CMSC414 Computer and Network Security

Introduction, Memory Layout and Buffer Overflows

Yizheng Chen | University of Maryland
surrealyz.github.io

Feb 3, 2026

Agenda

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

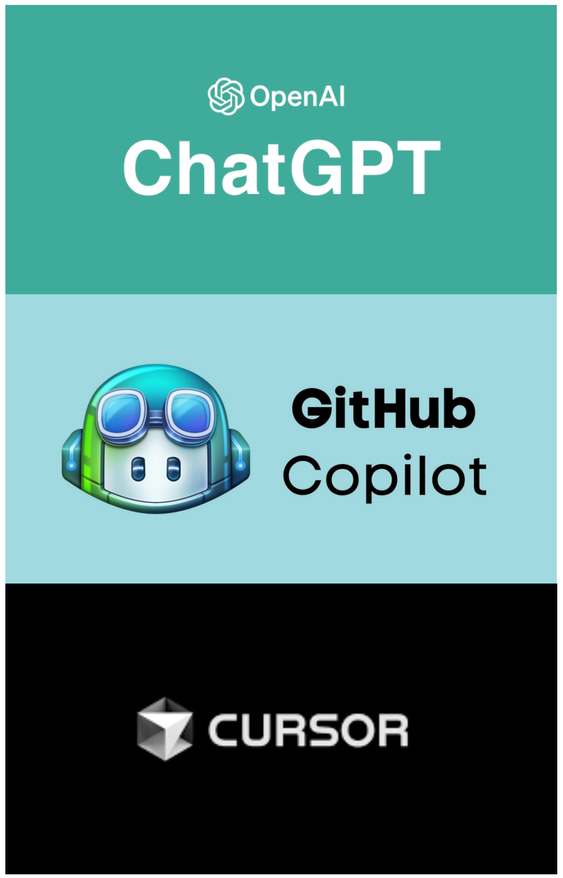
About Me

- Assistant Professor in Computer Science
- I work on AI and Security

Securing AI Coding Assistants

LLM Agents for Cybersecurity

Large Language Models Trained on Code



- Summarize Code
- Generate Code from Description
- Translate Code between Programming Languages
- Autocomplete a partial program
- …

ML-Enhanced Code Completion Improves Developer Productivity

July 26, 2022 ·

Gartner Says 75% of Enterprise Software Engineers Will Use AI Code Assistants by 2028

STAMFORD, Conn., April 11, 2024

Insecurity of Code Generation

Given 89 different prompts for GitHub Copilot to complete the program, 40% of generated programs are vulnerable

"Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions" Pearce et al., IEEE S&P 2022

Code Completion Benchmark Leaderboard

https://secrepobench.github.io/

#	Setup	secure-pass@1 (%)
1	Codex + GPT-5.1-Codex-Max (review)	60.1
2	Codex + GPT-5.1-Codex-Max	56.3
3	OpenHands + OpenAI o3	53.5
4	Codex + GPT-5	52.5
5	OpenHands + GPT-5	51.3
6	Aider + GPT-5	50.6
7	Claude Code + Claude Sonnet 4.5	50.3
8	Aider + Claude Sonnet 4.5	45.6
9	OpenHands + Claude Sonnet 4.5	43.7

