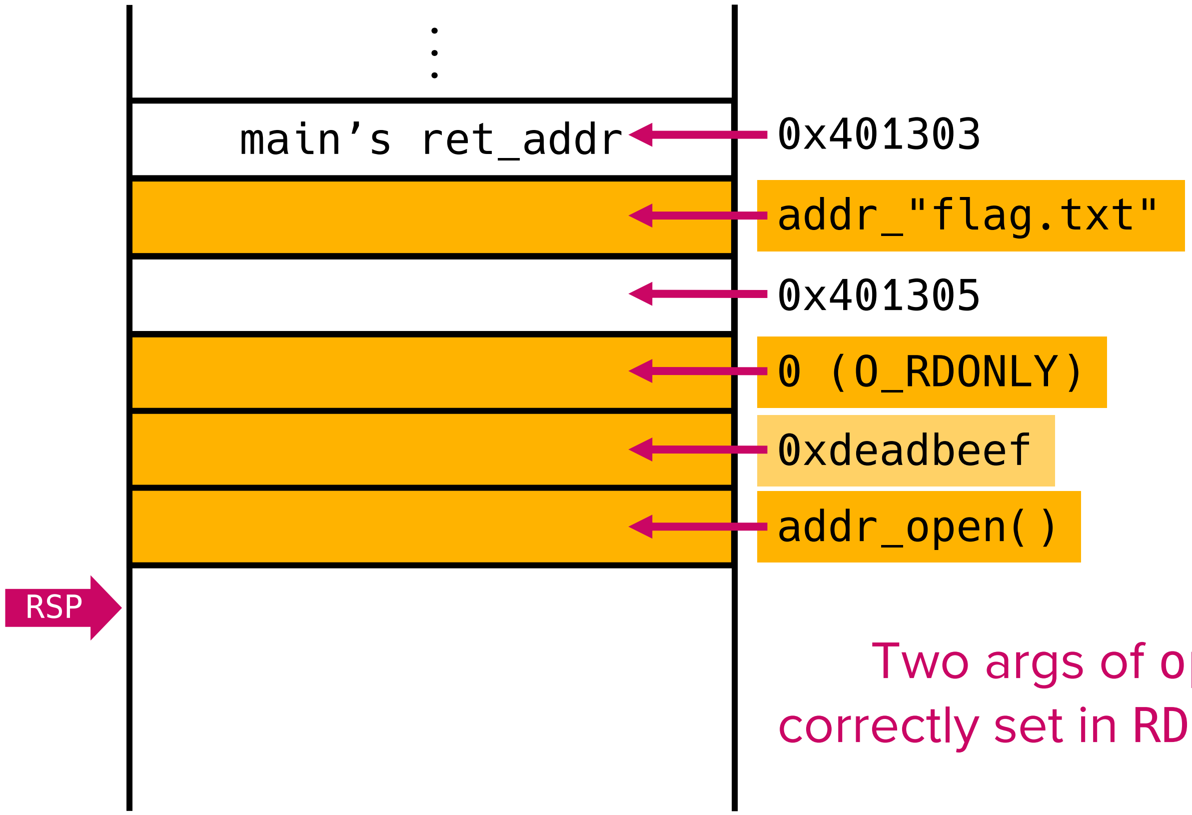
REG	VALUE
RDI	addr_"flag.txt"
RSI	0
RDX	?

ROP for x64

- Setting up args using gadgets

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

Returns to open(addr_"flag.txt", 0);




```
0x401303: pop rdi
0x401304: ret

0x401305: pop rsi
0x401306: pop r15
0x401307: ret

0x401309: pop rdx
0x40130a: ret
```

RIP → 0x401307

REG	VALUE
RDI	addr_"flag.txt"
RSI	0
RDX	?

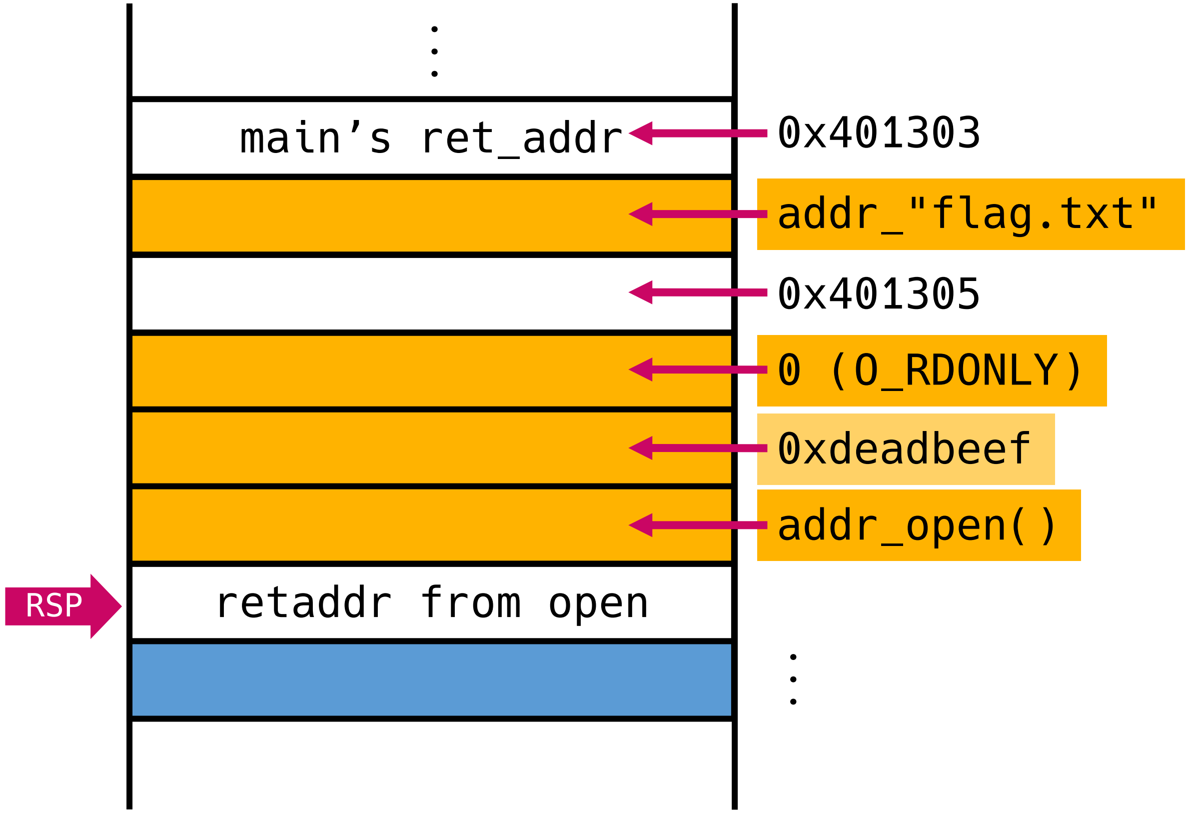
Two args of open( ) are correctly set in RDI and RSI

ROP for x64

- Setting up args using gadgets

We can keep the chain going!

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




```
0x401303: pop rdi
0x401304: ret
```

```
0x401305: pop rsi
0x401306: pop r15
0x401307: ret
```

```
0x401309: pop rdx
0x40130a: ret
```

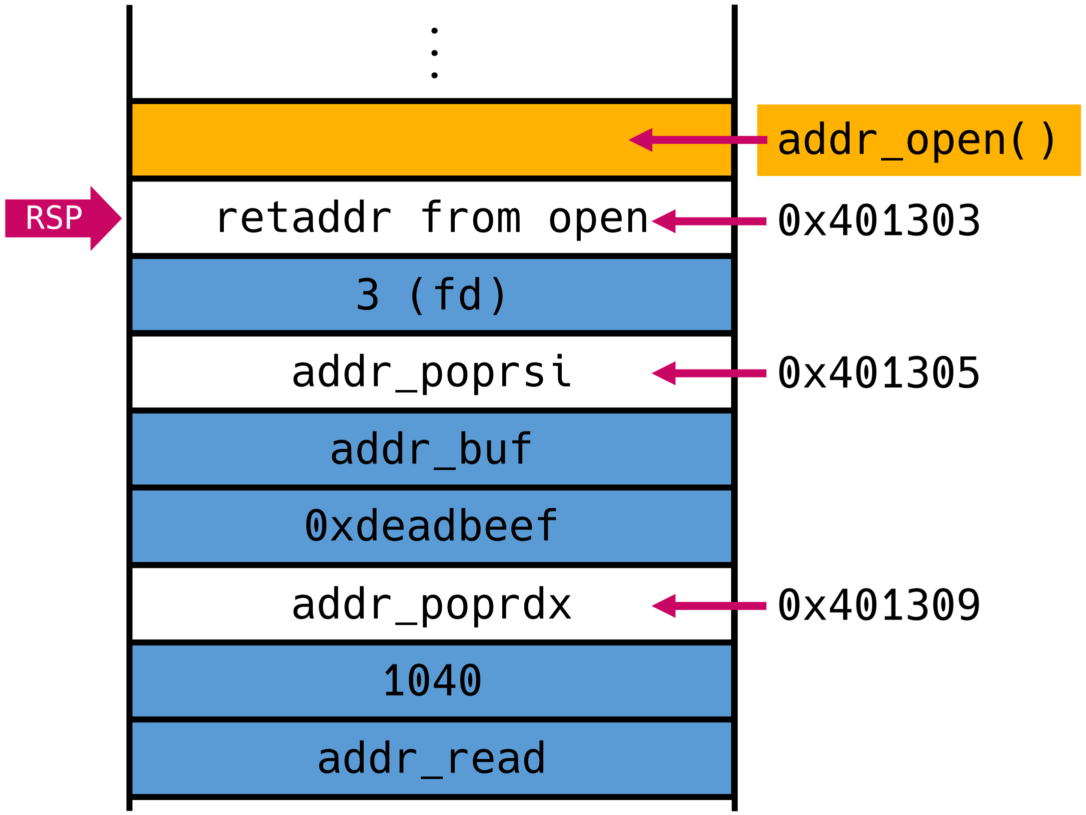
REG	VALUE
RDI	addr_"flag.txt"
RSI	0
RDX	?

