CMSC414 Computer and Network Security

Memory Layout and Buffer Overflows

Yizheng Chen | University of Maryland
surrealyz.github.io

Jan 30, 2024

Announcements

- Instructor office hour, **Tuesday** 2:15pm - 3:15pm, in IRB 5224
- TA office hours, online, will post URLs on ELMS
  - Maurice Shih, maurices@umd.edu, **Wednesday** 9am - 11am
  - Benjamin Sela, benjsela@umd.edu, **Thursday** 9am - 11am
  - Nathan Reitinger, nlr@umd.edu, **Friday** 2pm - 4pm
- Keep an eye out for Project 1 Announcement from ELMS

Agenda

- Number representation, how computers run programs
- Memory layout
- x86 assembly, stack frames
- Buffer overflow

Threat Model

- Attacker's goal:
  - Take over the entire target machine (e.g., web server)
- Attacker's knowledge:
  - Need to know CPU and OS used on the target machine
  - Our examples are for x86 running Linux
  - Details vary slightly between CPUs and OSs

Number Representation

- In computers, all data is represented as bits
  - Bit: a binary digit, 0 or 1
  - Byte: 8 bits
  - Nibble: 4 bits, one hexadecimal digit
- 0b10001000100010001000100010001000:
  - **32 bit, 8 hexadecimal digits, 4 bytes**

Hexadecimal

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Binary	Hexadecimal
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

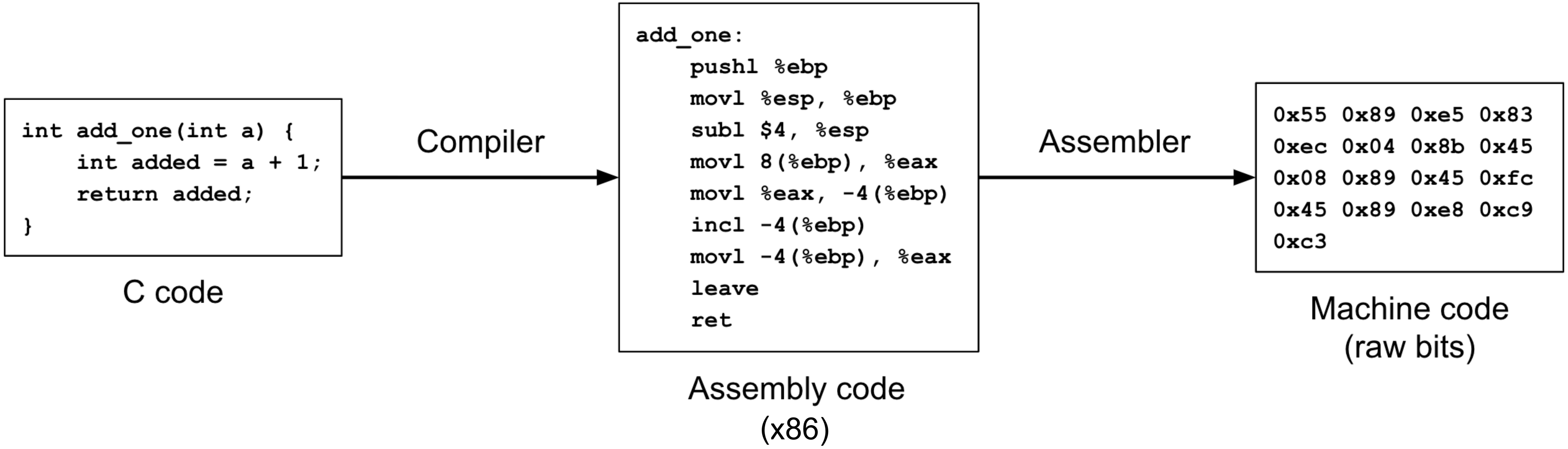
Hexadecimal

The byte **0b**11000110 can be written as **0x**C6 in hex
For clarity, we add **0b** in front of bits and **0x** in front of hex

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Binary	Hexadecimal
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Compiler, Assembler, Linker, Loader



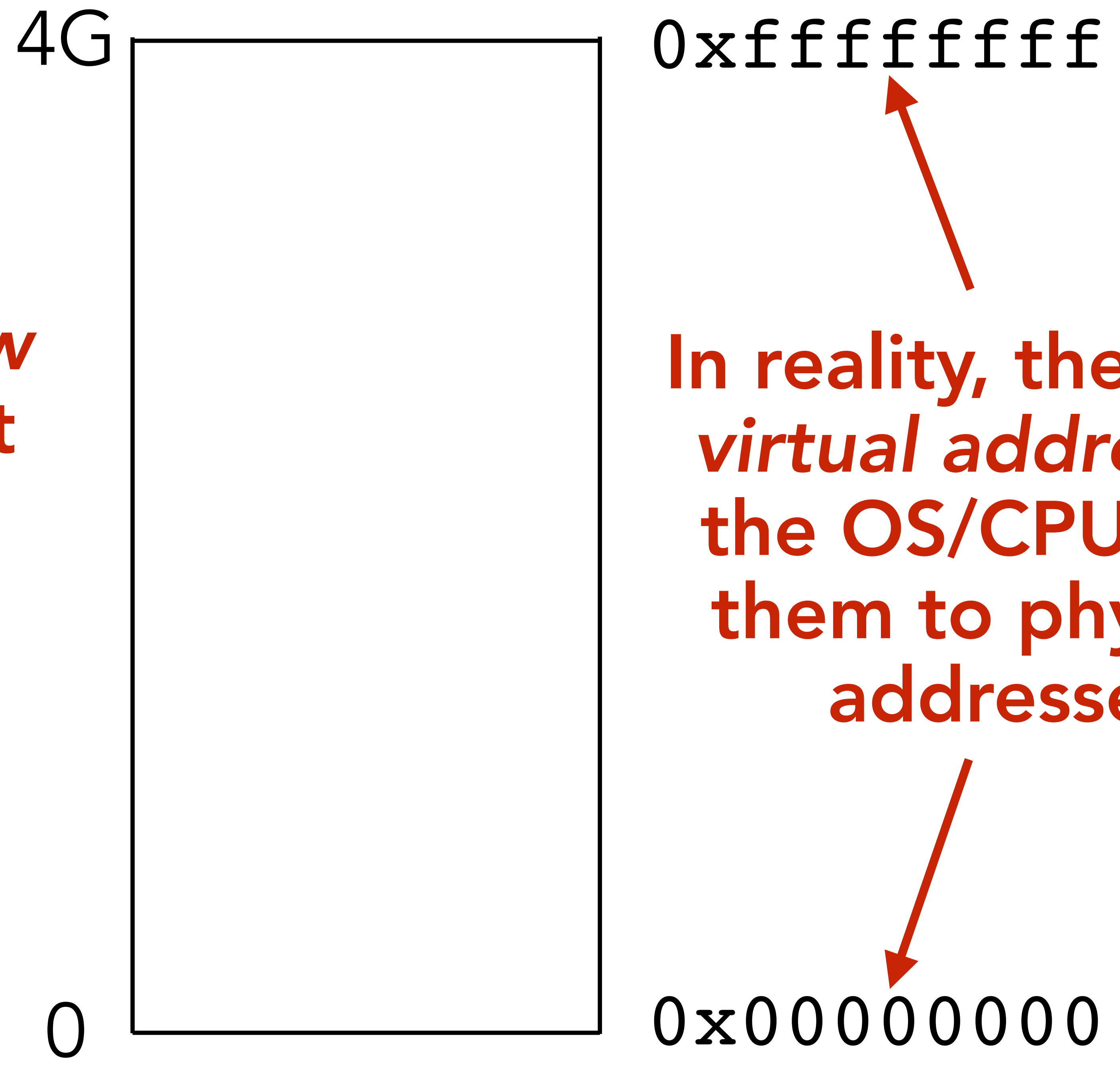
- Compiler: Converts C code into assembly code (e.g., x86)
- Assembler: Converts assembly code into machine code (raw bits)
- Linker: Deals with dependencies and libraries
- Loader: Sets up memory space and runs the machine code

Memory Layout

- At runtime, the loader tells the OS to give your program a big blob of memory
- On a 32-bit system, the memory has 32-bit addresses
  - On a 64-bit system, memory has 64-bit addresses
  - We use 32-bit systems in this class
- Each address refers to one byte, so 2^{32} bytes of memory

Memory Layout

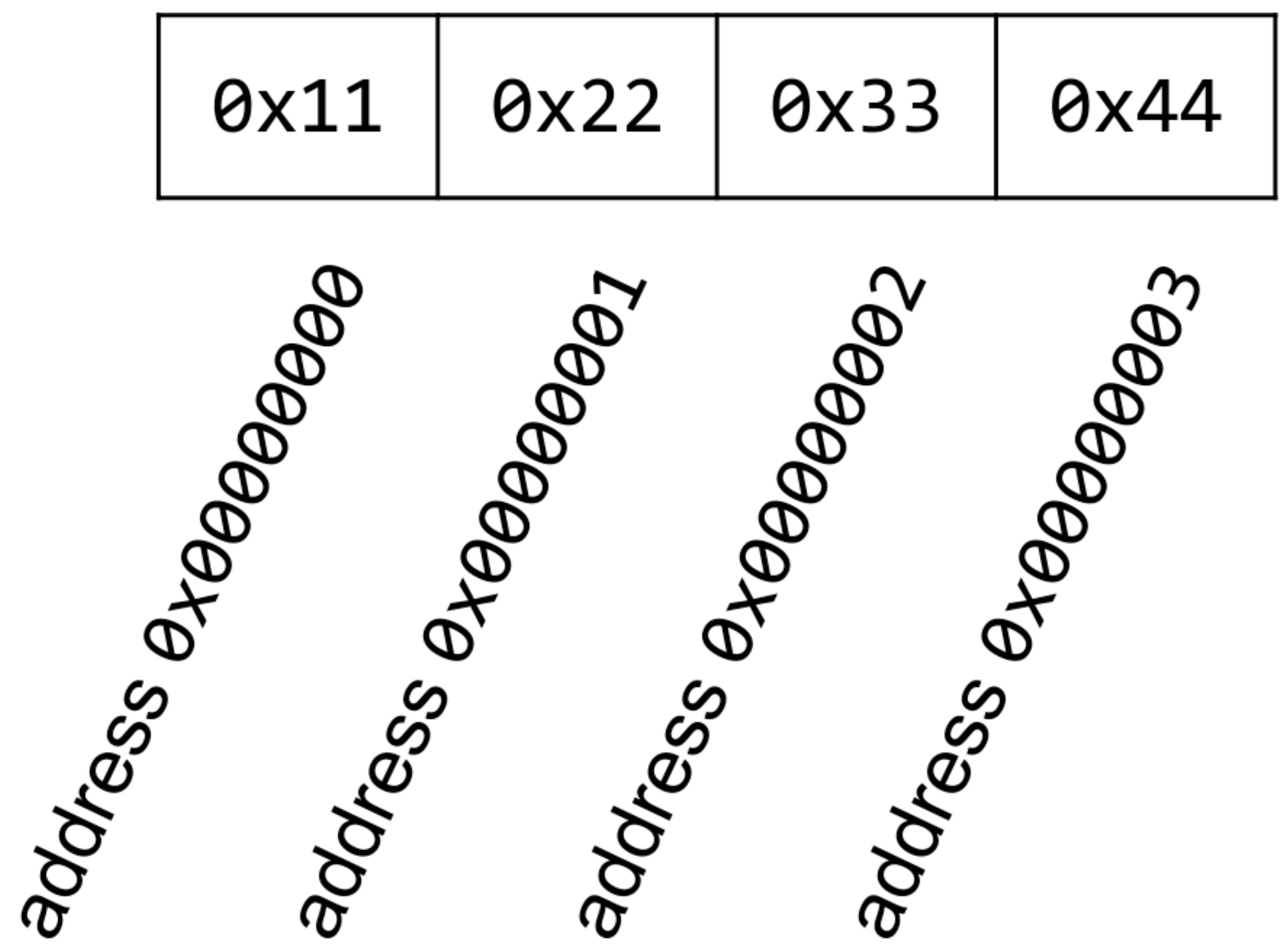
The *process's view* of memory is that it owns all of it



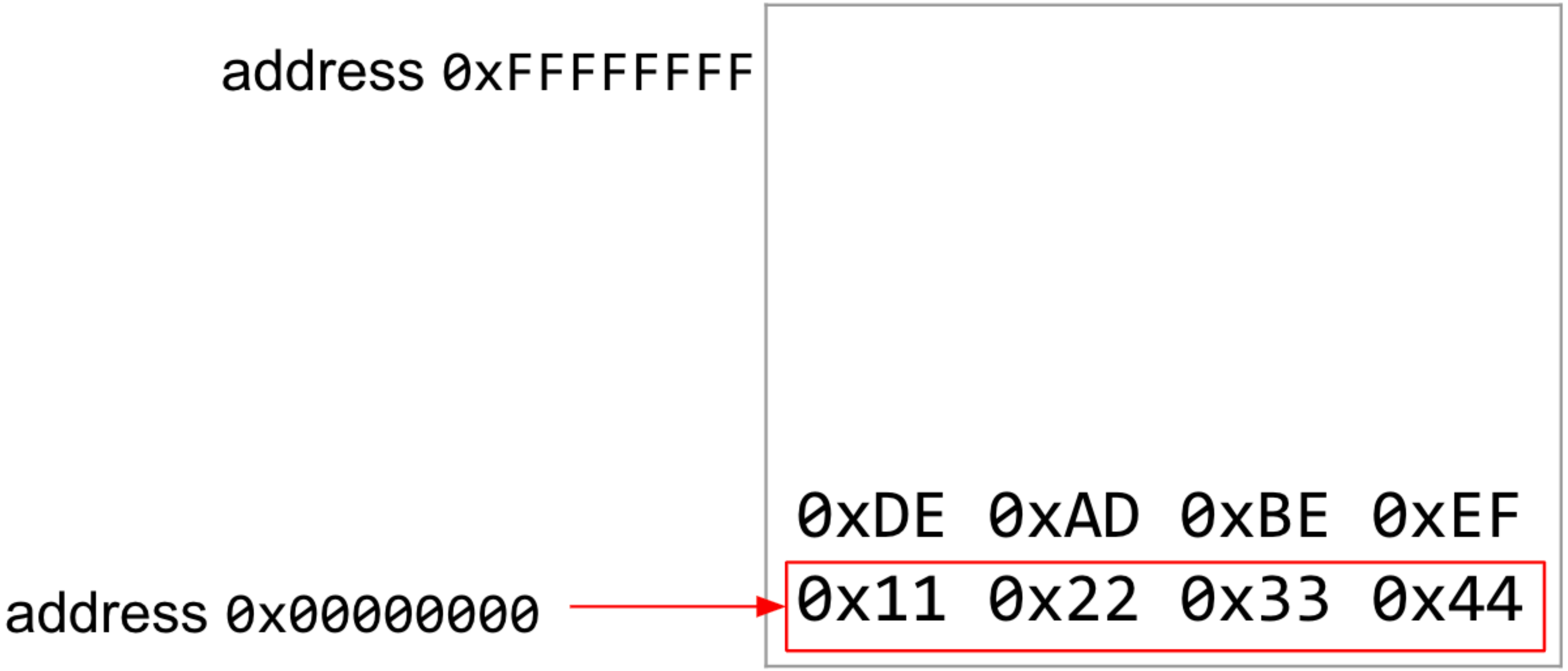
In reality, these are *virtual addresses*; the OS/CPU map them to physical addresses

