ECE 382N-Sec (FA25):

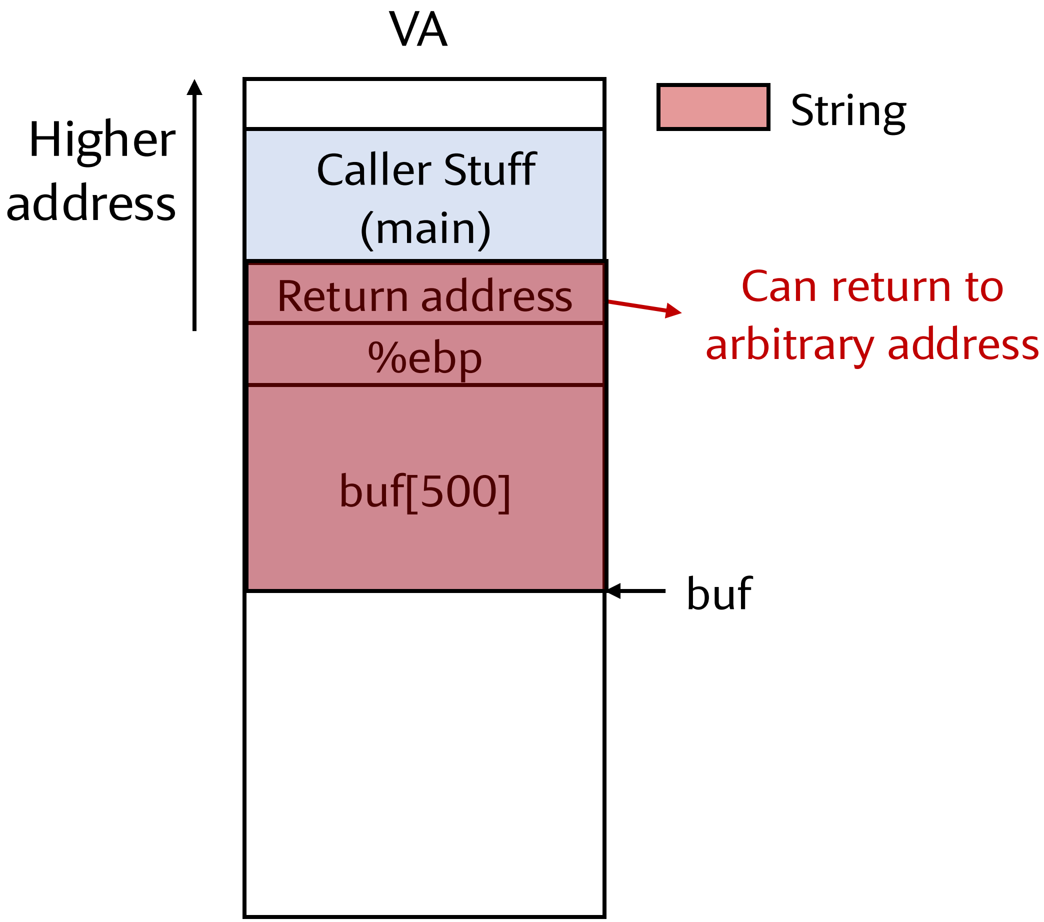
L10: Memory Safety

Neil Zhao

neil.zhao@utexas.edu

Buffer Overflow Attacks

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



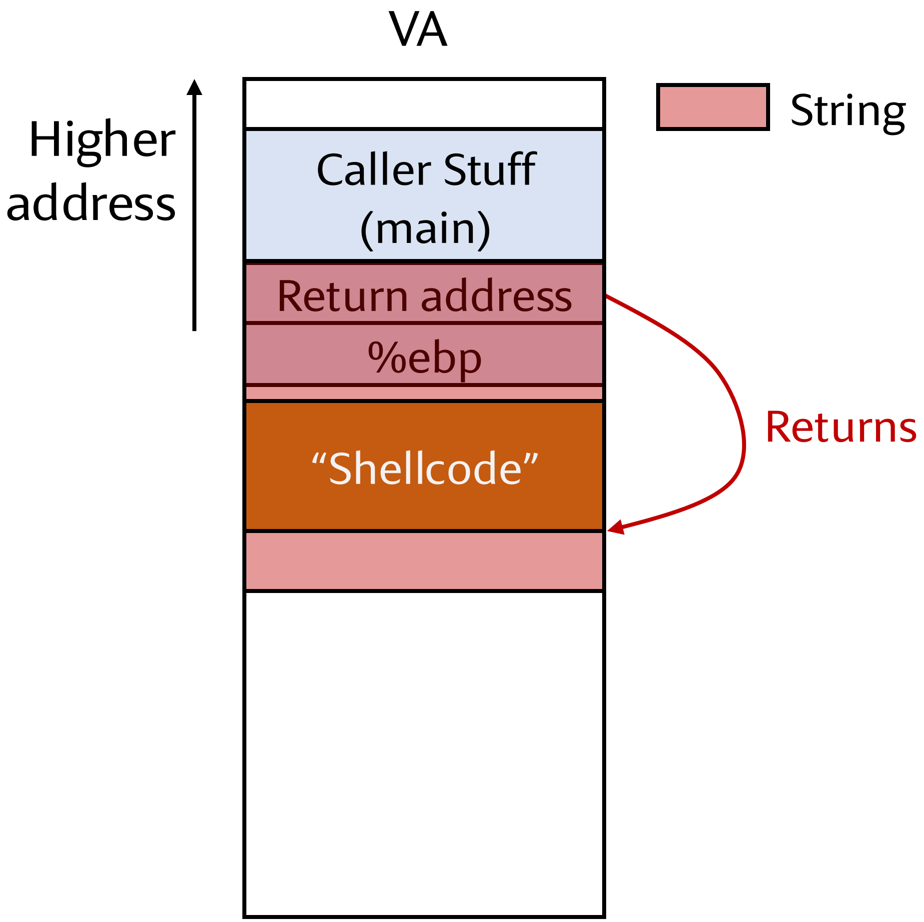

```
int main(int argc, char **argv) {
  char buf[500];

  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

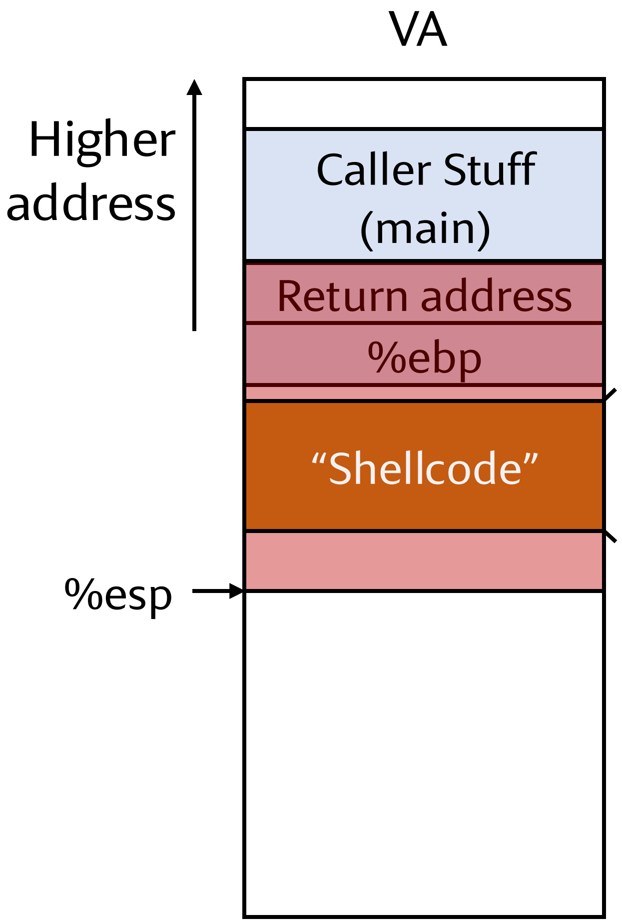