ROP for x64

- Setting up args using gadgets

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

(Scrolled down)




```
0x401303: pop rdi
0x401304: ret
```

```
0x401305: pop rsi
0x401306: pop r15
0x401307: ret
```

```
0x401309: pop rdx
0x40130a: ret
```

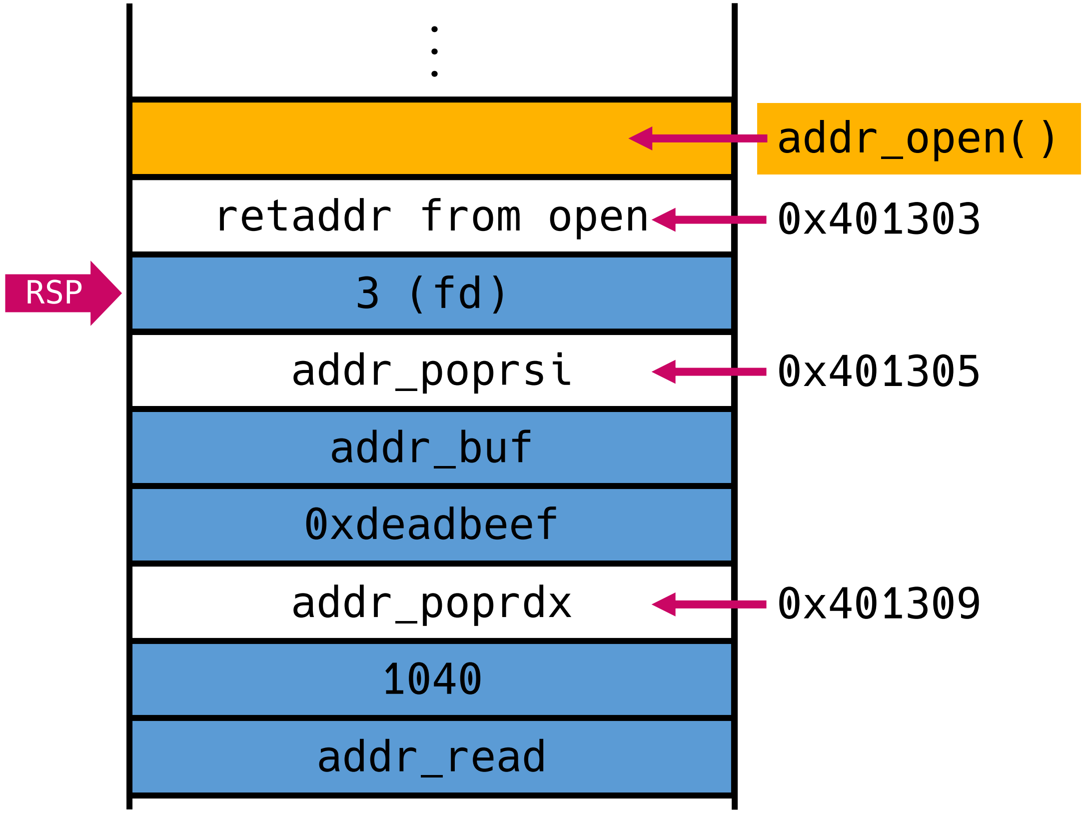
REG	VALUE
RDI	addr_"flag.txt"
RSI	0
RDX	?

ROP for x64

- Setting up args using gadgets

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

(return to the “pop rdi” gadget for arg1)



RIP

```
0x401303: pop rdi
0x401304: ret
```

```
0x401305: pop rsi
0x401306: pop r15
0x401307: ret
```

```
0x401309: pop rdx
0x40130a: ret
```

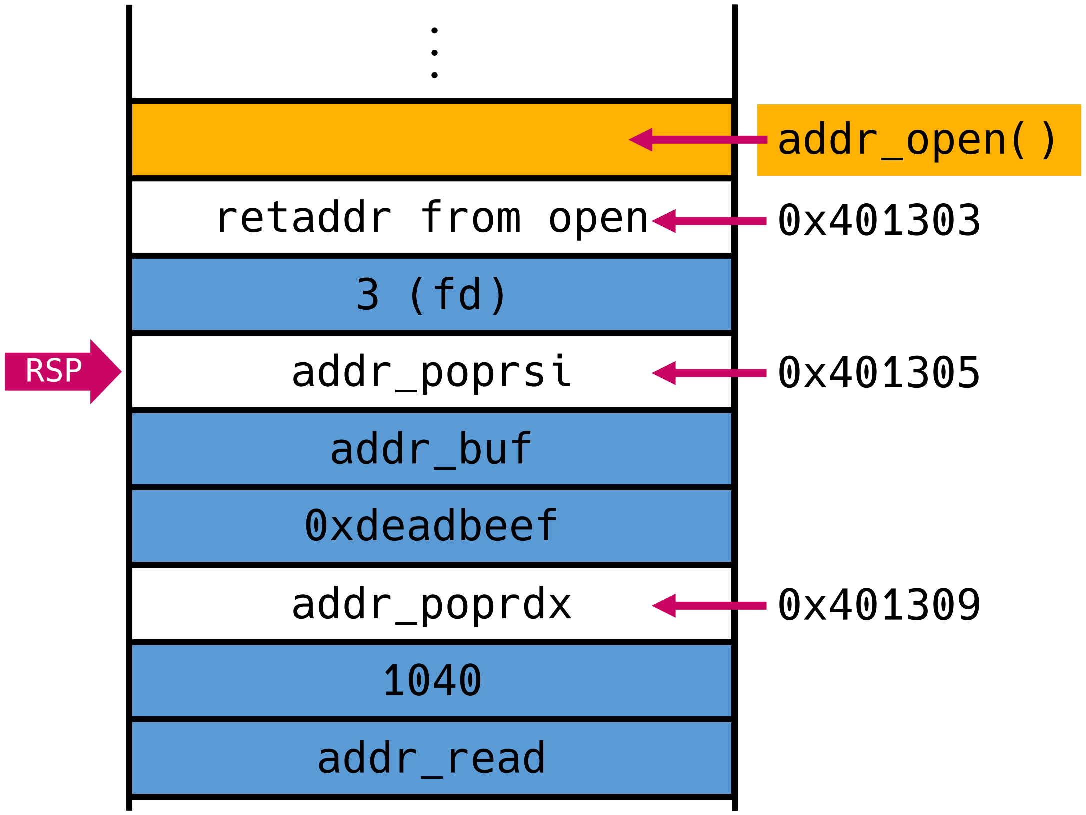
REG	VALUE
RDI	addr_"flag.txt"
RSI	0
RDX	?

ROP for x64

- Setting up args using gadgets

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

(RDI becomes 3)



RIP

```
0x401303: pop rdi
0x401304: ret
```

```
0x401305: pop rsi
0x401306: pop r15
0x401307: ret
```

```
0x401309: pop rdx
0x40130a: ret
```

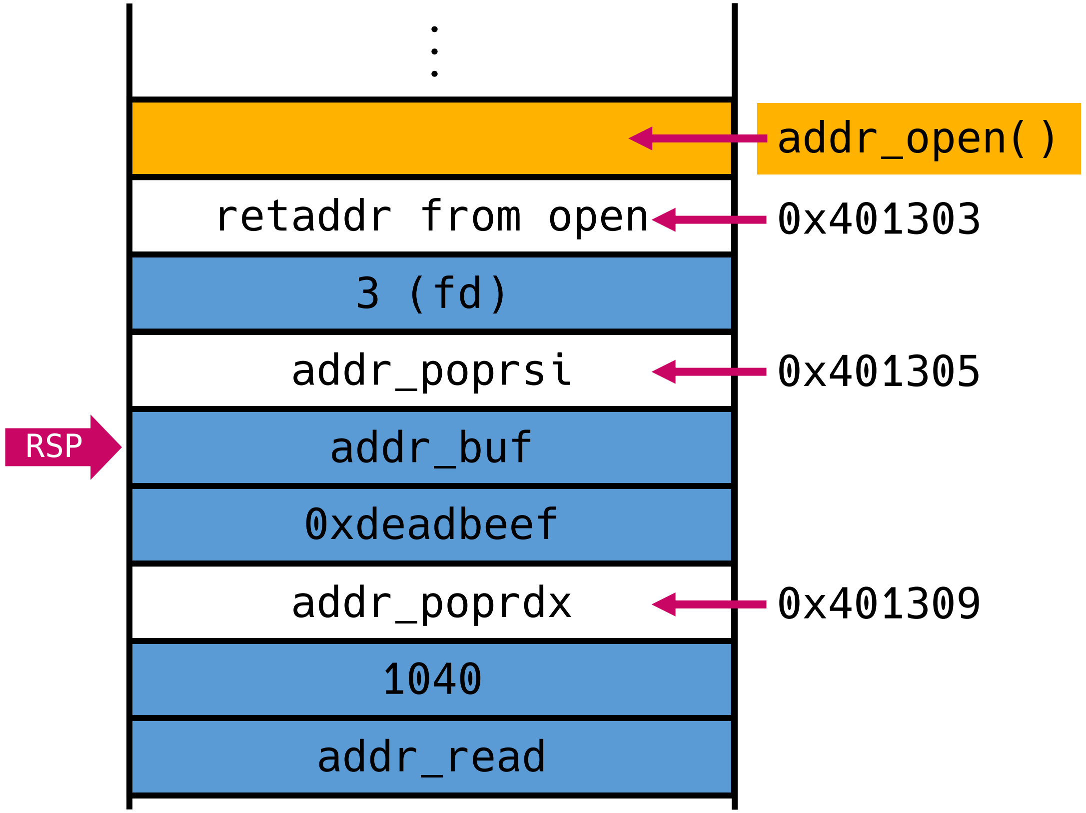
REG	VALUE
RDI	3
RSI	0
RDX	?

ROP for x64

- Setting up args using gadgets

(return to the “pop rsi” gadget for arg2)

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



RIP

```
0x401303: pop rdi
0x401304: ret
```

```
0x401305: pop rsi
0x401306: pop r15
0x401307: ret
```

```
0x401309: pop rdx
0x40130a: ret
```

REG	VALUE
RDI	3
RSI	0
RDX	?