Little-endian words

- One world: 4 bytes, 32 bits
- x86 is a **little-endian system**: the least significant byte is stored at the lowest address, and the most significant byte is stored at the highest address
- e.g., to store word 0x44332211

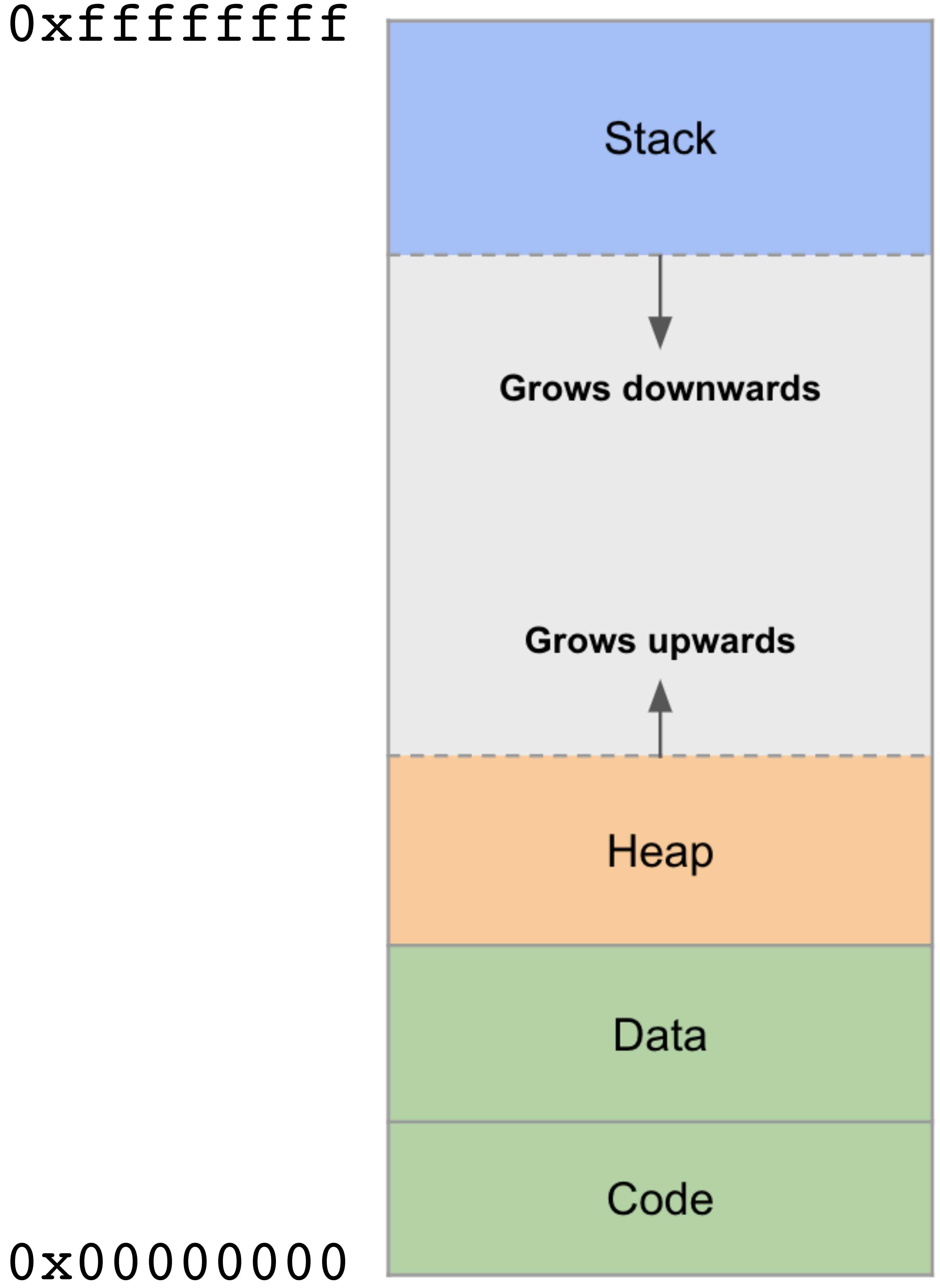


Combine Bytes on a Row to Form a Word



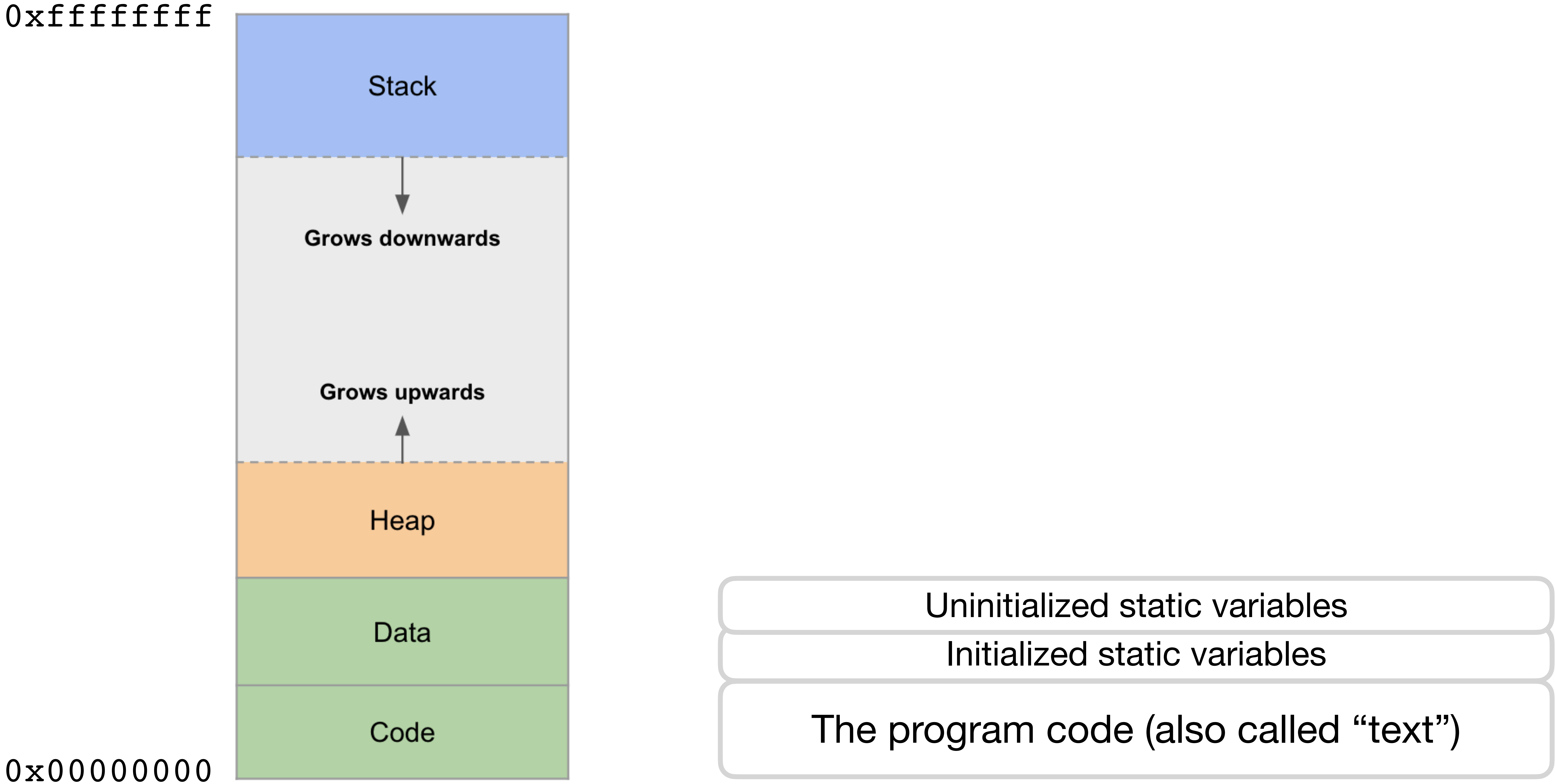
- What is the byte at address 0x00000000? 0x11
- What is the word at address 0x00000000? 0x44332211
- What is the byte / word at address 0x00000004?

x86 Memory Layout

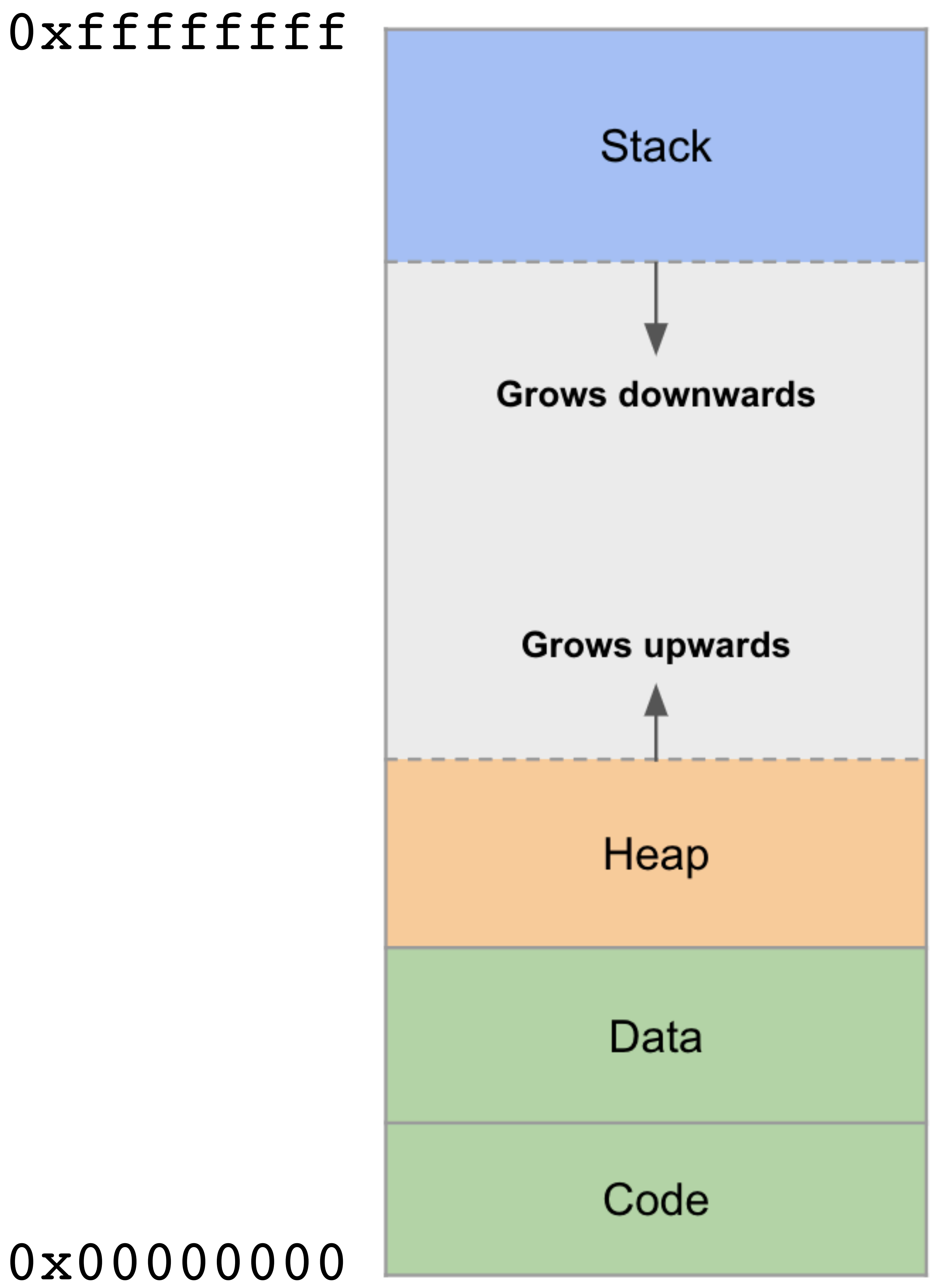


The program code (also called “text”)