```
int main(int argc, char **argv) {
  char buf[500];

  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

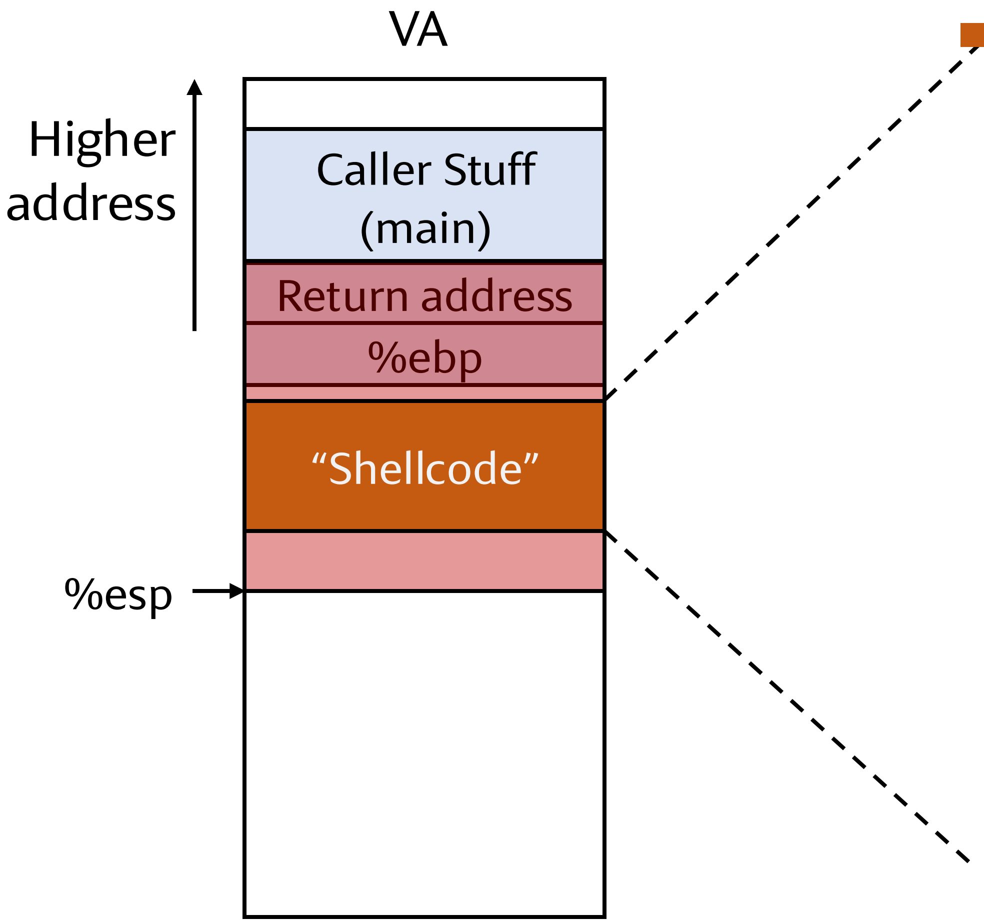



```
xor     eax,eax
sub     esp,0x1
mov     BYTE PTR [esp],al
push    0x68737a2f
push    0x6e69622f
push    0x7273752f
mov     esi,esp
push    eax
push    esi
mov     al,0xb
mov     ebx,esi
mov     ecx,esp
xor     edx,edx
int     0x80
mov     al,0x1
xor     ebx,ebx
int     0x80
```

```
Calling syscall 0xb (32-bit Linux):
execve(
  const char *pathname, // ebx
  char *const argv[],   // ecx
  char *const envp[]    // edx
)
```

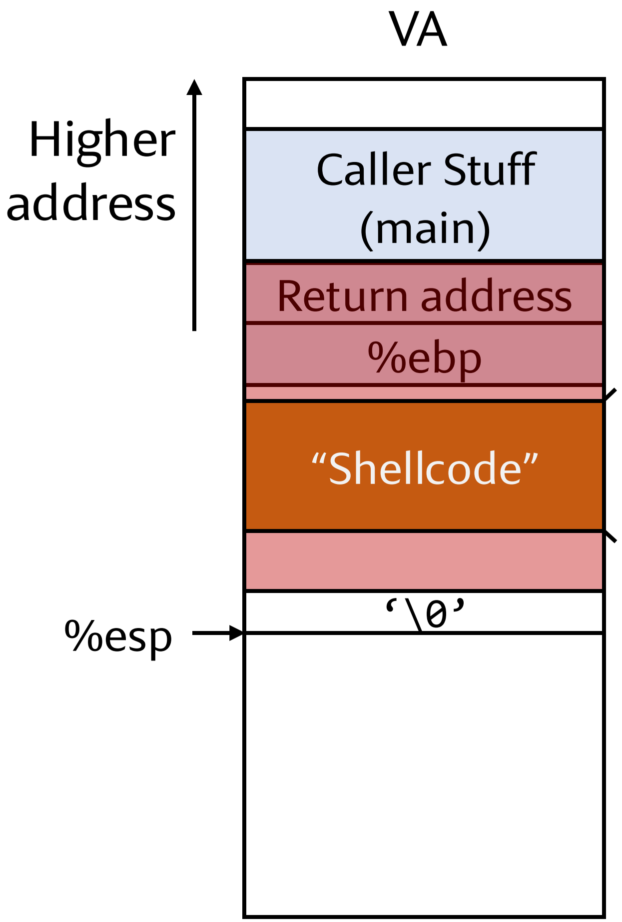
```
Goal: execve(“/usr/bin/zsh”,
{“/usr/bin/zsh”, NULL}, NULL);
```

The Shell Code



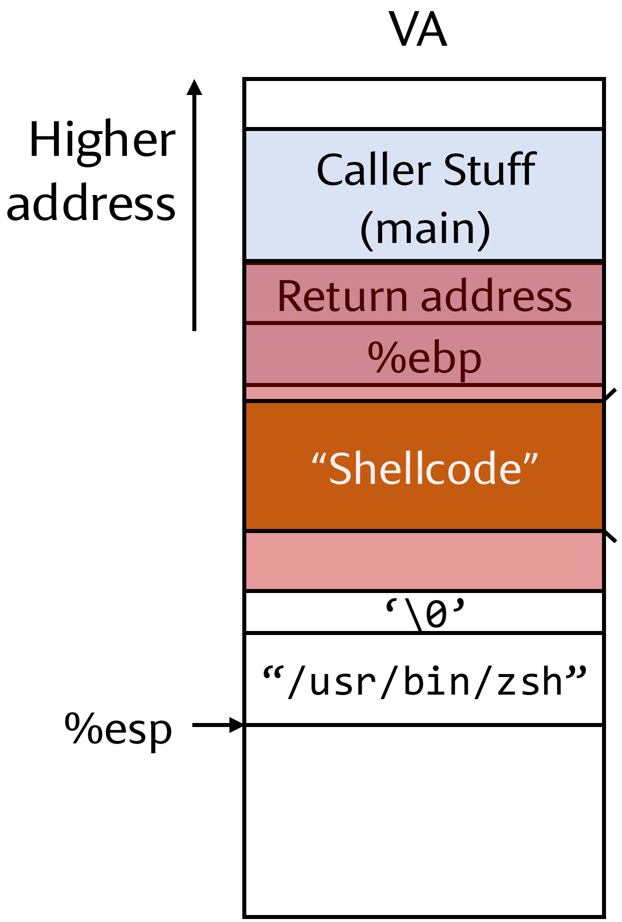

```
xor     eax,eax                  // %eax = 0;
sub     esp,0x1
mov     BYTE PTR [esp],al
push    0x68737a2f
push    0x6e69622f
push    0x7273752f
mov     esi,esp
push    eax
push    esi
mov     al,0xb
mov     ebx,esi
mov     ecx,esp
xor     edx,edx
int     0x80
mov     al,0x1
xor     ebx,ebx
int     0x80
```