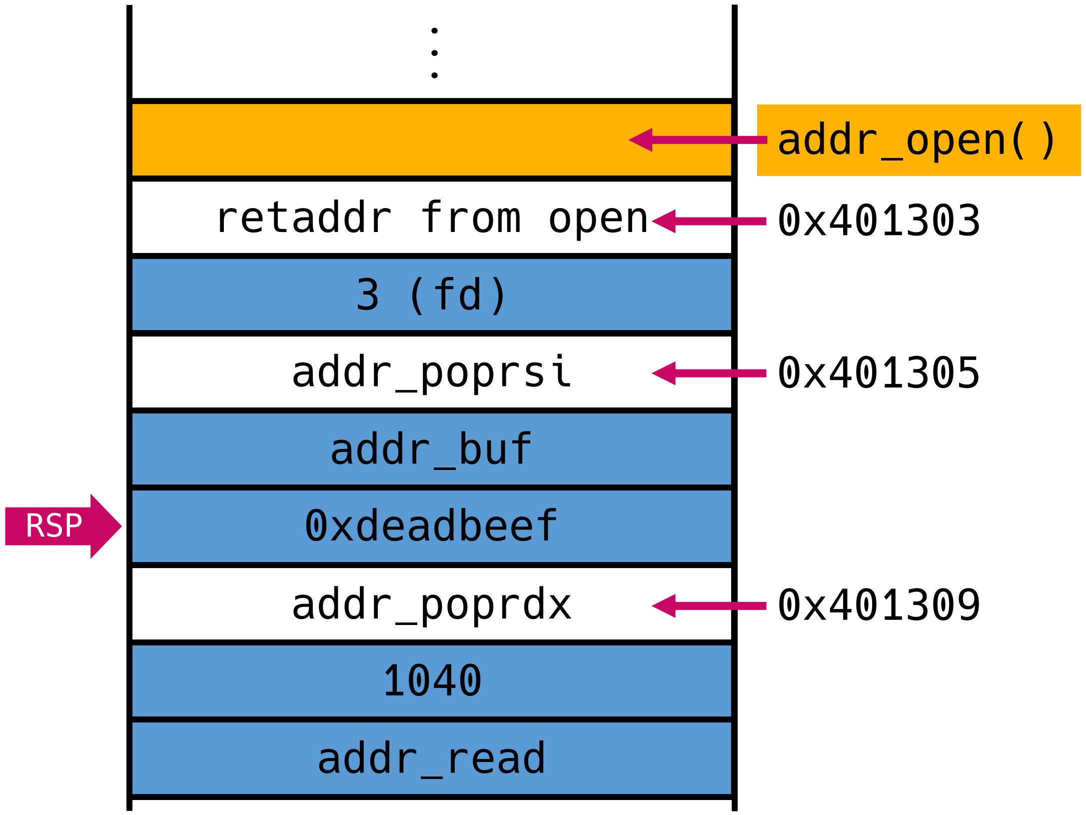
ROP for x64

POSTECH

- Setting up args using gadgets

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

(addr_buf is popped into RSI)




```
0x401303: pop rdi
0x401304: ret
```

RIP

```
0x401305: pop rsi
0x401306: pop r15
0x401307: ret
```

```
0x401309: pop rdx
0x40130a: ret
```

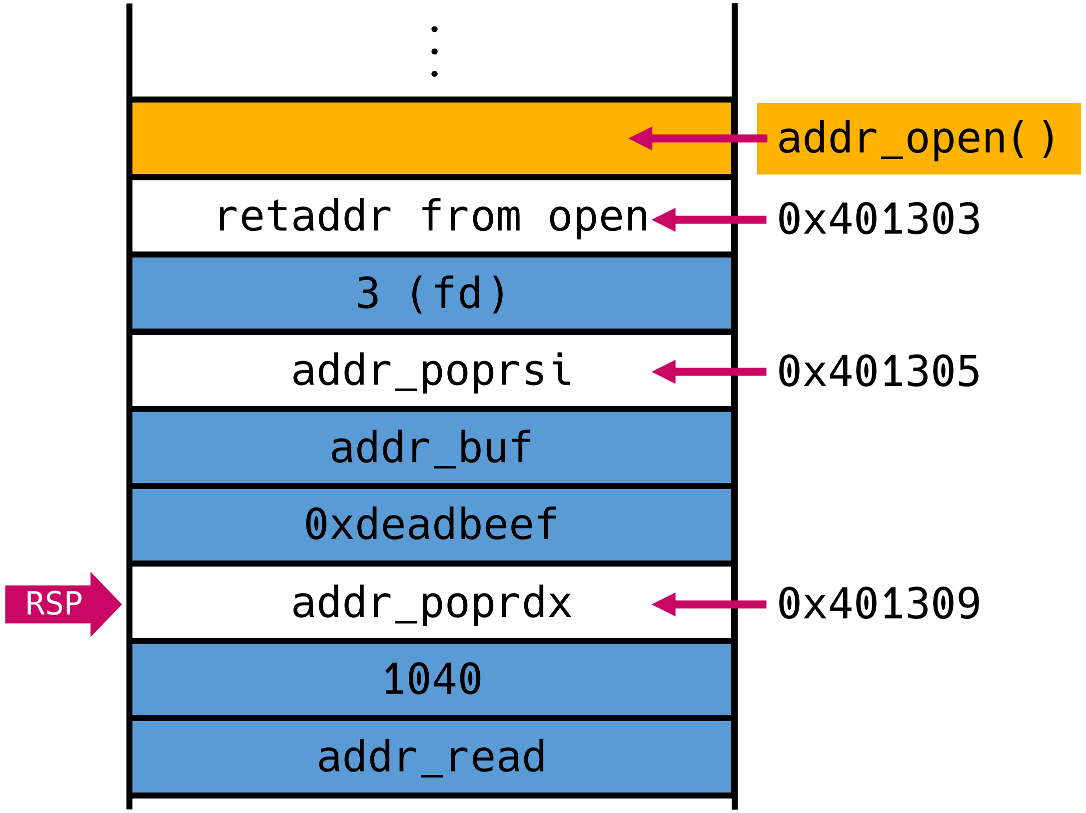
REG	VALUE
RDI	3
RSI	addr_buf
RDX	?

ROP for x64

- Setting up args using gadgets

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

(A dummy in r15)




```
0x401303: pop rdi
0x401304: ret

0x401305: pop rsi
0x401306: pop r15
0x401307: ret

0x401309: pop rdx
0x40130a: ret
```

RIP → 0x401307

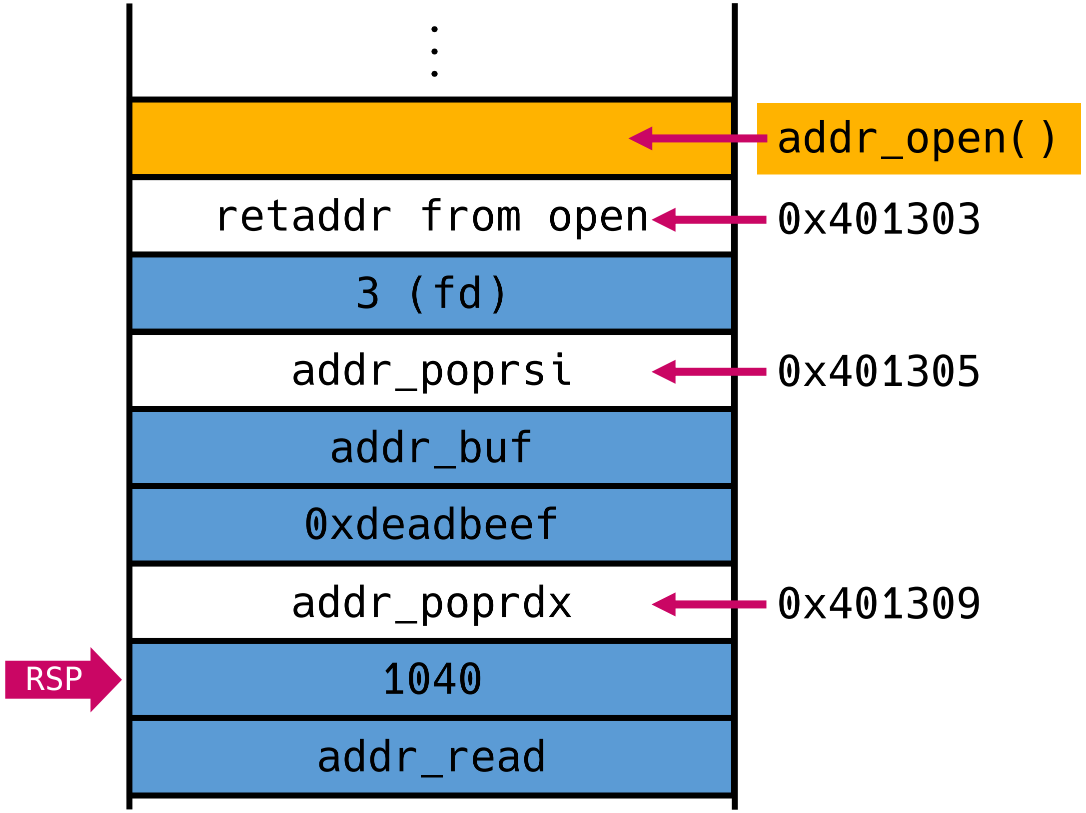
REG	VALUE
RDI	3
RSI	addr_buf
RDX	?

ROP for x64

- Setting up args using gadgets

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

(return to the “pop rdx” gadget for arg3)




```
0x401303: pop rdi
0x401304: ret

0x401305: pop rsi
0x401306: pop r15
0x401307: ret

0x401309: pop rdx
0x40130a: ret
```

RIP → 0x401309

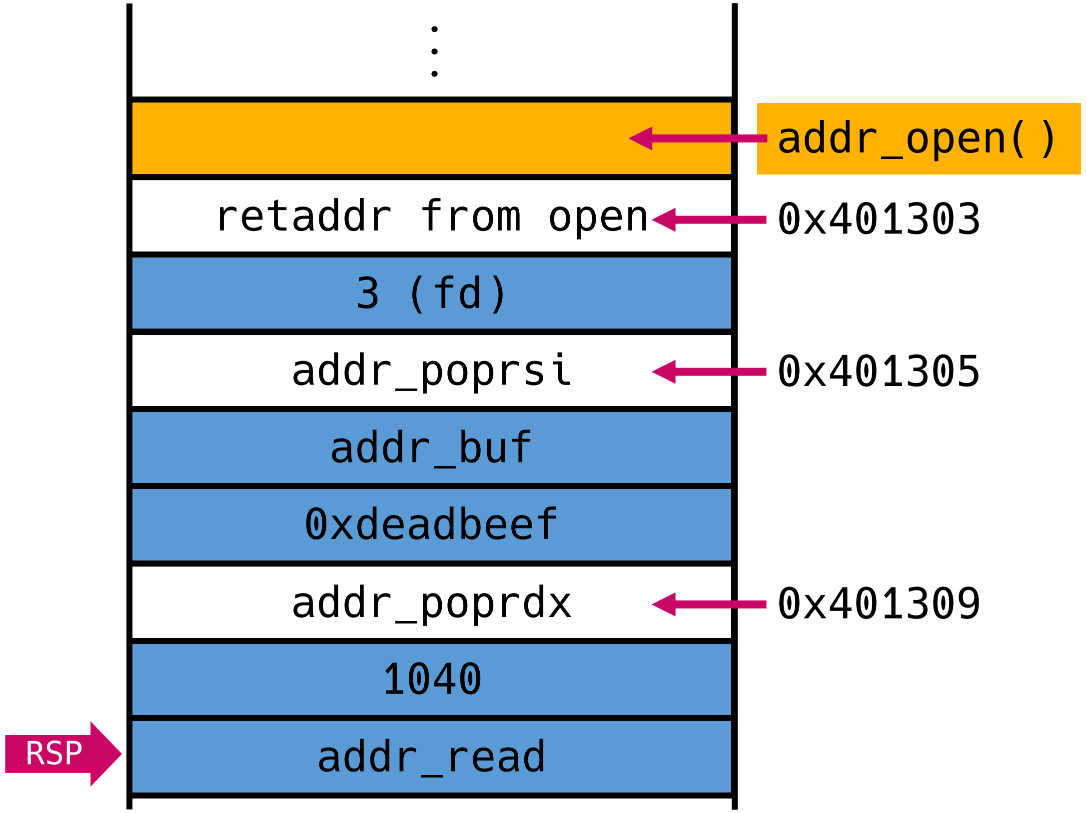
REG	VALUE
RDI	3
RSI	addr_buf
RDX	?