x86 Memory Layout



x86 Memory Layout



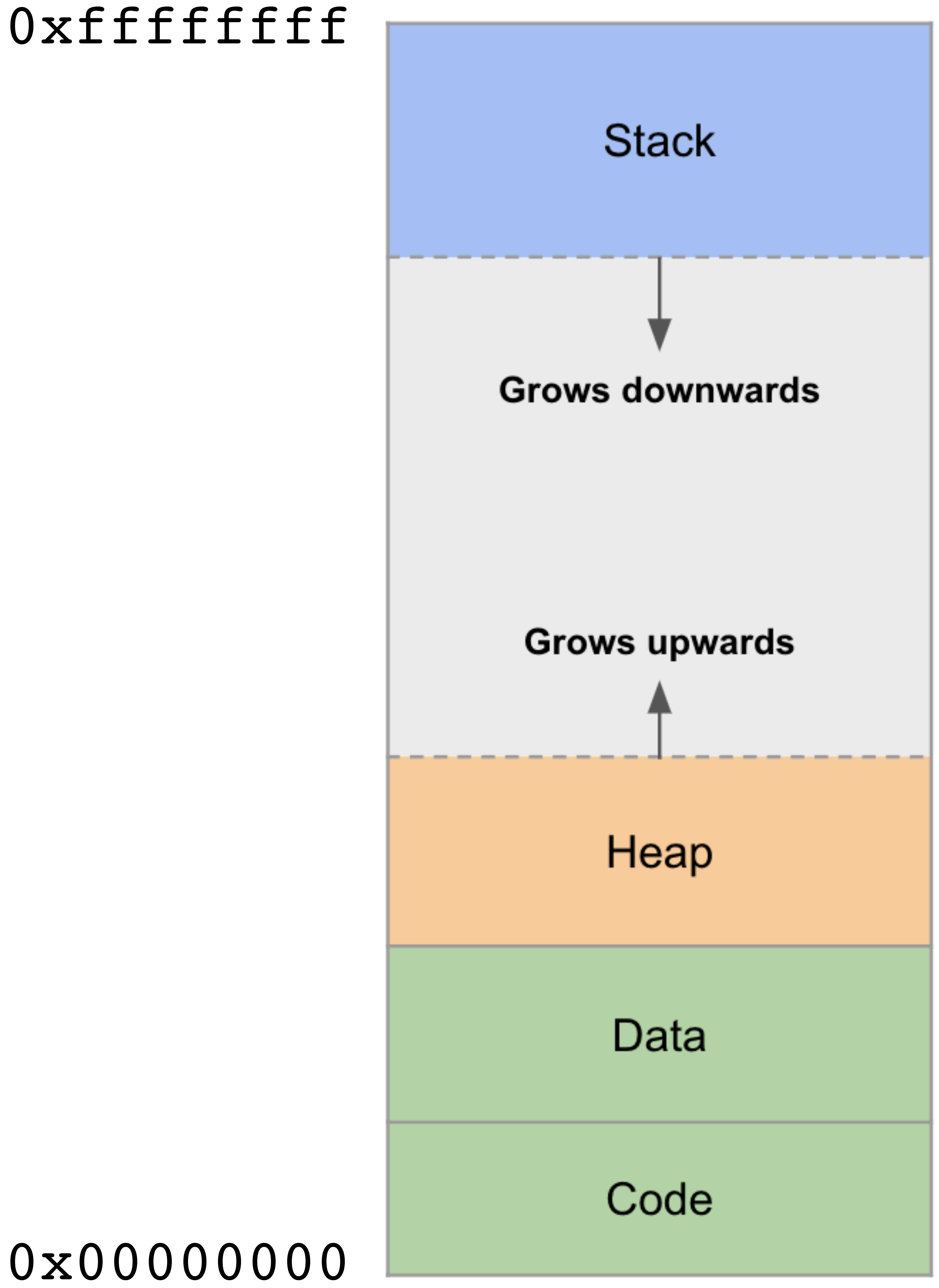
Dynamically allocated memory, e.g., using malloc and free

Uninitialized static variables

Initialized static variables

The program code (also called “text”)

x86 Memory Layout



Local variables and stack frames

Dynamically allocated memory, e.g., using malloc and free

Uninitialized static variables

Initialized static variables

The program code (also called “text”)

x86 instructions

- To understand how run-time attack works, we need to know just enough about x86 instructions
- Little-endian. Variable-length instructions: 1 to 16 bytes.
- Storage units on CPU (not in memory)
- Only 6 general-purpose registers:
  - eax, ebx, ecx, edx, esi, edi
- esp: stack pointer
- ebp: base pointer
- eip: instruction pointer

x86 instructions

- To understand how run-time attack works, we need to know just enough about x86 instructions
- Little-endian. Variable-length instructions: 1 to 16 bytes.
- Storage units on CPU (not in memory)
- Only 6 general-purpose registers:
  - eax, ebx, ecx, edx, esi, edi
- esp: stack pointer
- ebp: base pointer
- eip: instruction pointer

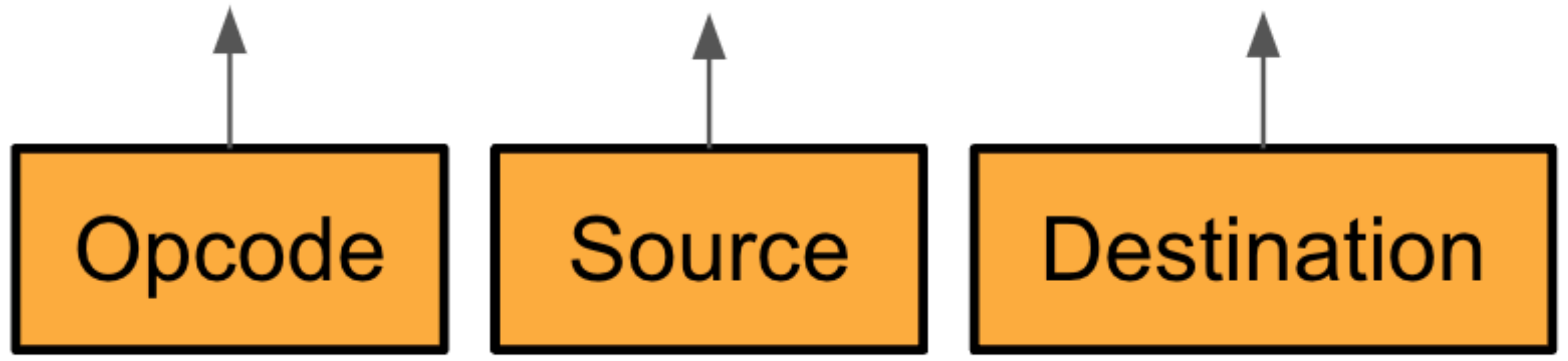
e: extended from 16 bit

x86 Syntax

- Register references are preceded with a percent sign `%`
  - **`Example: %eax, %esp, %edi`**
- Immediates are preceded with a dollar sign `$`
  - **`Example: $1, $414, $0xff`**
- Memory references use parentheses and can have immediate offsets
  - **`Example: 8(%esp) dereferences memory 8 bytes above the address contained in the stack pointer`**

x86 Assembly

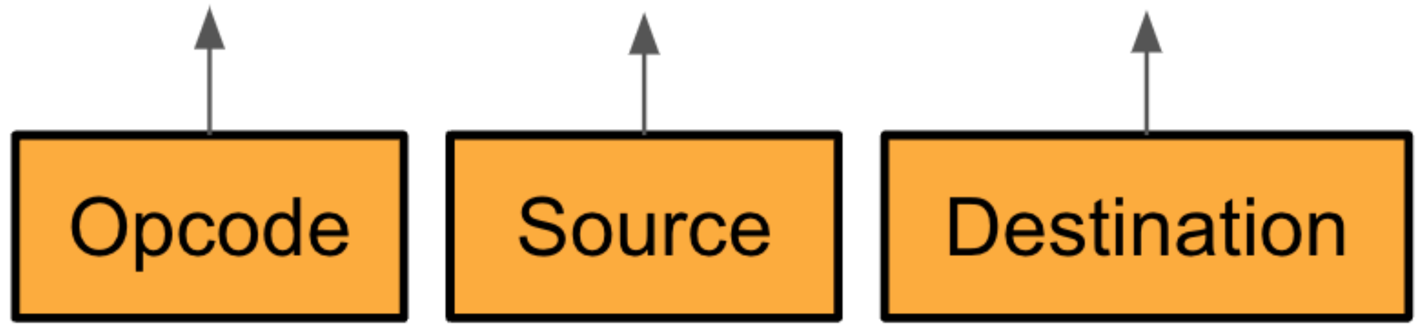
- Instructions are composed of an opcode and zero or more operands.
- `add     $0x8,    %ebx`



- Pseudocode: **`EBX = EBX + 0x8`**

x86 Assembly

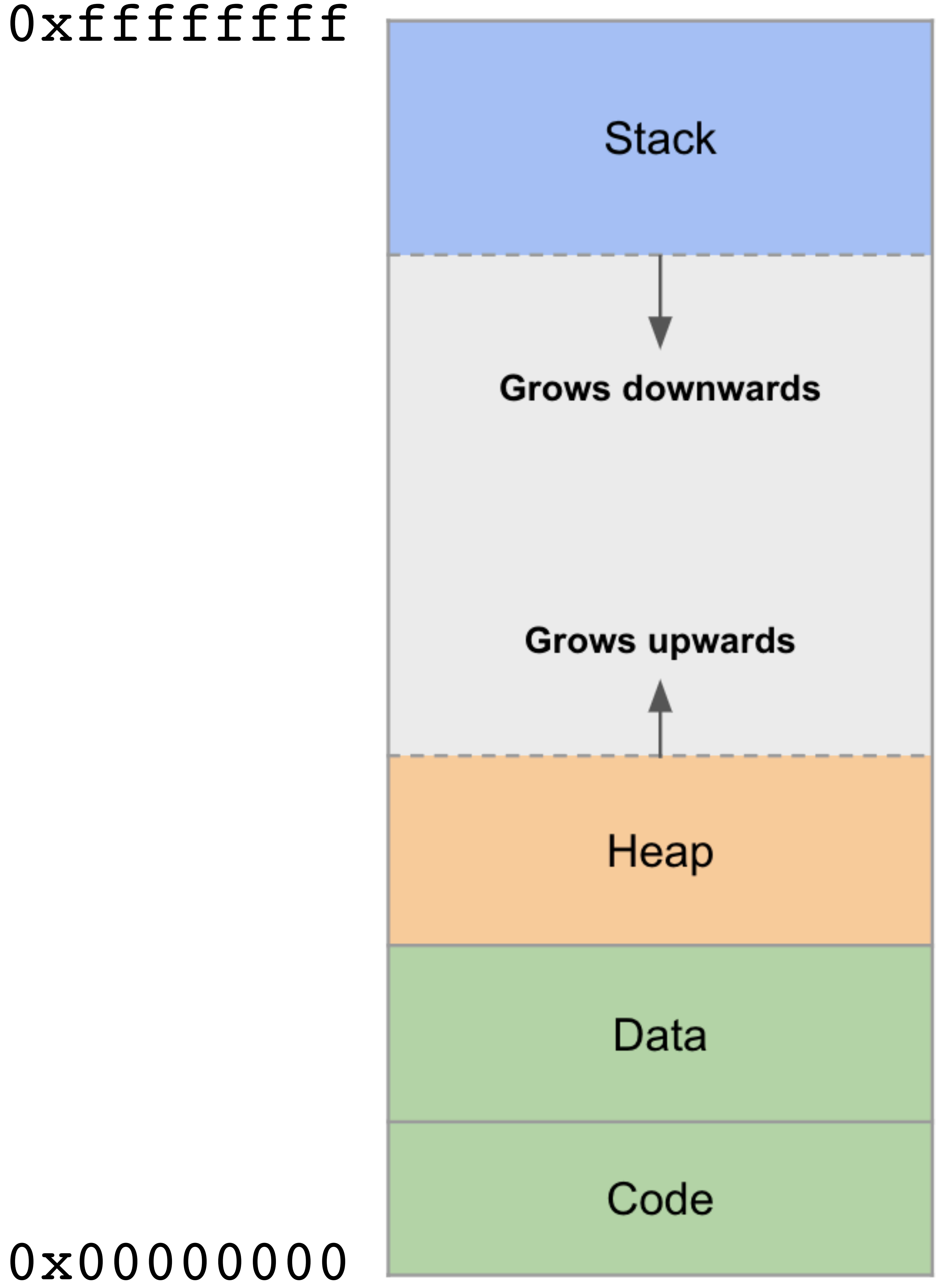
- **`xorl 4(%esi), %eax`**



- Pseudocode: **`EAX = EAX ^ *(ESI + 4)`**
- This is a memory reference, where the value at 4 bytes above the address in ESI is dereferenced, XOR'd with EAX, and stored back into EAX

Search for "x86 reference sheet"