The Shell Code



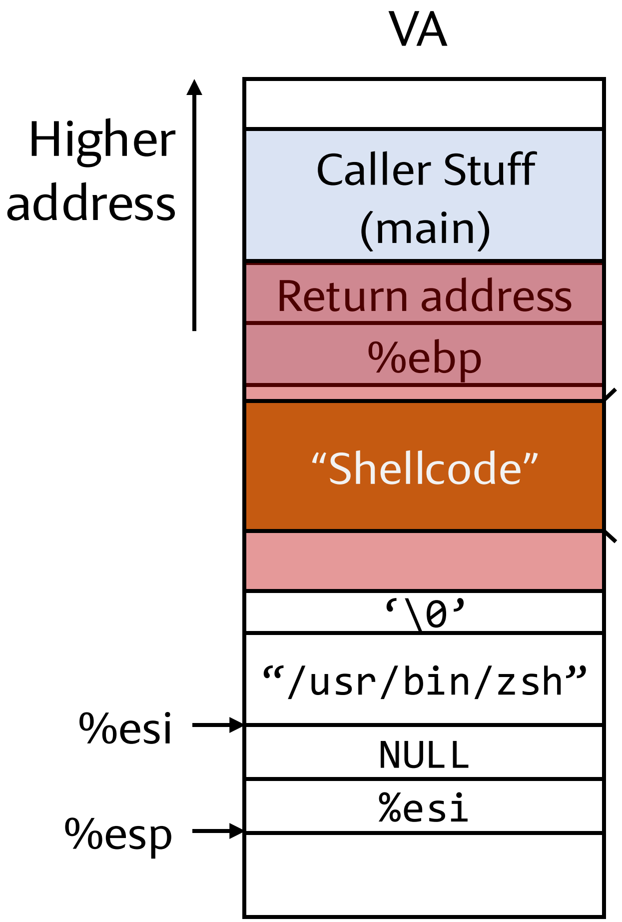

```
xor    eax,eax              // %eax = 0;
sub    esp,0x1
mov    BYTE PTR [esp],al    // Add the trailing ‘\0’
push   0x68737a2f
push   0x6e69622f
push   0x7273752f
mov    esi,esp
push   eax
push   esi
mov    al,0xb
mov    ebx,esi
mov    ecx,esp
xor    edx,edx
int    0x80
mov    al,0x1
xor    ebx,ebx
int    0x80
```

The Shell Code




```
xor     eax,eax              // %eax = 0;
sub     esp,0x1
mov     BYTE PTR [esp],al    // Add the trailing ‘\0’
push    0x68737a2f           // “/zsh”
push    0x6e69622f           // “/bin”
push    0x7273752f           // “/usr”
mov     esi,esp
push    eax
push    esi
mov     al,0xb
mov     ebx,esi
mov     ecx,esp
xor     edx,edx
int     0x80
mov     al,0x1
xor     ebx,ebx
int     0x80
```

The Shell Code




```
xor     eax,eax                // %eax = 0;
sub     esp,0x1
mov     BYTE PTR [esp],al      // Add the trailing '\0'
push    0x68737a2f             // "/zsh"
push    0x6e69622f             // "/bin"
push    0x7273752f             // "/usr"
mov     esi,esp
push    eax
push    esi
mov     al,0xb
mov     ebx,esi
mov     ecx,esp
xor     edx,edx
int     0x80
mov     al,0x1
xor     ebx,ebx
int     0x80
```

Calling syscall 0xb:

```
execve(
  const char *pathname, // ebx
  char *const argv[],   // ecx
  char *const envp[]    // edx
)

execve(
  "/usr/bin/zsh",             // ebx
  {"/usr/bin/zsh", NULL},     // ecx
  NULL                        // edx
)
```

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

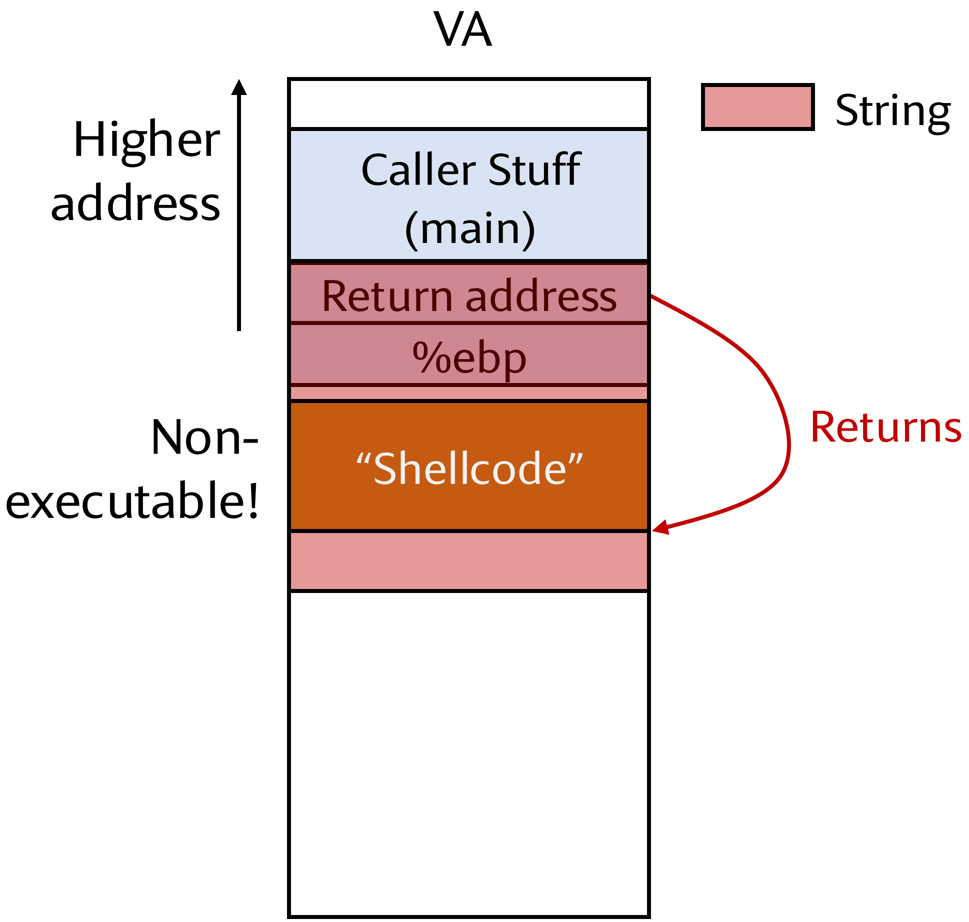
CVSS Version 4.0 | CVSS Version 3.x | CVSS Version 2.0

