# Lecture 03 – Control Flow

Stephen Checkoway

CS 343 – Fall 2020

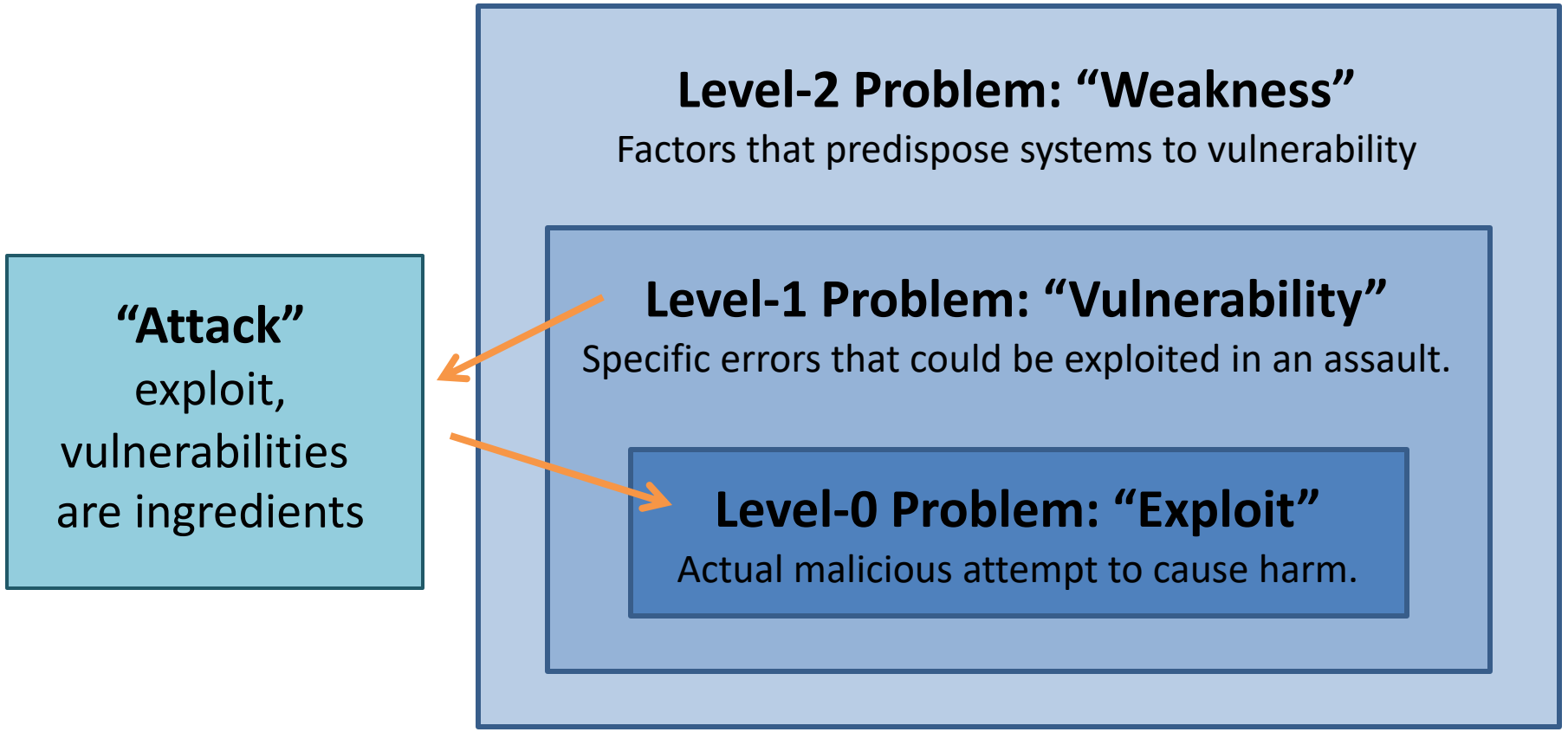
Adapted from Michael Bailey's ECE 422

# Outline

- Computer
  - CPU
  - Instructions
- The Stack (x86)
  - What is a stack
  - How it is used by programs
  - Technical details
- Attacks
- Buffer overflows

- Adapted from Aleph One's "Smashing the Stack for Fun and Profit"

# “Insecurity”?


Level-2 Problem: “Weakness”
Factors that predispose systems to vulnerability
Level-1 Problem: “Vulnerability”
Specific errors that could be exploited in an assault.
Level-0 Problem: “Exploit”
Actual malicious attempt to cause harm.
“Attack”
exploit,
vulnerabilities
are ingredients

# Why Study Attacks?

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

```
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;
}
```

# Virtual memory

- Each running process has its own virtual memory space
  - Your computer has a bunch of RAM
  - RAM is an array of bytes indexed from 0
  - It would be bad if any process could read/write any byte of memory
  - The OS and hardware carve up memory and hand it out to processes


Image from Wikipedia

# Virtual address space

- OS presents each process with the fiction that it has access to the entire valid range of memory from index 0 to the maximum index (2^32 - 1 or 2^64 - 1)
- It does this by mapping virtual addresses used by processes to physical addresses used by the hardware

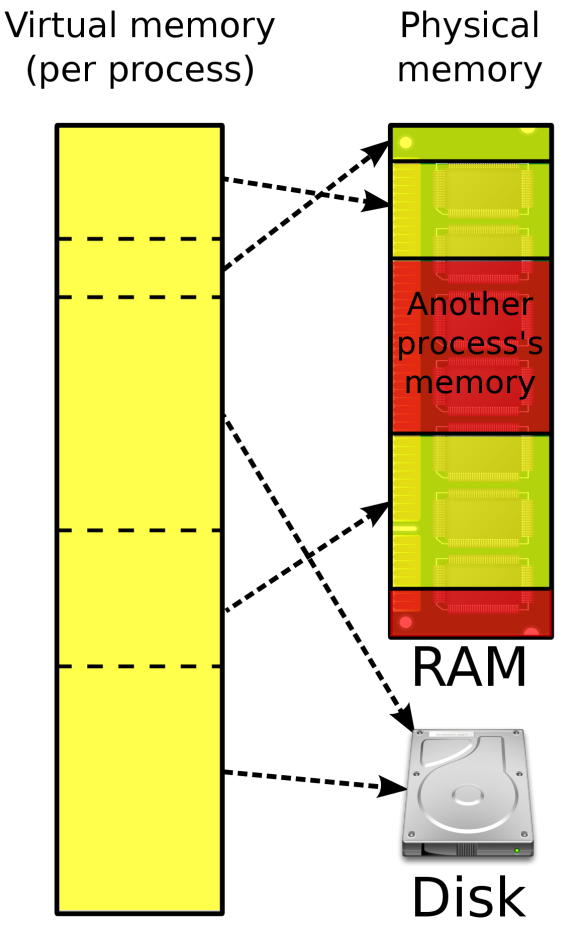



Image from Wikipedia

# Virtual address space layout

- Each function called in a program is allocated a *stack frame* on the call stack; it stores
  - The return address
  - Local variables
  - Arguments to functions it calls
- The software maintains two pointers
  - Stack pointer: points to the top (lowest address) of the stack
  - Frame pointer: points to the call frame (optional)

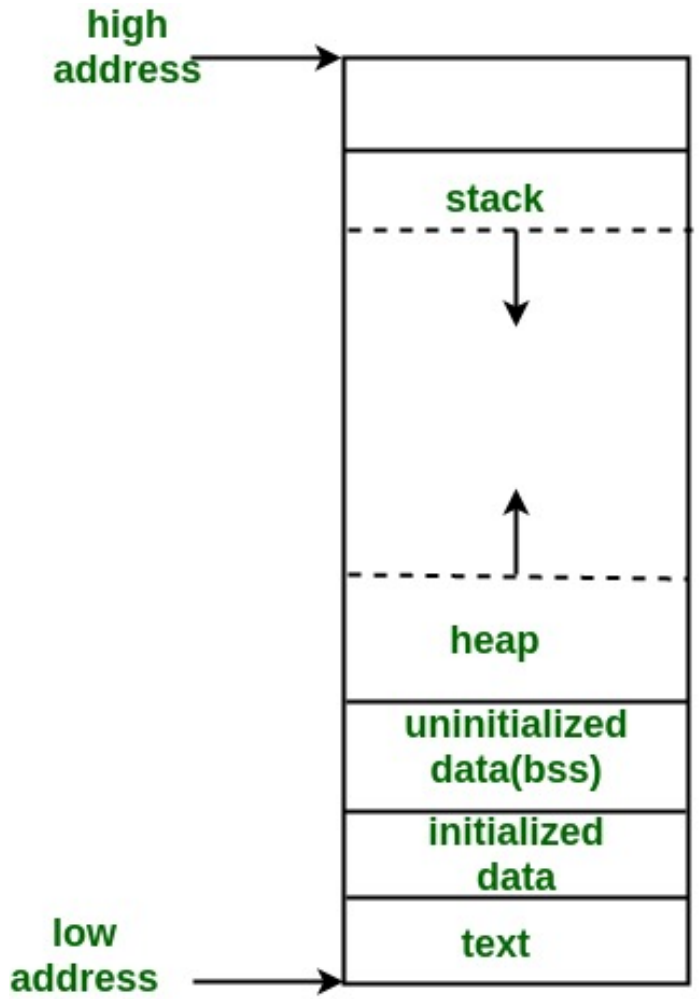
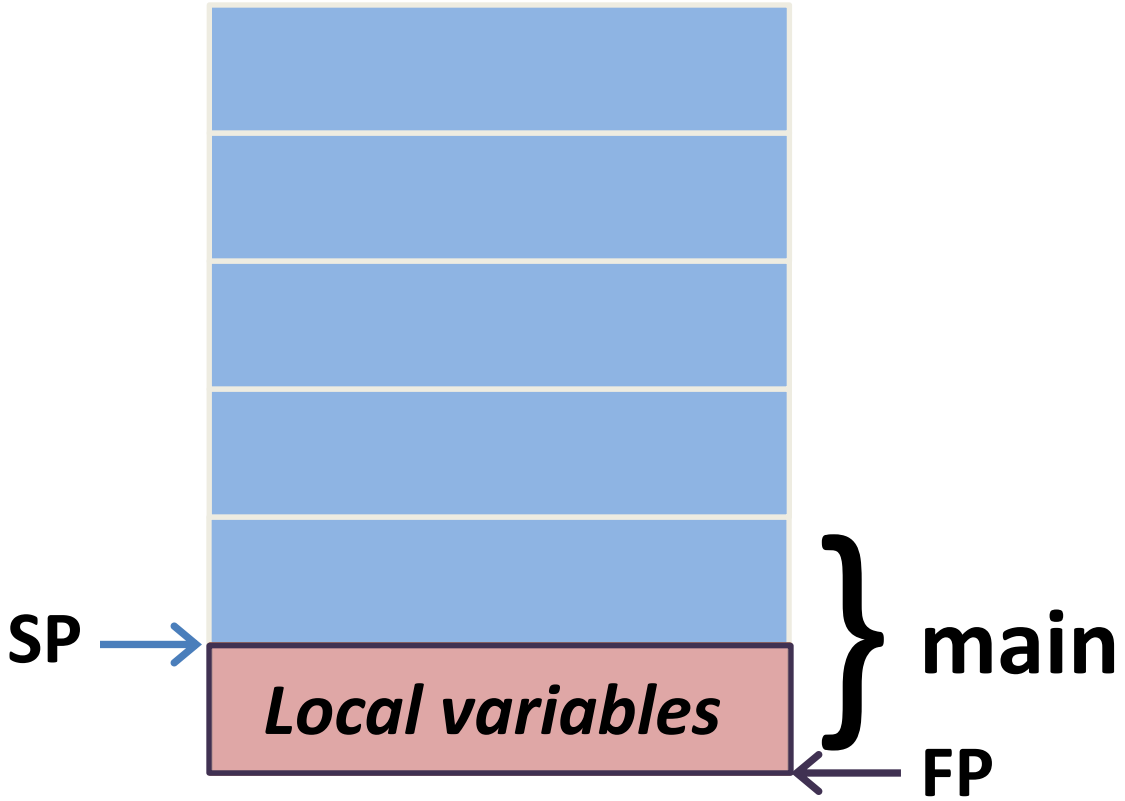



