

ECE 382N-Sec (FA25):

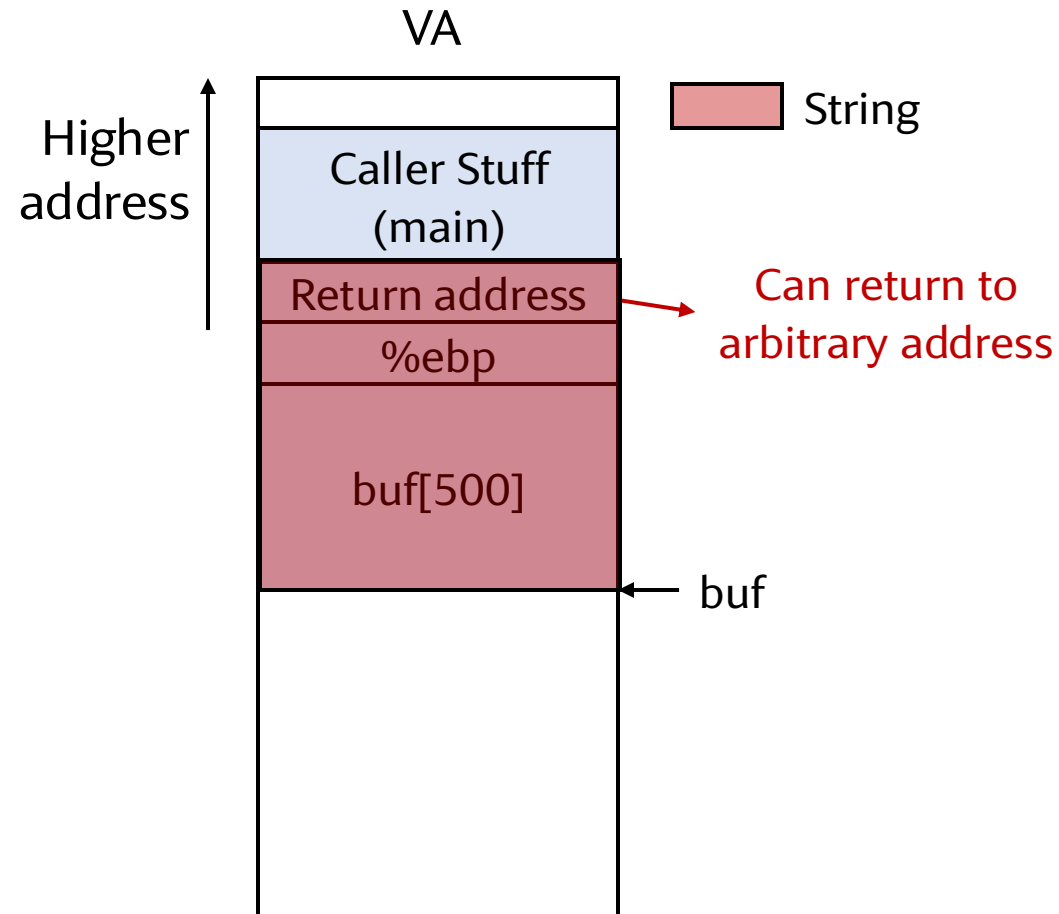
# L10: Memory Safety

Neil Zhao

neil.zhao@utexas.edu

# Buffer Overflow Attacks

<https://www.youtube.com/watch?v=1S0aBV-Waeo>



```
int main(int argc, char **argv) {  
    char buf[500];
```

```
    strcpy(buf, argv[1]);  
    return 0;
```

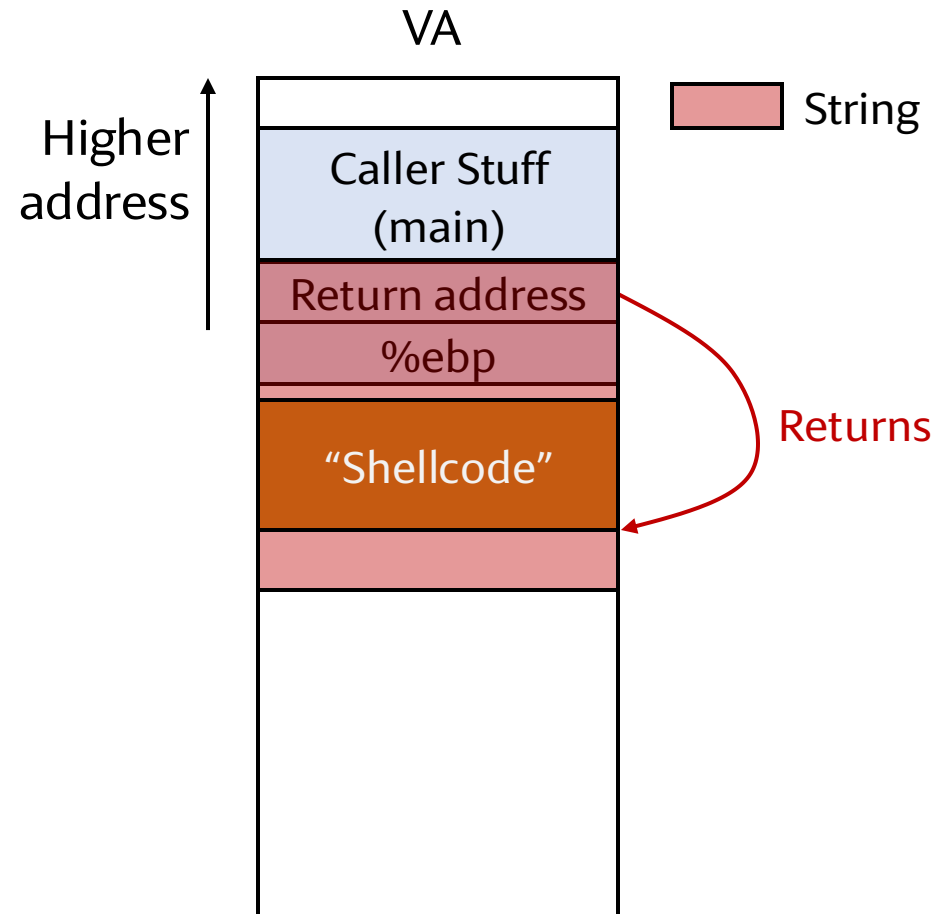
```
}
```

No bounds check

argv[1] is under the attacker's control:  
Set argv[1] to "a long long string ..."

# Buffer Overflow Attacks

<https://www.youtube.com/watch?v=1S0aBV-Waeo>



```
int main(int argc, char **argv) {  
    char buf[500];
```

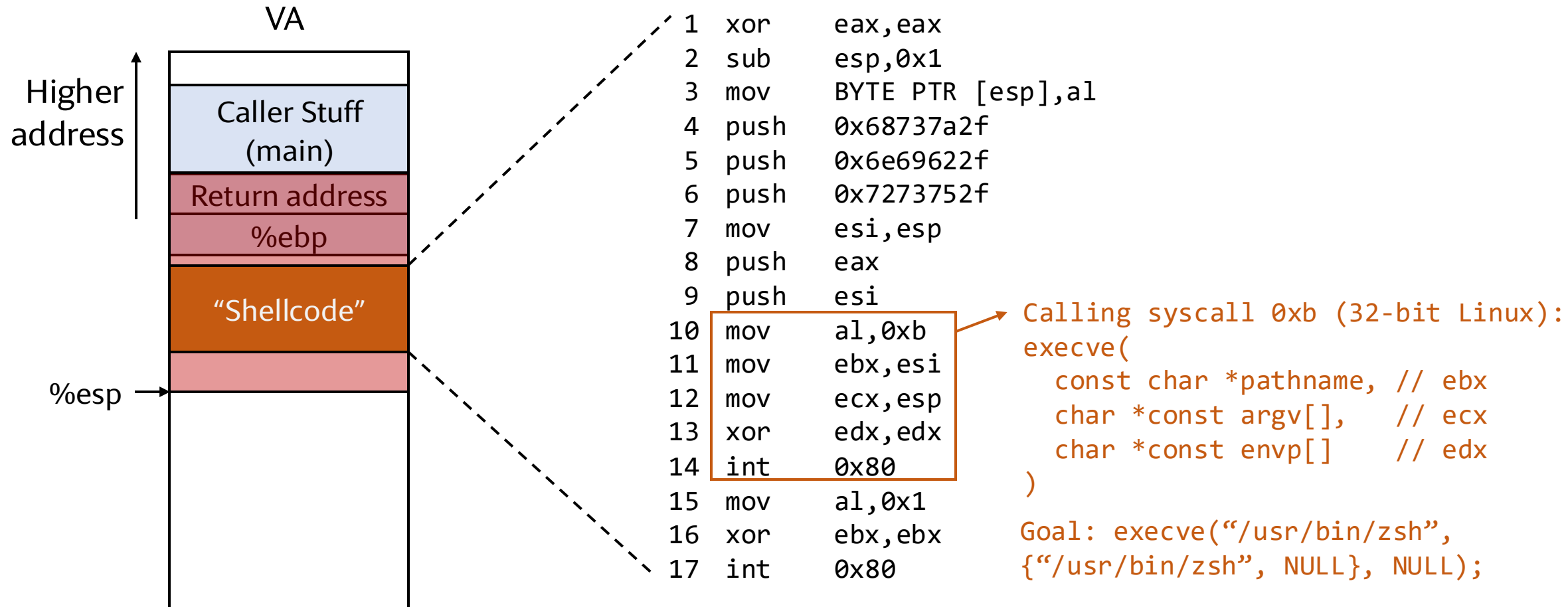
```
    strcpy(buf, argv[1]);  
    return 0;
```

```
}
```