ROP for x64

- Setting up args using gadgets

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

(1040 is popped into RDX)




```
0x401303: pop rdi
0x401304: ret

0x401305: pop rsi
0x401306: pop r15
0x401307: ret

0x401309: pop rdx
0x40130a: ret
```

RIP

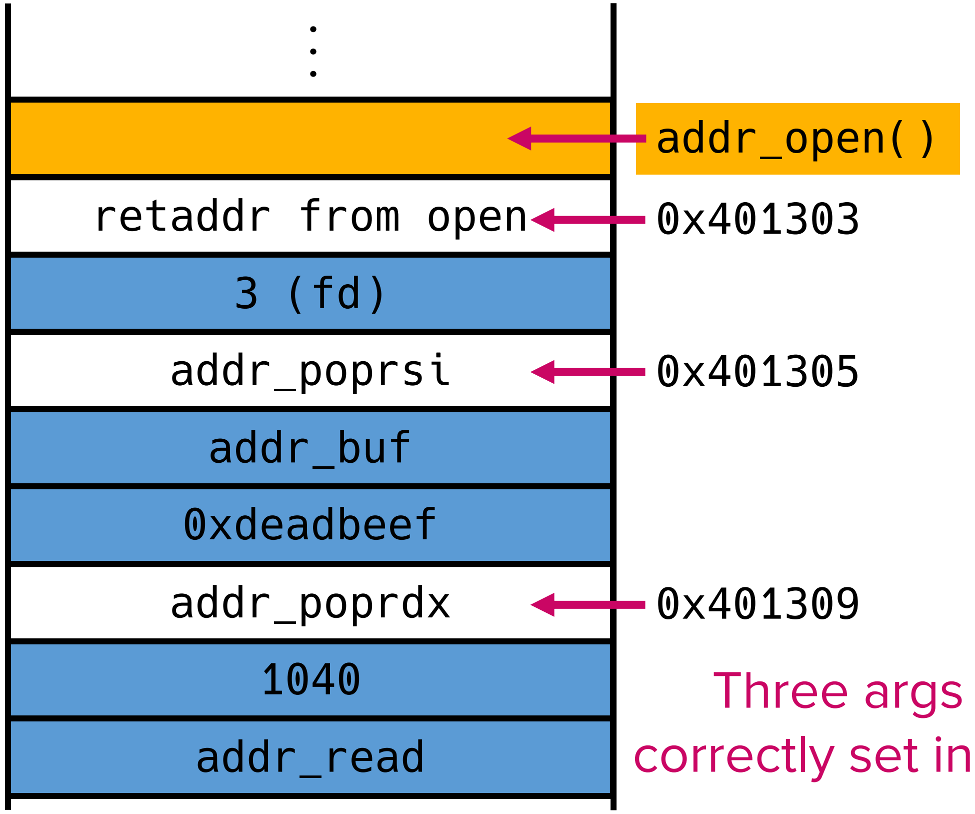
REG	VALUE
RDI	3
RSI	addr_buf
RDX	1040

ROP for x64

- Setting up args using gadgets

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

Returns to read(3, addr_buf, 1040);




```
0x401303: pop rdi
0x401304: ret
```

```
0x401305: pop rsi
0x401306: pop r15
0x401307: ret
```

```
0x401309: pop rdx
0x40130a: ret
```

Three args of read( ) are correctly set in RDI, RSI, RDX

REG	VALUE
RDI	3
RSI	addr_buf
RDX	1040

More on ROP Gadgets

ROP gadgets

- Small sequences of instructions found within the existing code of a program or the libraries it links to
- Key property
  - ROP gadget ends with an instruction that affects the instruction pointer (`rip` of x86_64, `eip` of x86)
    - `ret            ; == pop rip`
    - `call <label> ; == push next_rip + jmp <label>`
    - `jmp  <label> ; == rip = <label>`

ROP gadgets in Lab 02's target binary

- Finding gadgets through objdump
  - Search for ret instruction from target's disassembly

```
csed415-lab02@csed415:~$ objdump -D ./target -M intel | grep -B3 ret
...
--
    1267:   e8 64 ff ff ff          call   11d0 <deregister_tm_clones>
    126c:   c6 05 b5 2d 00 00 01    mov    BYTE PTR [rip+0x2db5],0x1
    1273:   5d                      pop    rbp
    1274:   c3                      ret
--
    14b8:   48 83 c4 08             add    rsp,0x8
    14bc:   c3                      ret
--
```

No useful gadget found ☹

ROP gadgets in Lab 02's target binary

- Finding gadgets through objdump
  - Search for ret instruction from target's disassembly

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

Can we find more gadgets?

```
--
    14b8:    48 83 c4 08                add    rsp,0x8
    14bc:    c3                         ret
--
```

No useful gadget found ☹

Background: Variable-length instructions

- Intel's CISC architecture employs variable-length instructions
  - Length of encoded instructions vary

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
  (We can assign shorter lengths to frequently used instructions)

Q) Disadvantage?

- Potential security issues
  (See next slides)

Background: Variable-length instructions

```
csed415-lab02@csed415:~$ objdump -D ./target -M intel
...
    1457:   f3 0f 1e fa             endbr64
    145b:   55                      push   rbp
    145c:   48 89 e5                mov    rbp,rsp
    145f:   48 83 ec 10             sub    rsp,0x10
    1463:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
    1466:   48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
    146a:   e8 1a fe ff ff          call   1289 <init>
    146f:   48 8d 05 2a 0c 00 00    lea    rax,[rip+0xc2a]
    1476:   48 89 c7                mov    rdi,rax
    1479:   e8 72 fc ff ff          call   10f0 <puts@plt>
...
```

Variable length!

Disassembling x86_64

- Bytecode

```
0x45 0x08 0x5d 0x48 0x01 0xc8 0xc3 ...
```

- Disassembled code from ⬆

```
or      BYTE PTR [r13+0x48],r11b
add     eax,ecx
ret
mov     rdi,rbp
```

Disassembling x86_64

- Bytecode

```
0x45 0x08 0x5d 0x48 0x01 0xc8 0xc3 ...
```

- Disassembled code from ⬆

```
or      BYTE PTR [r13+0x48],bl
add     eax,ecx
ret
mov     rdi,rbp
```

Completely different results, but instructions are still valid

Disassembling x86_64

- Bytecode

```
0x45 0x08 0x5d 0x48 0x01 0xc8 0xc3 ...
```

- Disassembled code from ⬆

```
pop    rbp
add    rax,rcx
ret
mov    rdi,rbp
```

Completely different results, but instructions are still valid

Disassembling x86_64

- Bytecode

```
0x45 0x08 0x5d 0x48 0x01 0xc8 0xc3 ...
```

- Disassembled code from ⬆

```
add    ecx,ecx
ret
mov    rdi,rbp
mov    rdx,rax
```

Completely different results, but instructions are still valid

Disassembling x86_64

- Bytecode

```
0x45 0x08 0x5d 0x48 0x01 0xc8 0xc3 ...
```

- Disassembled code from ⬆

```
enter  0x48c3,0x89
out    dx,eax
mov    rdx,rax
pop    rbp
```

Completely different results, but instructions are still valid

Variable-length instructions

- Variable-length instructions can be disassembled (decoded) from any address
  - Q) Is it legal to jump to the middle of variable-length instructions?
    - Why not? It's perfectly legal in terms of program execution
    - Recall Lec 07: "The CPU is agnostic of the execution semantics"
  - Security problem:
    - We can find unintended ROP gadgets

Variable-length instructions

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

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

- If disassembled from the 2nd byte:

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

Unintended `ret` instructions may appear

ropper: A tool to find ROP gadgets

- ropper scans a binary for all ROP gadgets
  - Including unintended `ret`, `call`, and `jmp` instructions

```
csed415-lab02@csed415:~$ ropper -f ./target
...                                        (Search for ROP gadgets in the target's bytecode)
0x0000000000001017: add esp, 8; ret;
0x0000000000001016: add rsp, 8; ret;
0x0000000000001014: call rax;
0x00000000000011ef: jmp rax;
0x0000000000001273: pop rbp; ret;
...
```

ropper: A tool to find ROP gadgets

- ropper scans a binary for all ROP gadgets
  - Including unintended **`ret`**, **`call`**, and **`jmp`** instructions

```
csed415-lab02@csed415:~$ ropper -f ./target
...
0x0000000000001017: add esp, 8; ret;
0x0000000000001016: add rsp, 8; ret;
0x0000000000001014: call rax;
0x00000000000011ef: jmp rax;
0x0000000000001273: pop rbp; ret;
...
```

But.. What if the gadgets required for x86_64 rop attack do not exist in the binary?