Image from https://blog.adafruit.com/2019/07/19/the-pitfalls-of-uninitialized-memory-programming-c-rust/

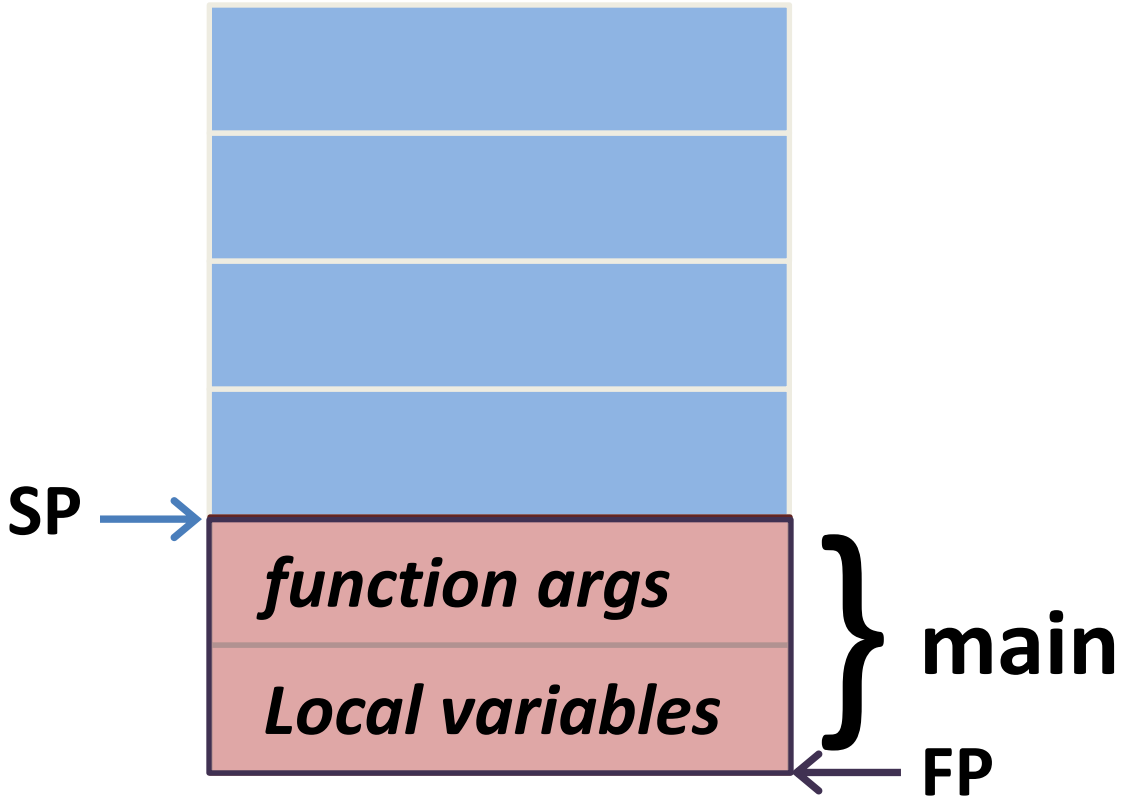
# example.c

```
void foo(int a, int b) {
    char buf1[10];
}

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

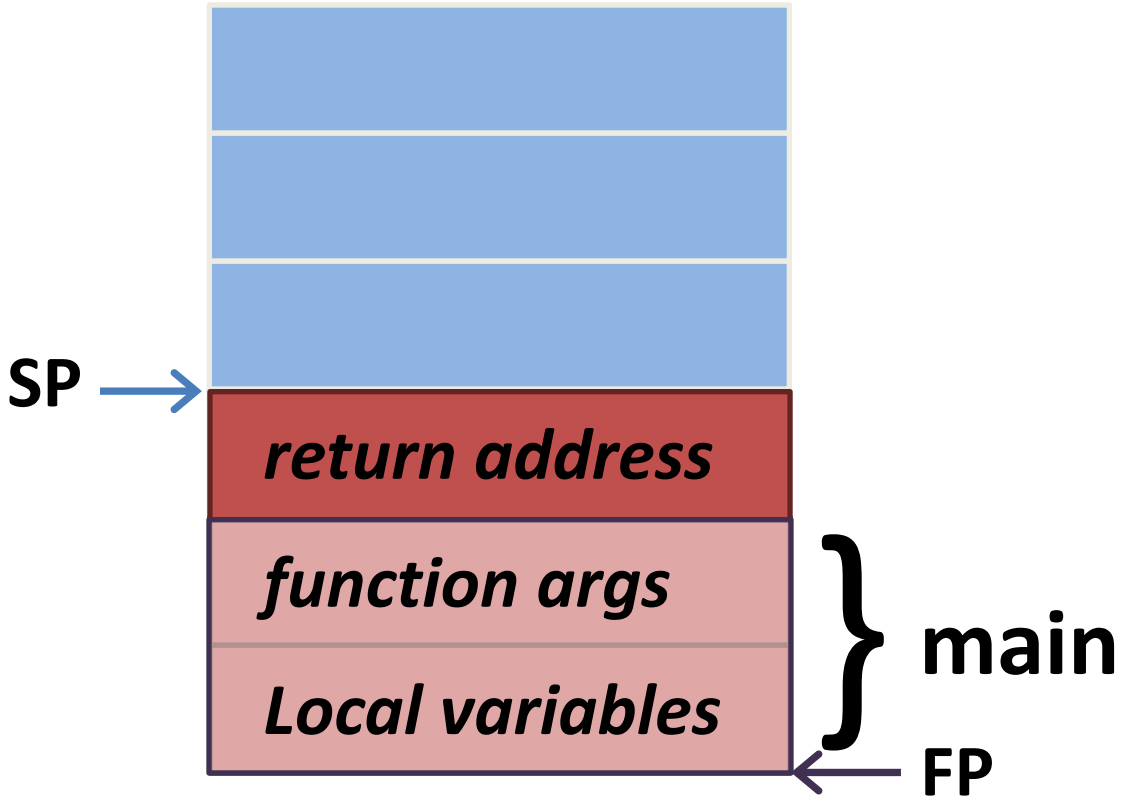
# C stack frames



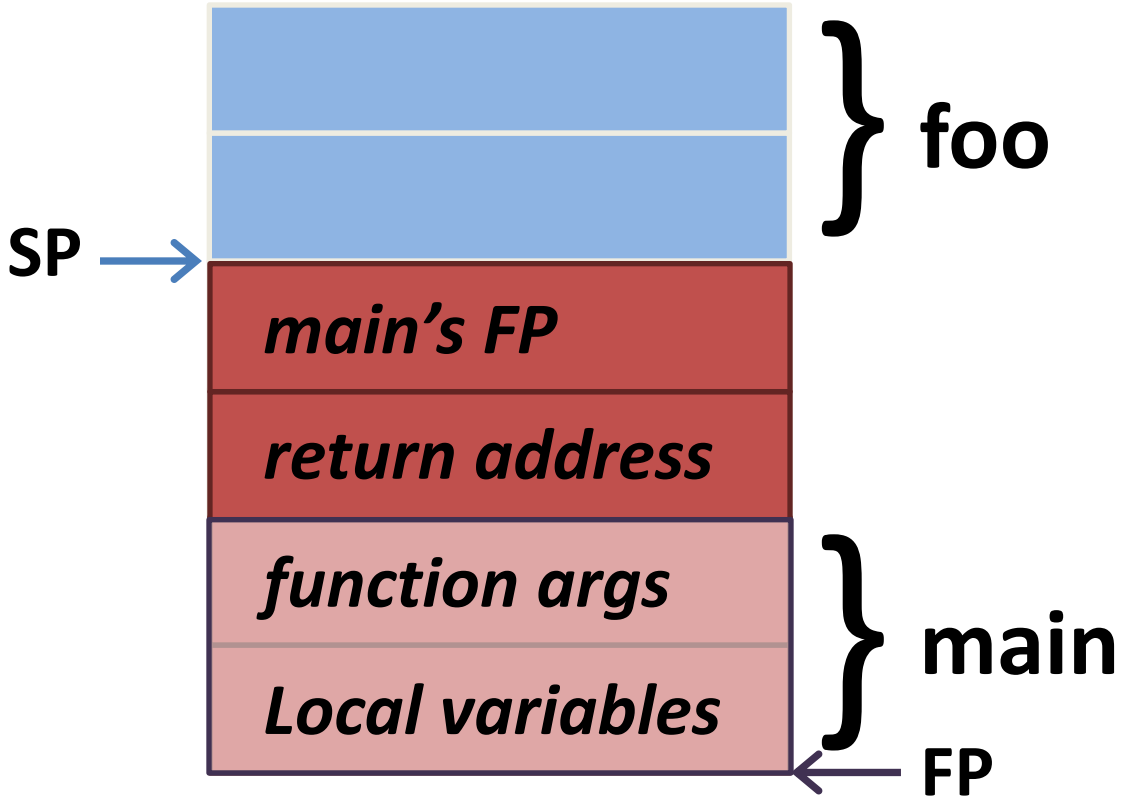
# C stack frames



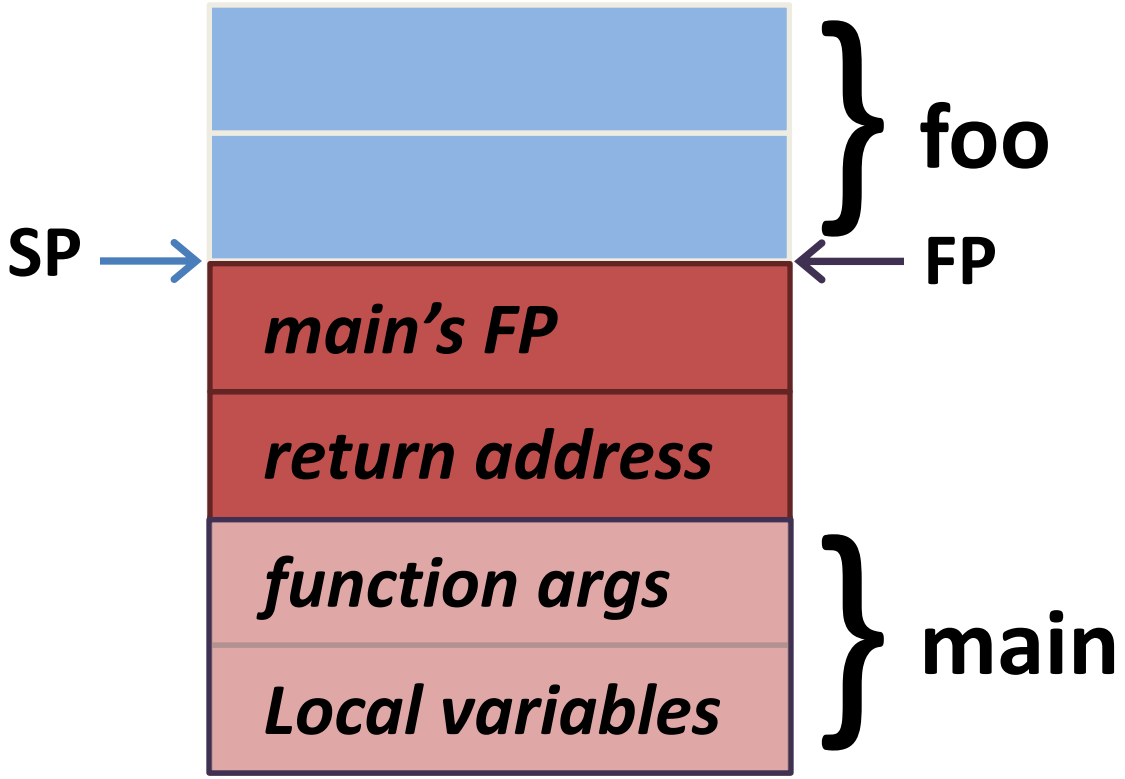
# C stack frames



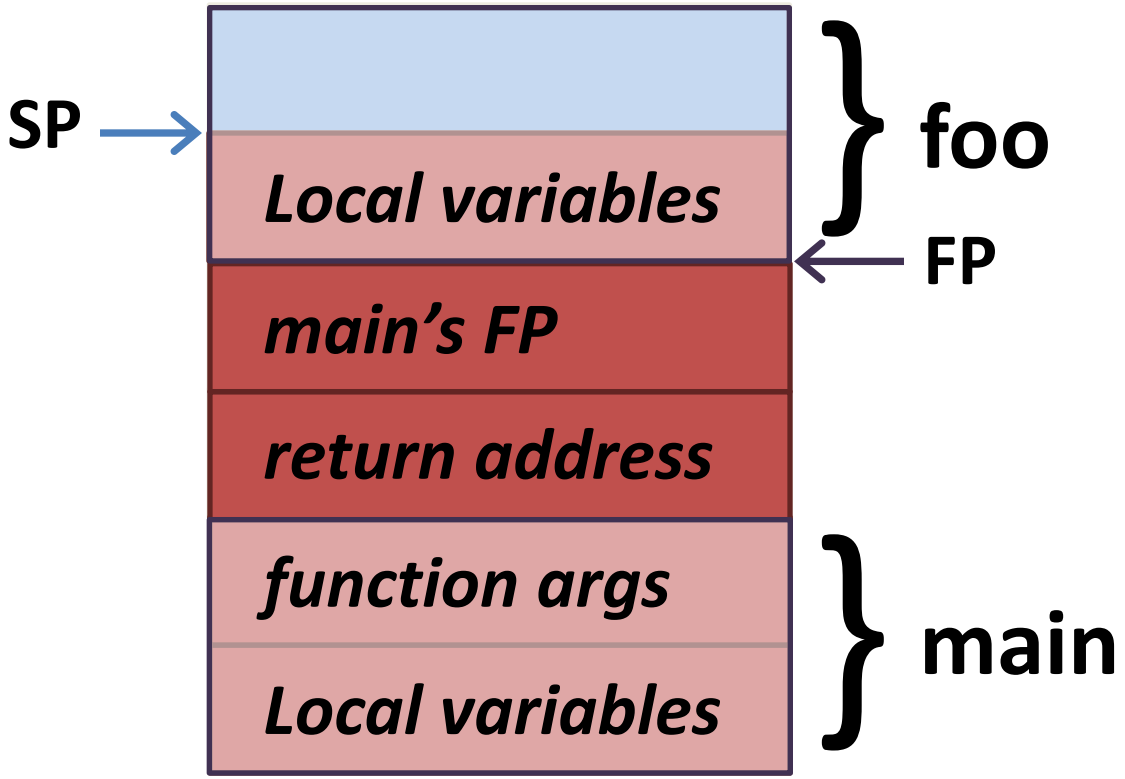
# C stack frames



# C stack frames



# C stack frames



# 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

# Some x86 instructions (AT&T notation)

- add src, dest ; Computes dest + src, stores in dest
- sub src, dest ; Computes dest – src, stores in dest
- mov src, dest ; Copies src to dest
- push src ; Decrements esp by 4, writes src to stack
- pop dest ; Reads top of stack into dest, increments esp by 4