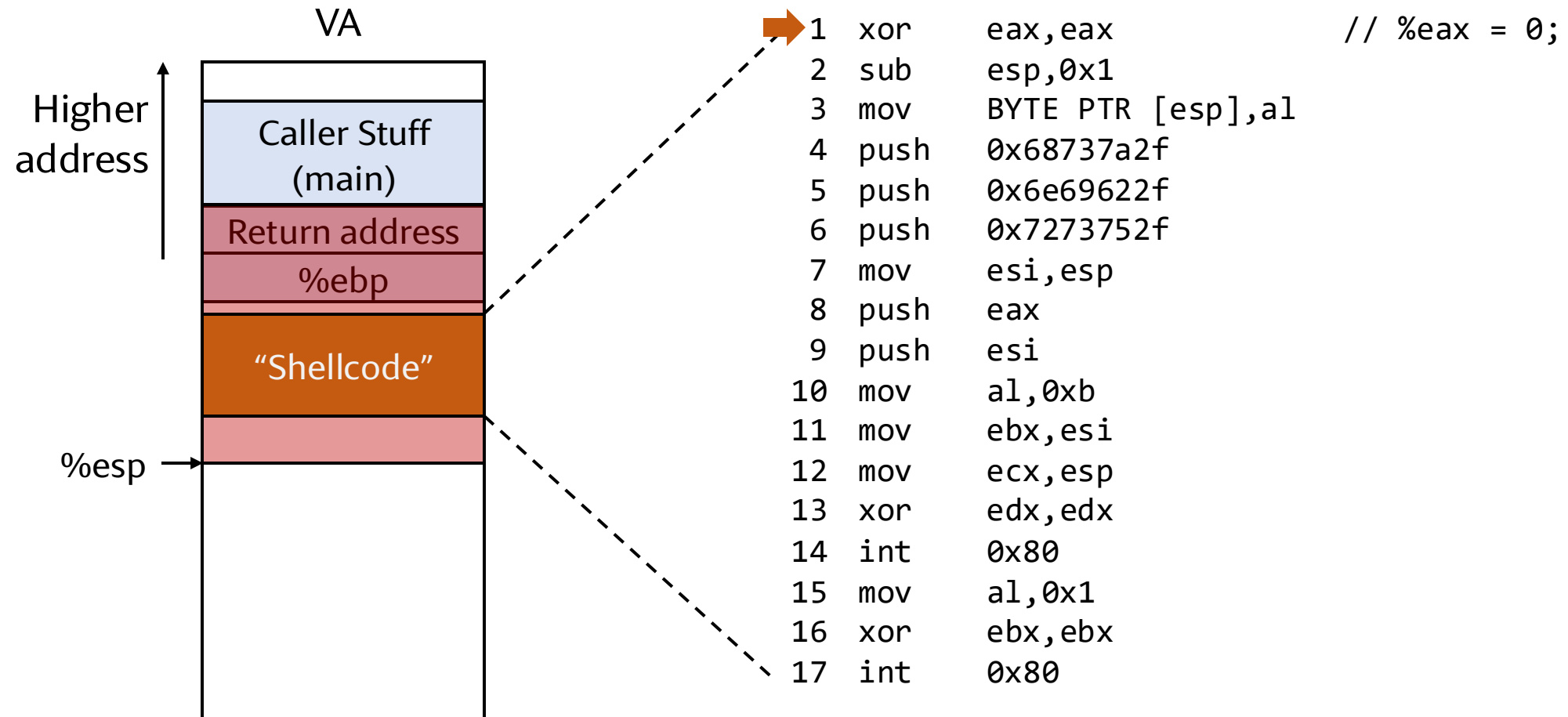
No bounds check

argv[1] is under the attacker's control:  
Set argv[1] to "a long long string ..."