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:

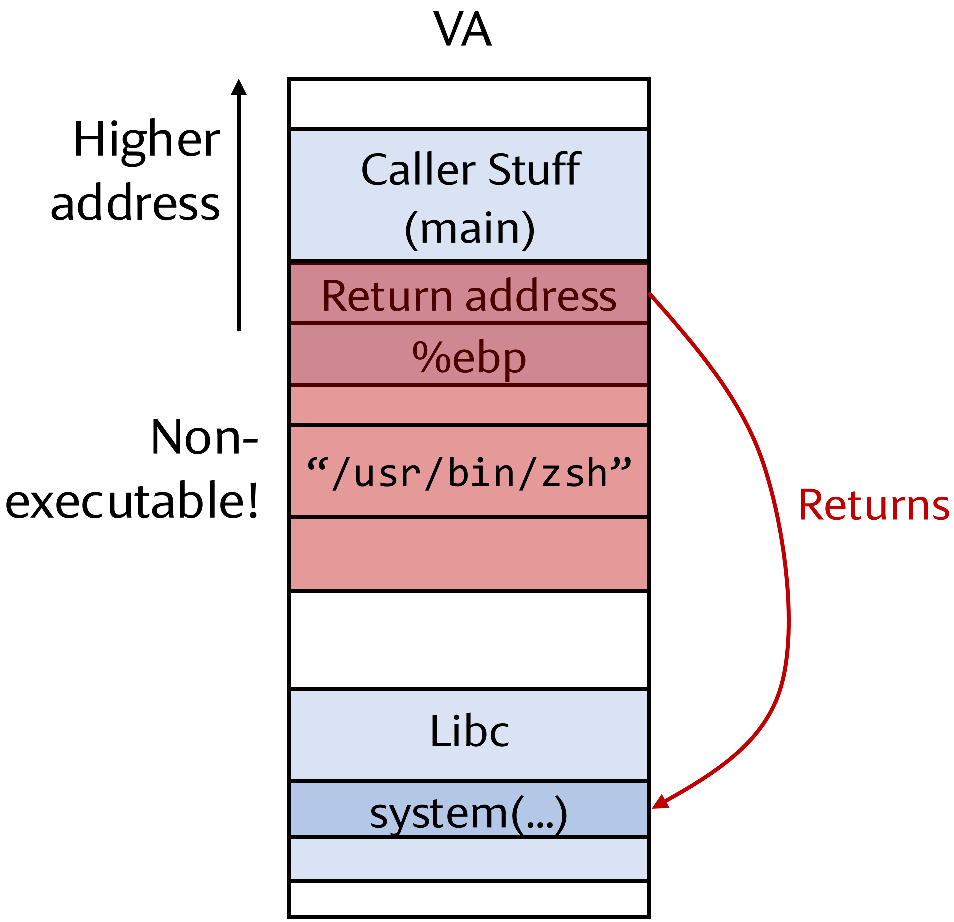
NIST: NVD	**Base Score:** 7.8 HIGH	**Vector:** CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H
ADP: CISA-ADP	**Base Score:** 7.8 HIGH	**Vector:** CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H

Mitigation 1: Data Execution Prevention (DEP)



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

Code-Reuse Attack: Return-to-Library



Libc "system" function: executes a shell command

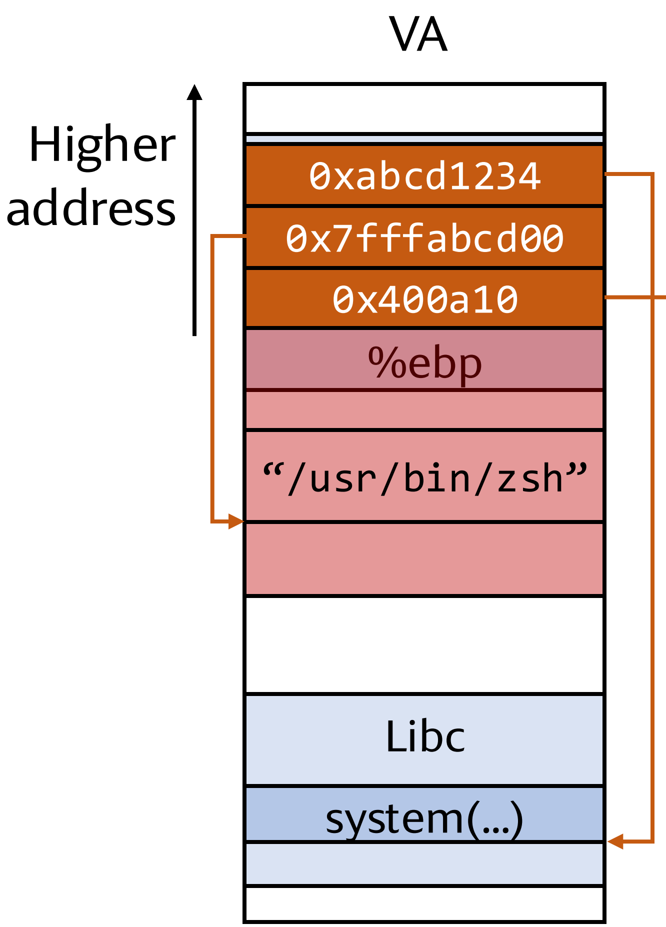
`int system(const char *command);`

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)



Somewhere in the binary:

```
...
0x400a10: pop rdi;
0x400a11: ret;
```

“\x5f\xc3”

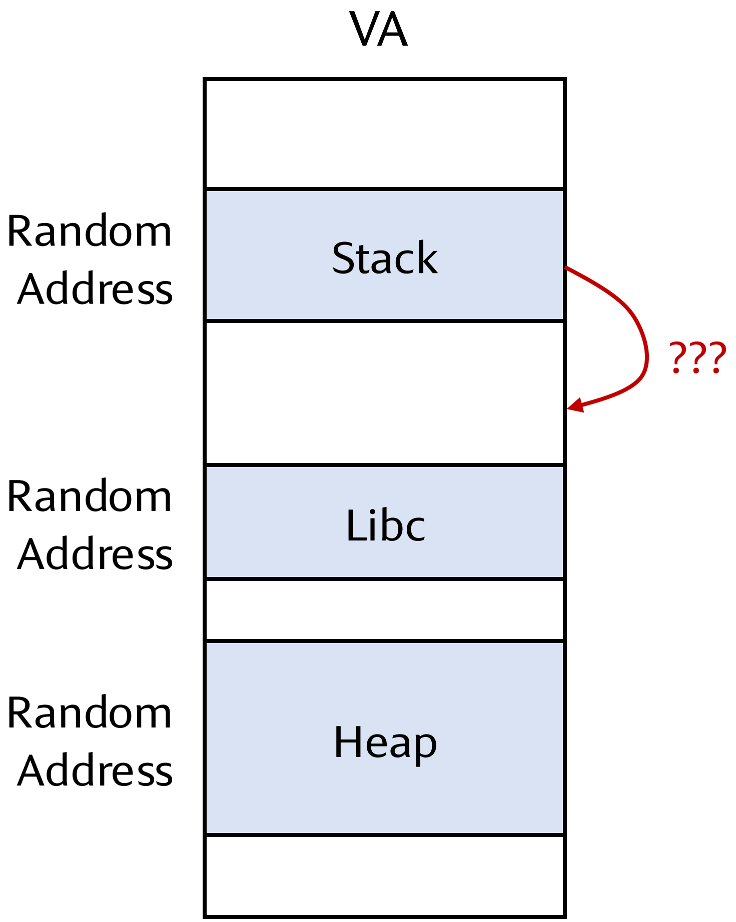
Exploiting variable-length, unaligned instruction.