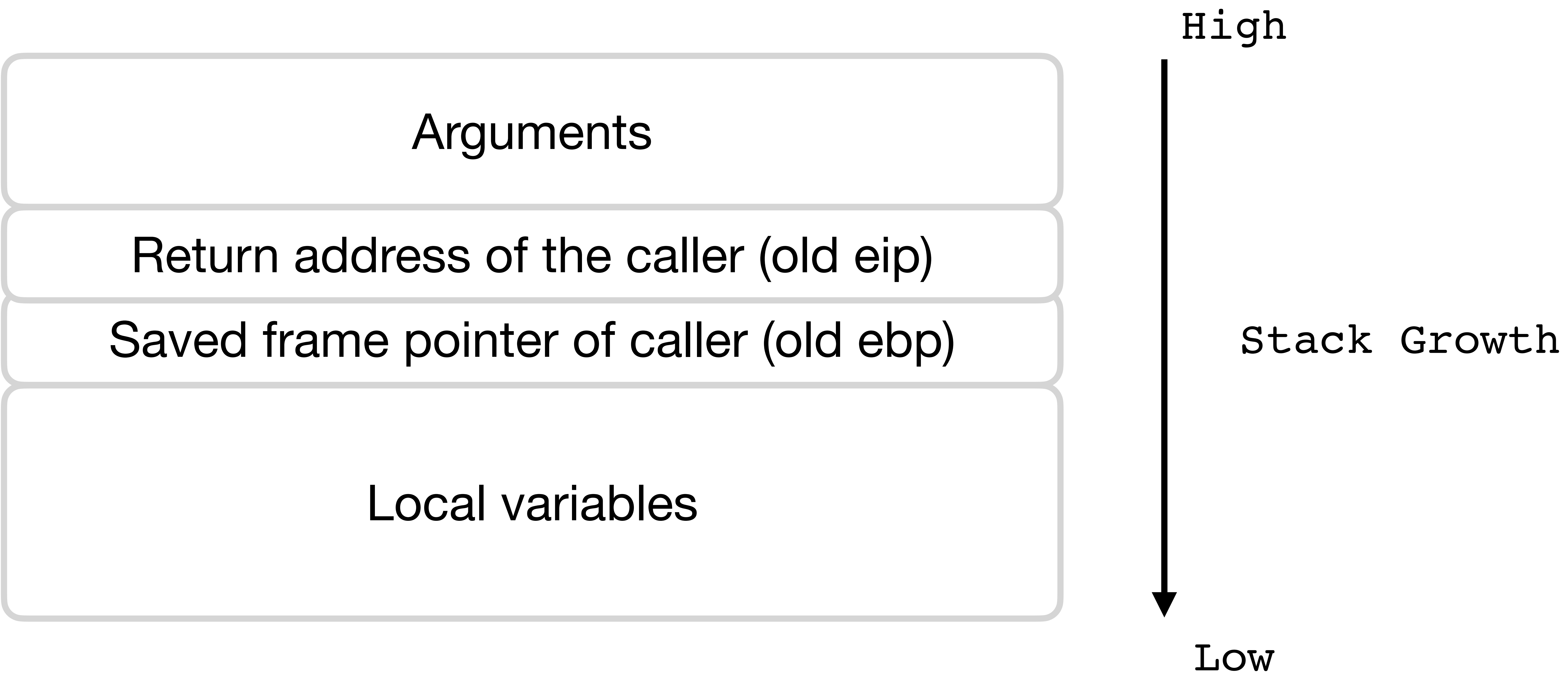
x86 Memory Layout



Local variables and stack frames

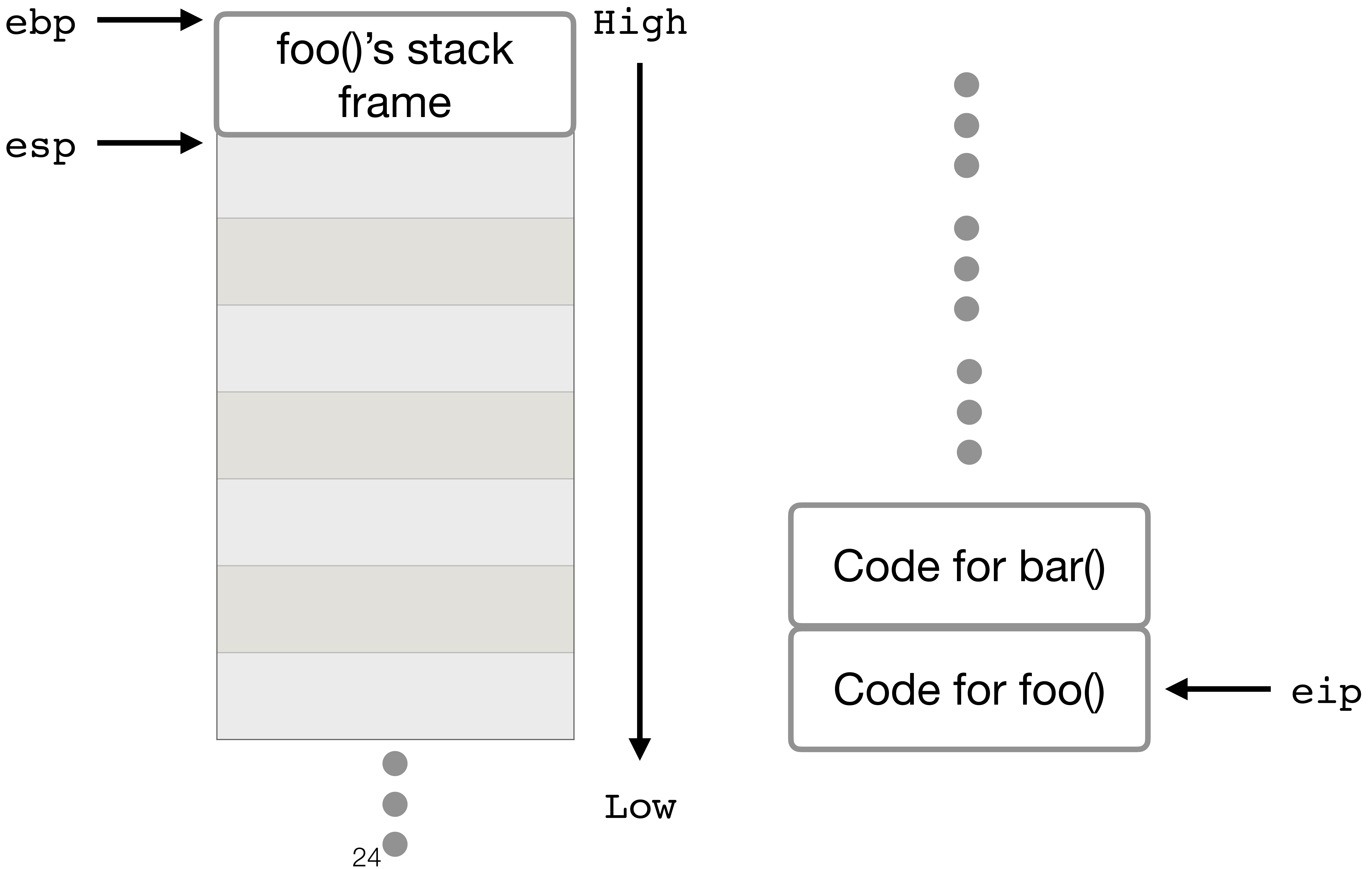
Where buffer overflow happens

Stack Frame of a Function



Stack Frames: Calling a Function

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1,
int arg2) {
    int  loc1;
    long loc2;
    ...
}
```



Note:

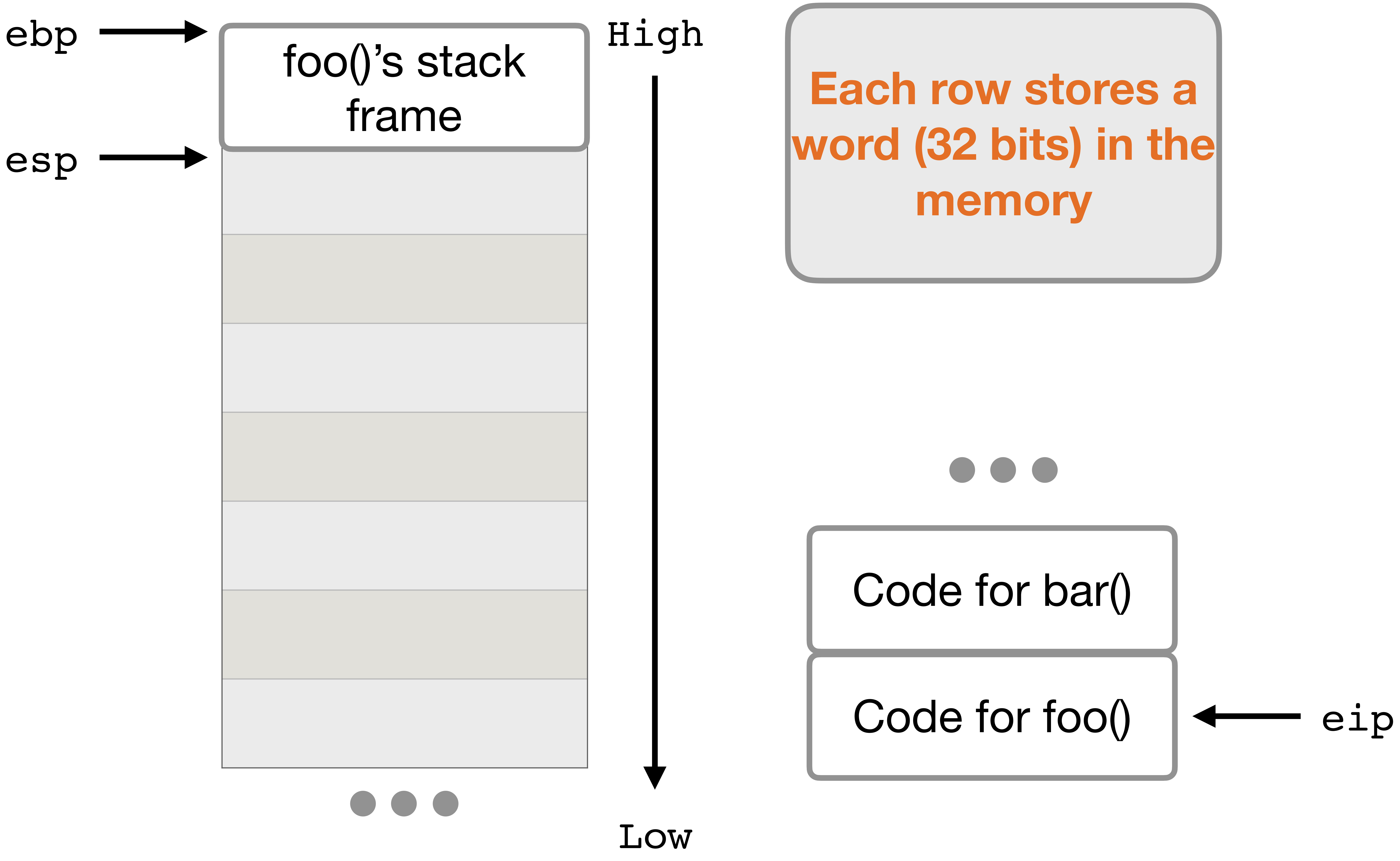
- Arguments
- Return address
- Saved Frame Pointer
- Local Variables

Stack Frames: Calling a Function

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1,
int arg2) {
    int  loc1;
    long loc2;
    ...
}
```

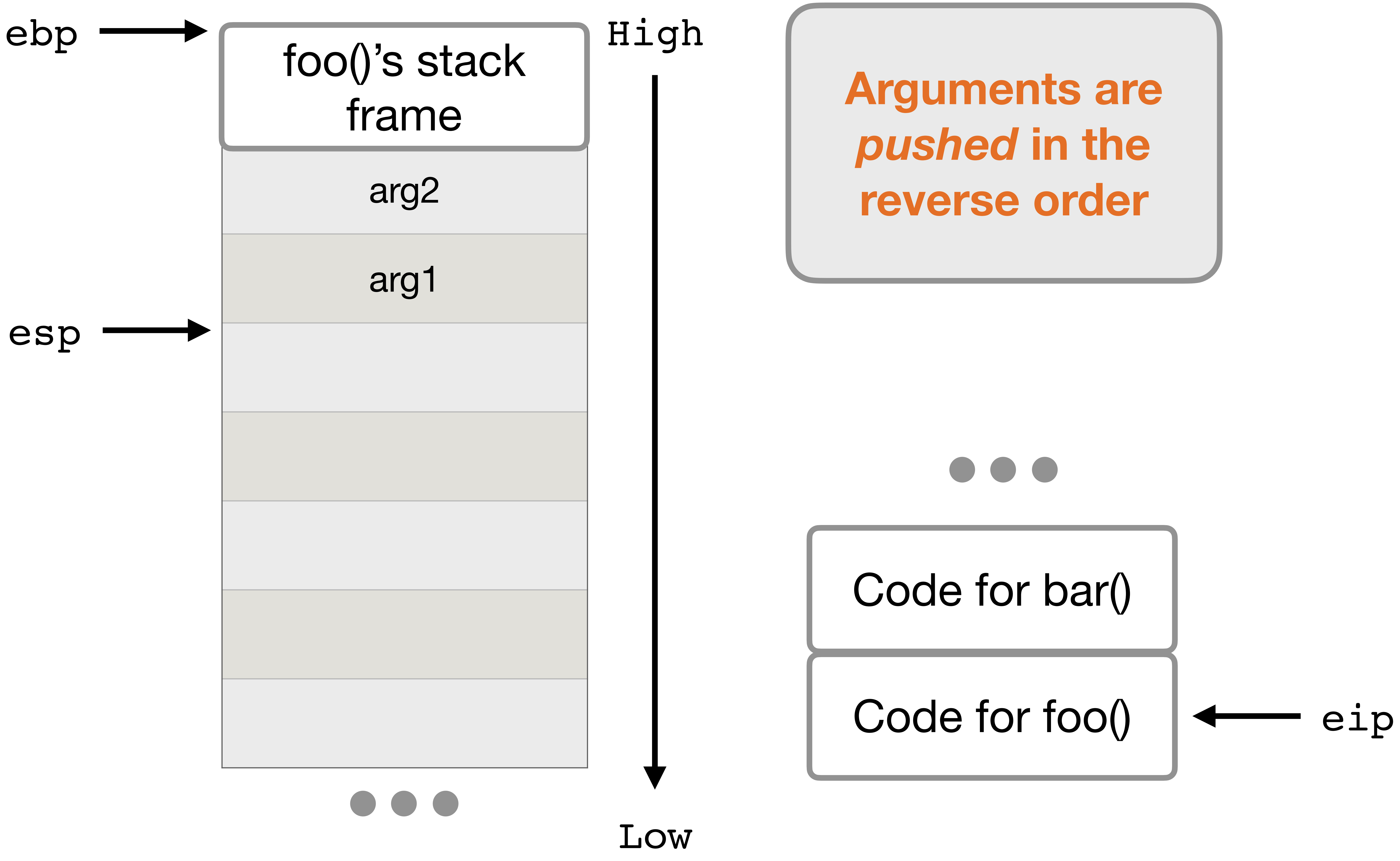
Note:

- Arguments
- Return address
- Saved Frame Pointer
- Local Variables



Stack Frames: Calling a Function

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1,
int arg2) {
    int  loc1;
    long loc2;
    ...
}
```

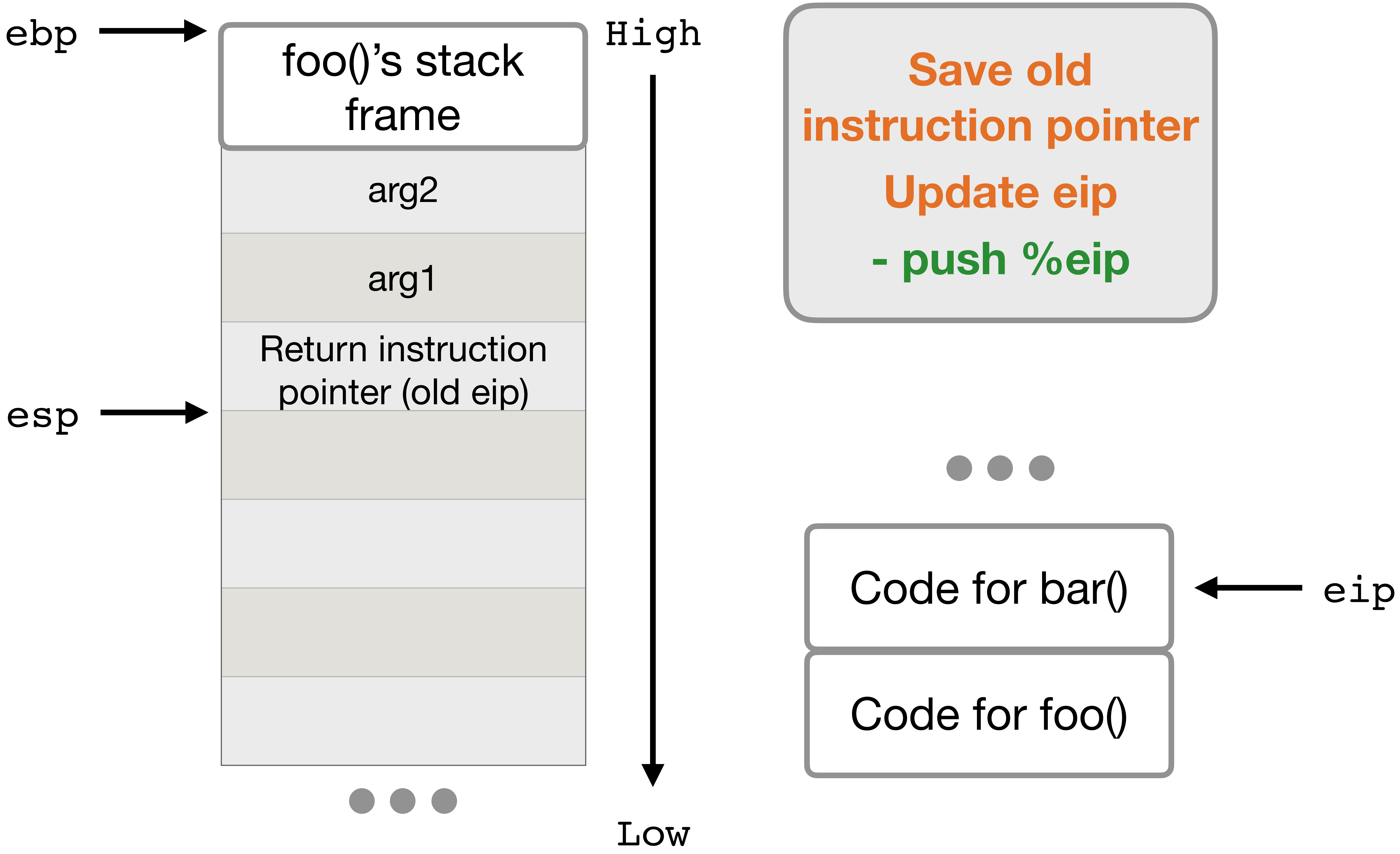


Note:

- Arguments
- Return address
- Saved Frame Pointer
- Local Variables

Stack Frames: Calling a Function

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1,
int arg2) {
    int  loc1;
    long loc2;
    ...
}
```

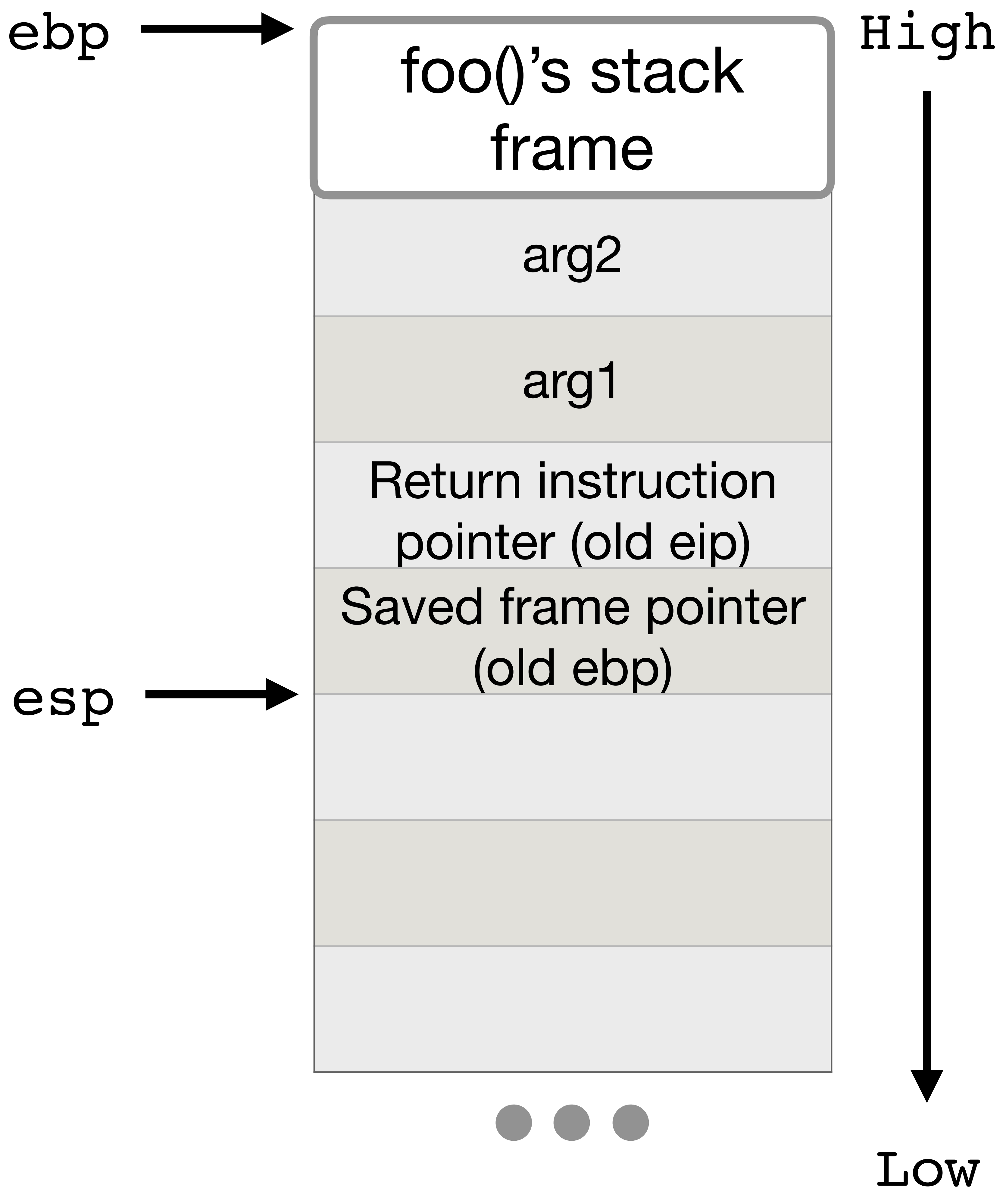


Note:

- Arguments
- Return address
- Saved Frame Pointer
- Local Variables

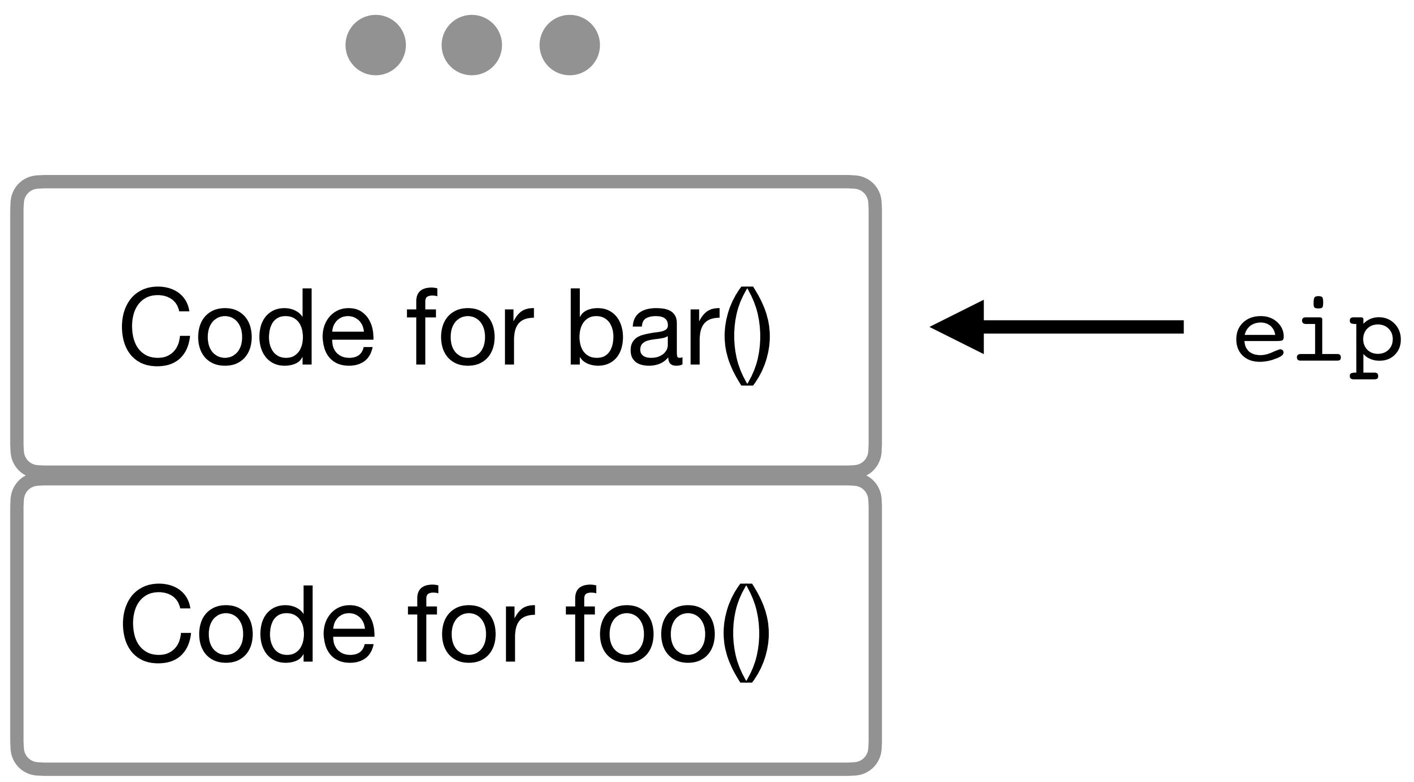
Stack Frames: Calling a Function

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1,
int arg2) {
    int  loc1;
    long loc2;
    ...
}
```



Save old ebp

- push %ebp

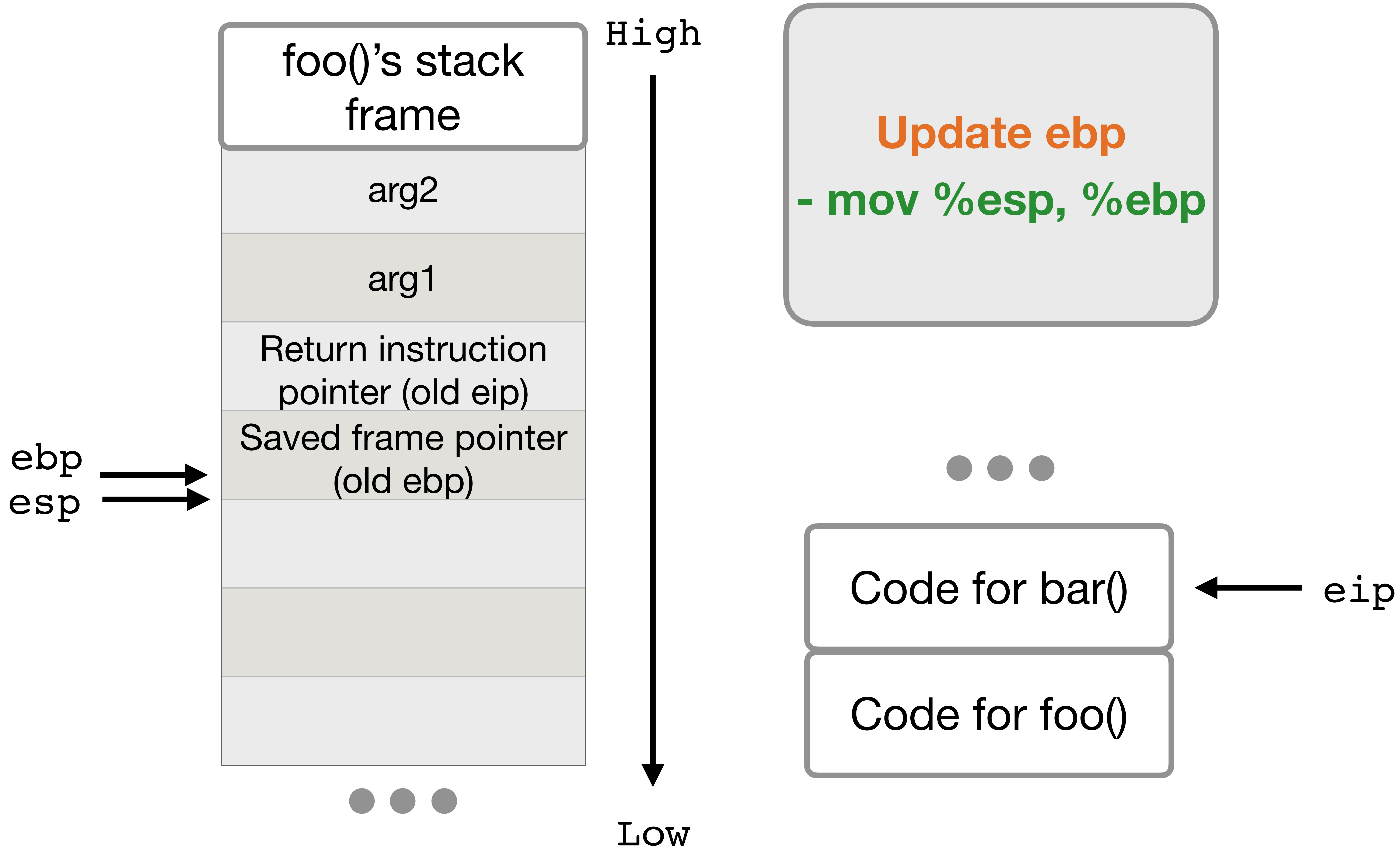


Note:

- Arguments
- Return address
- Saved Frame Pointer
- Local Variables

Stack Frames: Calling a Function

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1,
int arg2) {
    int  loc1;
    long loc2;
    ...
}
```