Software Security Section

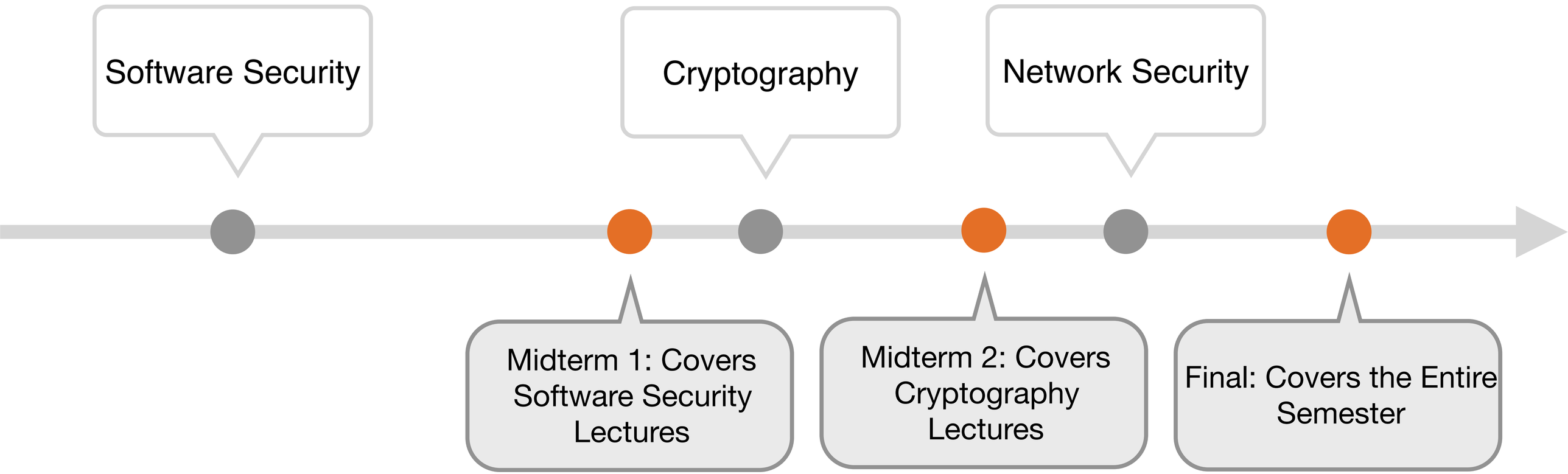
Vulnerable source code?

Web vulnerabilities?

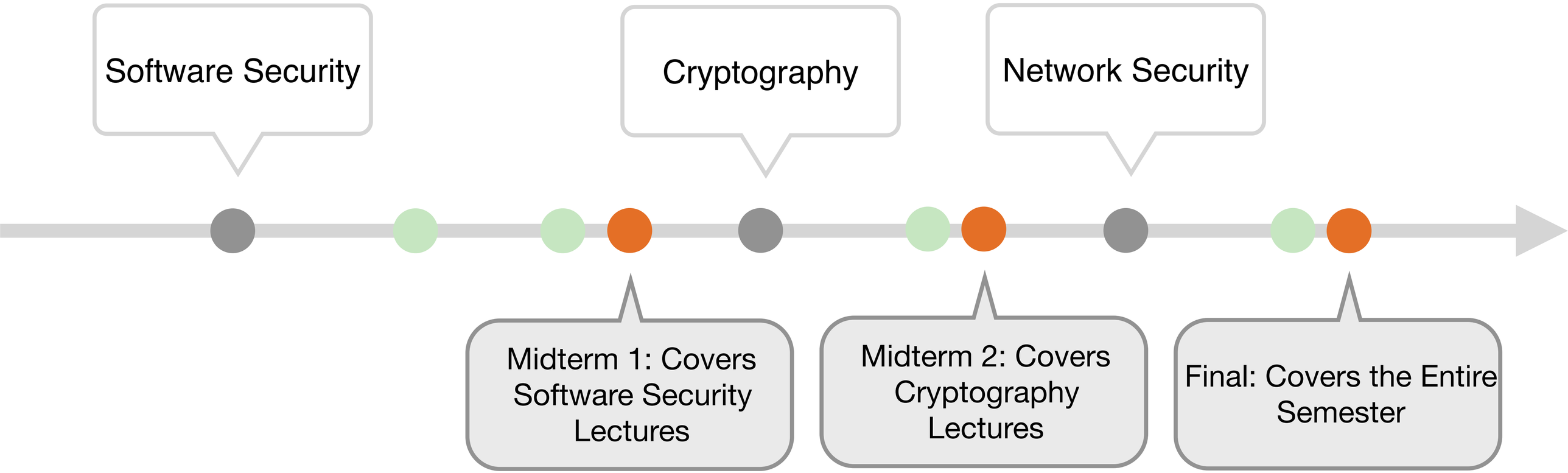
How to mitigate them?

...

Course Topics and Timeline



Course Topics and Timeline



Programming Projects

Exam Dates: Major Scheduled Grading Events

Midterm 1

- March 10
- In Classroom

Midterm 2

- April 9
- In Classroom

Final Exam

- May 14
- Location TBD

Let the instructor know ASAP if date has conflicts due to excused absences (syllabus)

Late Policy

- Projects may be submitted up to 24 hours late for a 10% penalty.
- We will grade the last submitted version.
- We do not grant project extensions.