Example:

```
0x400bf0: mov QWORD PTR [rdi-0x3d],rbx
```

“\x48\x89\x5f\xc3”

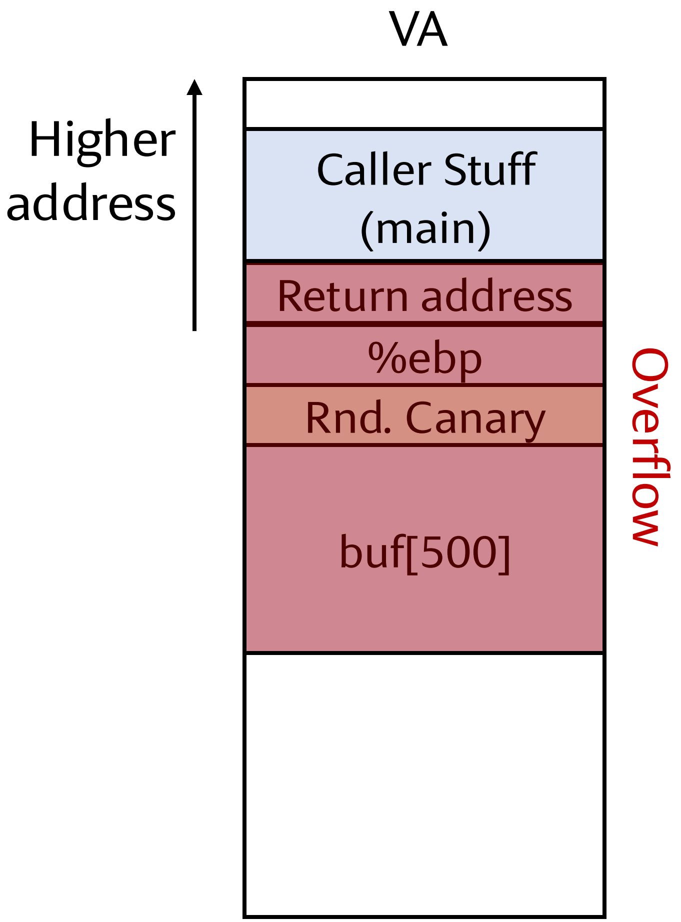
Mitigation 2: Address Space Layout Randomization



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

Mitigation 3: Control-Flow Integrity (CFI)

Protecting returns: Stack canary

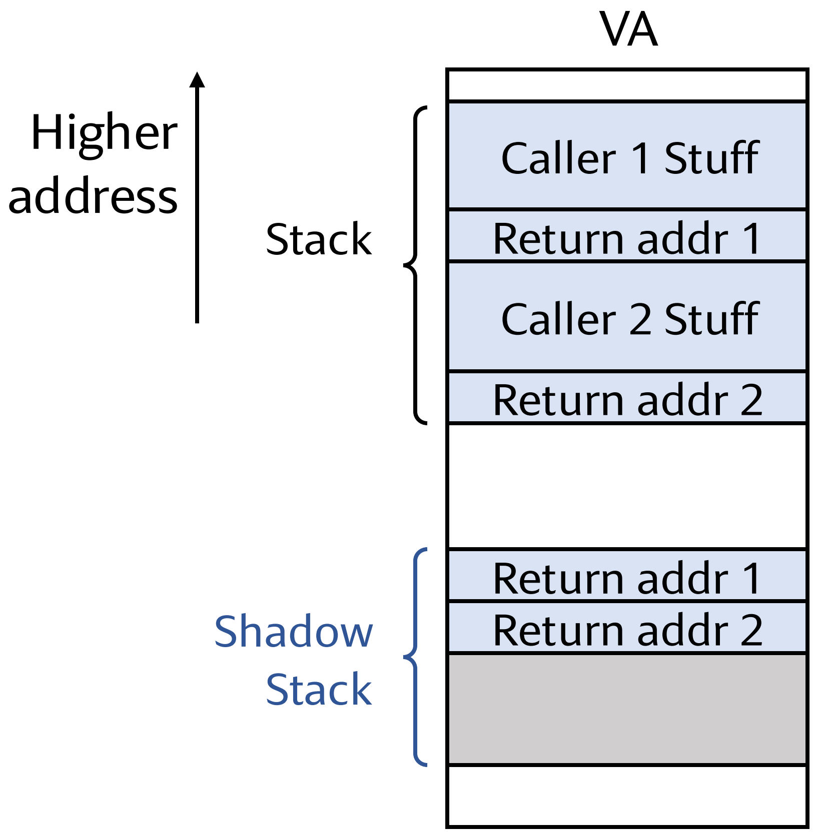


Approach: put a random (64-bit) canary value between the return address & %ebp and the current stack frame
⇒ Check if the canary is modified before returning

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 ⇒ 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

Allowed

Blocked

```
-------- Function starts ---------
0x00001189      endbr64
0x0000118d      push rbp
0x0000118e      mov rbp, rsp
0x00001191      mov dword [var_4h], edi
0x00001194      mov eax, dword [var_4h]
0x00001197      add eax, 1
0x0000119a      pop rbp
0x0000119b      ret
```