Note:

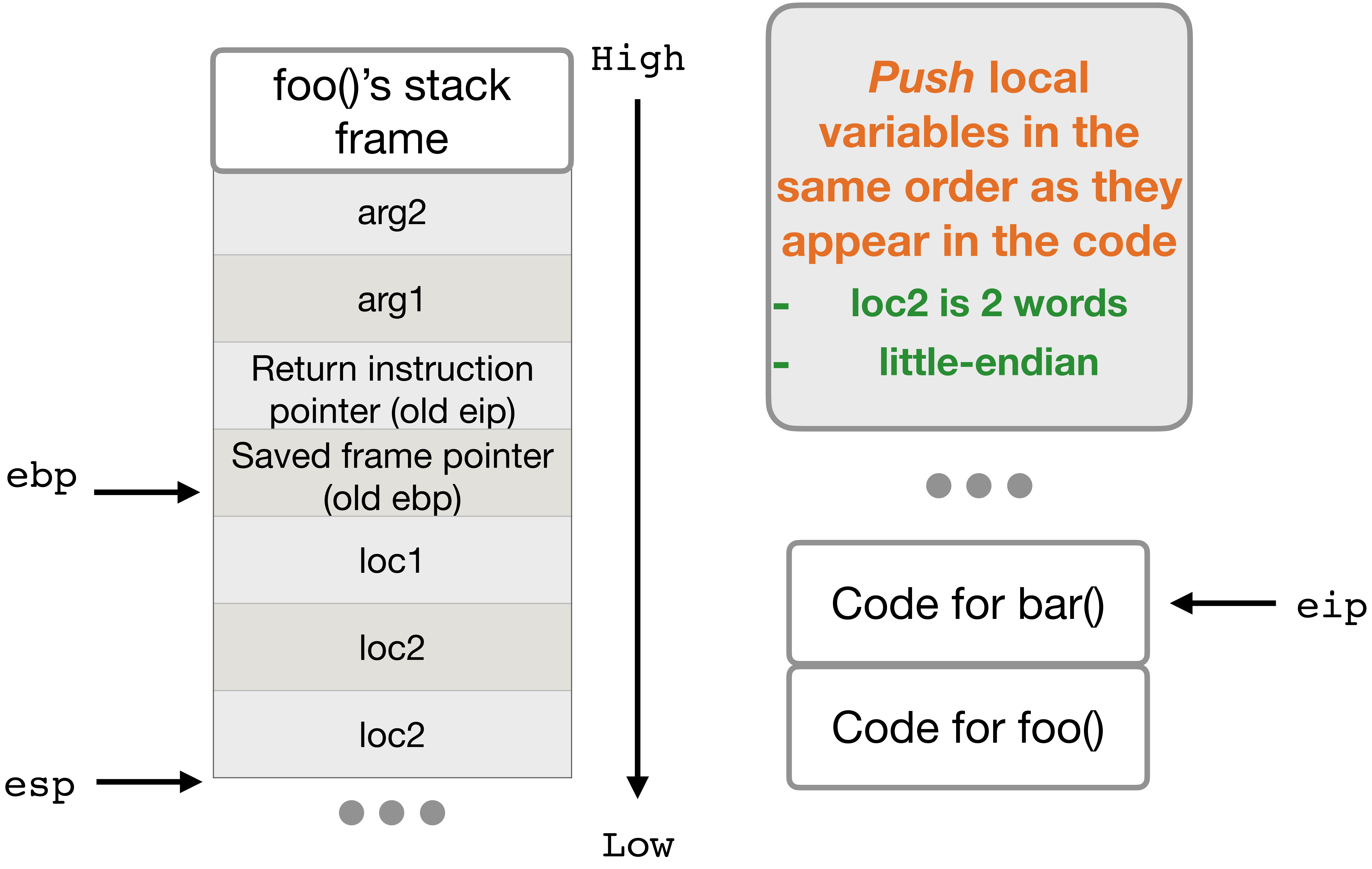
- Arguments
- Return address
- Saved Frame Pointer
- Local Variables

Stack Frames: Calling a Function

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1,
int arg2) {
    int  loc1;
    long loc2;
    ...
}
```

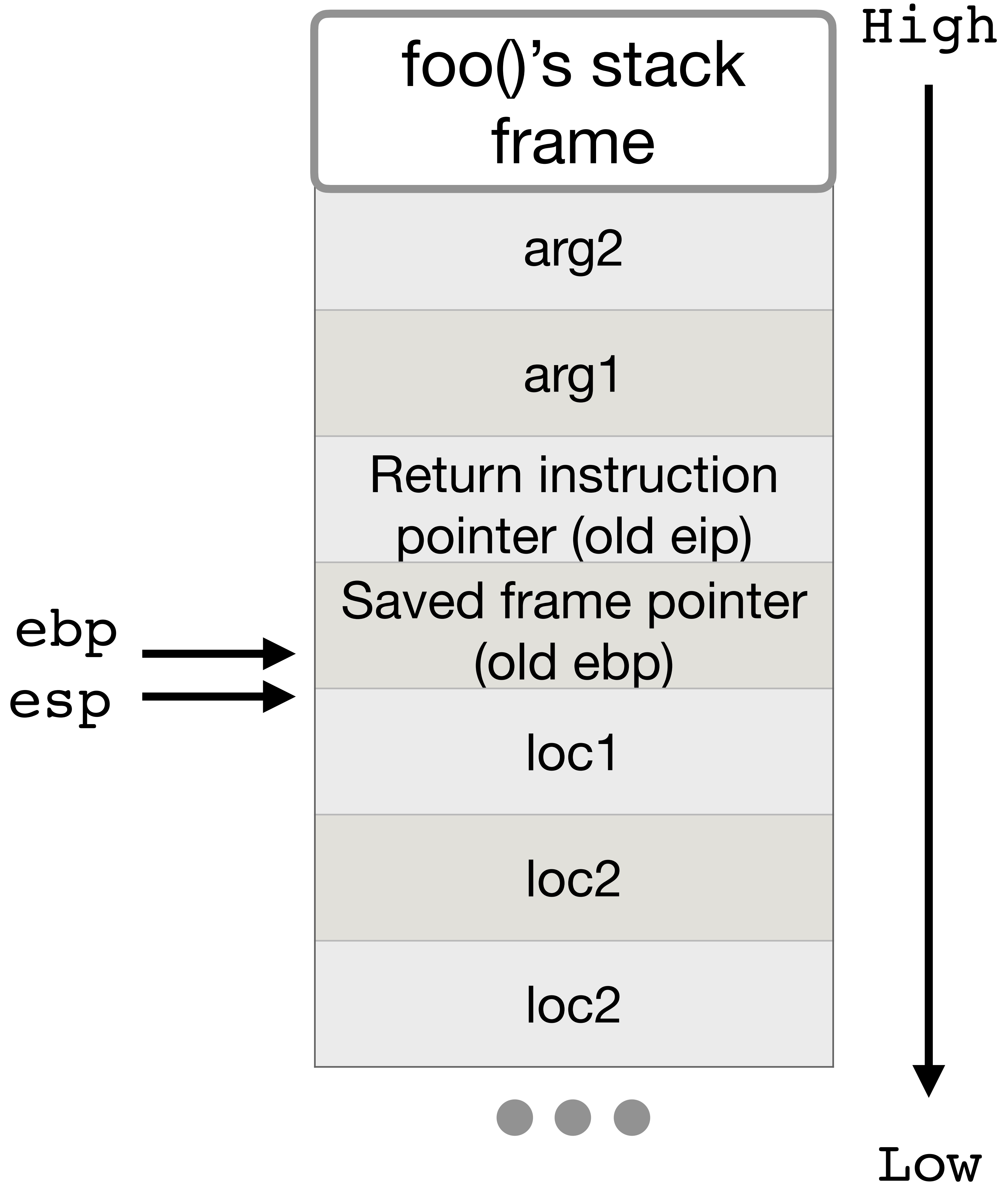
Note:

- Arguments
- Return address
- Saved Frame Pointer
- Local Variables



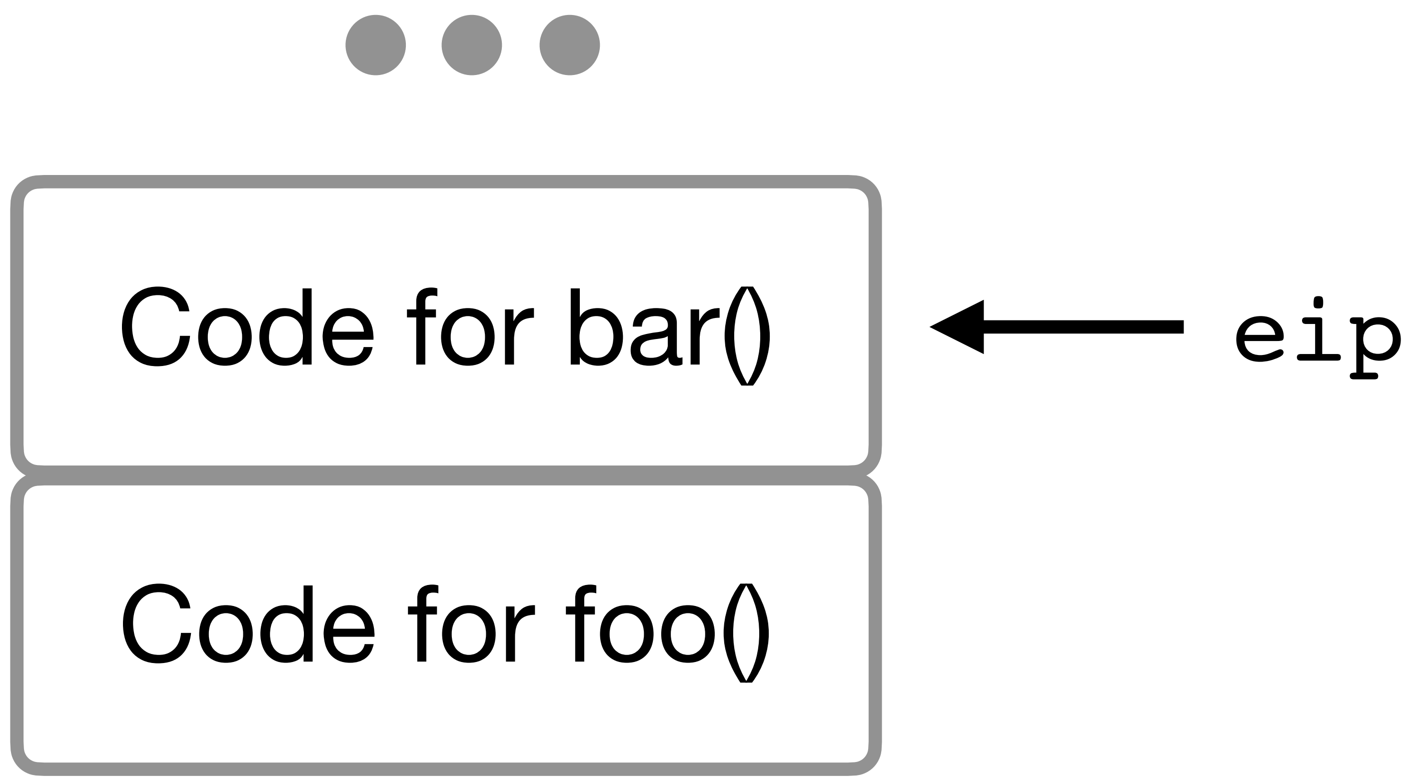
Stack Frames: Return from a Function

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1,
int arg2) {
    int  loc1;
    long loc2;
    ...
}
```



Restore the stack pointer

- mov %ebp, %esp
- This does not wipe off loc1 and loc2 from the memory
- They are undefined