e.g.,

```
pop rdi
ret
```

```
pop rsi
ret
```

```
pop rdx
ret
```

ropper: A tool to find ROP gadgets

- Finding more gadgets:
  - LIBC inevitably contains many gadgets as it is compiled from a huge code base

```
$ ldd ./target | grep libc  (Find the LIBC's shared object file that target links to)
      libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6

$ ropper -f /lib/x86_64-linux-gnu/libc.so.6 --search "pop rdi; ret"
0x000000000002a3e5: pop rdi; ret;             (Search for specific gadgets)

$ ropper -f /lib/x86_64-linux-gnu/libc.so.6 --search "pop rsi; ret"
0x0000000000163f88: pop rsi; ret;             (Search for specific gadgets)

$ ropper -f /lib/x86_64-linux-gnu/libc.so.6 --search "pop % ret"
(a lot of pop*ret gadgets)                    (% matches any character)
```

Defense #2: ASLR

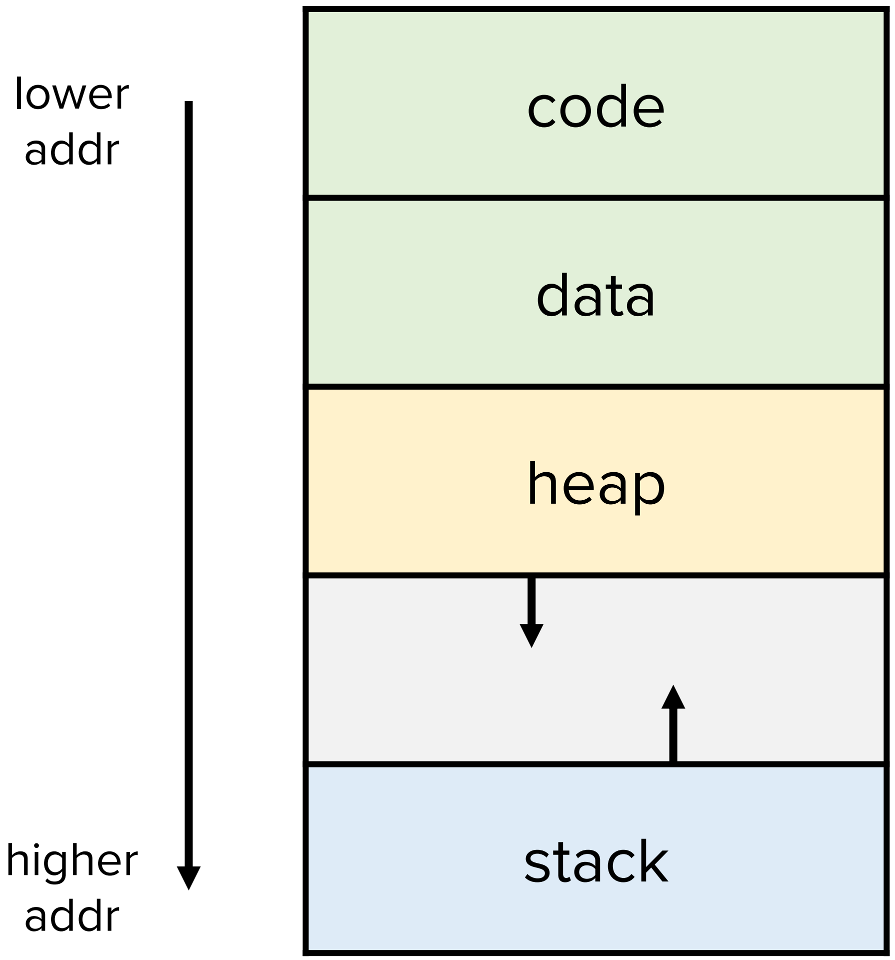
Address Space Layout Randomization

- 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, and ROP gadget addresses can be easily identified
  - Attackers can easily launch ret-to-libc or ROP attacks