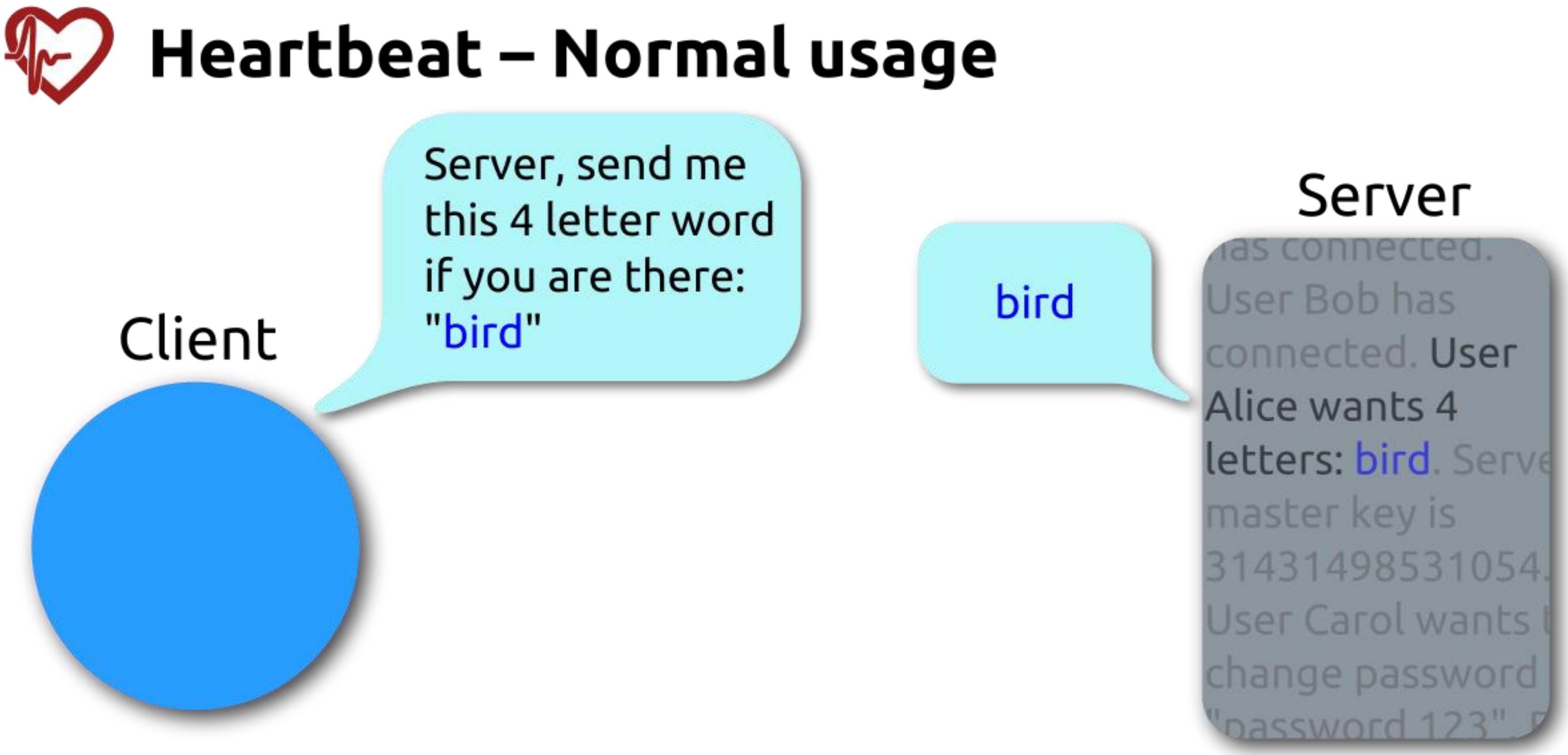
Spatial Memory Safety

De-referencing an out-of-bound pointer:

- Out-of-bound writes → Buffer overflow
    - Compromise program integrity

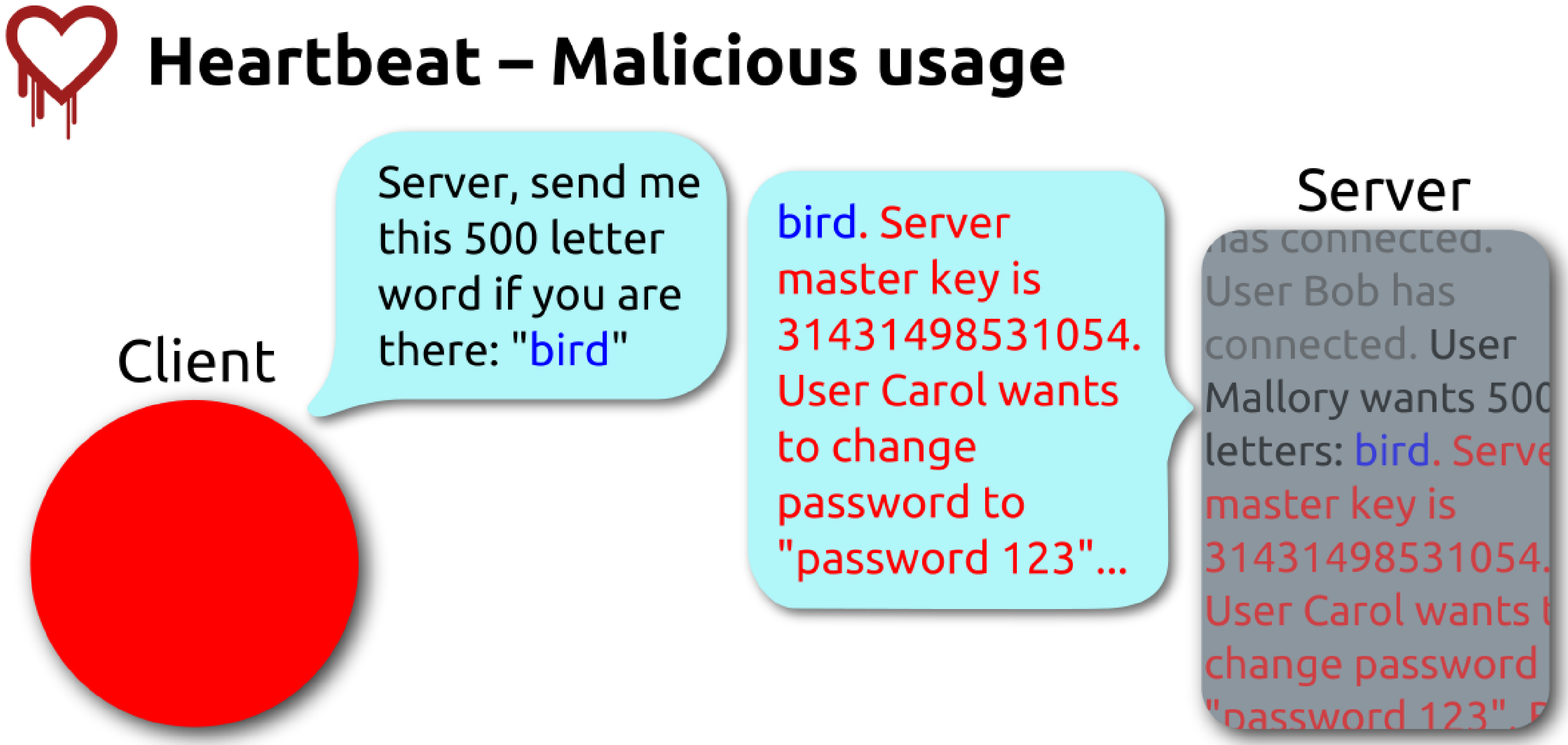
- Out-of-bound reads → Buffer over-read
    - Information leakage. Can lead to integrity attack
        - E.g., leaking the stack canary
    - Another high-profile example: Heartbleed

Heartbleed (CVE-2014-0160)



*https://en.wikipedia.org/wiki/Heartbleed#/media/File:Simplified_Heartbleed_explanation.svg

Heartbleed (CVE-2014-0160)



*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

```c
typedef struct {
    int id;
    char[60] secret;
} Obj; // 64B

char *buf = malloc(64);
// ...
free(buf);

Obj *obj = malloc(sizeof(Obj));

printf(“%s\n”, &buf[4]);
```

```c
typedef void (*myfunc)();
typedef struct {
    int id;
    myfunc *func;
} Obj; // 16B

char *buf = malloc(16);
// ...
free(buf);

Obj *obj = malloc(sizeof(Obj));

buf[…] = …;
obj->func(); // compromised!
```

Temporal Memory Safety – Double-Free

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

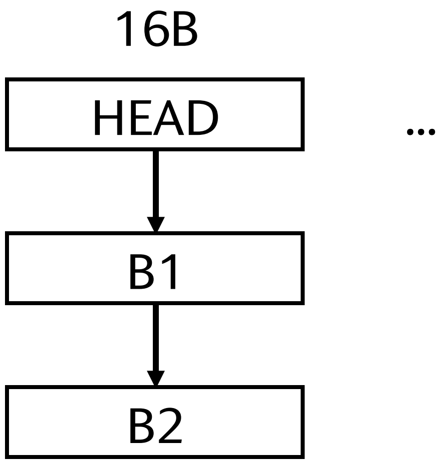
```
char *a = malloc(10); // 0x5683fe74e6b0
char *b = malloc(10); // 0x5683fe74e6d0
char *c = malloc(10); // 0x5683fe74e6f0

free(a);
free(b);
free(a);

char *d = malloc(10); // 0x5683fe74e6b0
char *e = malloc(10); // 0x5683fe74e6d0
char *f = malloc(10); // 0x5683fe74e6b0
```

But why? Is malloc really that lazy?