Please read the syllabus

Grades

- 50% Projects
  - 4 projects (10% projects 1-3, 20% project 4)
- 25% Midterms
  - 12.5% for each midterm exam
- 25% Final

Ethics and Legality

- You will be learning about *(and implementing and launching)* attacks, many of which are in active use today.

- This is not an invitation to perform these attacks without the express written consent of all parties involved. **To do otherwise would risk a violating University of Maryland policies and Maryland and U.S. laws.**

Office Hours

- Instructor office hour, **Thursday** 1:00pm - 2:00pm
- TA office hours, online, URLs on ELMS
  - Daniel Kiely, dmkiely@umd.edu, **Monday** 12pm - 2pm
  - Mia Yi, yi12@umd.edu, **Tuesday** 11am - 2pm
  - Steven Shen, stevencs@umd.edu, **Wednesday** 2pm - 4pm
  - Sarah Bransky, sbransky@umd.edu, **Friday** 11am - 1pm

Please Read the Syllabus

- Academic Integrity
- ADS Accommodation
- Use of external resources
- Grading & Regrading policies
- More …

Announcements

- Project 1 Released
- Please log onto gitlab https://gitlab.cs.umd.edu/ and verify that your username matches your directory ID
  - If not, change username to match directory ID

Announcements

- Sarah (TA) Tutorial Session on gdb: Feb 6, Friday, 1pm

Threat Model

- Attacker's goal:
  - Take over the entire target machine (e.g., web server)
- Attacker's knowledge:
  - CPU and OS on the target machine
  - Our examples: x86 running Linux