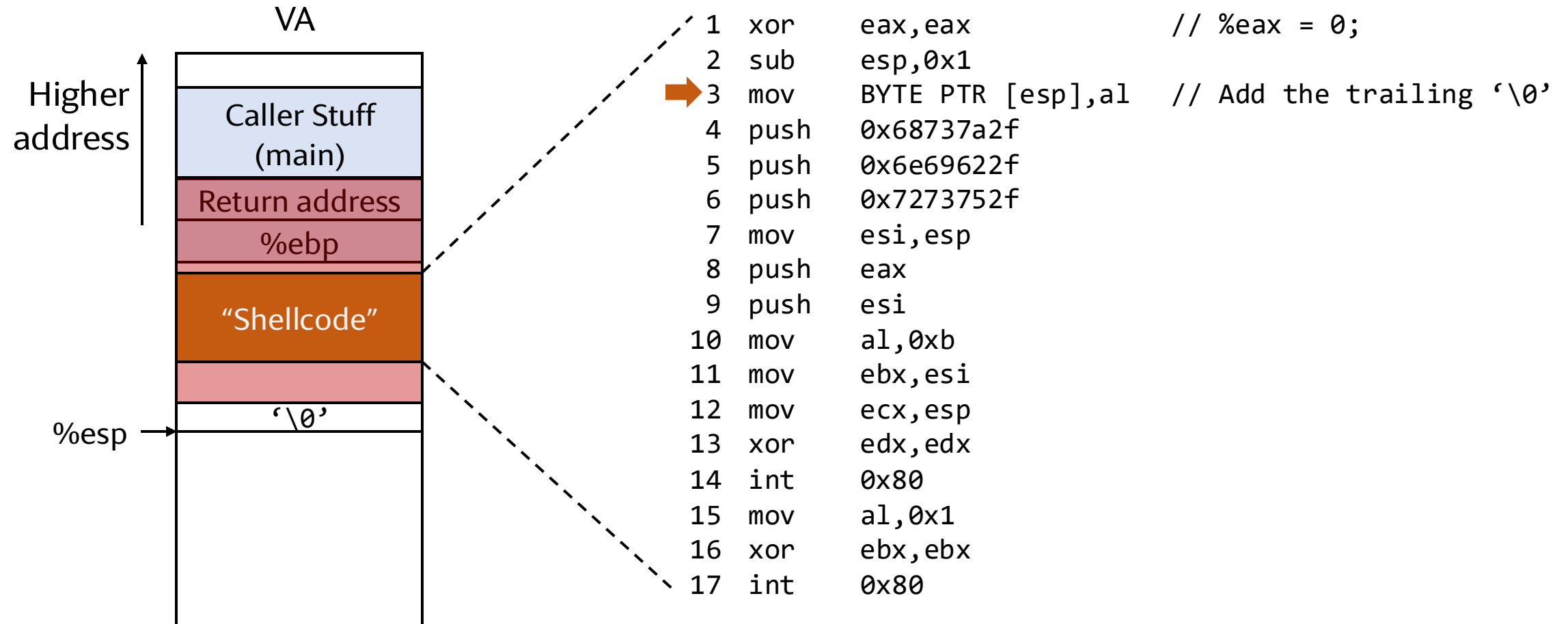
# The Shell Code



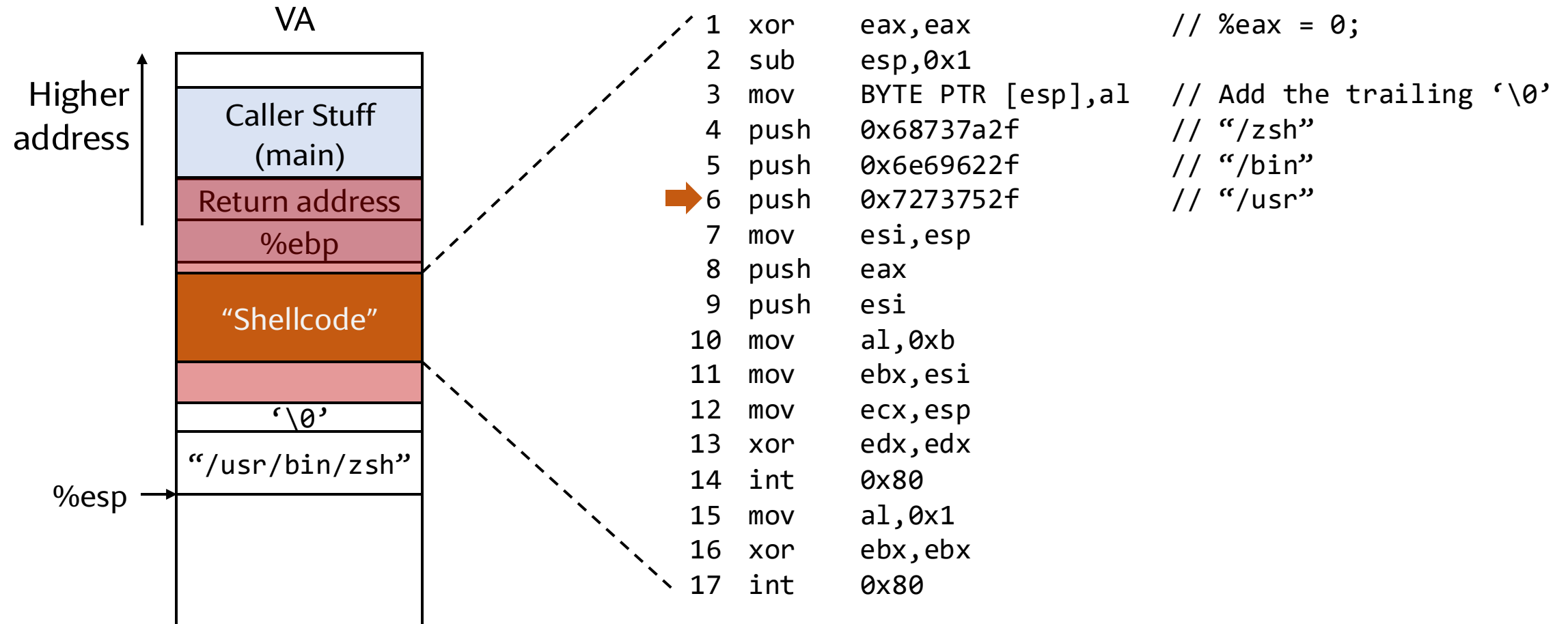
# The Shell Code



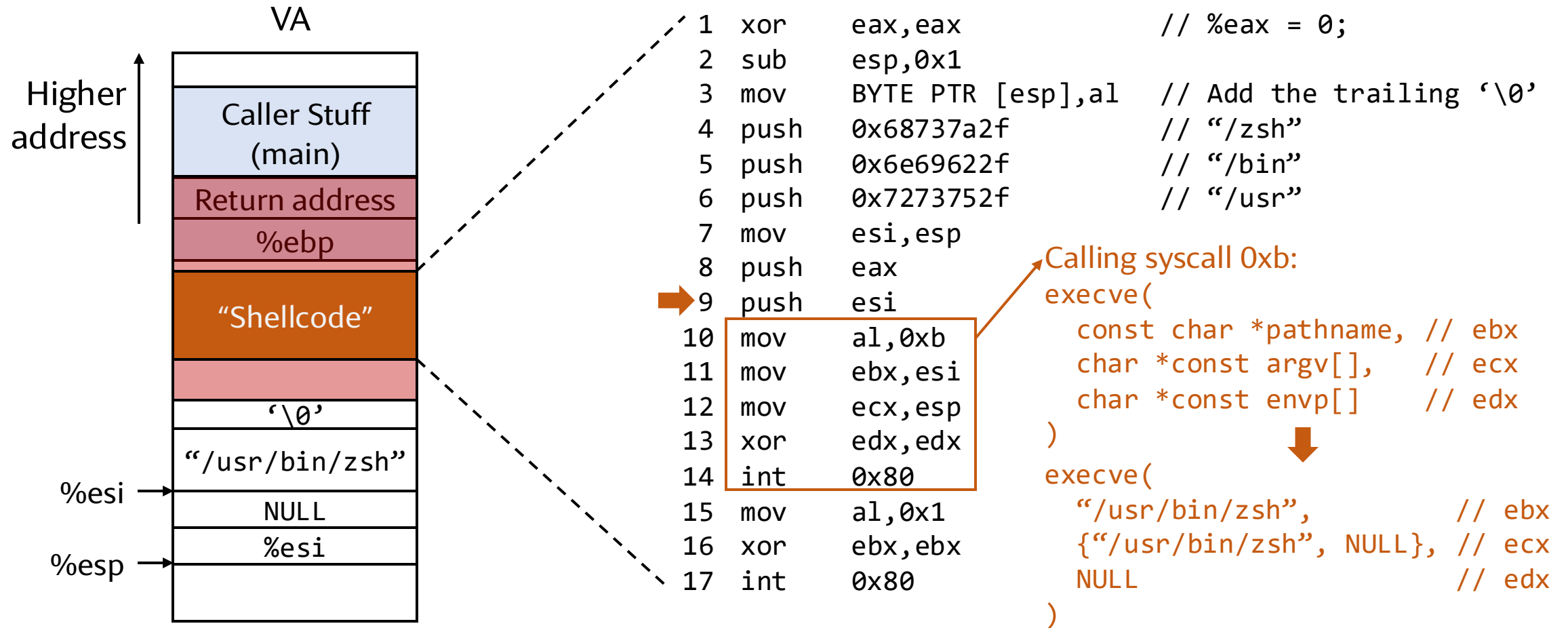
# The Shell Code



# The Shell Code



# The Shell Code





# The setuid Bit and the sudoedit Vulnerability (CVE-2021-3156)

setuid: Allows users to run a program with the privileges of the program's owner

```
neilz@meyer-lemon:~$ cat /etc/shadow
cat: /etc/shadow: Permission denied
neilz@meyer-lemon:~$ ls -l /etc/shadow
-rw-r----- 1 root shadow 1632 Oct  9 11:13 /etc/shadow
neilz@meyer-lemon:~$ ls -l $(which cat)
-rwxr-xr-x 1 root root 39384 Jun 22 11:21 /usr/bin/cat
neilz@meyer-lemon:~$ ls -l $(which passwd)
-rwsr-xr-x 1 root root 64152 May 30 2024 /usr/bin/passwd
```

# The setuid Bit and the sudoedit Vulnerability (CVE-2021-3156)

setuid: Allows users to run a program with the privileges of the program's owner

## CVE-2021-3156 Detail

### Description

Buffer overflow!

Sudo before 1.9.5p2 contains an off-by-one error that can result in a heap-based buffer overflow, which allows privilege escalation to root via "sudoedit -s" and a command-line argument that ends with a single backslash character.