- call foo ; Calls the function foo, pushes the address of the next instruction onto the stack
- ret ; Pops the top of the stack into eip (returns from a function)

# 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

# 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
  - movl %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)

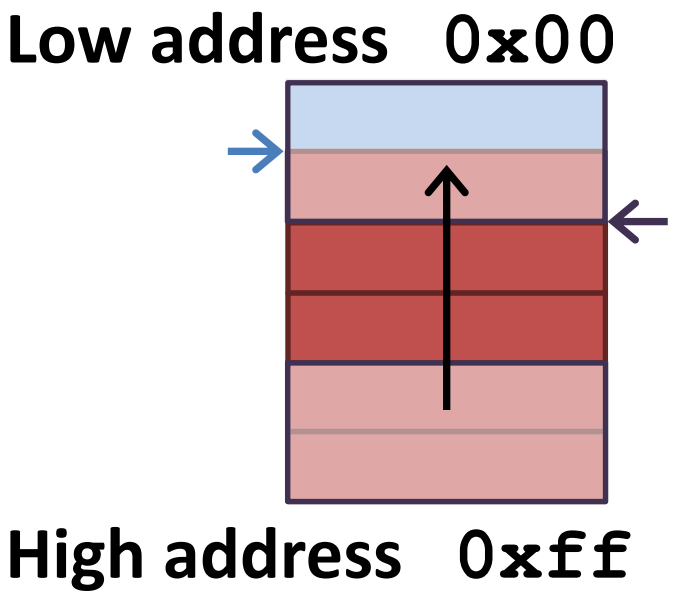
# C stack frames (x86 specific)

Grows toward lower address

Starts ~end of VA space

Two related registers

- %esp - Stack Pointer
- %ebp - Frame Pointer

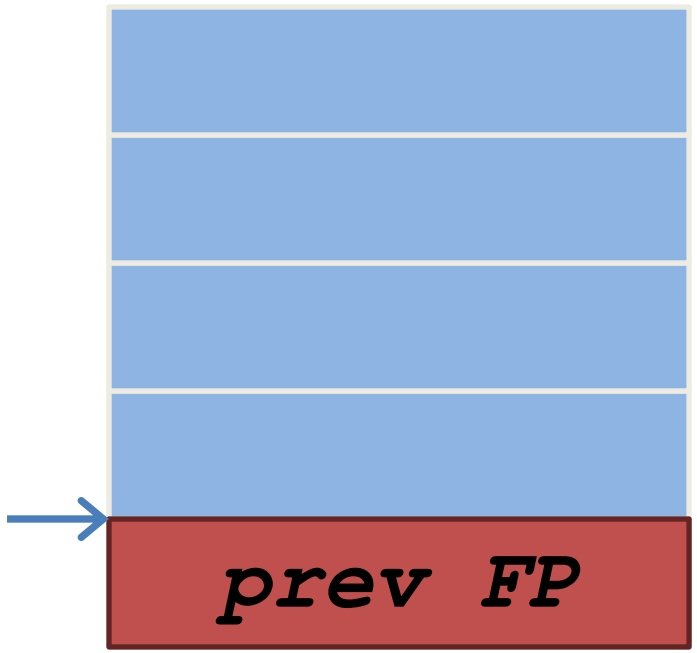


# example.c

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

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

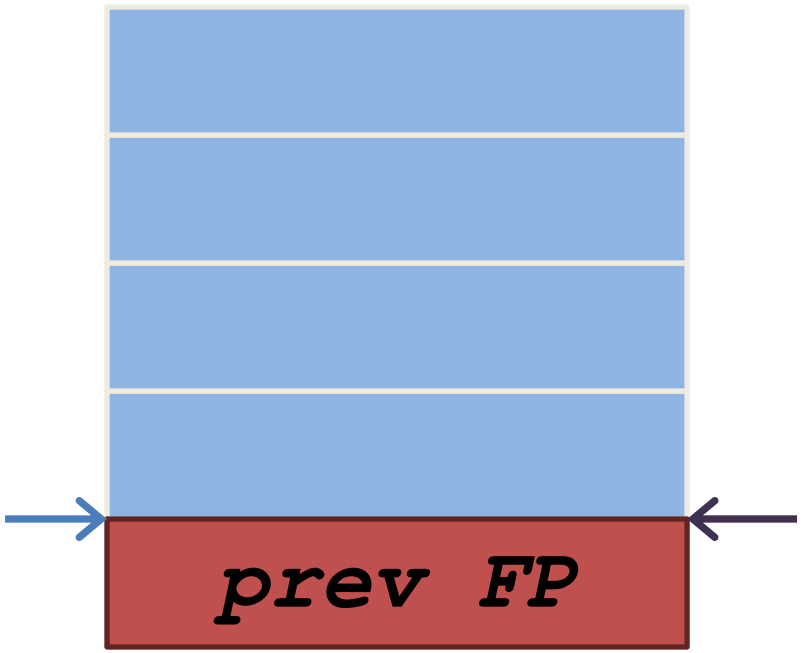
# example.s (x86)

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



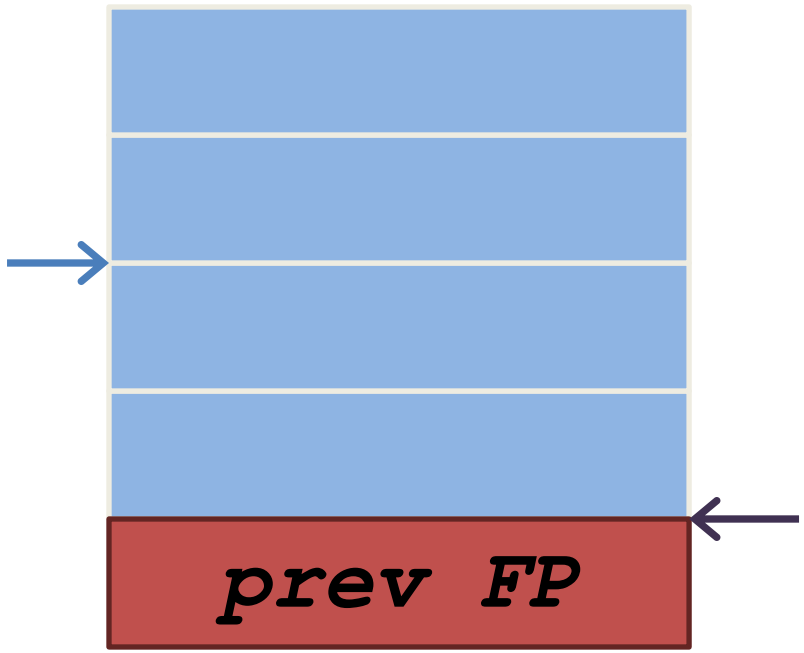
# example.s (x86)

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