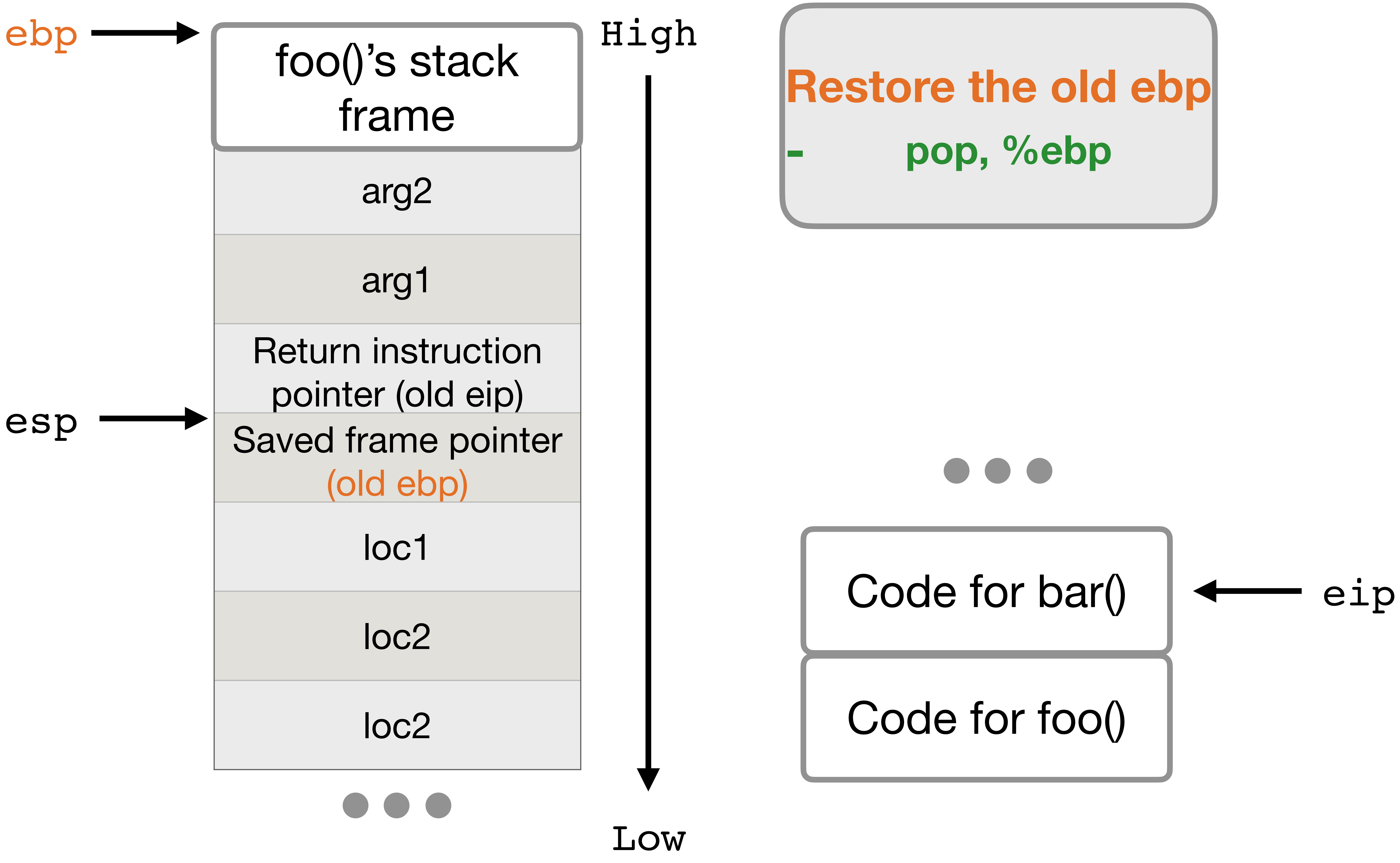


Note:

- Arguments
- Return address
- Saved Frame Pointer
- Local Variables

Stack Frames: Return from a Function

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1,
int arg2) {
    int  loc1;
    long loc2;
    ...
}
```

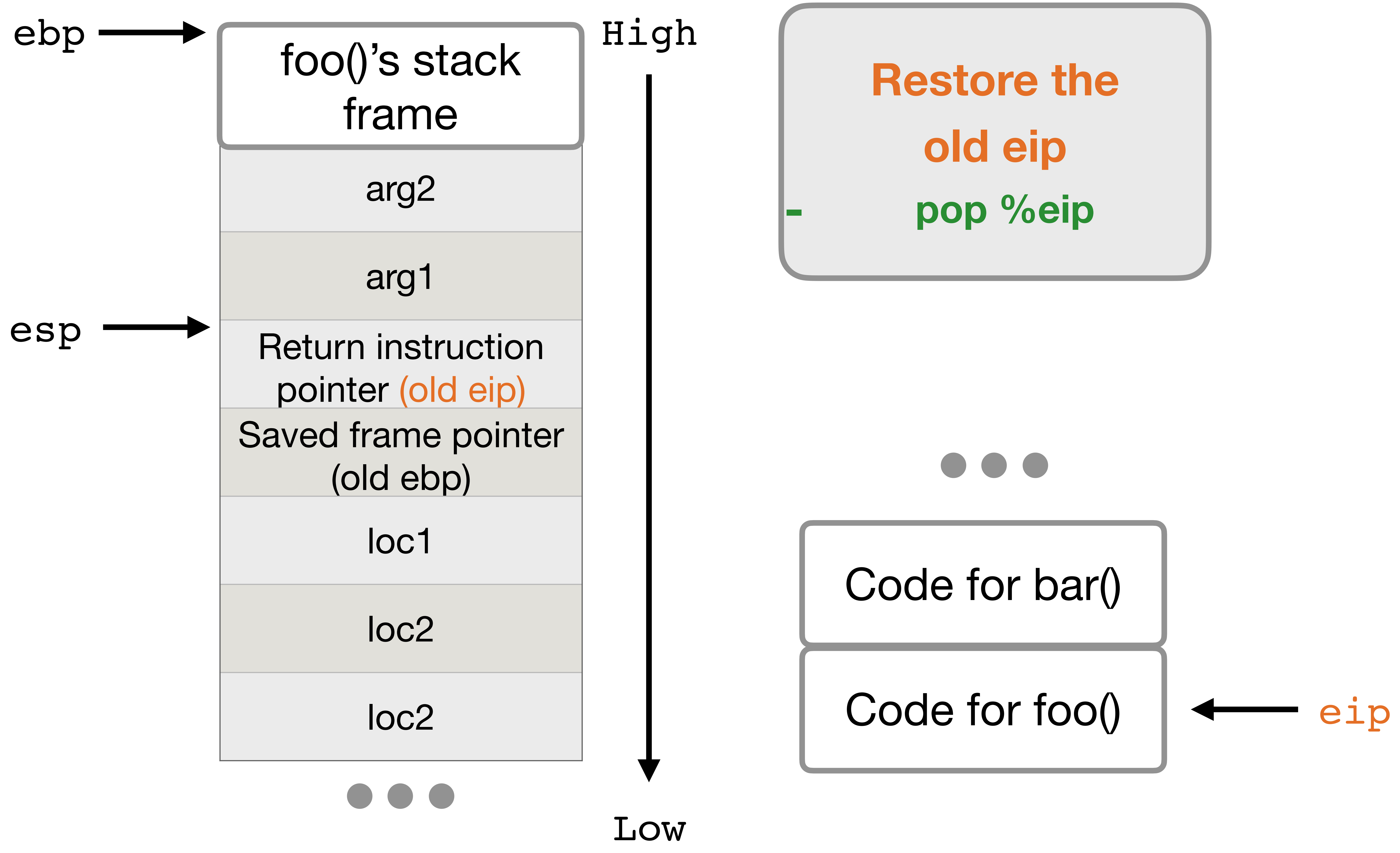


Note:

- Arguments
- Return address
- Saved Frame Pointer
- Local Variables

Stack Frames: Return from a Function

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1,
int arg2) {
    int  loc1;
    long loc2;
    ...
}
```

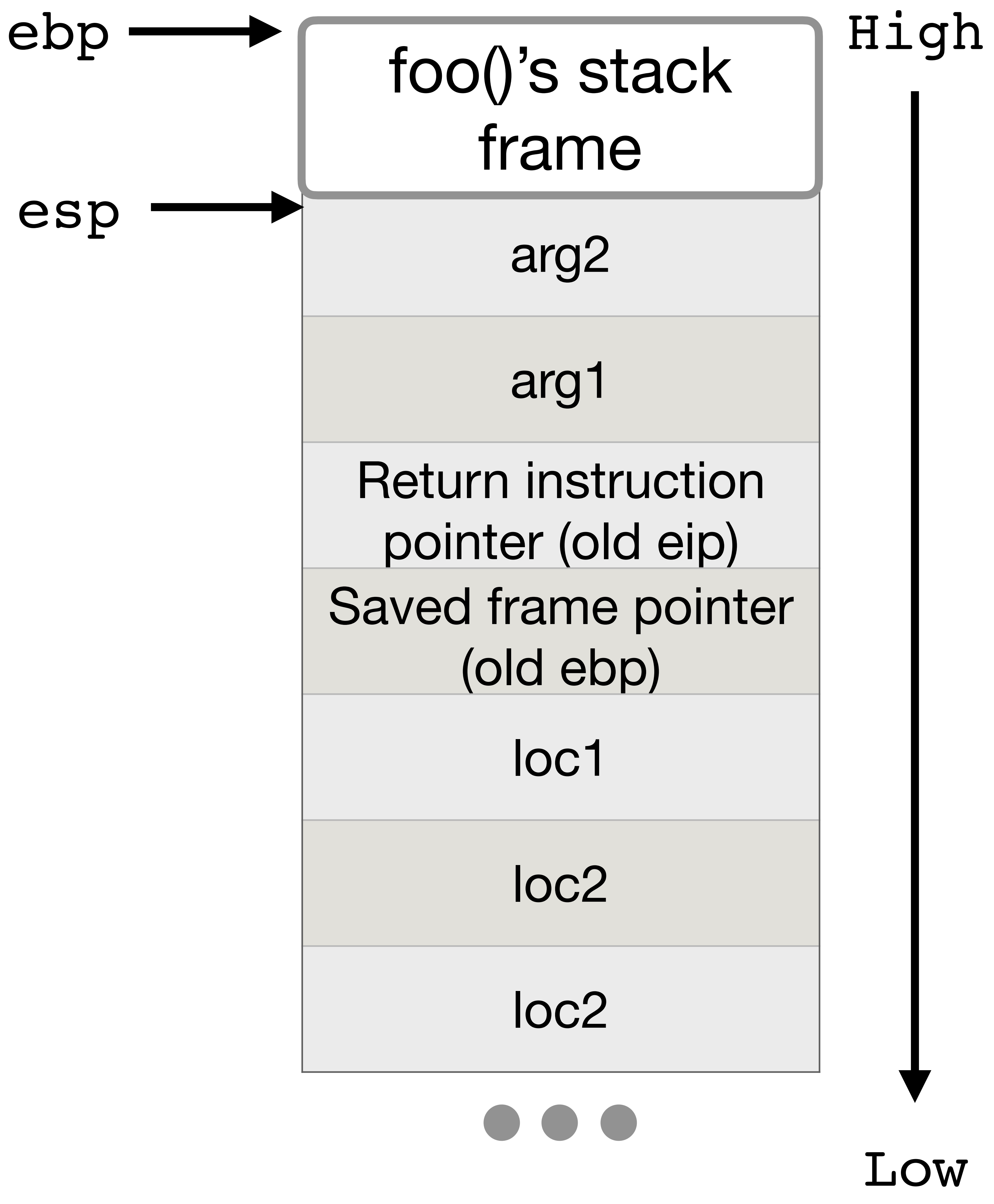


Note:

- Arguments
- Return address
- Saved Frame Pointer
- Local Variables

Stack Frames: Return from a Function

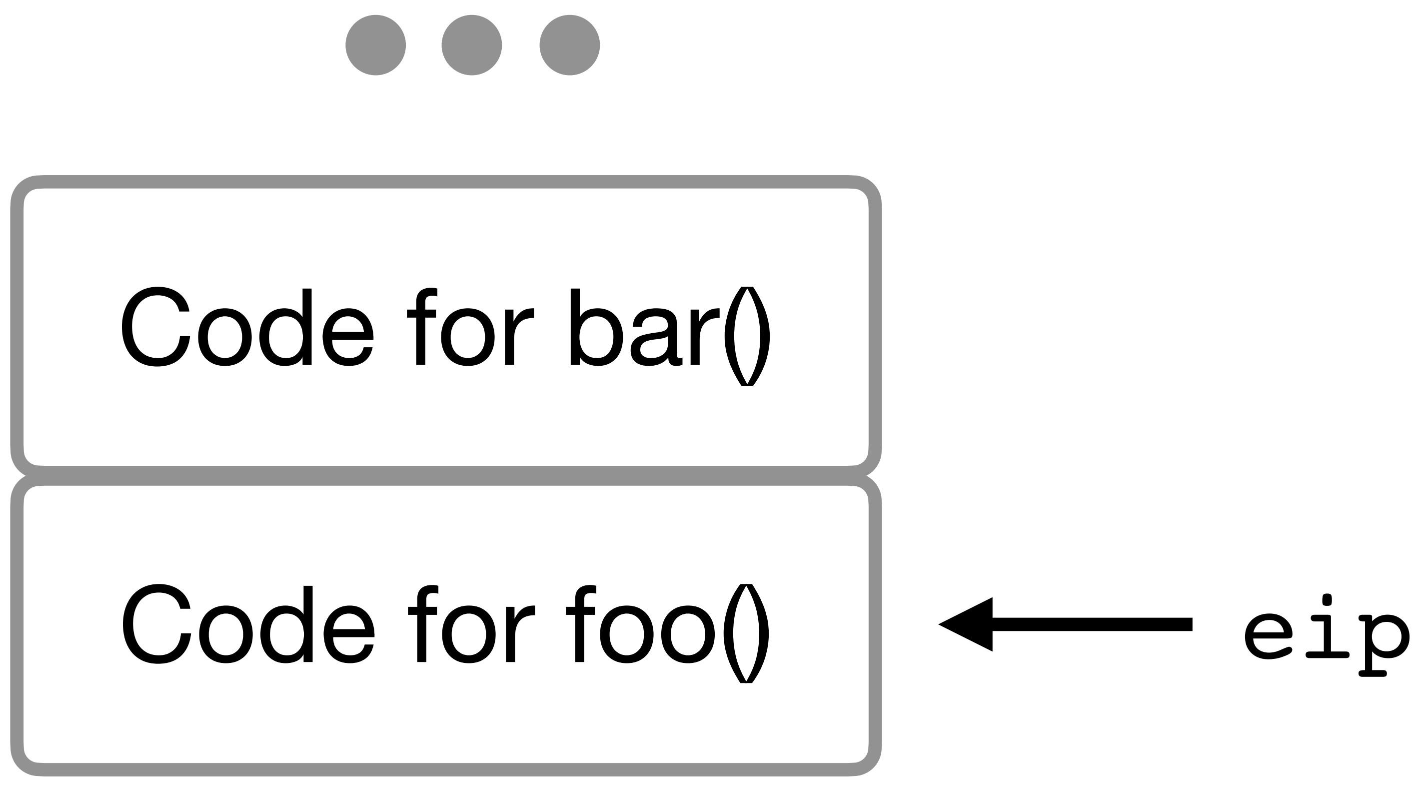
```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1, int arg2) {
    int  loc1;
    long loc2;
    ...
}
```