*Requires tcache disabled on modern glic

Fastbins

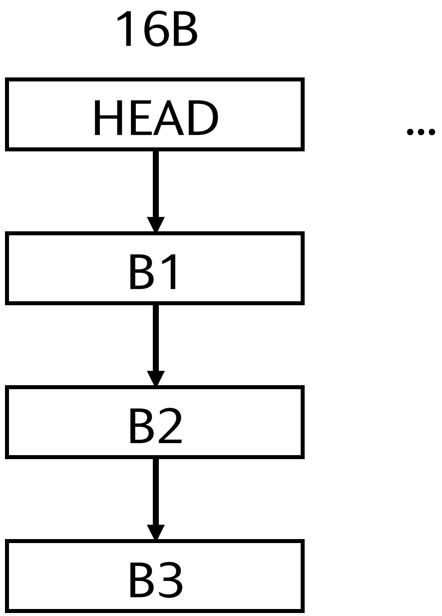
A small software cache for holding recently freed small memory chunks



free(B2)

Fastbins

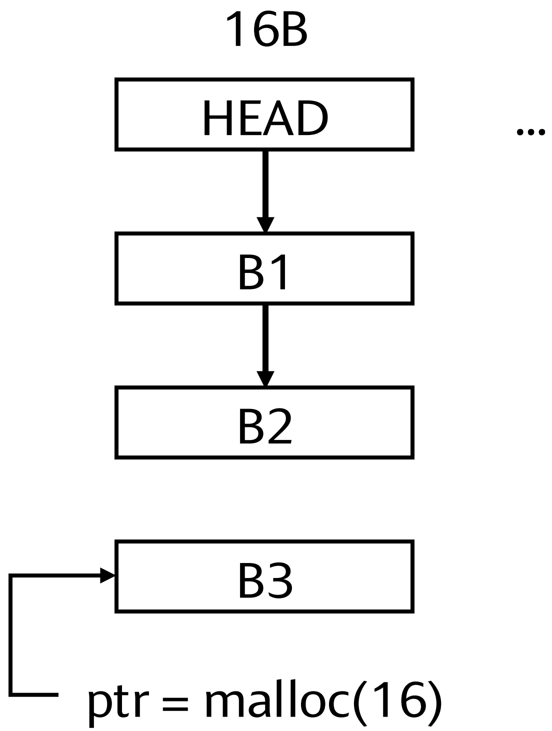
A small software cache for holding recently freed small memory chunks



free(B3)

Fastbins

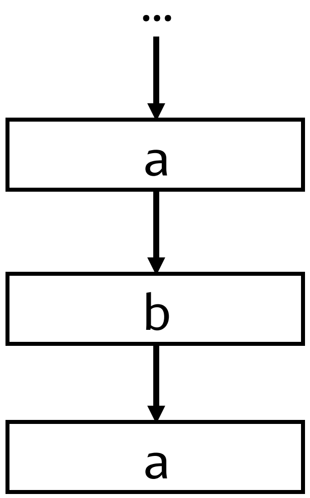
A small software cache for holding recently freed small memory chunks



Temporal Memory Safety – Double-Free

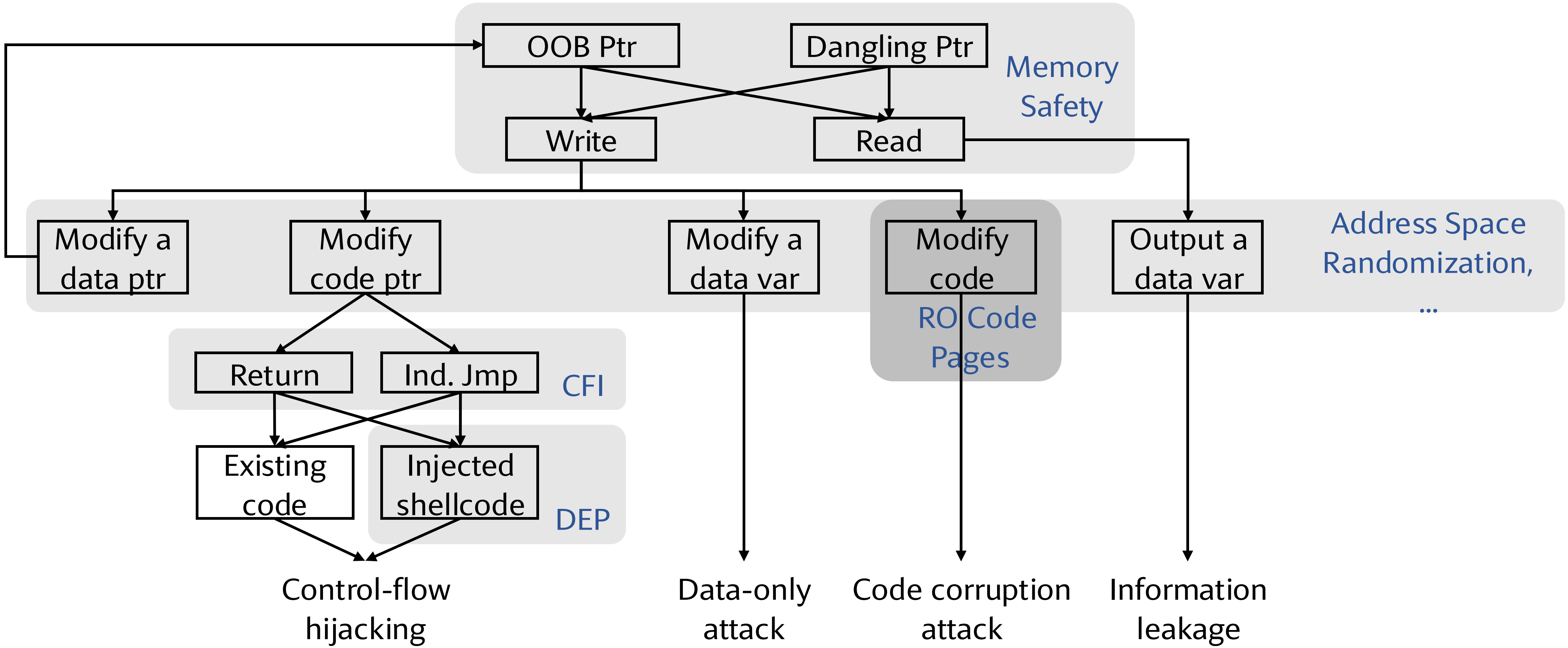
The only thing worse than double-free is double-dipping!

```
char *a = malloc(10); // 0x5683fe74e6b0
char *b = malloc(10); // 0x5683fe74e6d0
char *c = malloc(10); // 0x5683fe74e6f0

free(a);
free(b);
free(a);

char *d = malloc(10); // 0x5683fe74e6b0
char *e = malloc(10); // 0x5683fe74e6d0
char *f = malloc(10); // 0x5683fe74e6b0
```