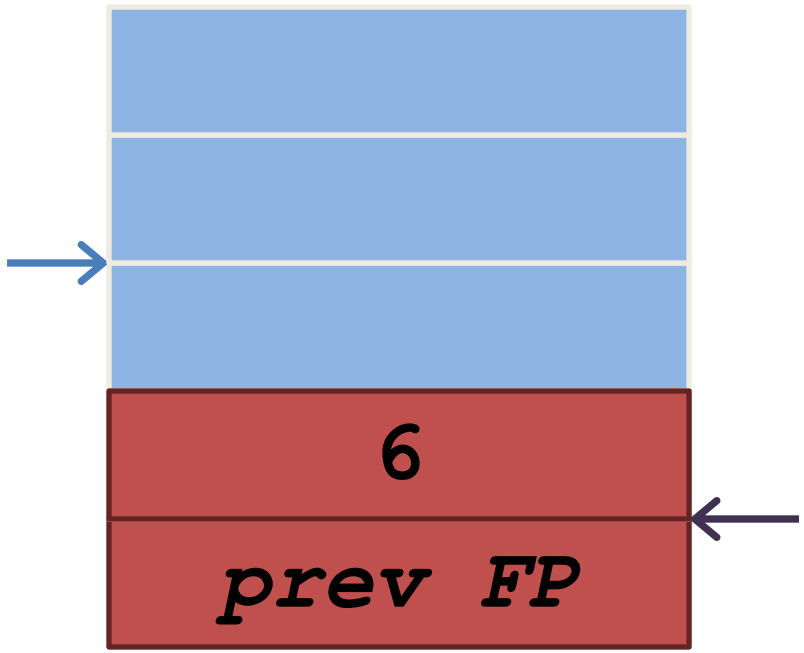
# example.s (x86)

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



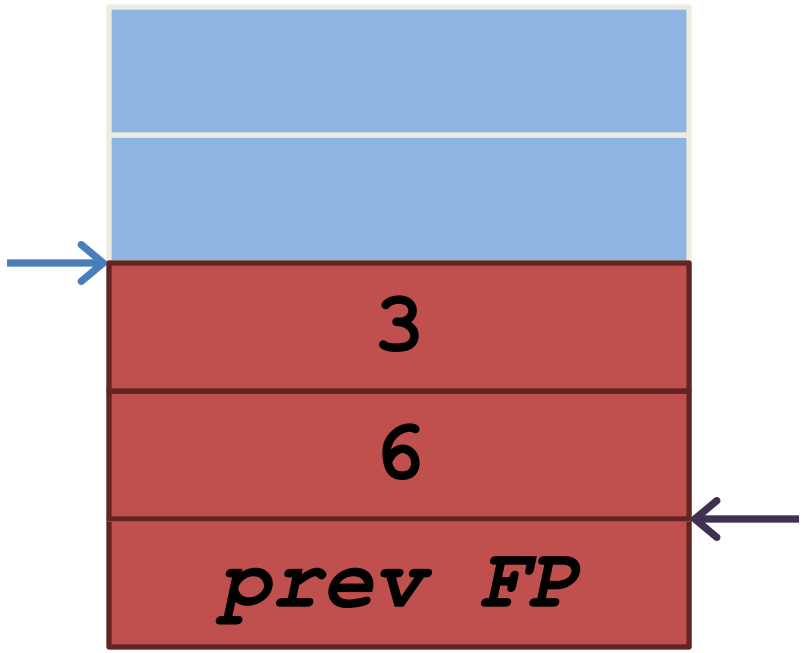
# example.s (x86)

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



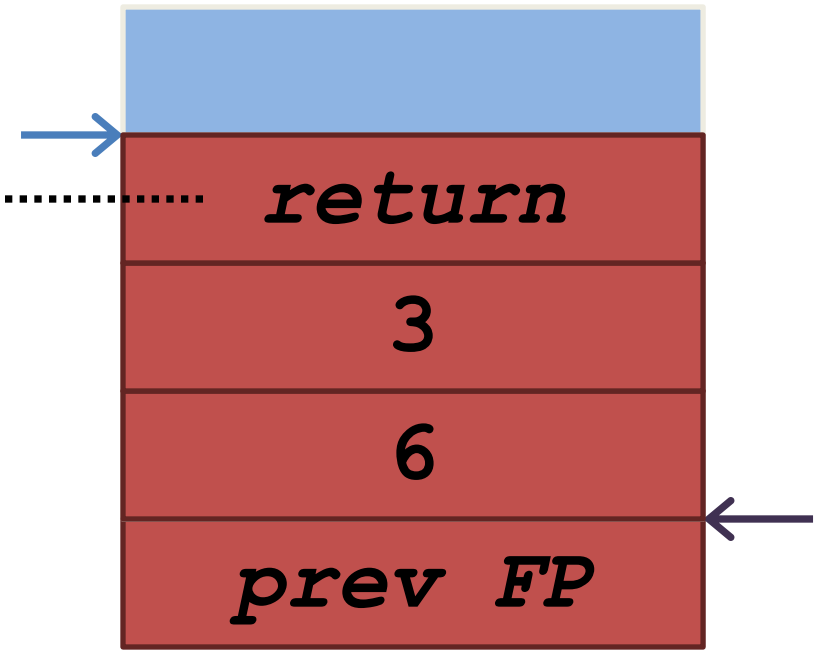
# example.s (x86)

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



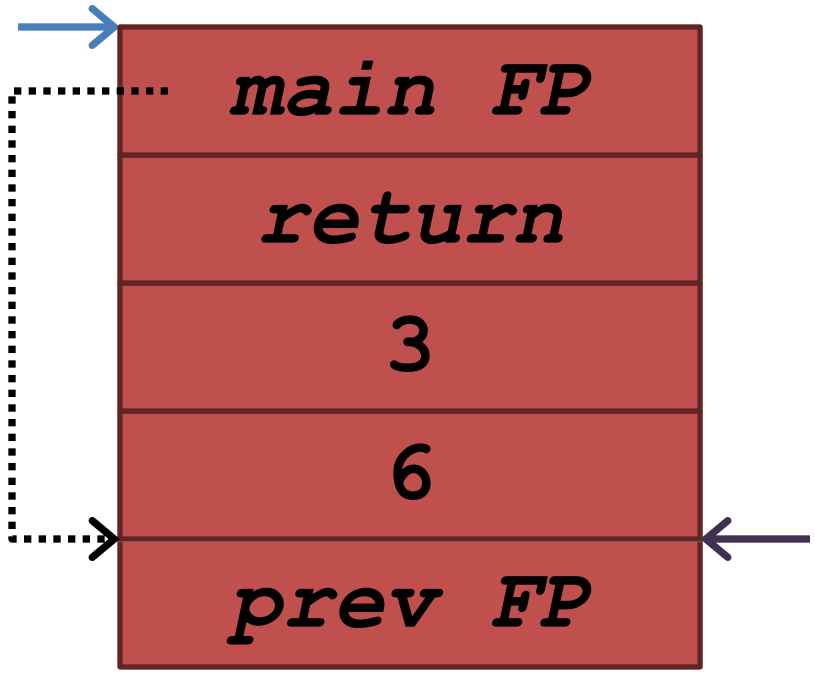
# example.s (x86)

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



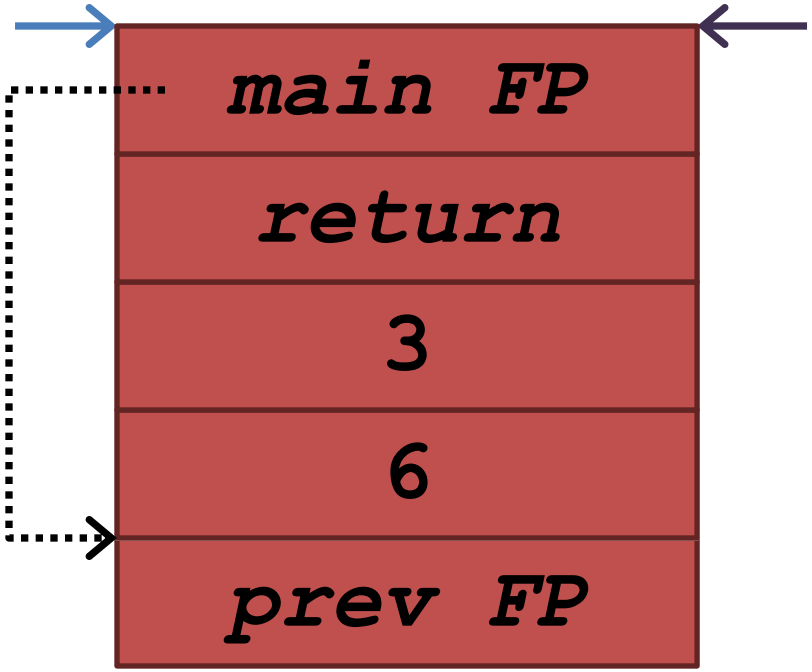
# example.s (x86)

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



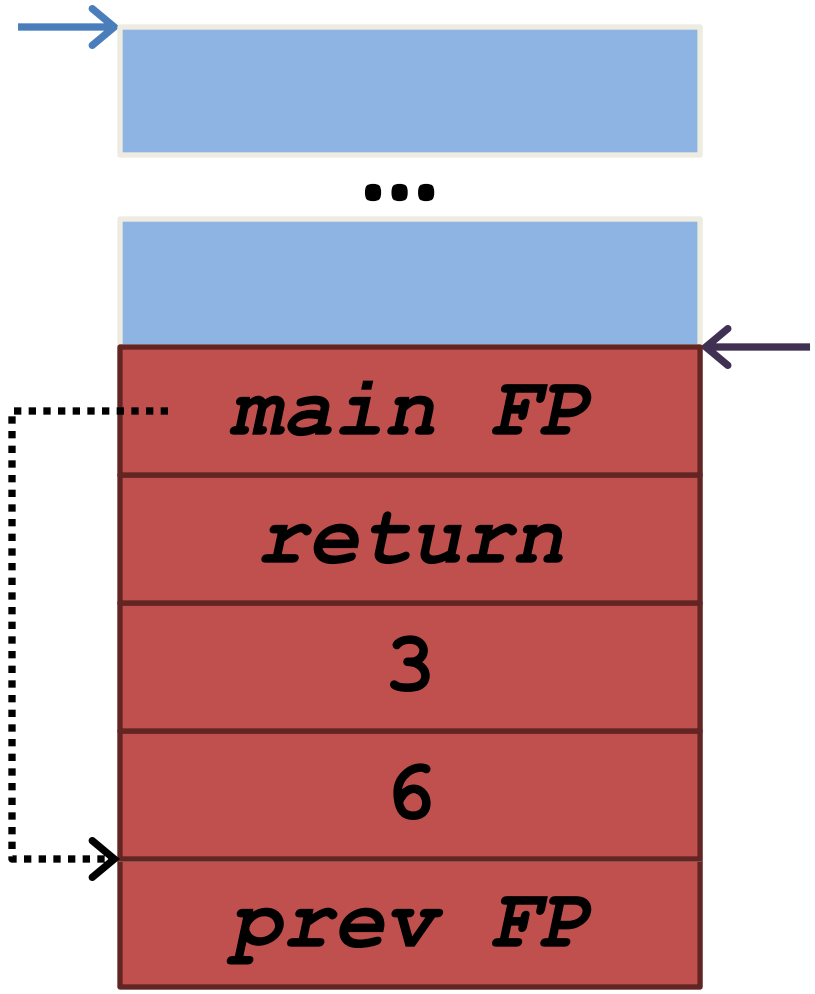
# example.s (x86)

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



# example.s (x86)

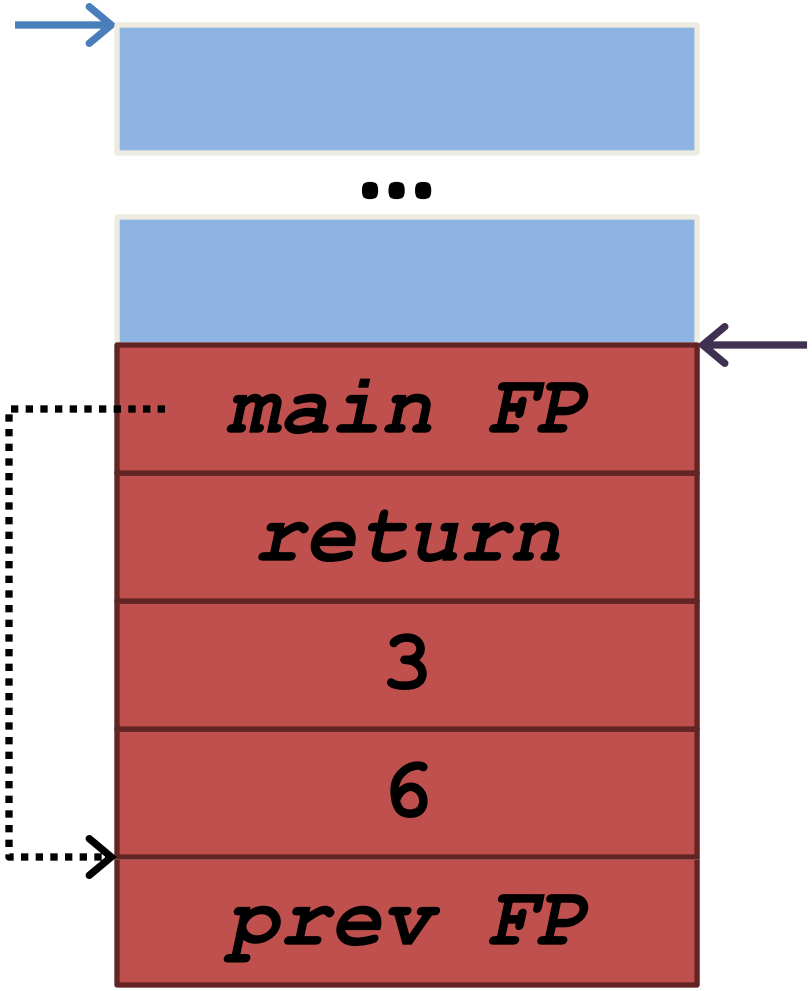
```
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    leave
    ret
```



# example.s (x86)

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