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

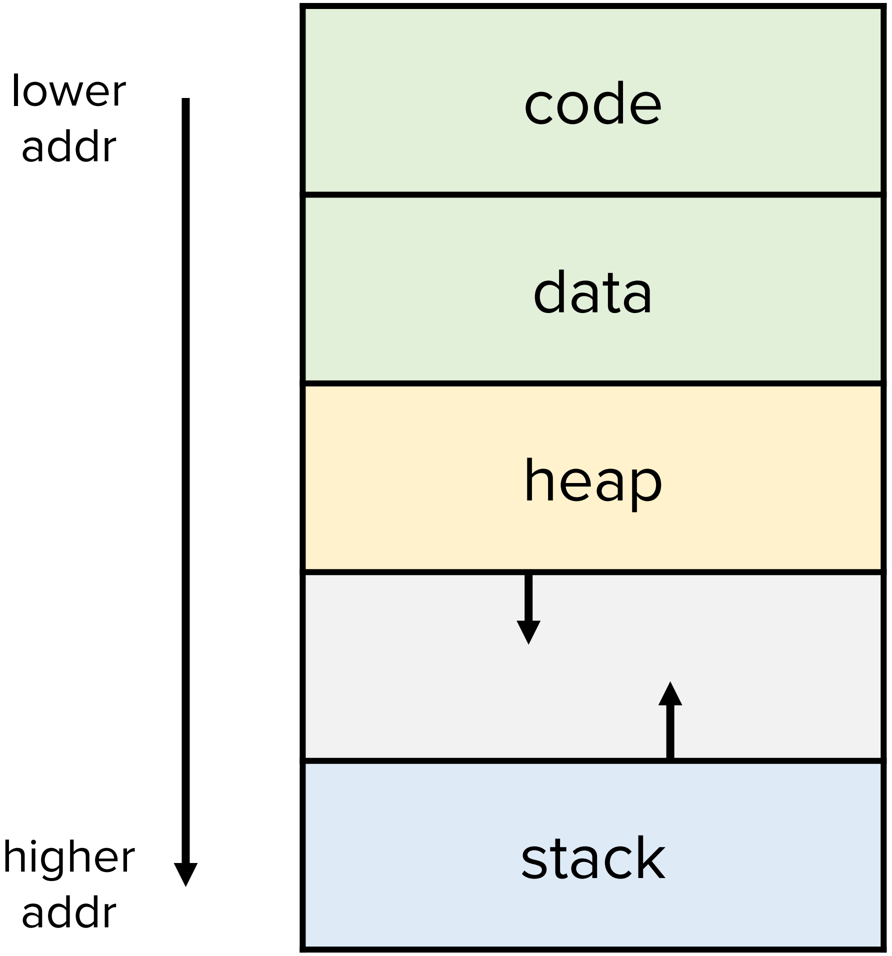
Recall: Process memory layout

- In theory: Packed

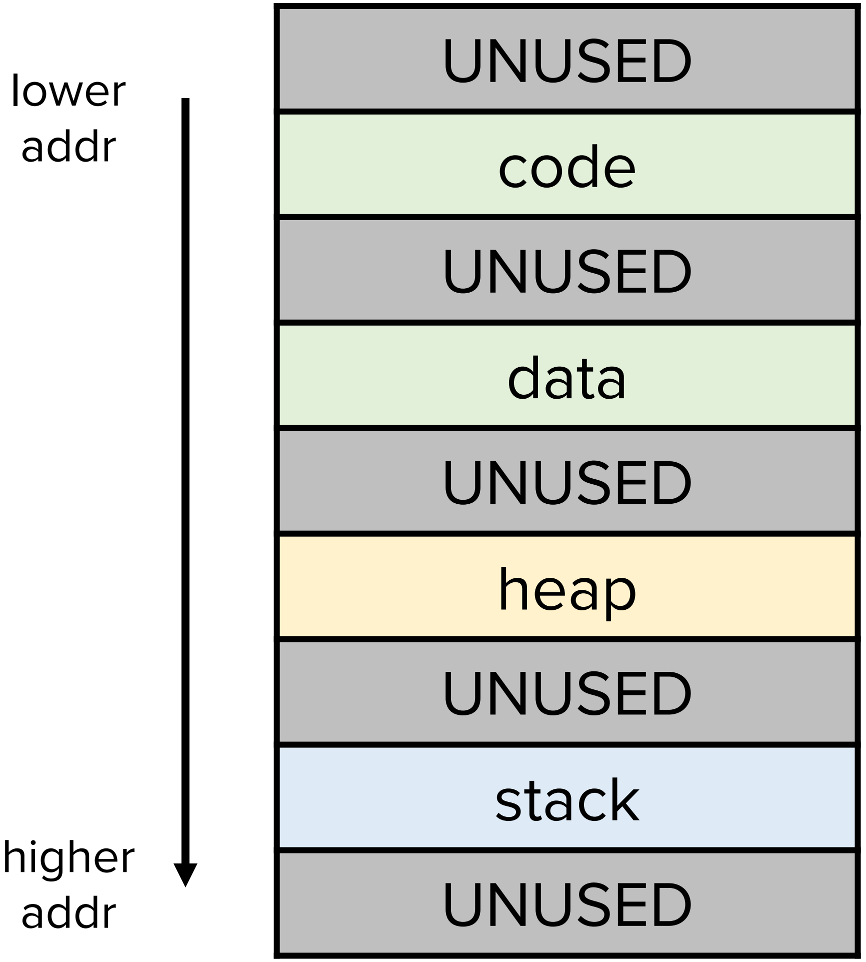


Recall: Process memory layout

- In theory: Packed

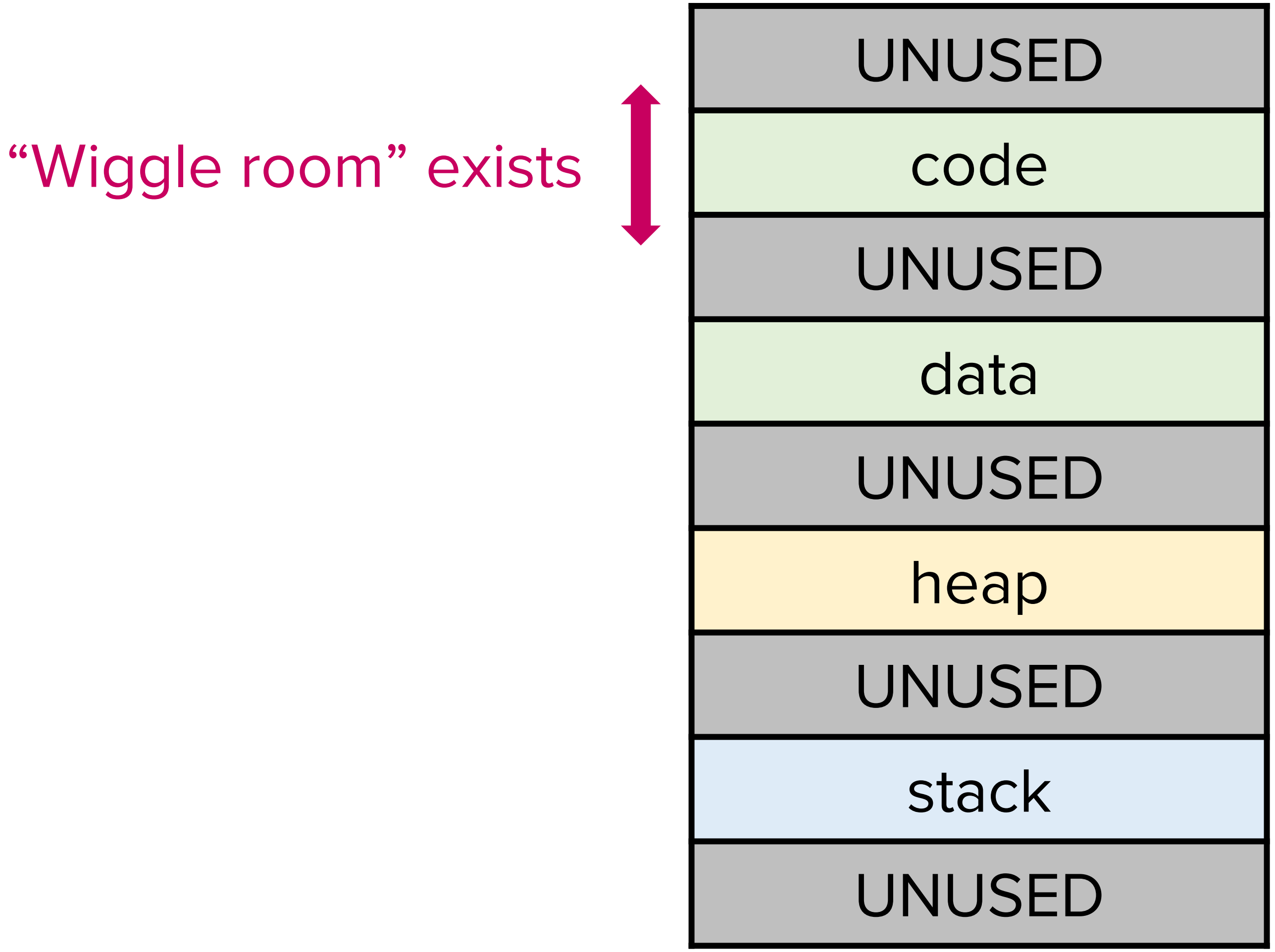


- In practice: Mostly empty

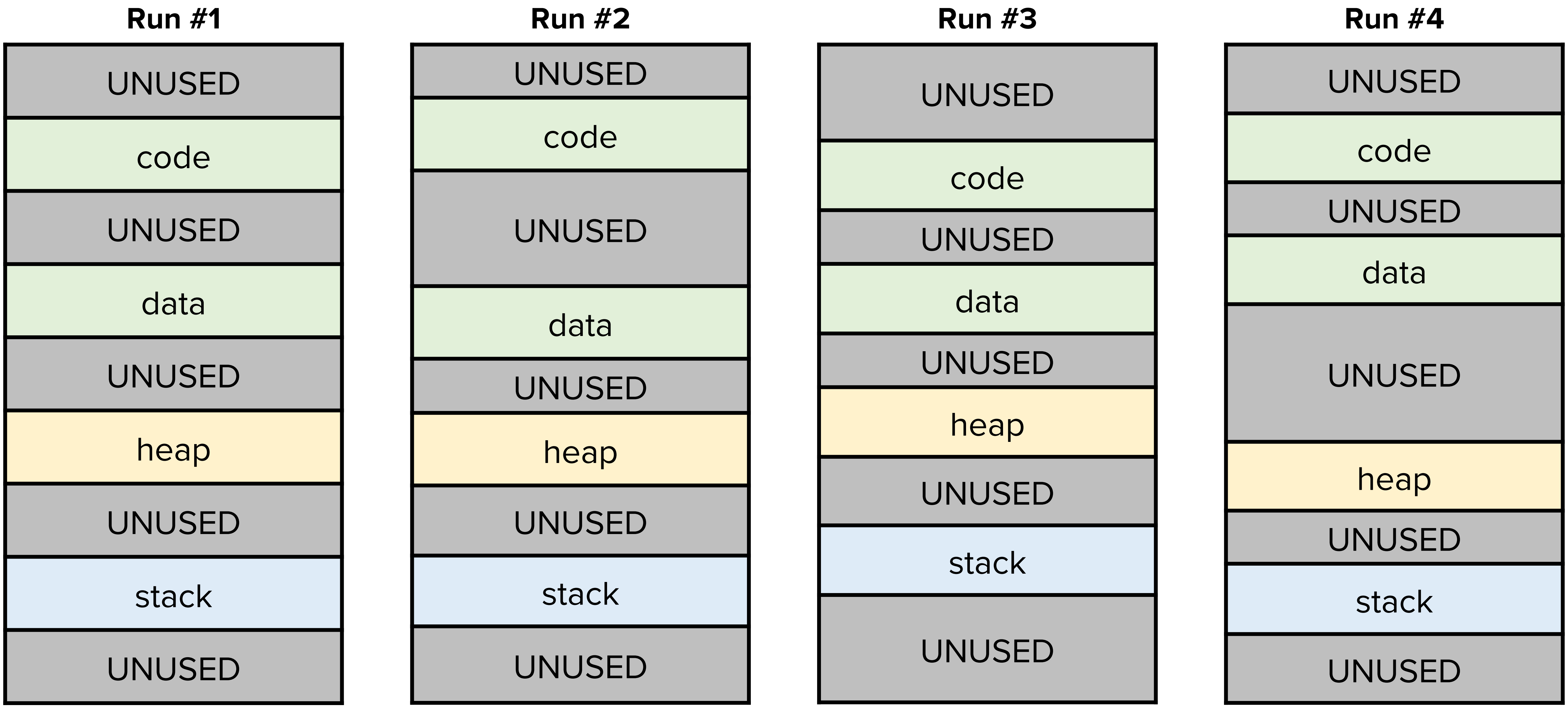


Recall: Process memory layout

- In practice: Mostly empty



Idea: Load each segment at different address



ASLR can shuffle all four segments

- Randomized stack:
  - Can’t return to shellcode on stack without knowing the address of the stack buffer or environment variable
- Randomized heap:
  - Can’t return to shellcode on the heap without knowing chunk’s addr
- Randomized code:
  - Can’t ret-to-libc without knowing libc function addresses (e.g., system)
- Randomized data:
  - Can’t reuse existing data (e.g., do not know the address of /bin/sh)