### Metrics

CVSS Version 4.0

CVSS Version 3.x

CVSS Version 2.0

*NVD enrichment efforts reference publicly available information to associate vector strings. CVSS information contributed by other sources is also displayed.*

#### CVSS 3.x Severity and Vector Strings:



NIST: NVD

Base Score: 7.8 HIGH

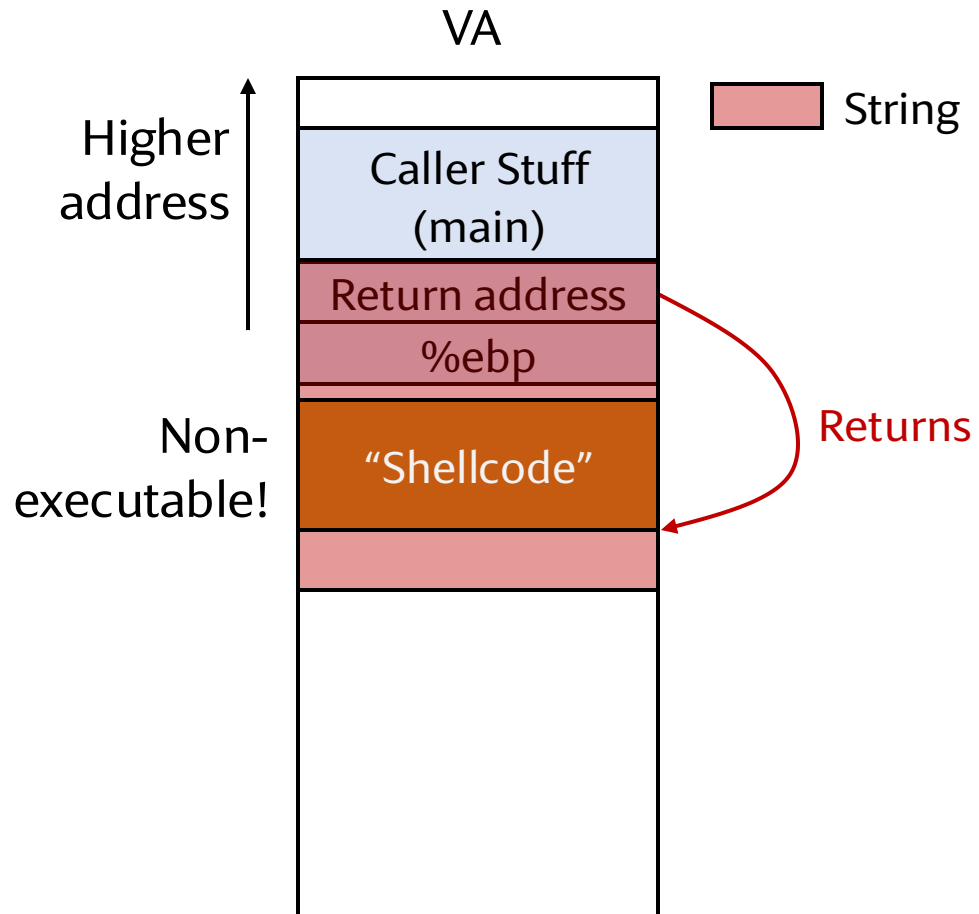
Vector: CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H

ADP: CISA-ADP

Base Score: 7.8 HIGH

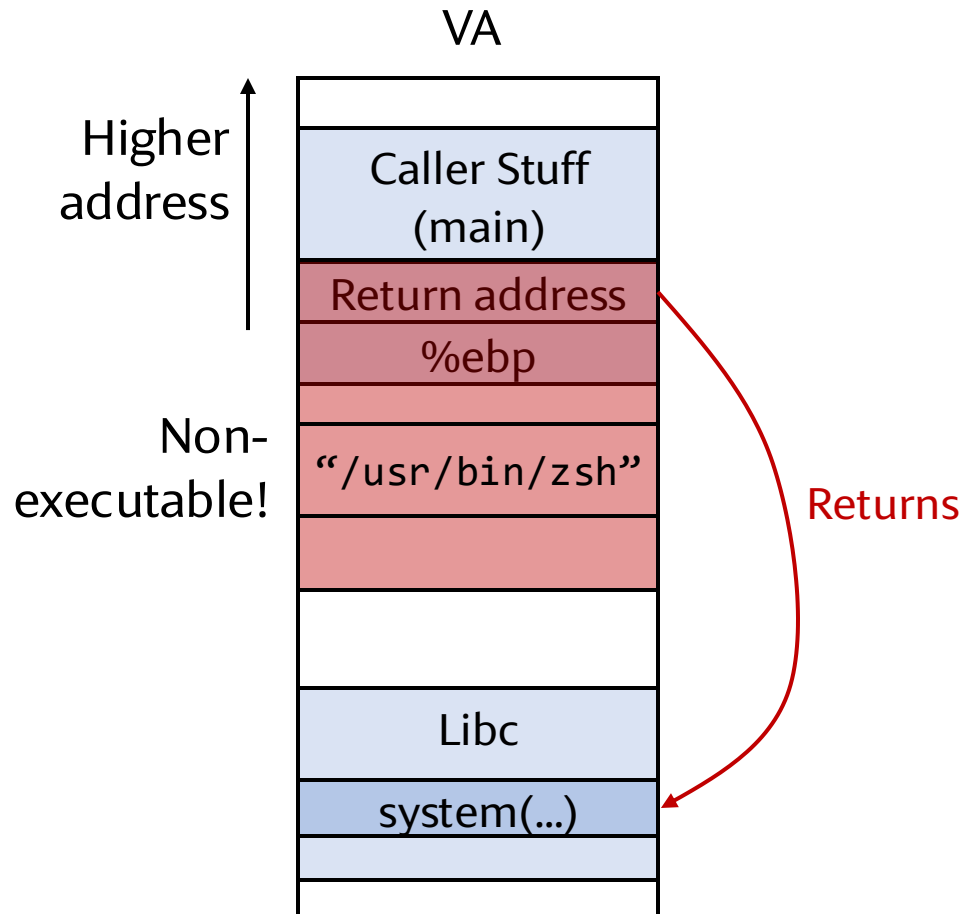
Vector: CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H

# Mitigation 1: Data Execution Prevention (DEP)



**Policy:** a page is either writable or executable, but not both! (i.e.,  $W \oplus X$ )

# Code-Reuse Attack: Return-to-Library



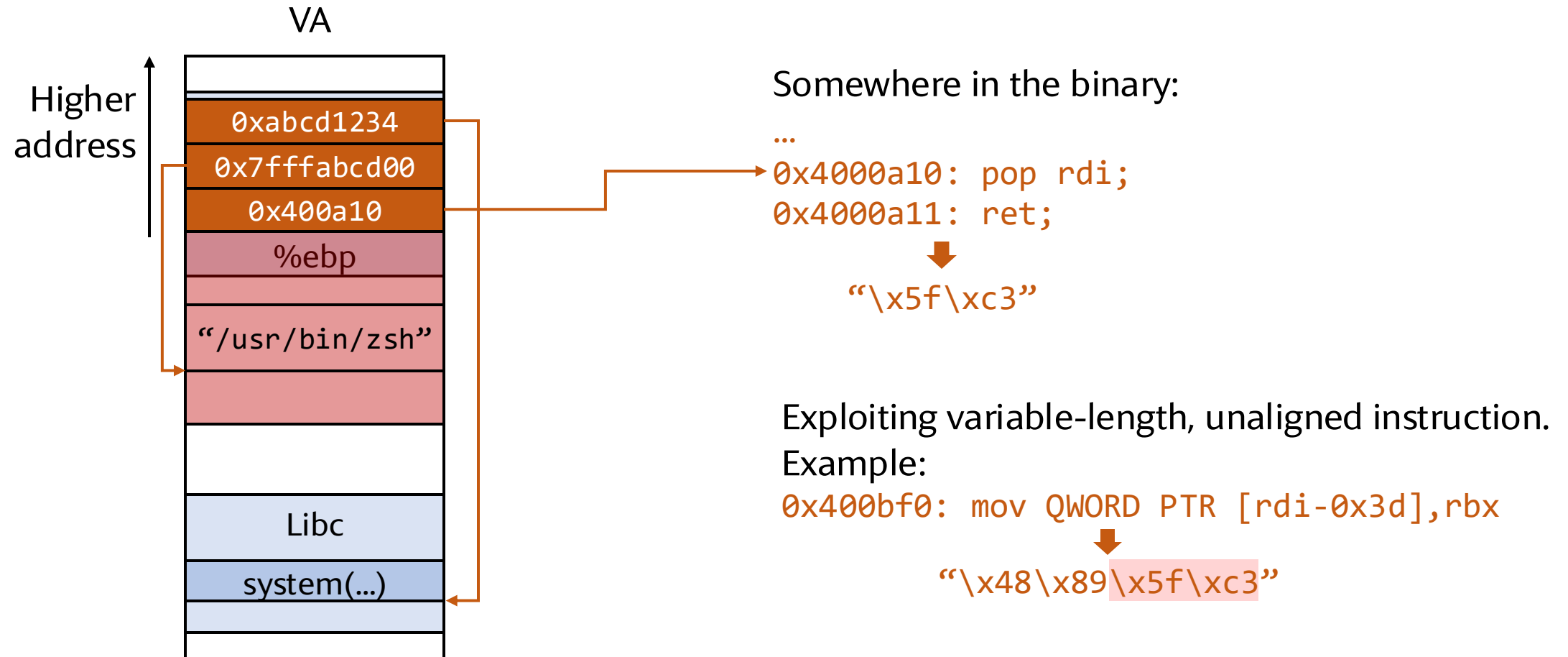
Libc "system" function: executes a shell command  
`int system(const char *command);`