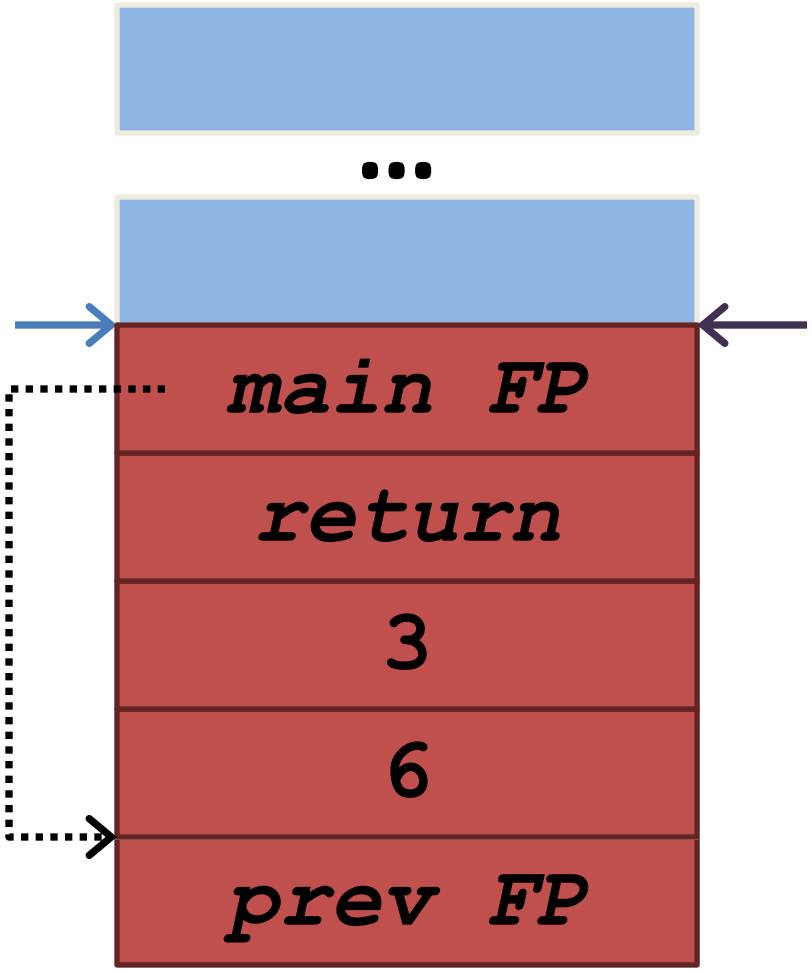
```
mov %ebp, %esp
pop %ebp
```



# example.s (x86)

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

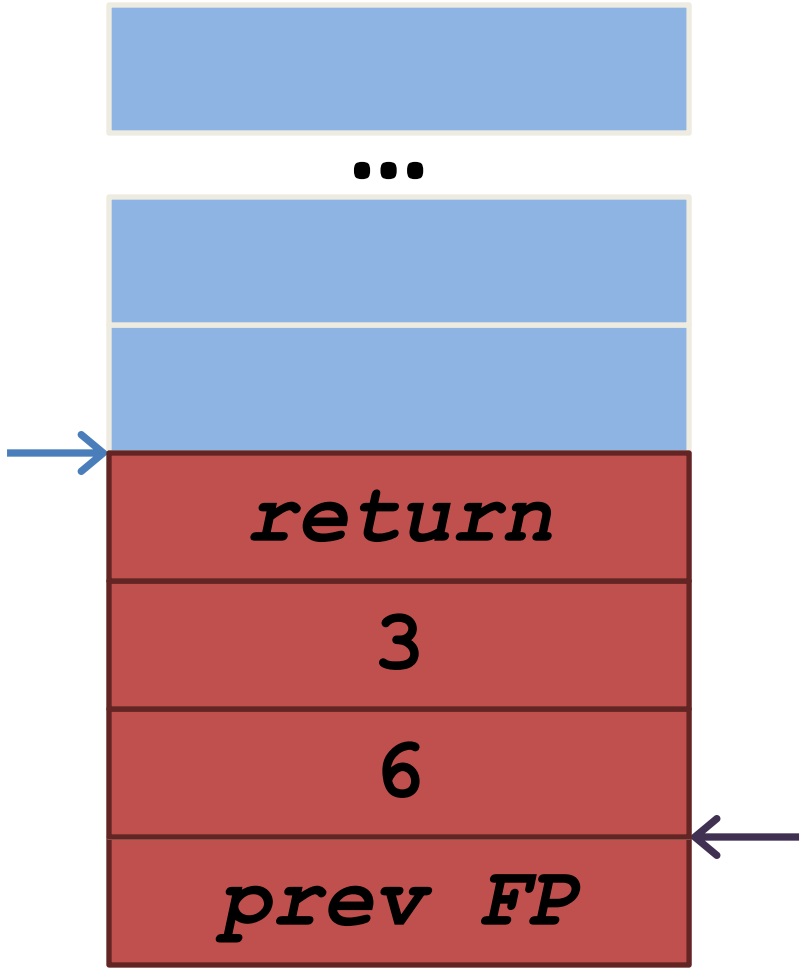
```
mov %ebp, %esp
pop %ebp
```



# example.s (x86)

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

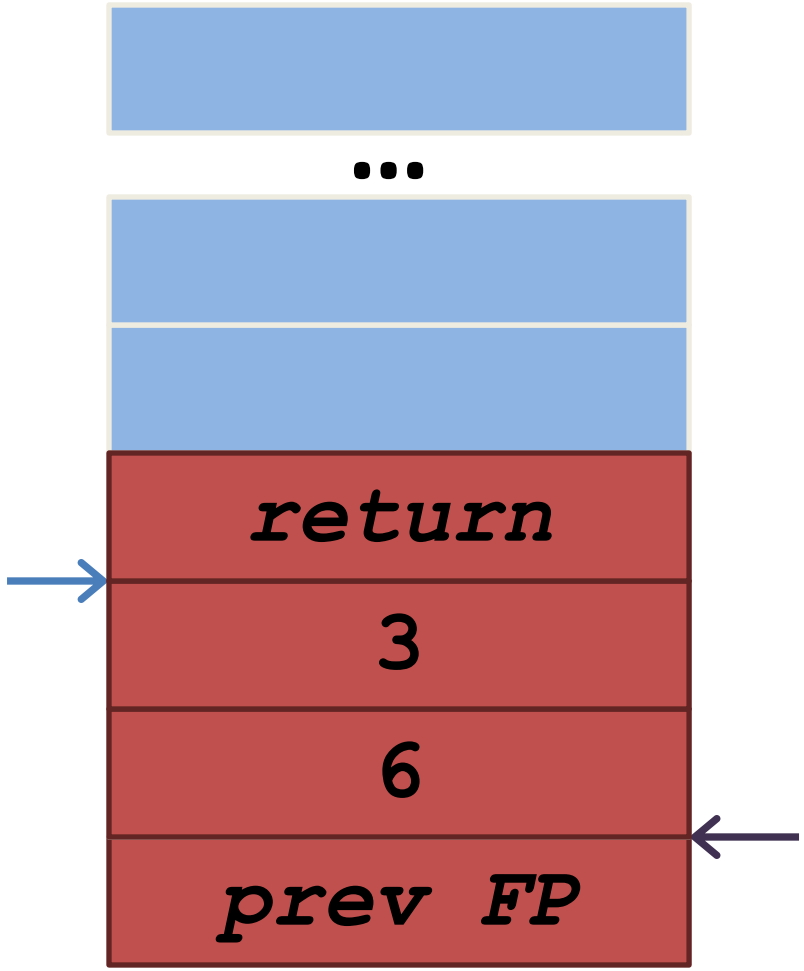
```
mov %ebp, %esp
pop %ebp
```



# example.s (x86)

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

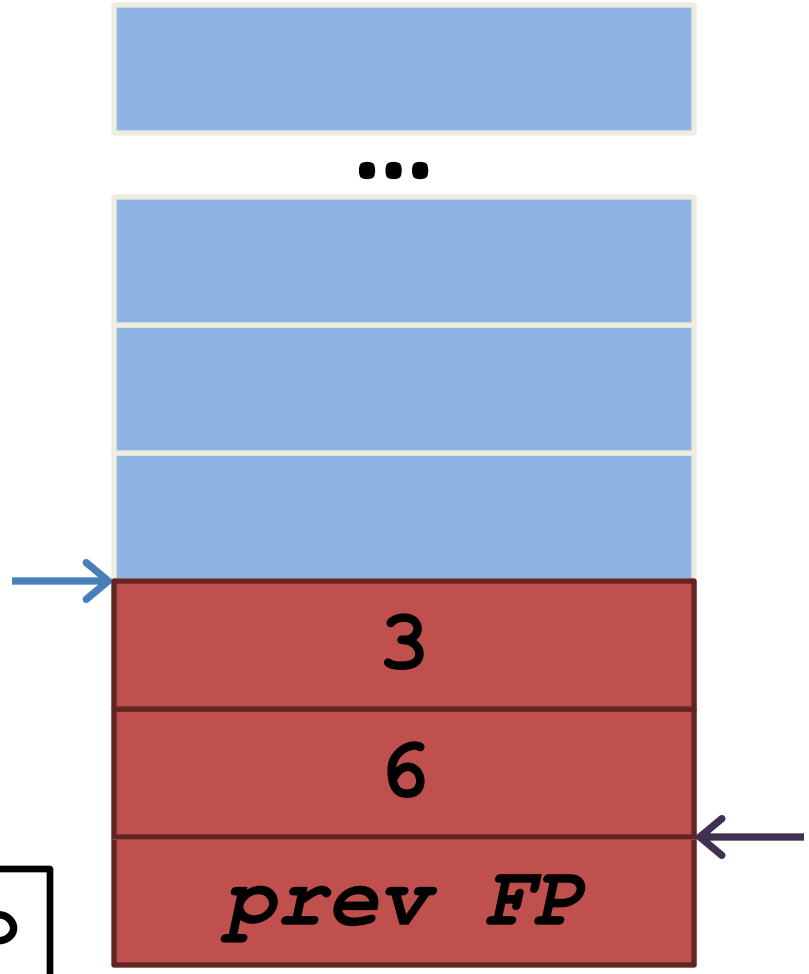
```
mov %ebp, %esp
pop %ebp
```



# example.s (x86)

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

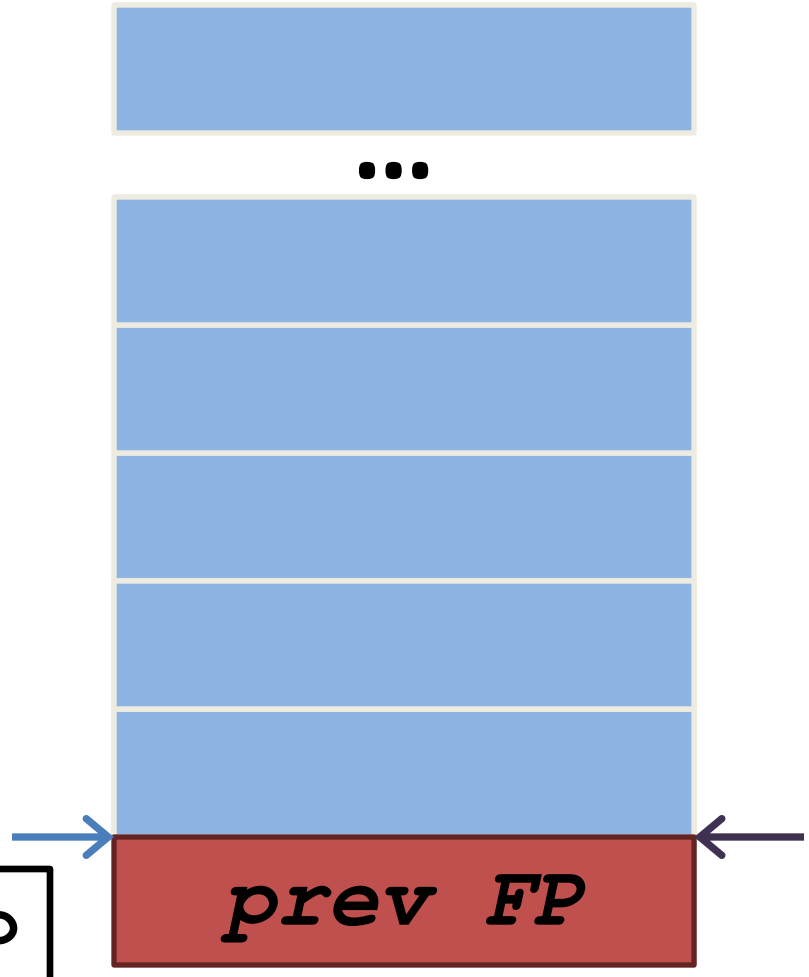
```
mov %ebp, %esp
pop %ebp
```



# example.s (x86)

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

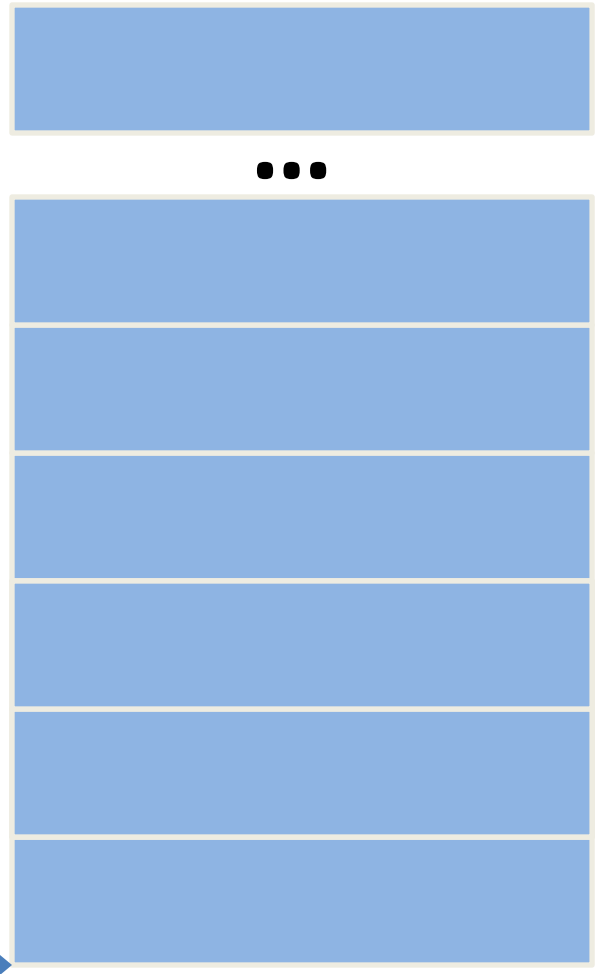
```
mov %ebp, %esp
pop %ebp
```



# 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
```