Checking ASLR – Try it yourself

```c
#include <stdio.h>

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

Code to print stack variable's address

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

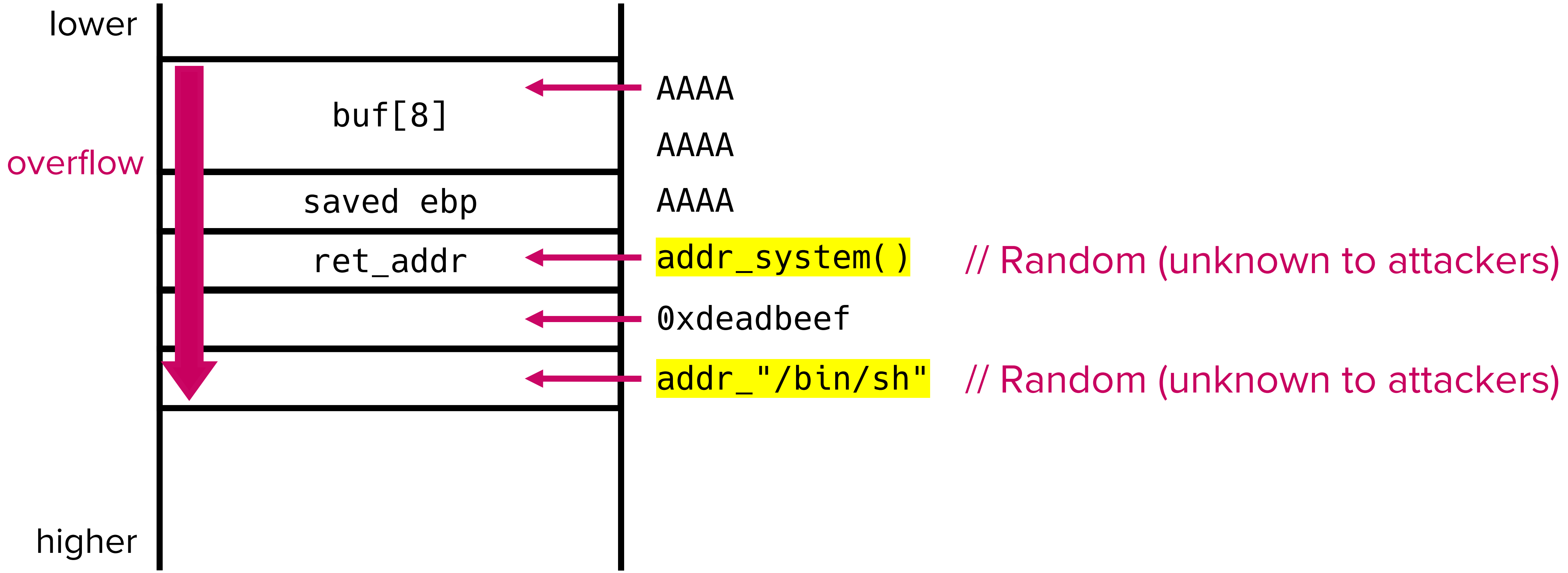
Compilation

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

Stack address is different across executions

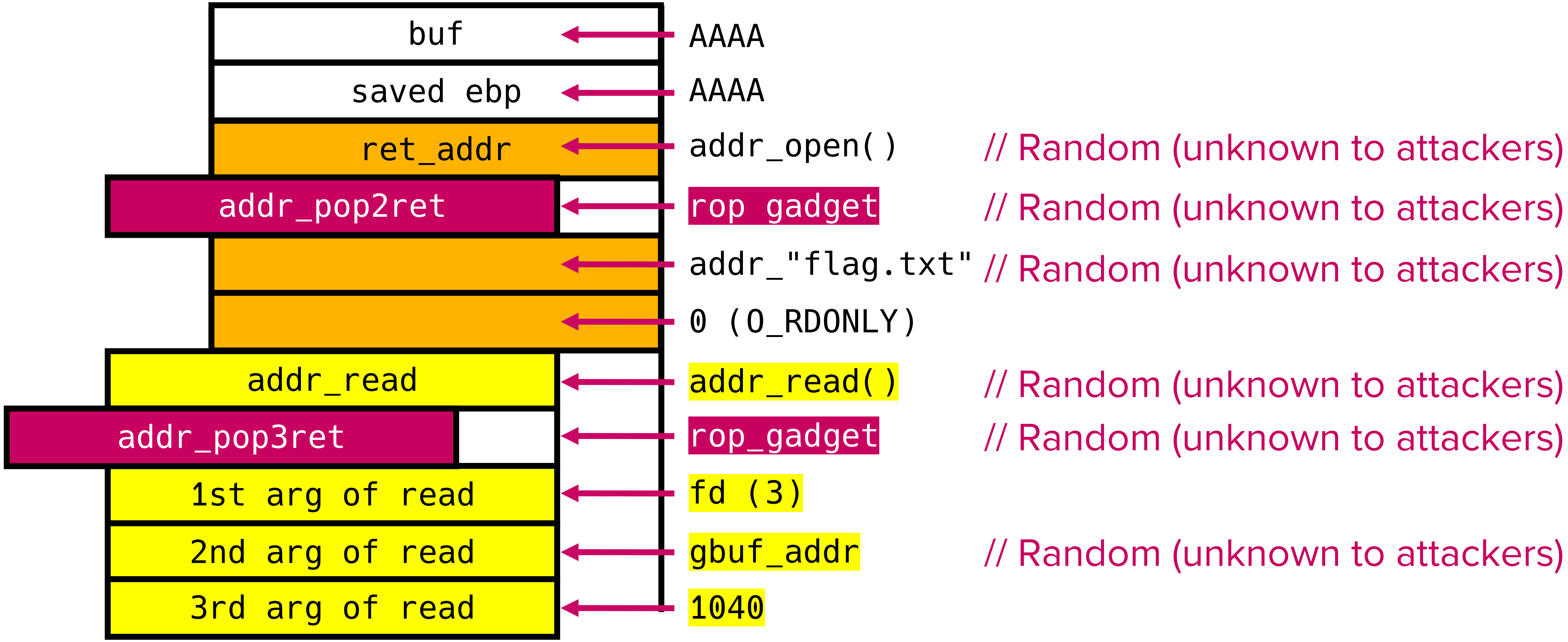
The effectiveness of ASLR

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



The effectiveness of ASLR

- ASLR prevents ROP attacks
  - Similarly, cannot do ROP without knowing addresses



The effectiveness of ASLR

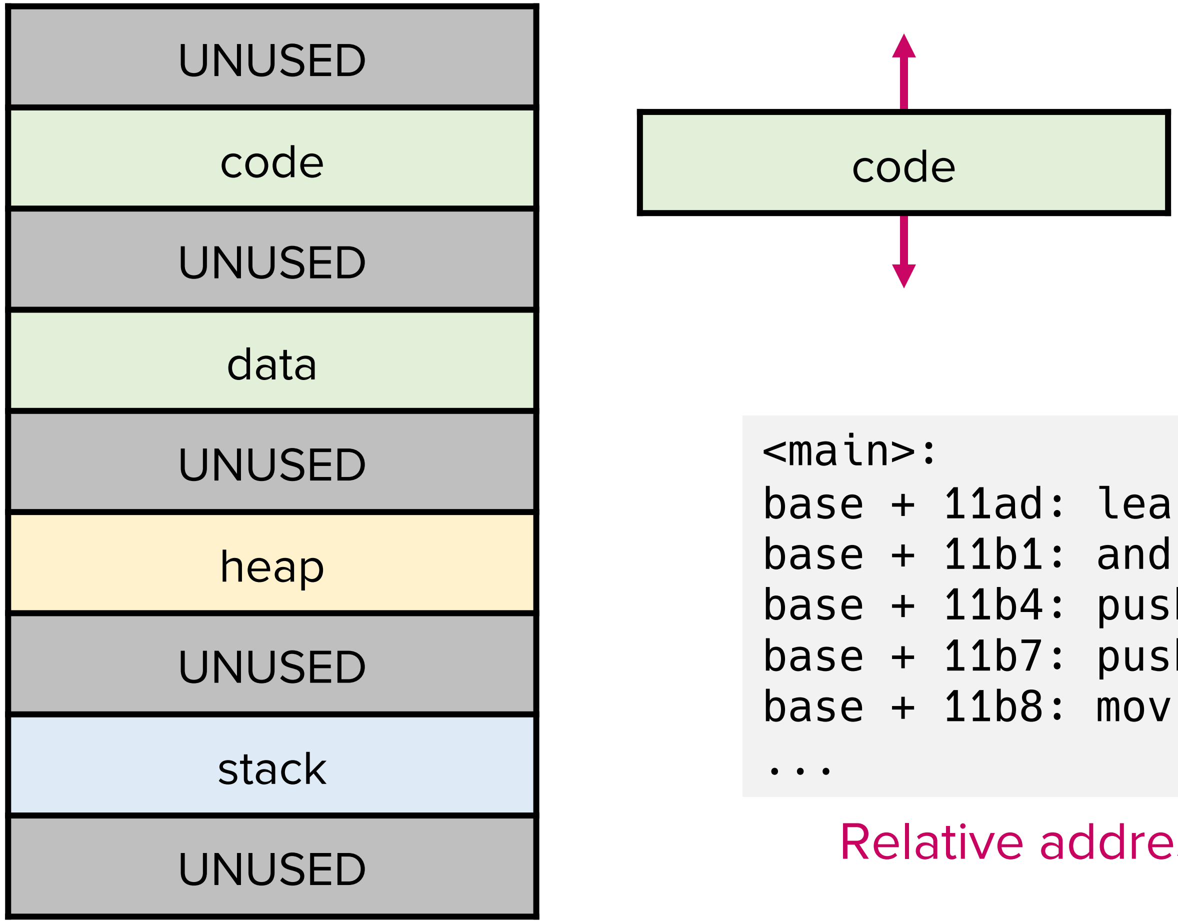
- ASLR prevents ROP attacks
  - Similarly, cannot do ROP without knowing addresses

buf	AAAA	
saved_ebp	AAAA	
addr_read	addr_read()	// Random (unknown to attackers)
addr_pop3ret	rop_gadget	// Random (unknown to attackers)
1st arg of read	fd (3)	
2nd arg of read	gbuf_addr	// Random (unknown to attackers)
3rd arg of read	1040	

So, are we safe now?

Subverting ASLR (1)

- ASLR only randomizes the base address of segments

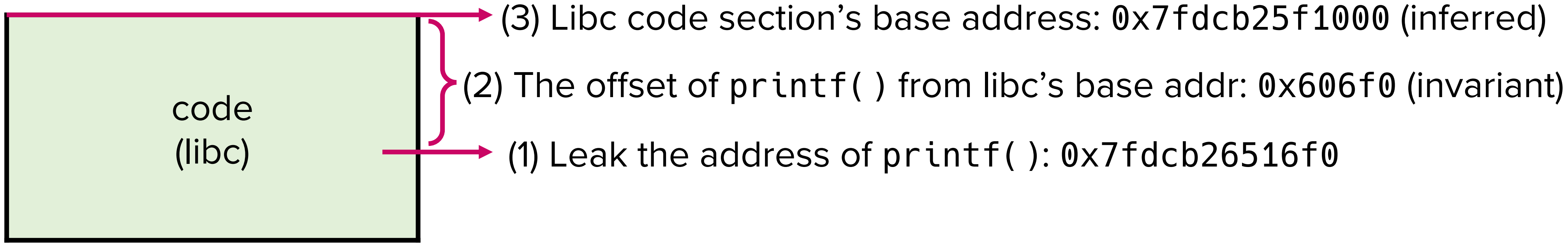