Goal:

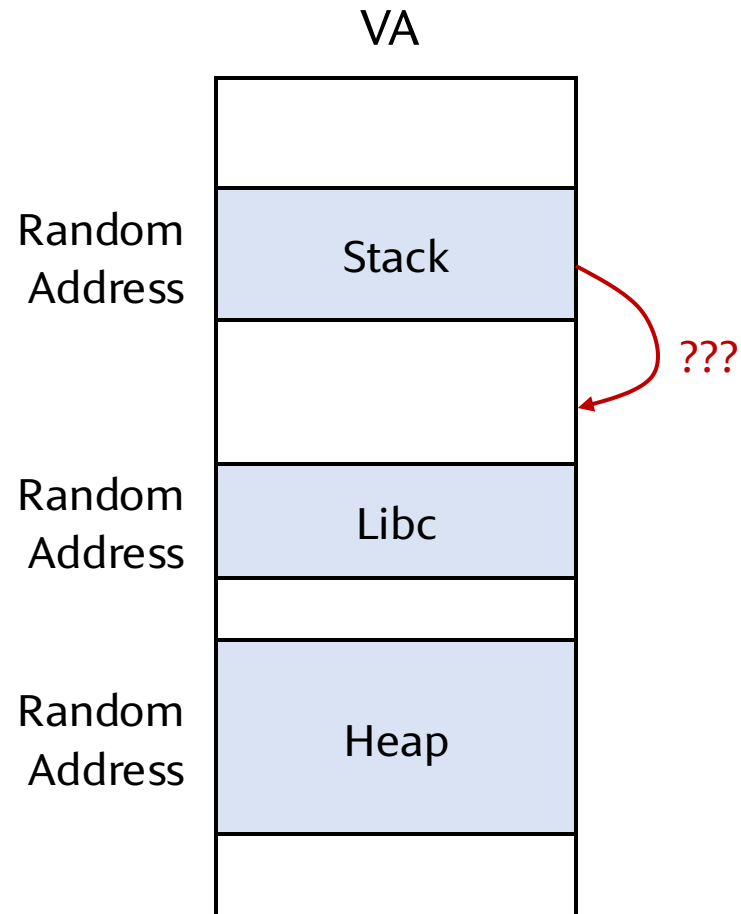
`system("/usr/bin/zsh");`

**Problem:** The first argument of the "system" function is passed through register `%rdi`, which is not under attacker's control

# Code-Reuse Attack: Return-Oriented Programming (ROP)



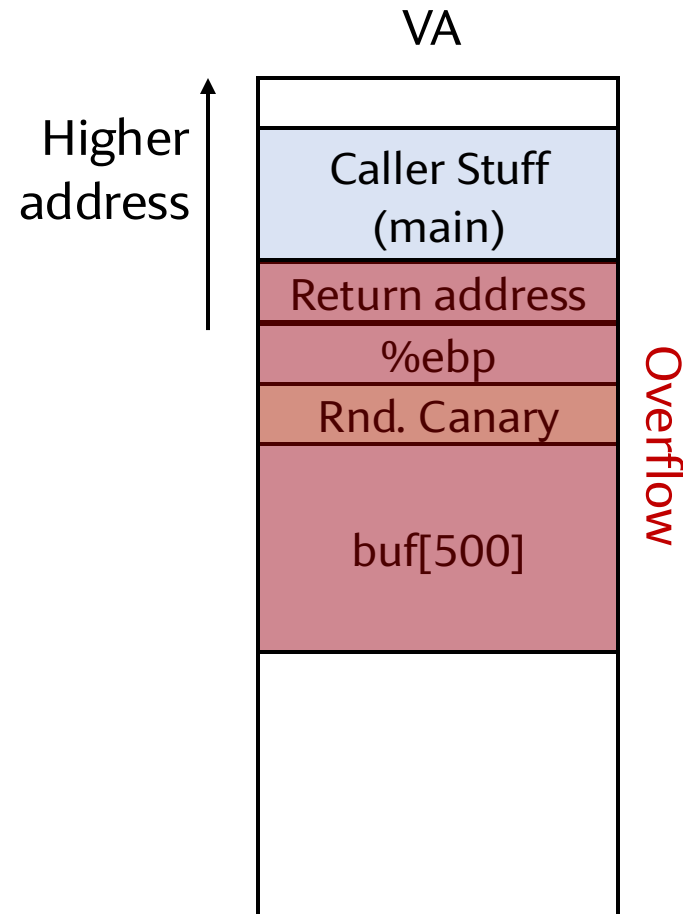
# Mitigation 2: Address Space Layout Randomization



**Approach:** allocate the stack, heap, and libraries at a random address before each execution

# Mitigation 3: Control-Flow Integrity (CFI)

Protecting returns: Stack canary

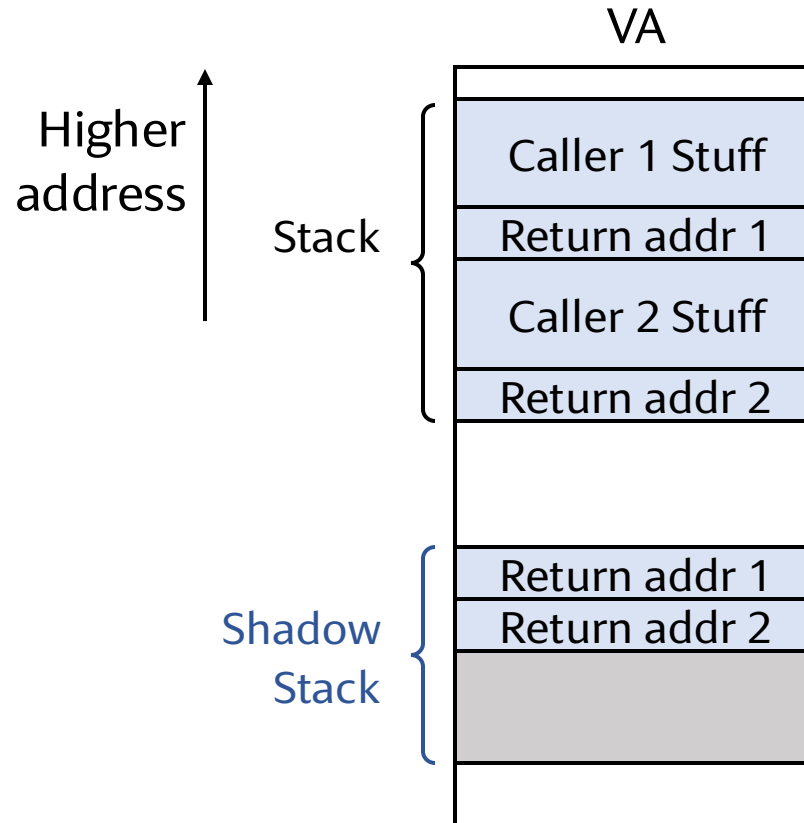


**Approach:** put a random (64-bit) canary value between the return address & %ebp and the current stack frame  
⇒ Check if the canary is modified before returning

The canary is a global random constant for every process or thread

# Mitigation 3: Control-Flow Integrity (CFI)

Hardware support: Intel Control-flow Enforcement Technology (CET)



- When executing a “call” instruction, the CPU saves a copy of the return address in a shadow stack
- The shadow stack is NOT writable by normal stores
- When the callee returns, the CPU pops the saved address from the shadow stack and compares it against the address popped from the stack
  - Mismatch  $\Rightarrow$  CPU raises a fault

What about indirect jumps/calls?