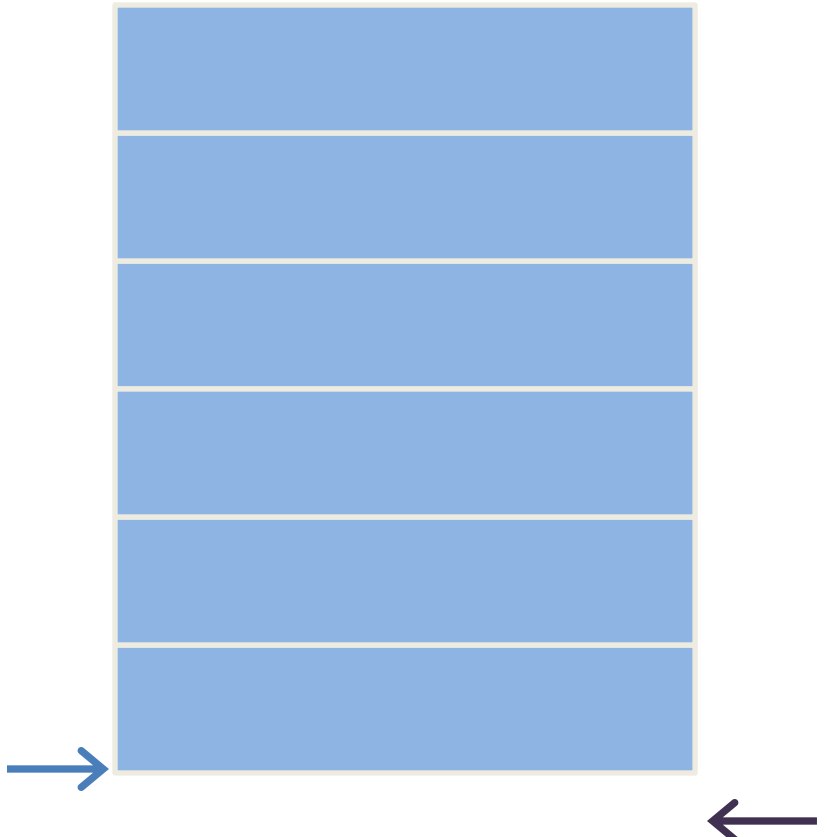


# Buffer overflow example

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

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

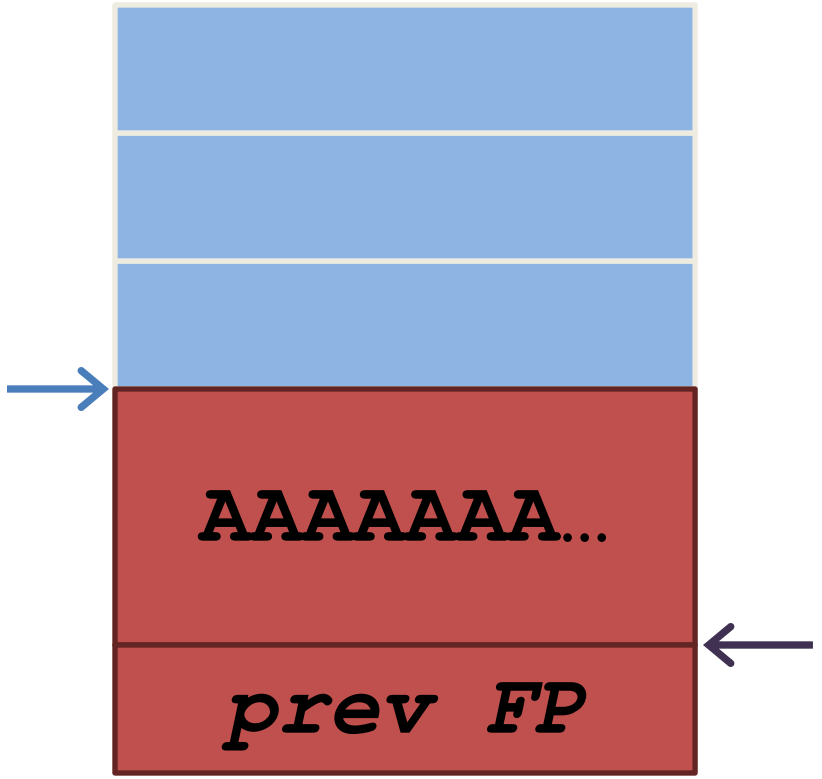
# Buffer overflow example

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

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



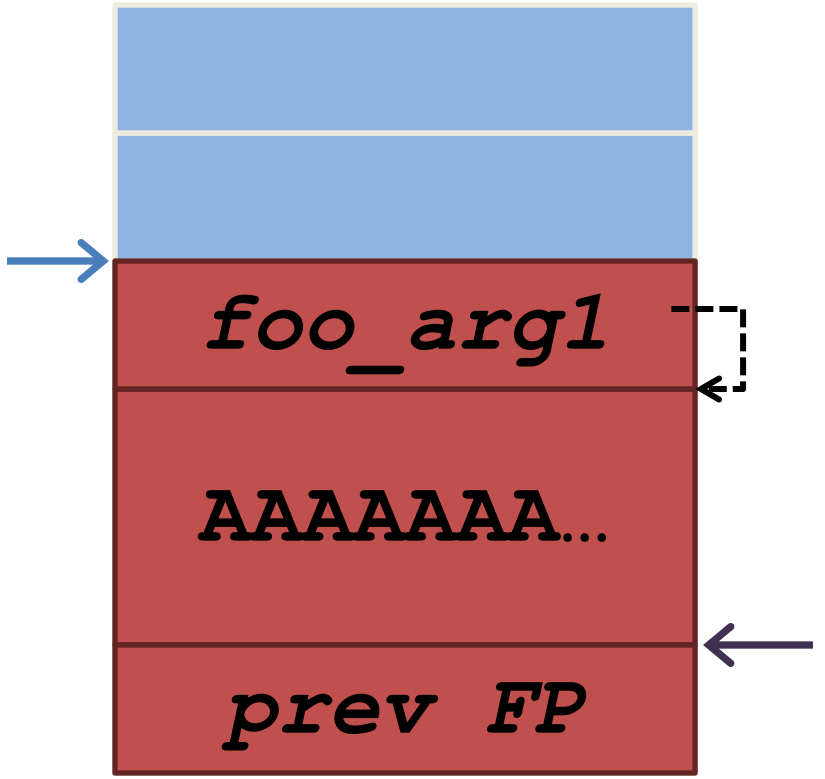
# Buffer overflow example

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

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



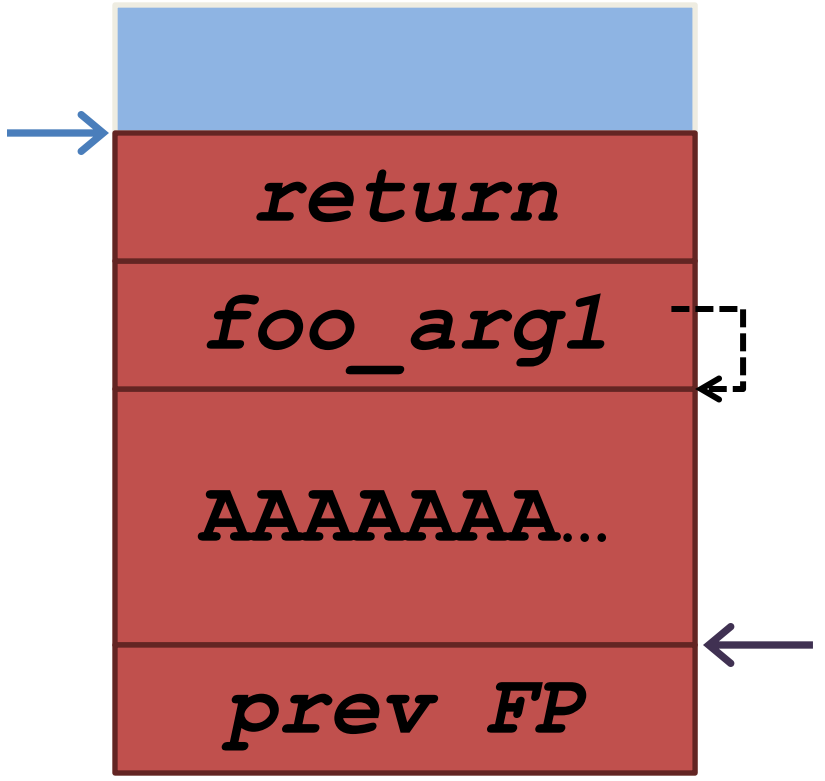
# Buffer overflow example

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

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



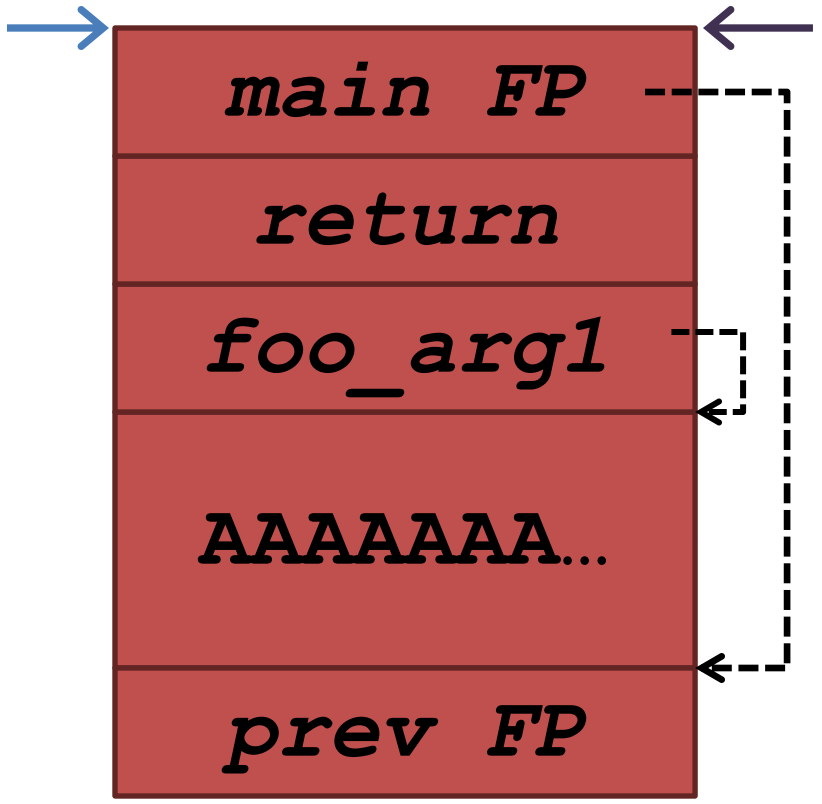
# Buffer overflow example

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

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



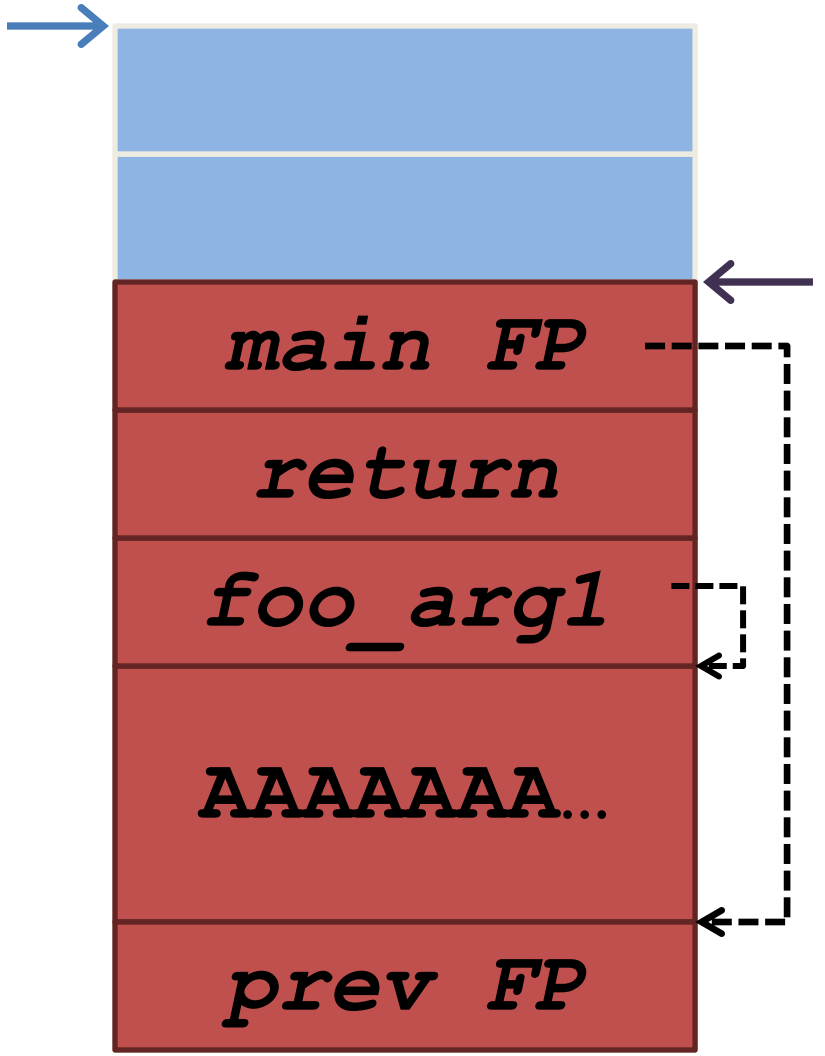
# Buffer overflow example

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

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



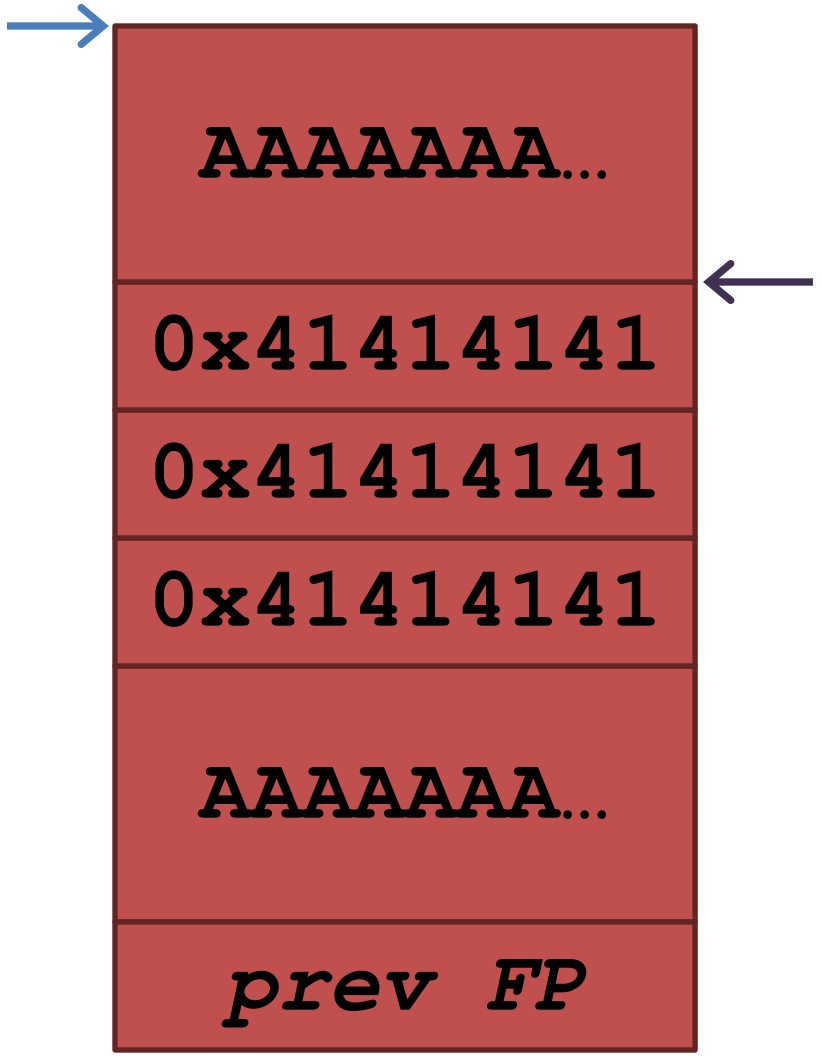