```
<main>:
base + 11ad: lea  ecx,[esp+0x4]
base + 11b1: and  esp,0xfffffff0
base + 11b4: push DWORD PTR [ecx-0x4]
base + 11b7: push ebp
base + 11b8: mov  ebp,esp
...
```

Relative addresses (offsets) are fixed!

Subverting ASLR (1)

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



- Then, the runtime addresses of other functions within the same segment can be inferred
  - e.g., Offset of `"/bin/sh"` in x86_64 libc: **`0x1d8678`**

    → `inferred_libc_base_addr` + `0x1d8678` = `addr_binsh`

Subverting ASLR (1)

- If the address of a pointer within a segment is leaked

Q) How can we leak code pointers?
→ Advanced topic. Take CSED702C!

(libc)

(1) Leak the address of `printf()`: `0x7fdcb26516f0`

asdf

Q) Why does ASLR only randomize segments' base address?
→ To ensure reasonable performance!

→ `inferred_libc_base_addr + 0x1d8678 = addr_binsh`

Subverting ASLR (2)

- Entropy (randomness) of ASLR on x86 Linux is small
  - x86 only randomizes 16 bits (out of 32 bits) of base address
    - There are only $2^{16} = 65536$ possible addresses for a function
    - Brute-forcible (Pick an address and repeat until the attack works)

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 (odorless, colorless)
- 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

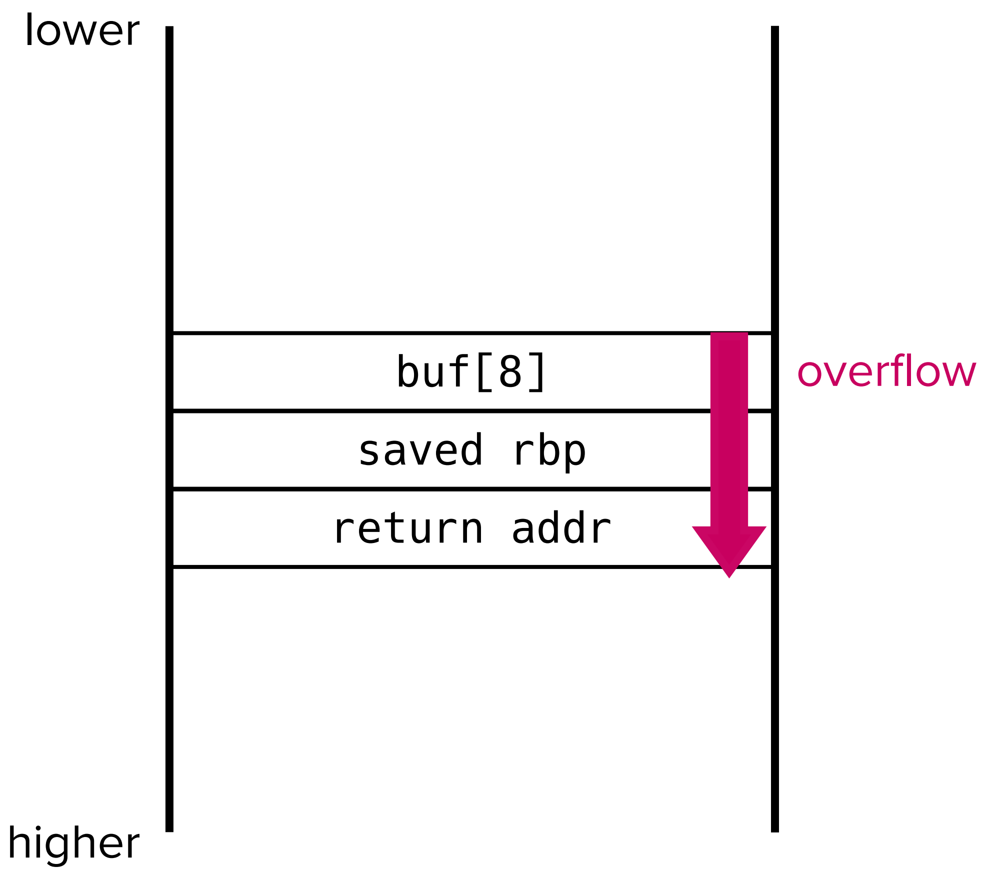
img: Times Higher Education

Canaries as stack protectors

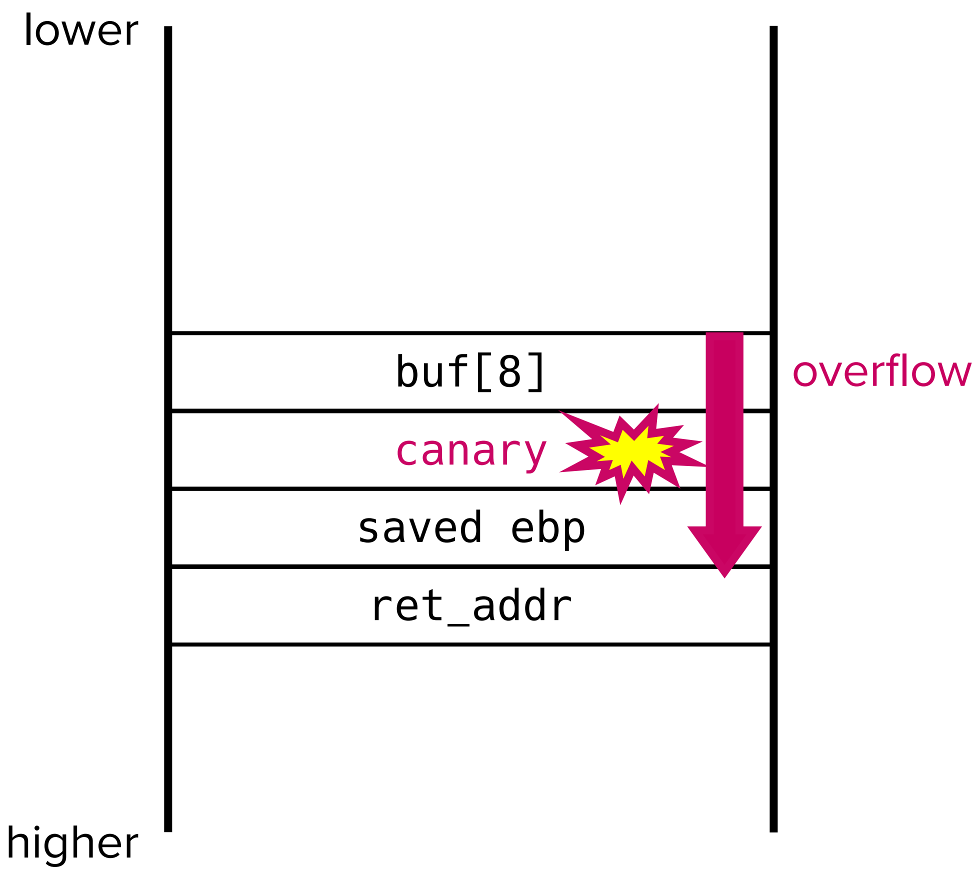
- Workflow of stack smashing attacks:
  - Overwrite the return address of a function
  - Wait for the function to complete and execute **RET** instruction
- Idea for protection:
  - Insert a canary value to the stack between the local variables and return address
    - To overwrite the return address, 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 the canary is dead

Stack diagram (x86_64)

- Without a canary



- With a canary

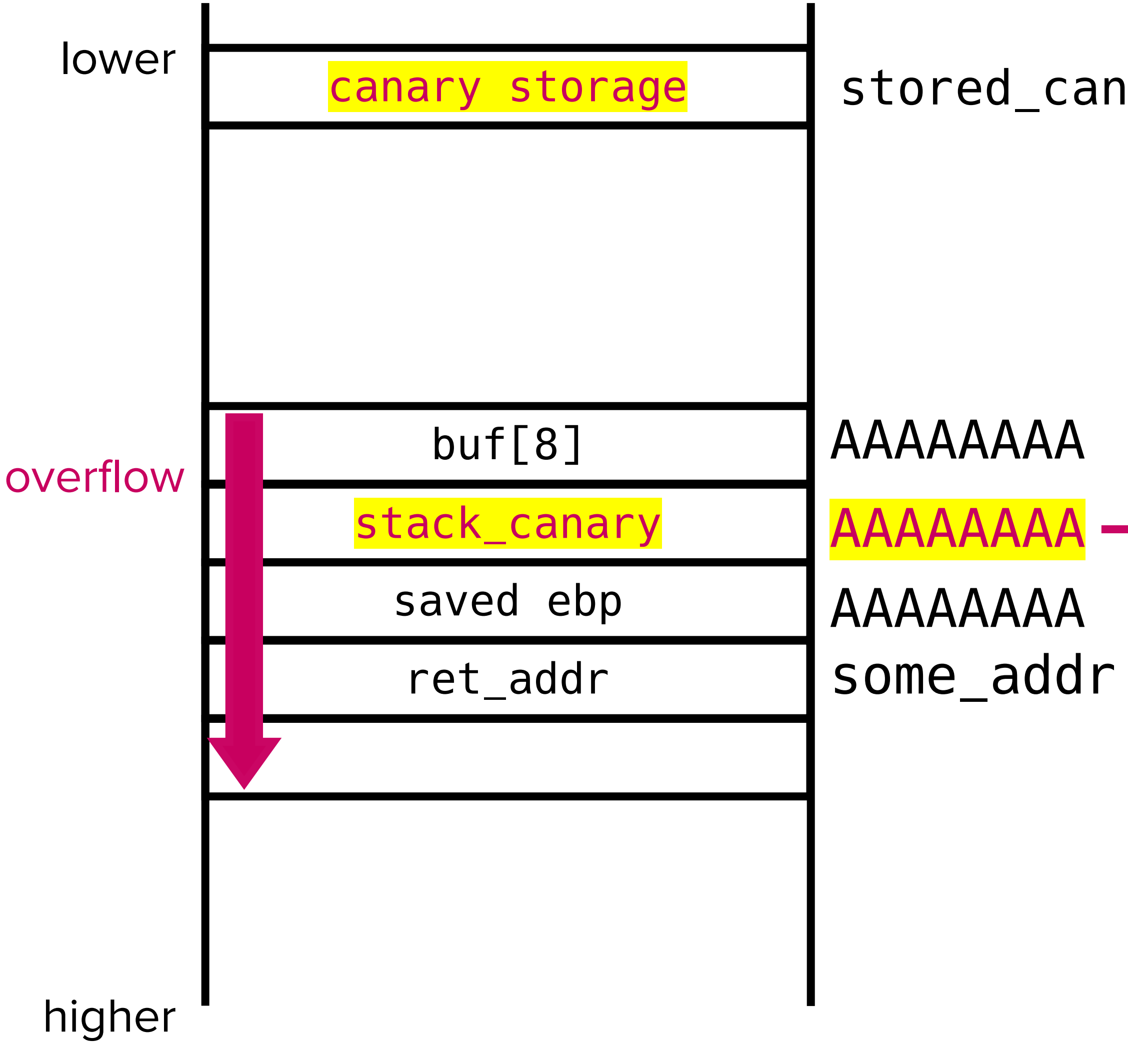


Canary mechanism

- 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 frame pointer 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 mechanism



Check if `stack_canary == stored_canary` before executing `ret`

Canary has been modified!
Stack is corrupted. Abort.

Stack canary in practice

- Implementing 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;
  char buf[8];
  memcpy(buf, argv[1], 64);
  if (canary != CANARY)
    exit(-1);
  return 0;
}
```

This is not what we do in practice.
It is not desirable to manually do this for every function of a software

Stack canary in practice

- 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

(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 aaaa
[1]    2304931 segmentation fault  ./without-canary aaaa
```

→ 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 aaaa
```

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

Subverting stack canary (1)

- Leak the canary's value
  - Exploit any vulnerability that can reveal stack memory
    - Examples:
      - The HeartBleed vulnerability allows attackers to read arbitrary memory
      - Format string bugs allows printing stack values at chosen addresses
  - Once the canary is leaked, overwrite the canary with the leaked value

Subverting stack canary (2)

- Guess the canary's value
  - The least significant byte (LSB) of canary is always `0x00` // Q) Why?
    - On x86 (32-bit) systems, this leaves 24 bits (out of 32 bits) to guess
    - There are 2^{24} (~16 million) possibilities, which is feasible for brute force
  - Once successfully guessed, overwrite the canary with the correct value
    - Q) Doesn't canary value change each time a process is run?
    - A) In many server applications, the process is forked per request, so the canary (inherited from the parent process) remains the same across child processes

Subverting stack canary (3)

- Bypass the canary
  - Stack canary can only mitigate sequential writes (e.g., via `strcpy()`)
  - Some vulnerabilities (e.g., format string bugs) allow you to do "arbitrary writes"
    - An attacker can write to any memory address directly
  - You can overwrite the return address without altering the stack canary!

Section 1 of CSED415 is over

- 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 trae-off between cost and effectiveness

Primary focus so far: System Integrity (ref: Lecture 02)
“A system performs its intended function in an unimpaired manner.”

Question: How can we preserve Confidentiality?

Coming up next

- Cryptographic primitives and their applications

Questions?