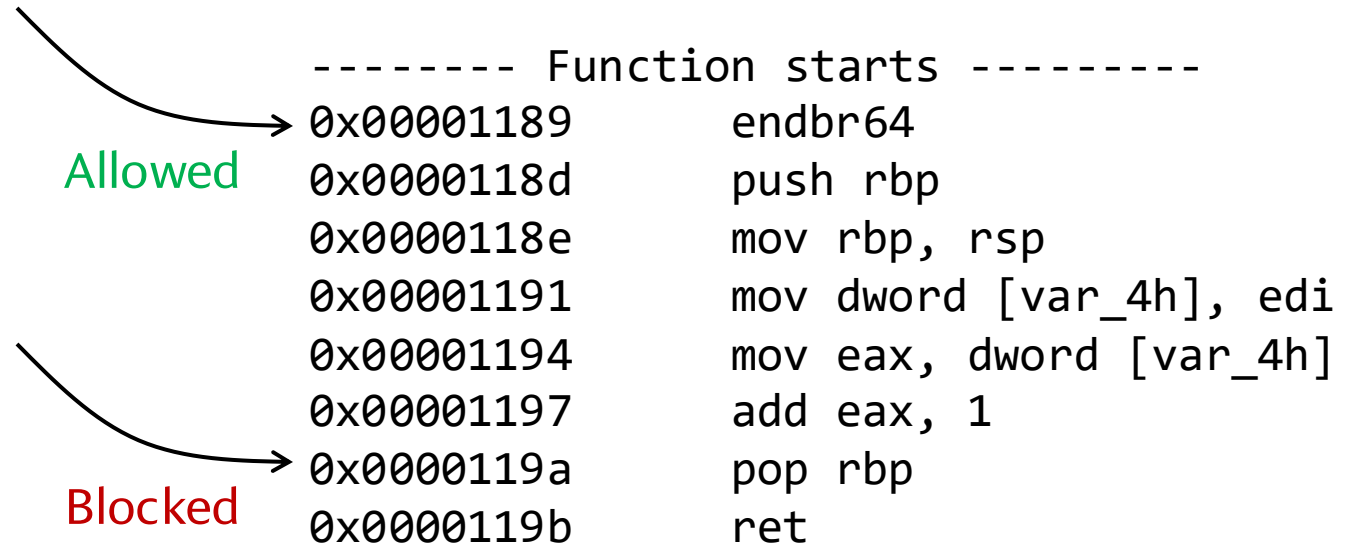


# Mitigation 3: Control-Flow Integrity (CFI)

Hardware support: Intel Control-flow Enforcement Technology (CET)

## Indirect Branch Tracking:

- New instruction “endbr”
  - 32-bit: endbr32
  - 64-bit: endbr64
- Indirect jumps and calls must land on an “endbr” instruction



# Spatial Memory Safety

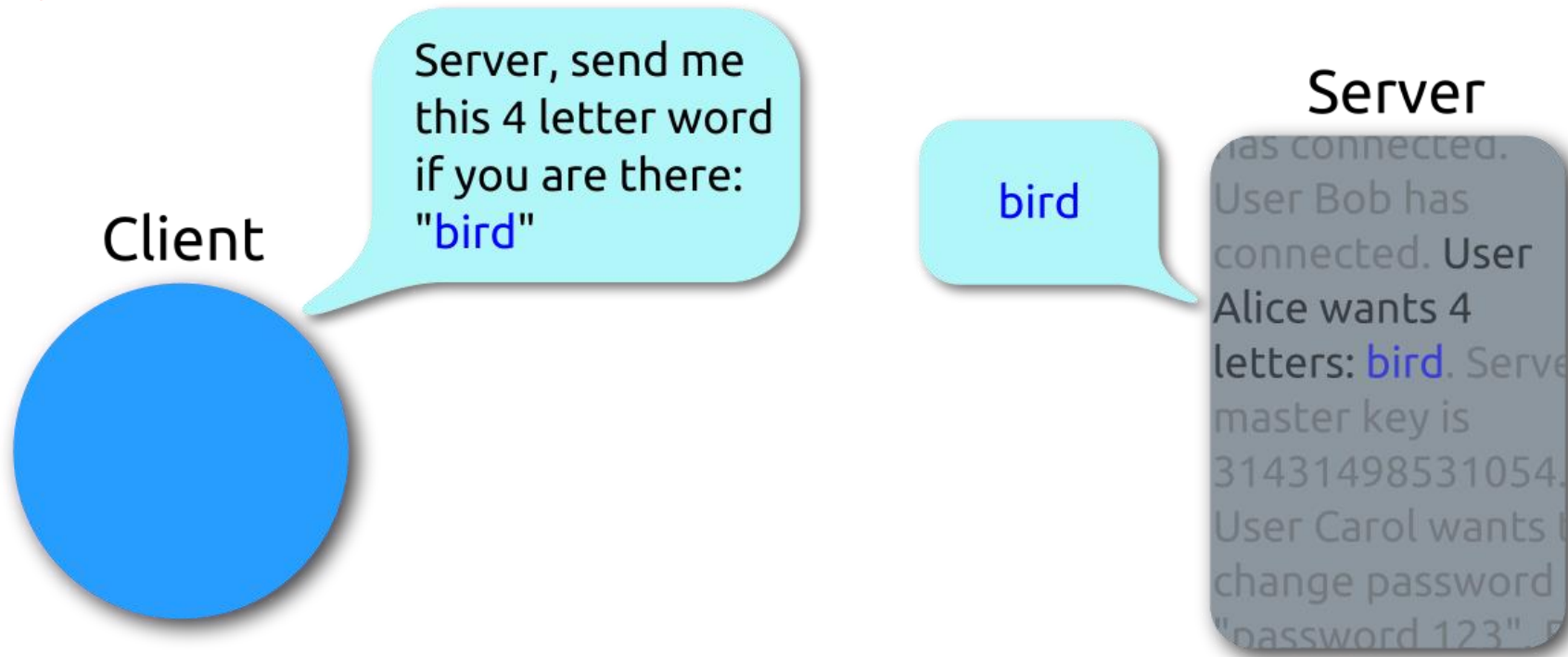
De-referencing an out-of-bound pointer:

- Out-of-bound writes → Buffer overflow
  - Compromise program integrity
- Out-of-bound reads → Buffer over-read
  - Information leakage. Can lead to integrity attack
    - E.g., leaking the stack canary
  - Another high-profile example: Heartbleed