Remove arguments from the stack

- add $8, %esp

- anything below esp is undefined



Note:

- Arguments
- Return address
- Saved Frame Pointer
- Local Variables

Return from a Function

In C

```
return;
```

In compiled assembly

```
leave:    mov %ebp %esp
          pop %ebp
ret:      pop %eip
```

- Leave: leave the stack frame of the callee
  - restore stack pointer (mov %ebp %esp)
  - restore the base pointer (pop %ebp)
- Ret: restore the instruction pointer (pop %eip)

Exercise

```c
#include <stdio.h>

int foo(int a, int b) {
    int c = a + b;
    int d = a - b;
    return c + d;
}

int main(void) {
    foo(1, 2);
    return 0;
}
```

Compile the program, use gdb to step through the x86 assembly instructions, understand the instructions, draw the stack

Summary

- C memory layout: code, data, heap, stack sections
- x86 registers
  - **ebp**: base pointer of the stack frame
  - **esp**: stack point for the stack frame (growth direction)
  - **eip**: next instruction to be executed
- x86 calling convention
  - When calling a function, the old **eip** is saved on the stack
  - When calling a function, the old **ebp** is saved on the stack
  - When the function returns, the old **ebp** and **eip** are restored from the stack

Buffer Overflow

```c
char name [2];
name[3] = 'a';
```

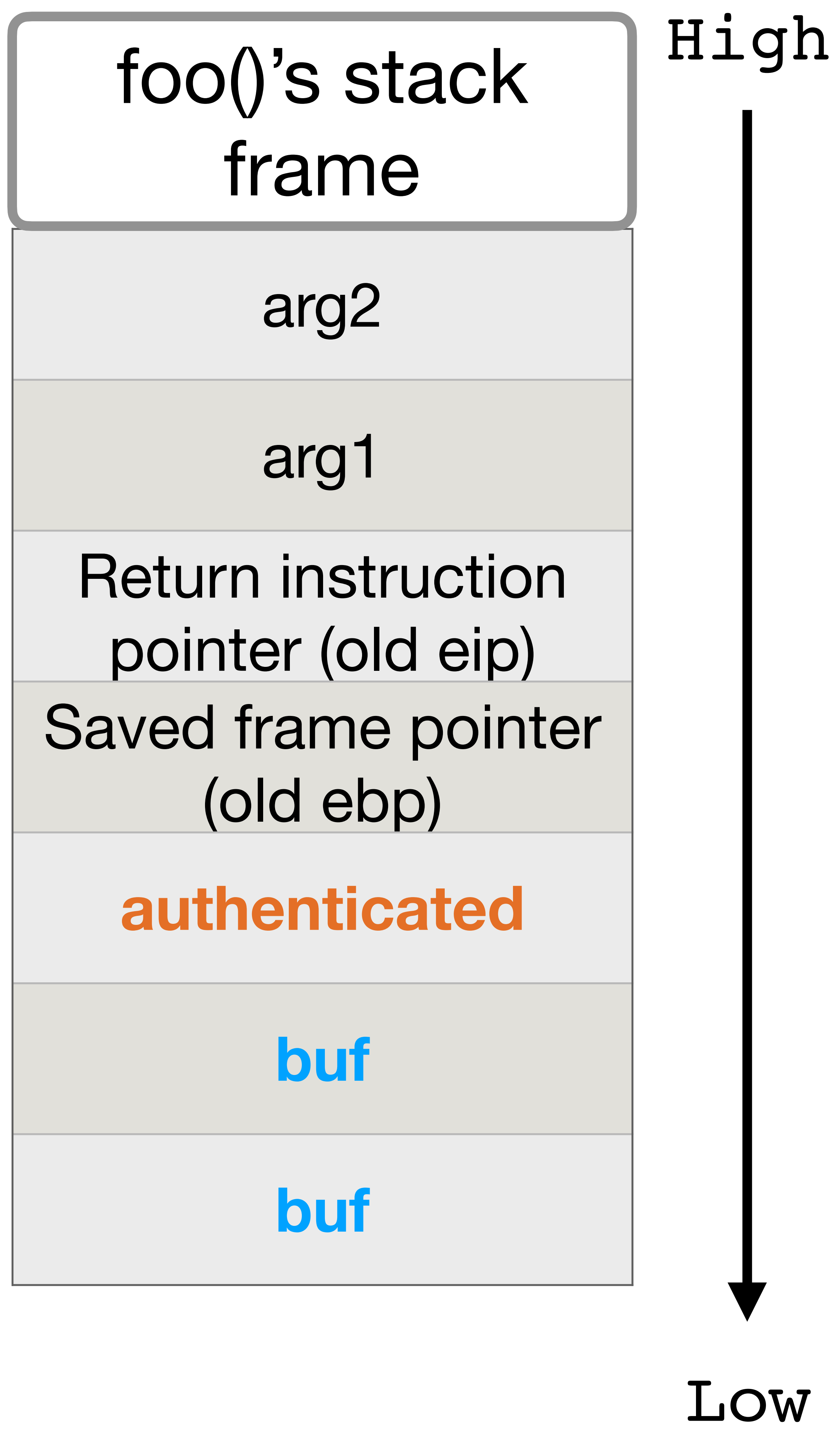
Buffer Overflow

```c
char name [2];
name[3] = 'a';
```

C does not check bounds

How to change authenticated to 1?

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1, int arg2) {
    int  authenticated = 0;
    char buf[8];
    ...
}
```

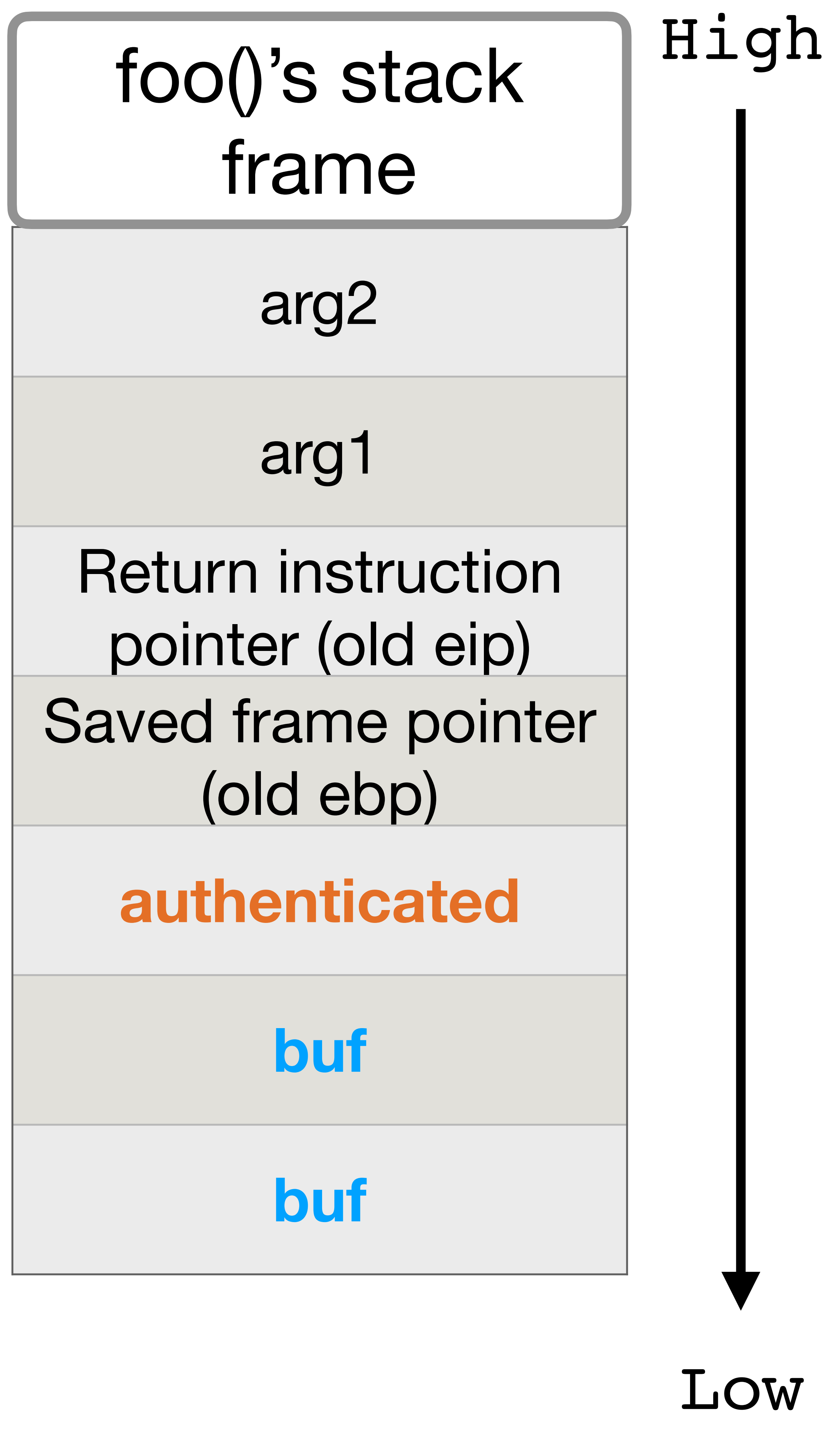


Buffer Overflow

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1, int arg2) {
    int  authenticated = 0;
    char buf[8];
    ...
}
```

Set buf[8] to non-zero

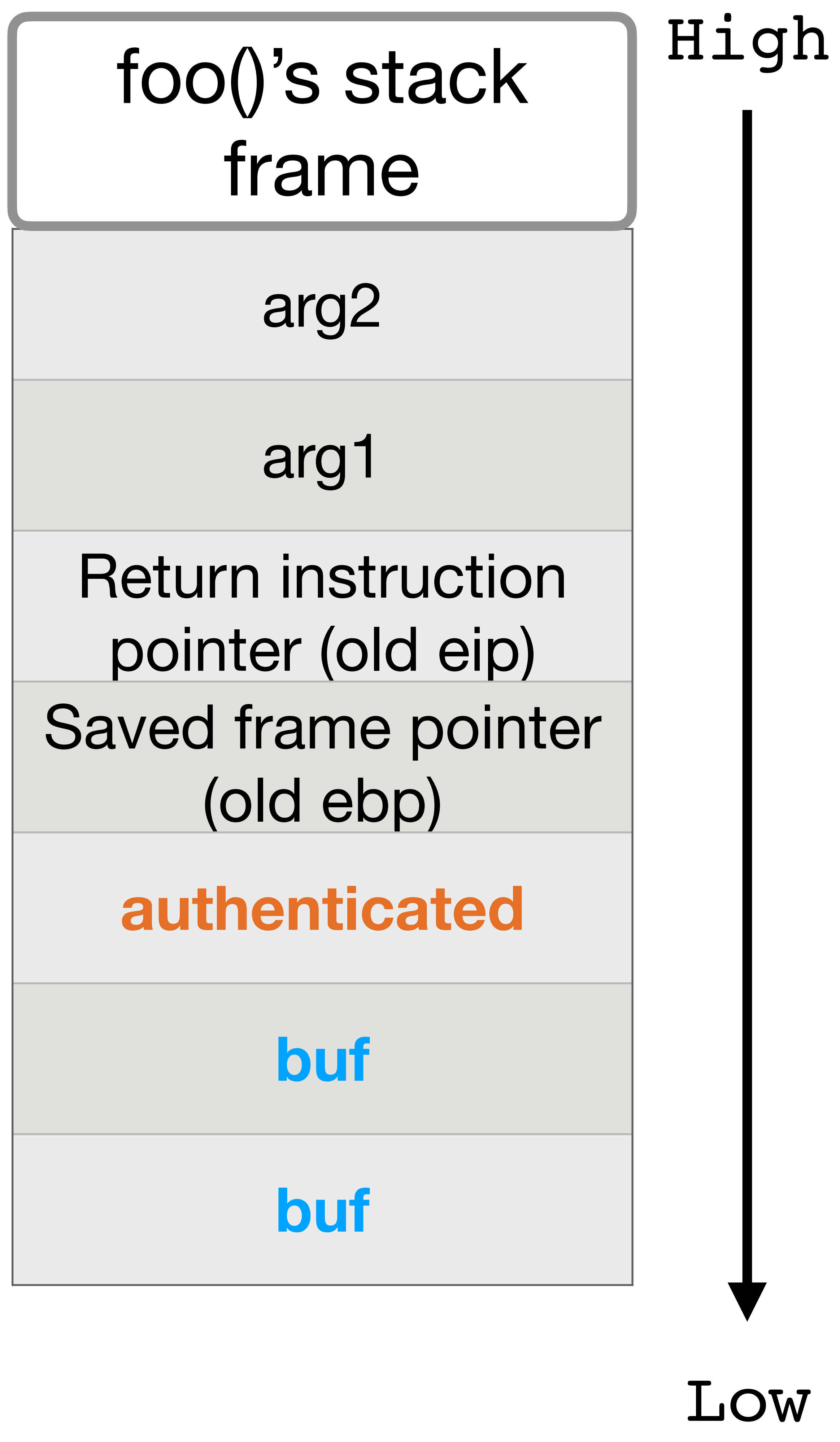
Hint: little-endian



Buffer Overflow

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1, int arg2) {
    int  authenticated = 0;
    char buf[8];
    ...
}
```

Exercise: write out the memory layout for **buf** and **authenticated** if we set **buf** as **“abcdefgh!”**



Stack Smashing

```
void foo() {
    …
    bar(arg1, arg2);
}

void bar(char *arg1, int arg2) {
    int  authenticated = 0;
    char buf[8];
    ...
}
```