# Buffer overflow example

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

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



# Buffer overflow example

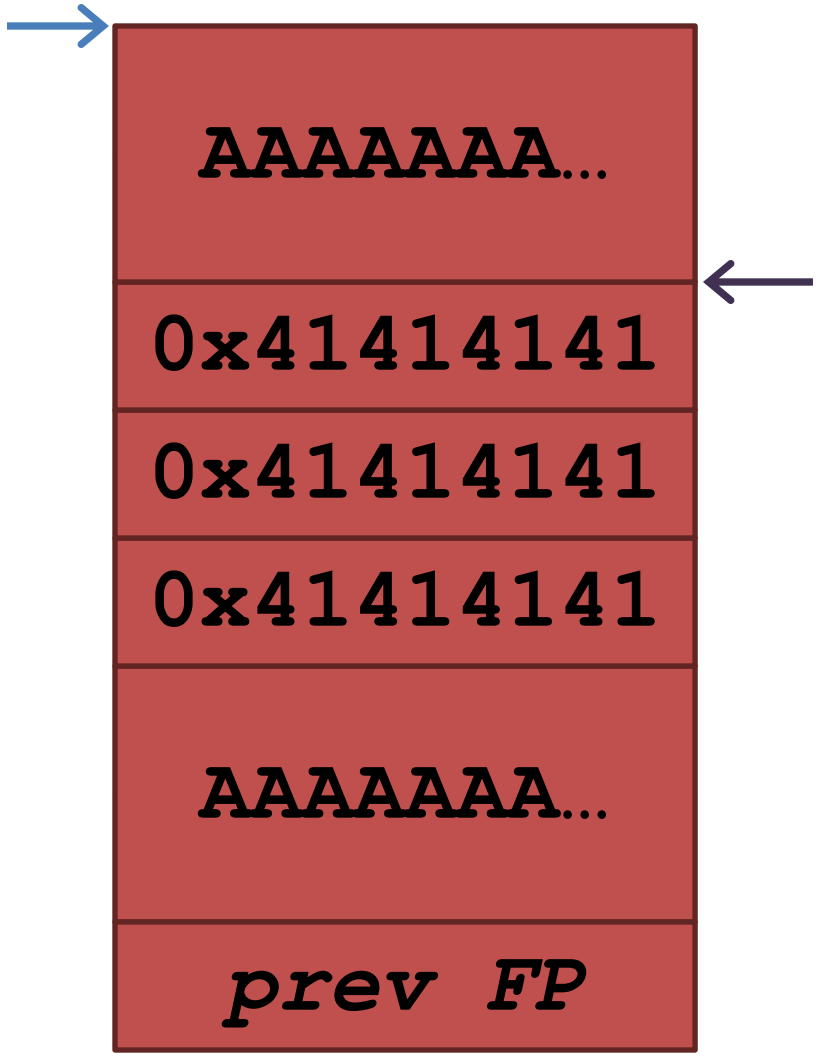
```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

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



# Buffer overflow example

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

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

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



# Buffer overflow example

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

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

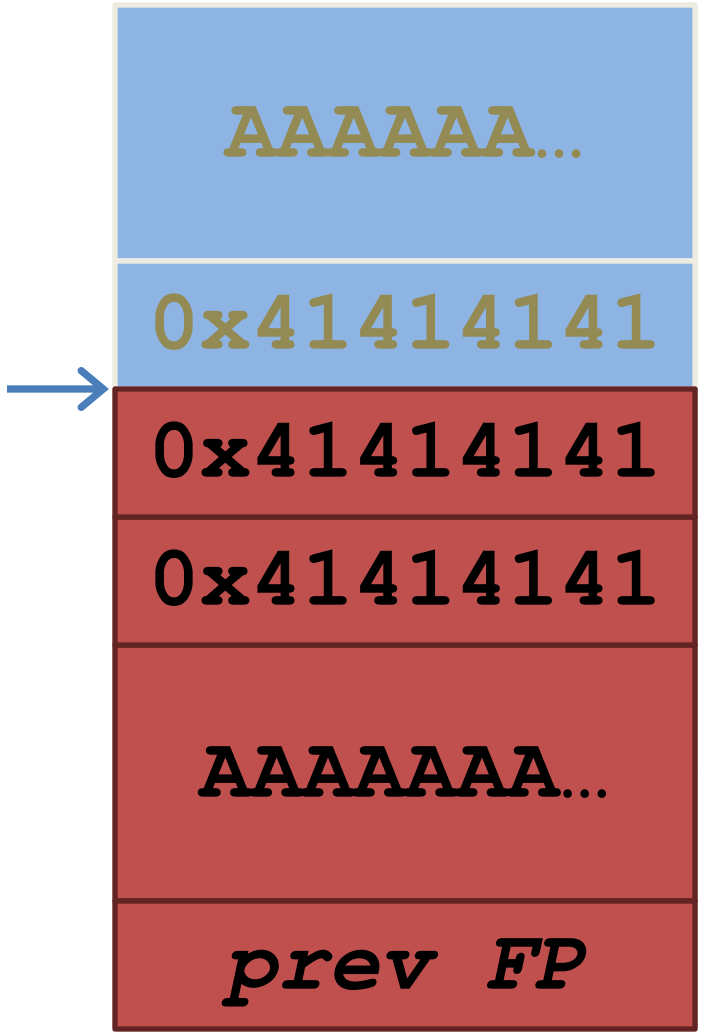
```
mov %ebp, %esp
pop %ebp
ret
```

# Buffer overflow example

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

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

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

?←

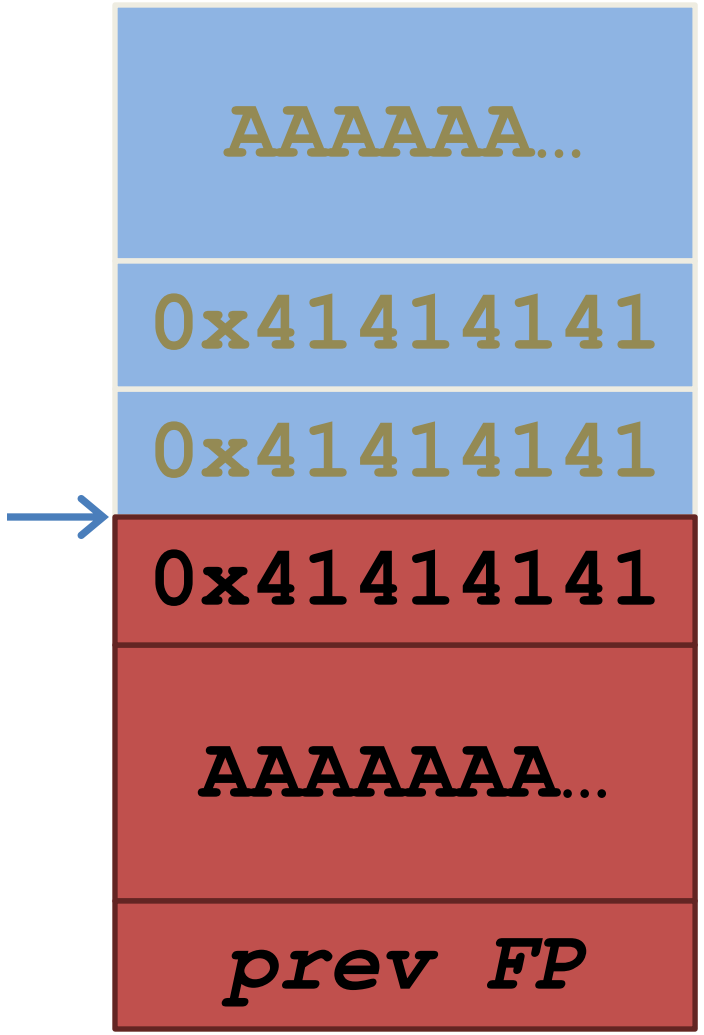


# Buffer overflow example

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

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

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

?←