# Heartbleed (CVE-2014-0160)



## Heartbeat – Normal usage

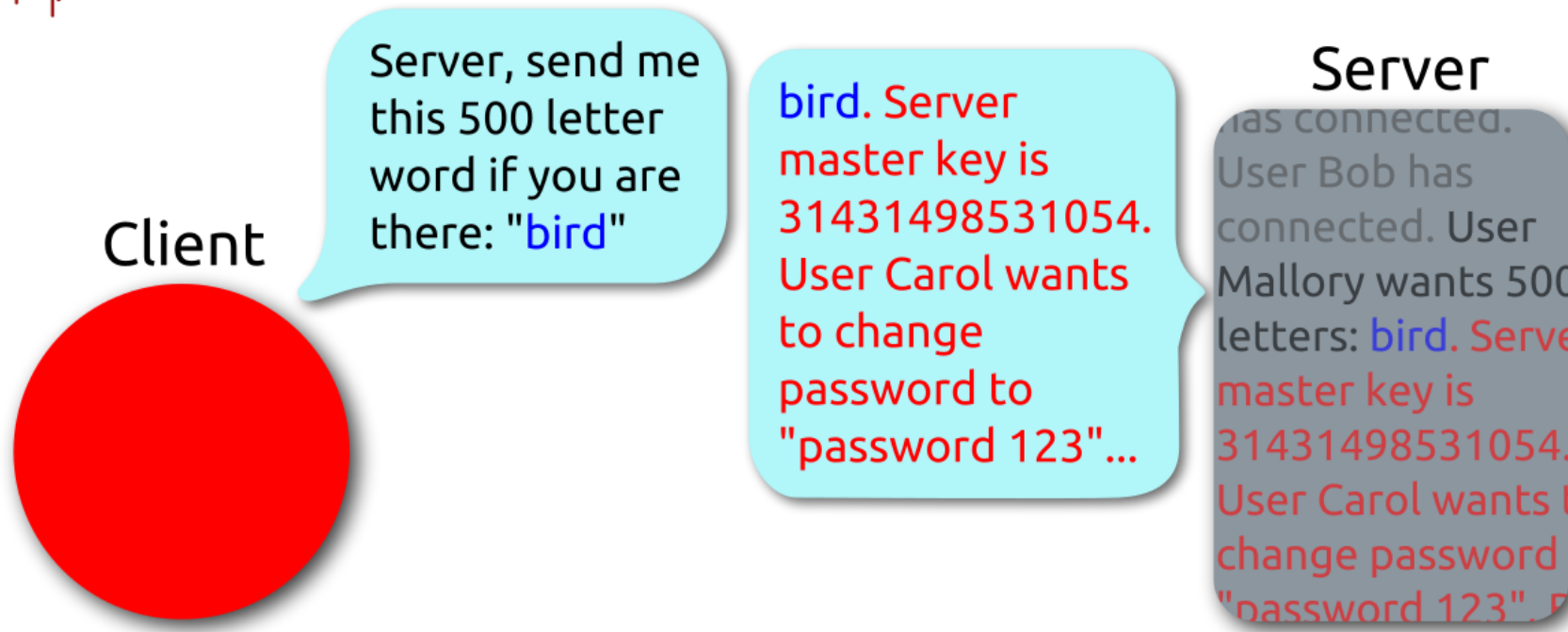


\*[https://en.wikipedia.org/wiki/Heartbleed#/media/File:Simplified\\_Heartbleed\\_explanation.svg](https://en.wikipedia.org/wiki/Heartbleed#/media/File:Simplified_Heartbleed_explanation.svg)

# Heartbleed (CVE-2014-0160)



## Heartbeat – Malicious usage



\*[https://en.wikipedia.org/wiki/Heartbleed#/media/File:Simplified\\_Heartbleed\\_explanation.svg](https://en.wikipedia.org/wiki/Heartbleed#/media/File:Simplified_Heartbleed_explanation.svg)

# Temporal Memory Safety - Use-after-Free

De-referencing a dangling pointer that points to a deleted object

```
1 typedef struct {
2     int id;
3     char[60] secret;
4 } Obj; // 64B
5
6 char *buf = malloc(64);
7 // ...
8 free(buf);
9
10 Obj *obj = malloc(sizeof(Obj));
11
12 printf("%s\n", &buf[4]);
```

```
1 typedef void (*myfunc)();
2 typedef struct {
3     int id;
4     myfunc *func;
5 } Obj; // 16B
6
7 char *buf = malloc(16);
8 // ...
9 free(buf);
10
11 Obj *obj = malloc(sizeof(Obj));
12
13 buf[...] = ...;
14 obj->func(); // compromised!
```

# Temporal Memory Safety – Double-Free

The only thing worse than double-free is double-dipping!

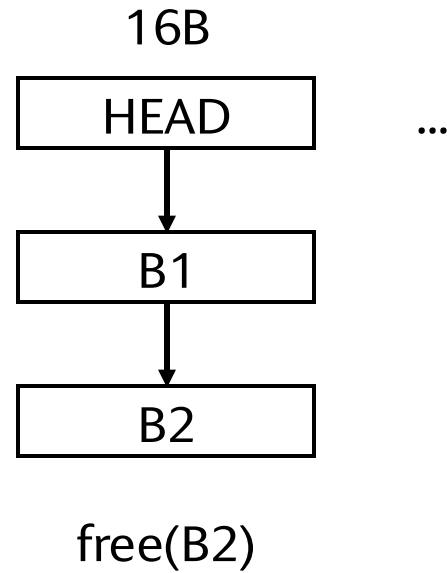
```
1 char *a = malloc(10); // 0x5683fe74e6b0
2 char *b = malloc(10); // 0x5683fe74e6d0
3 char *c = malloc(10); // 0x5683fe74e6f0
4
5 free(a);
6 free(b);
7 free(a);
8
9 char *d = malloc(10); // 0x5683fe74e6b0
10 char *e = malloc(10); // 0x5683fe74e6d0
11 char *f = malloc(10); // 0x5683fe74e6b0
```

But why? Is malloc really that lazy?

\*Requires tcache disabled on modern glibc

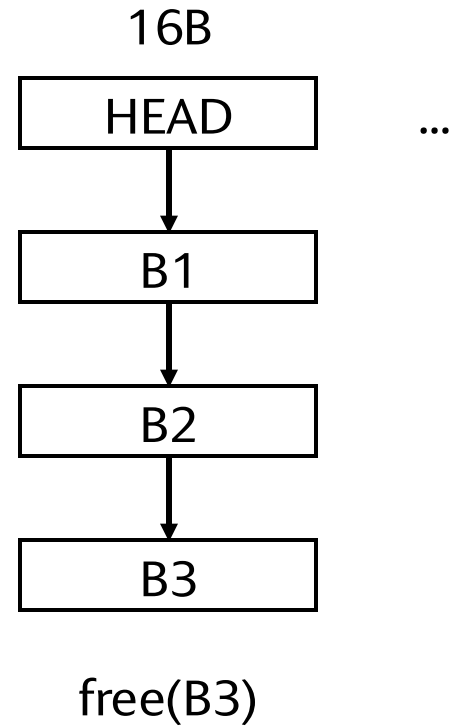
# Fastbins

A small software cache for holding recently freed small memory chunks



# Fastbins

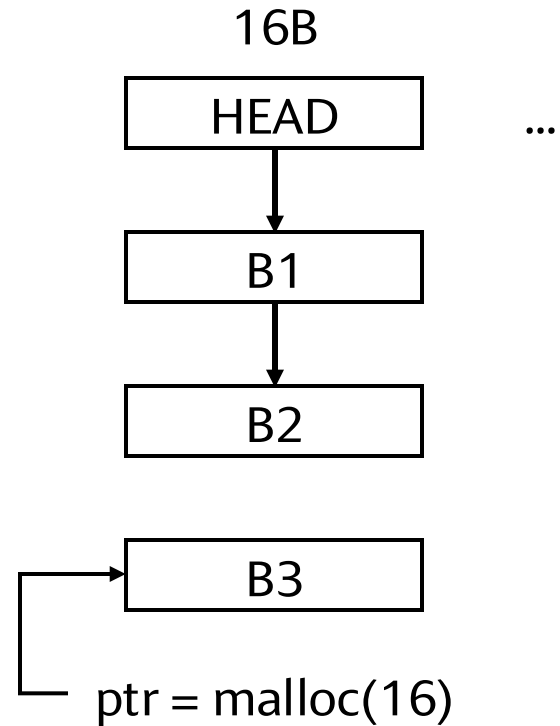
A small software cache for holding recently freed small memory chunks





# Fastbins

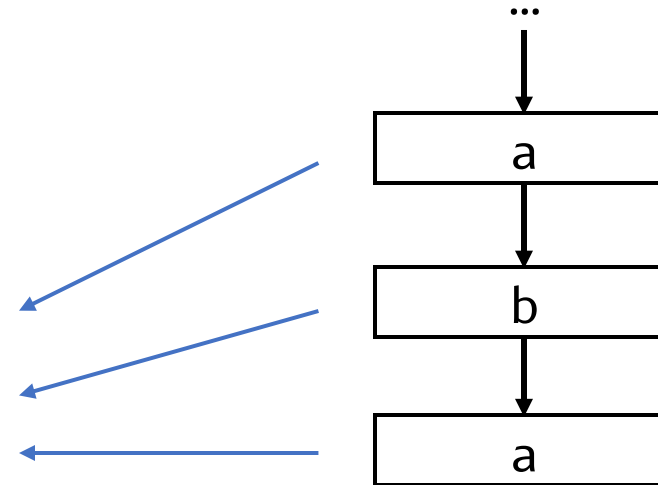
A small software cache for holding recently freed small memory chunks