*Requires tcache disabled on modern glic

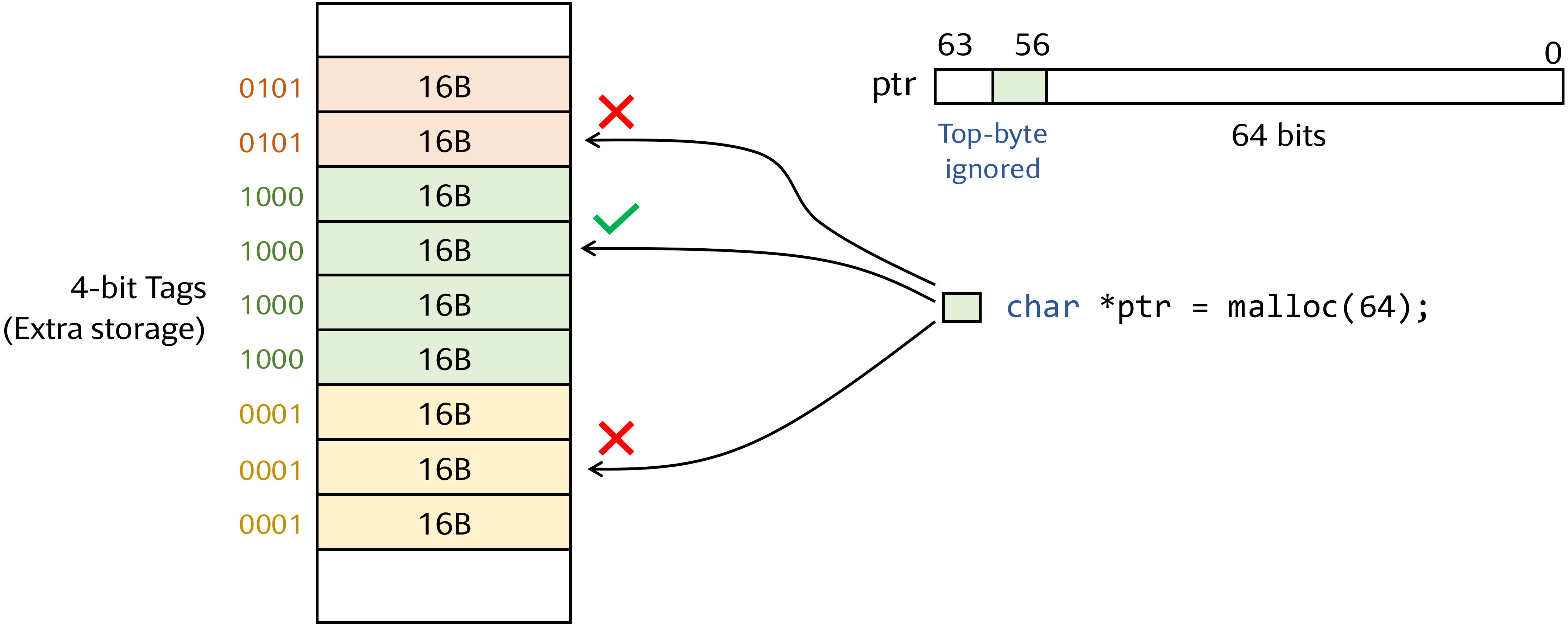
Adapted from “SoK: Eternal War in Memory”[1] (Simplified[2])



[1]Szekeres et al., “SoK: Eternal War in Memory,” SP ’13
[2]Only a subset of defenses are listed. Some adaptations are made.

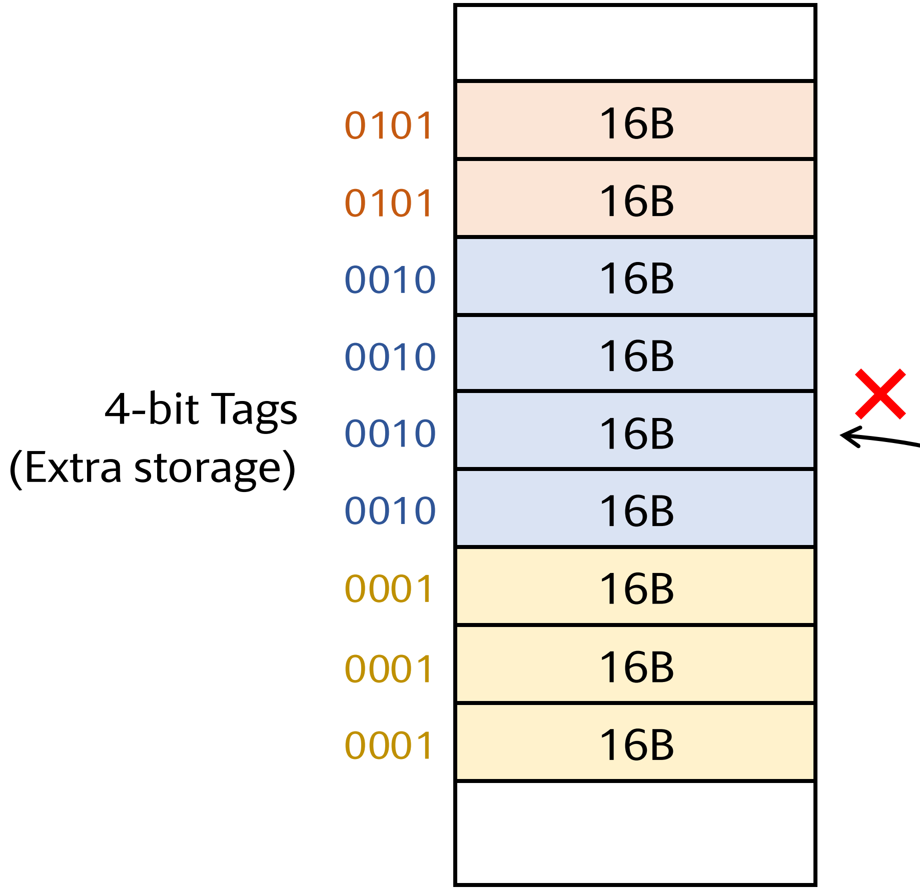
Memory Safety Technique – Memory Tag Extension (MTE)

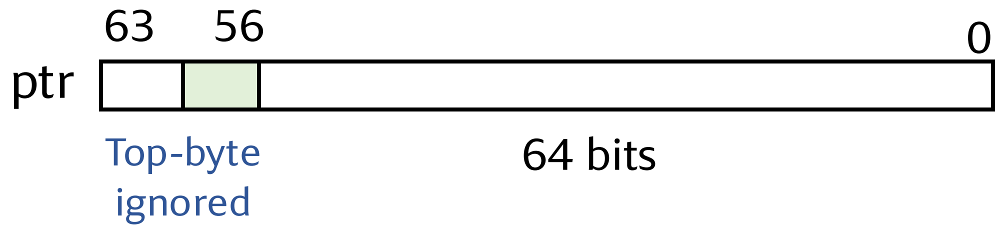
Both memory blocks and pointers are tagged



Memory Safety Technique – Memory Tag Extension (MTE)

Both memory blocks and pointers are tagged






```
char *ptr = malloc(64);
...
free(ptr); // re-color
...
*ptr;
```