# Buffer overflow example

**eip = 0x41414141**

**???**

# Buffer overflow FTW

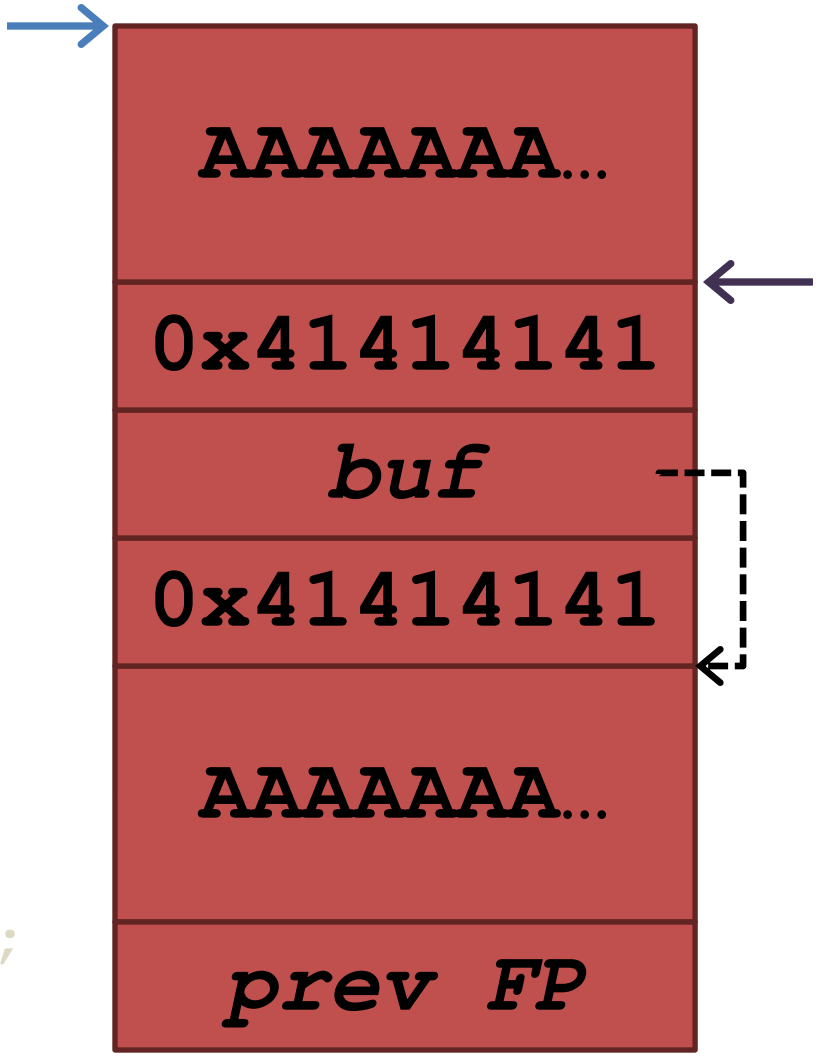
- Success! Program crashed!
- Can we do better?
  - Yes
    - How?

# 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

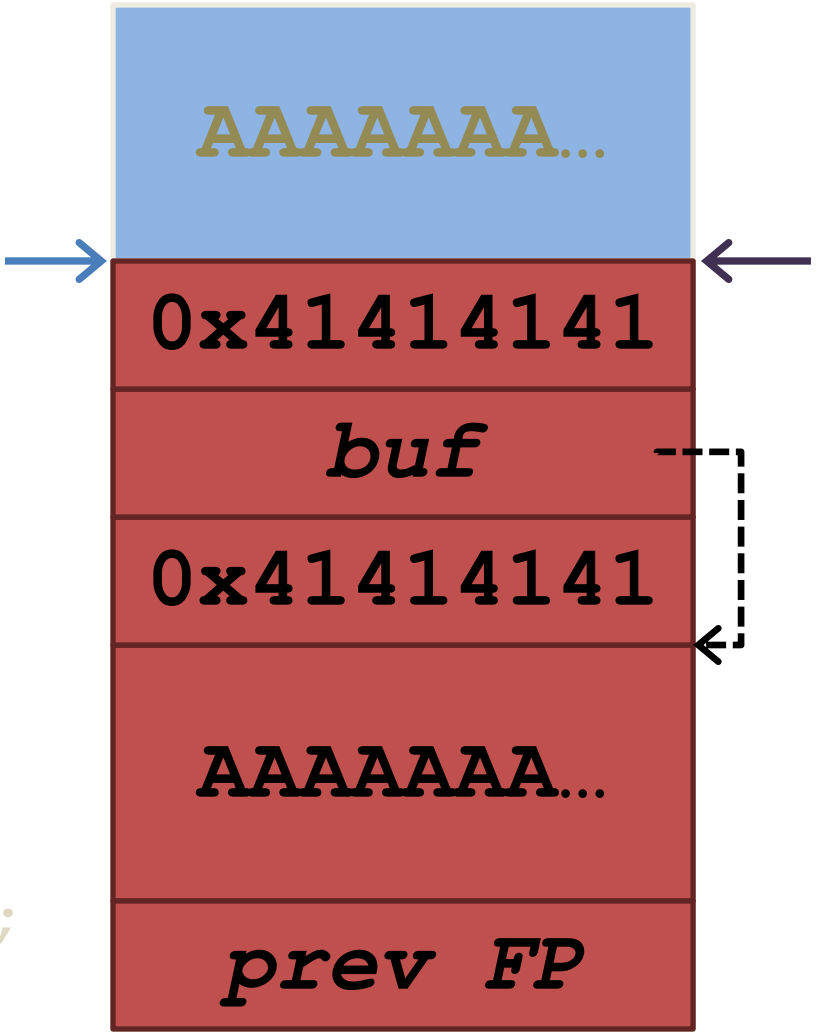
```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

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);
}
```

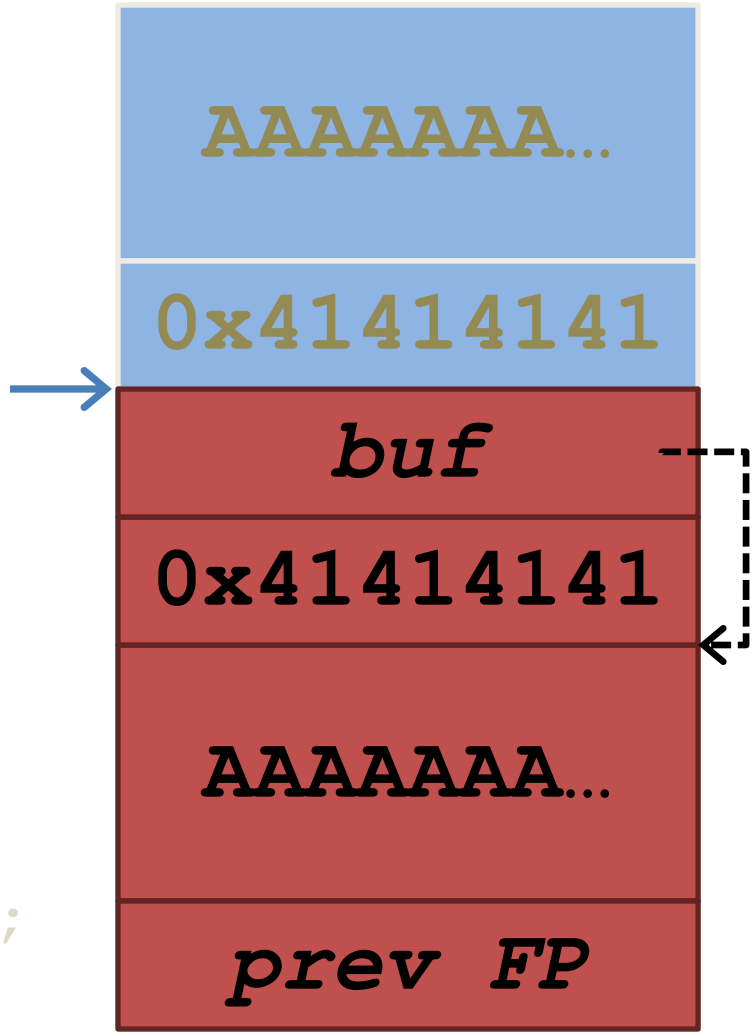
```
mov %ebp, %esp
pop %ebp
ret
```



# Exploiting buffer overflows

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

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

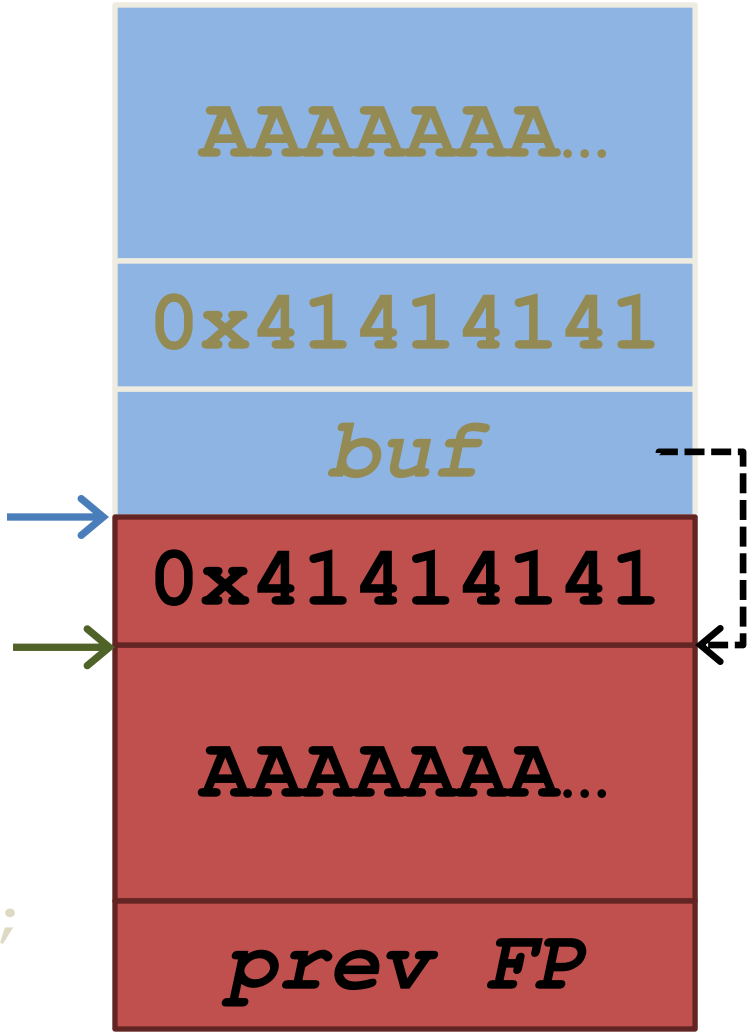
```
mov %ebp, %esp
pop %ebp
ret
```



# Exploiting buffer overflows

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

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

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



# What’s the Use?

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

# (slightly) more realistic vulnerability

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

# (slightly) more realistic vulnerability

```
int 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
```