# Temporal Memory Safety – Double-Free

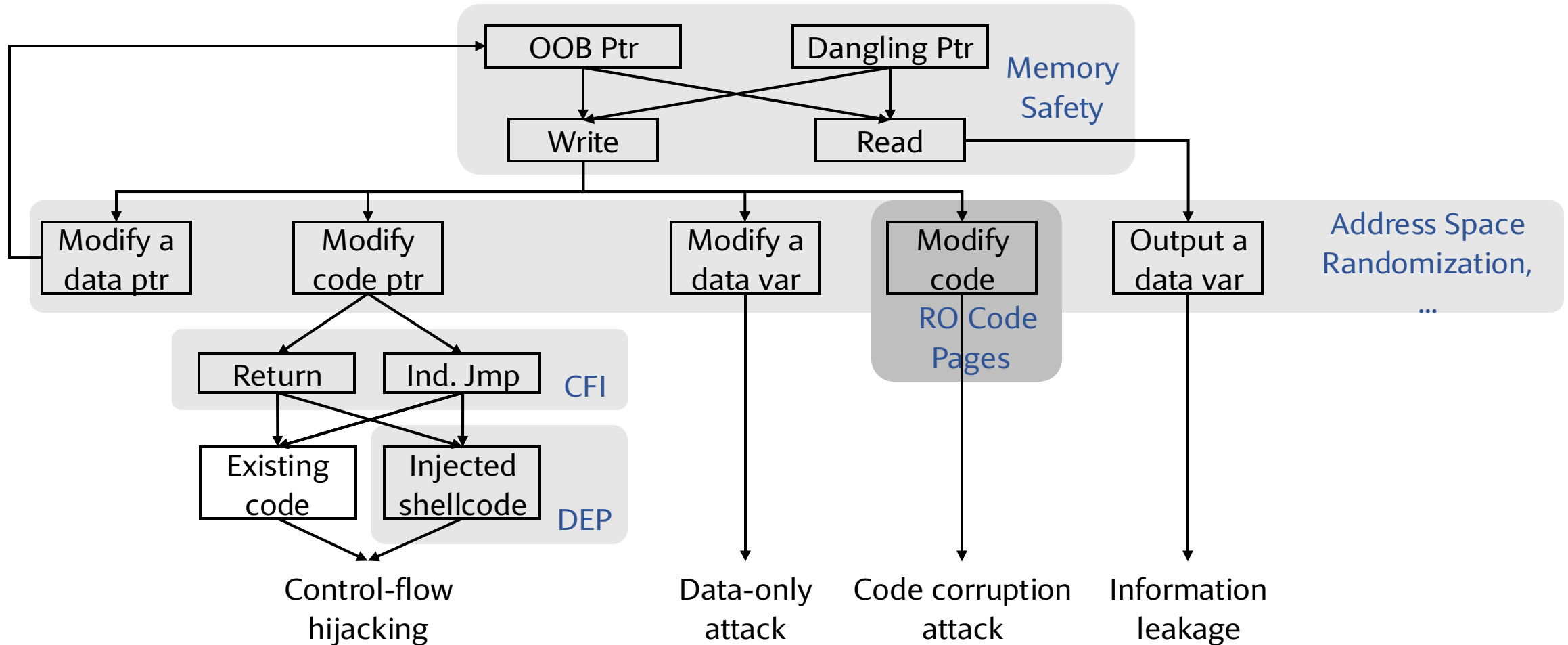
The only thing worse than double-free is double-dipping!

```
1 char *a = malloc(10); // 0x5683fe74e6b0
2 char *b = malloc(10); // 0x5683fe74e6d0
3 char *c = malloc(10); // 0x5683fe74e6f0
4
5 free(a);
6 free(b);
7 free(a);
8
9 char *d = malloc(10); // 0x5683fe74e6b0
10 char *e = malloc(10); // 0x5683fe74e6d0
11 char *f = malloc(10); // 0x5683fe74e6b0
```



\*Requires tcache disabled on modern glibc

# Adapted from “SoK: Eternal War in Memory”<sup>1</sup> (Simplified<sup>2</sup>)

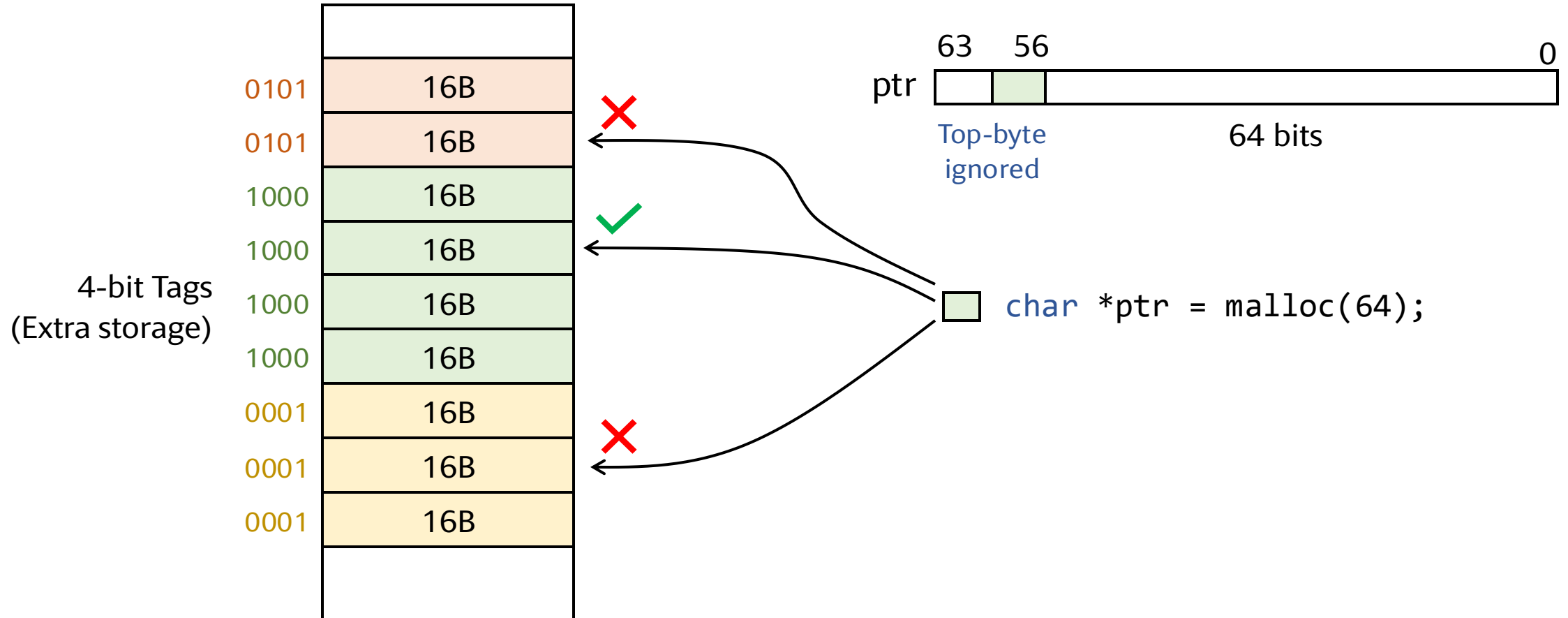


<sup>1</sup>Szekeres et al., “SoK: Eternal War in Memory,” SP ’13

<sup>2</sup>Only a subset of defenses are listed. Some adaptations are made.

# Memory Safety Technique – Memory Tag Extension (MTE)

Both memory blocks and pointers are tagged



# Memory Safety Technique – Memory Tag Extension (MTE)

Both memory blocks and pointers are tagged