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

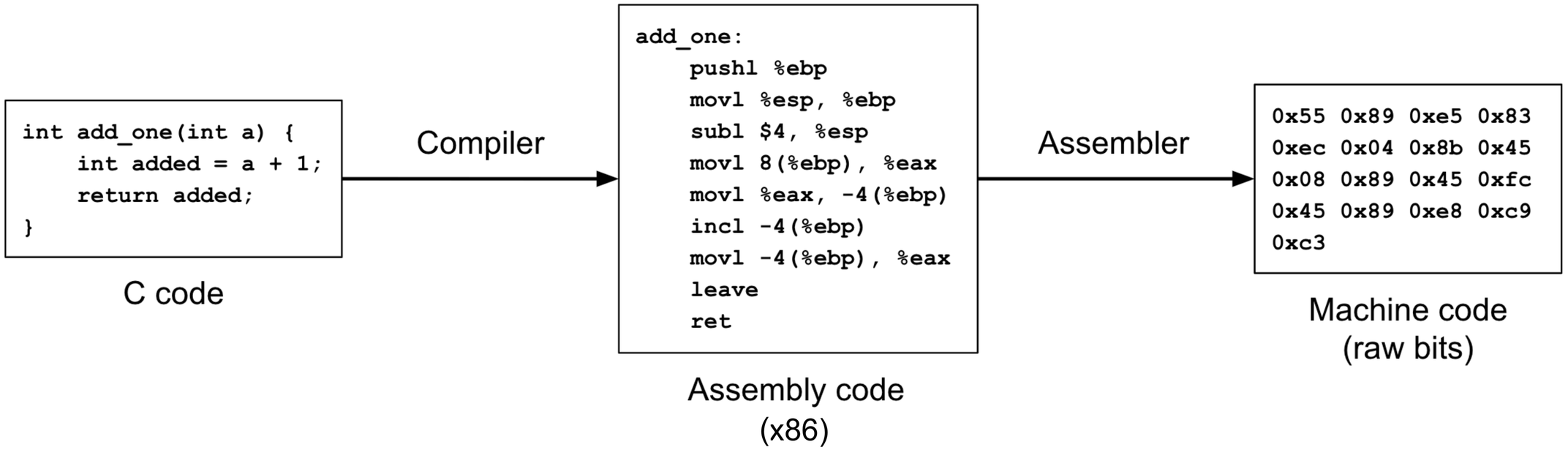
Hexadecimal

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

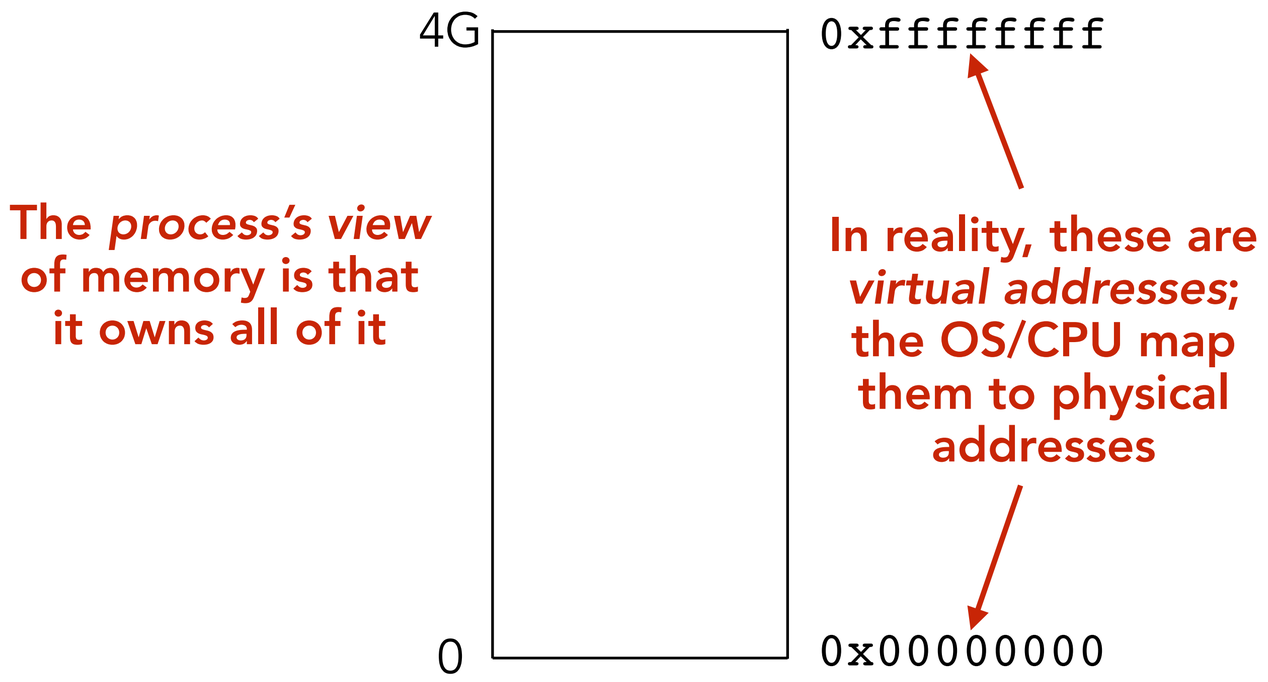


- Compiler: Converts C code into assembly code (e.g., x86)
- Assembler: Converts assembly code into machine code (raw bits)
- Linker: Deals with dependencies and libraries
- Loader: Sets up memory space and runs the machine code

Memory Layout

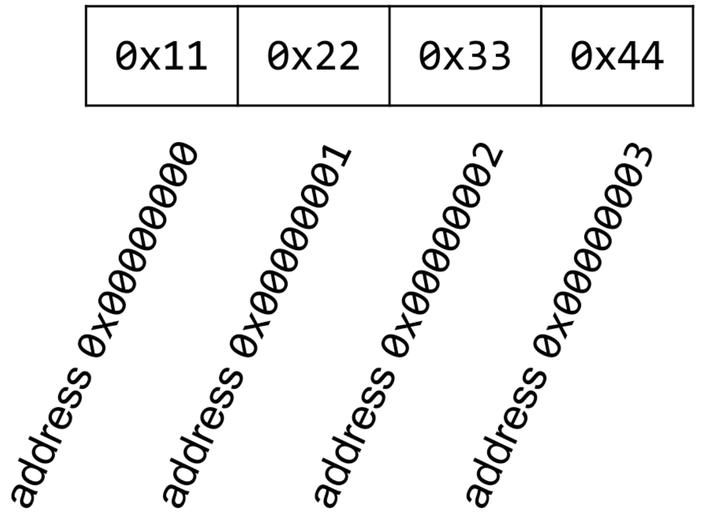
- At runtime, the loader tells the OS to give your program a big blob of memory
- On a 32-bit system, the memory has 32-bit addresses
  - On a 64-bit system, memory has 64-bit addresses
  - We use 32-bit systems in this class
- Each address refers to one byte, so 2^{32} bytes of memory

Memory Layout

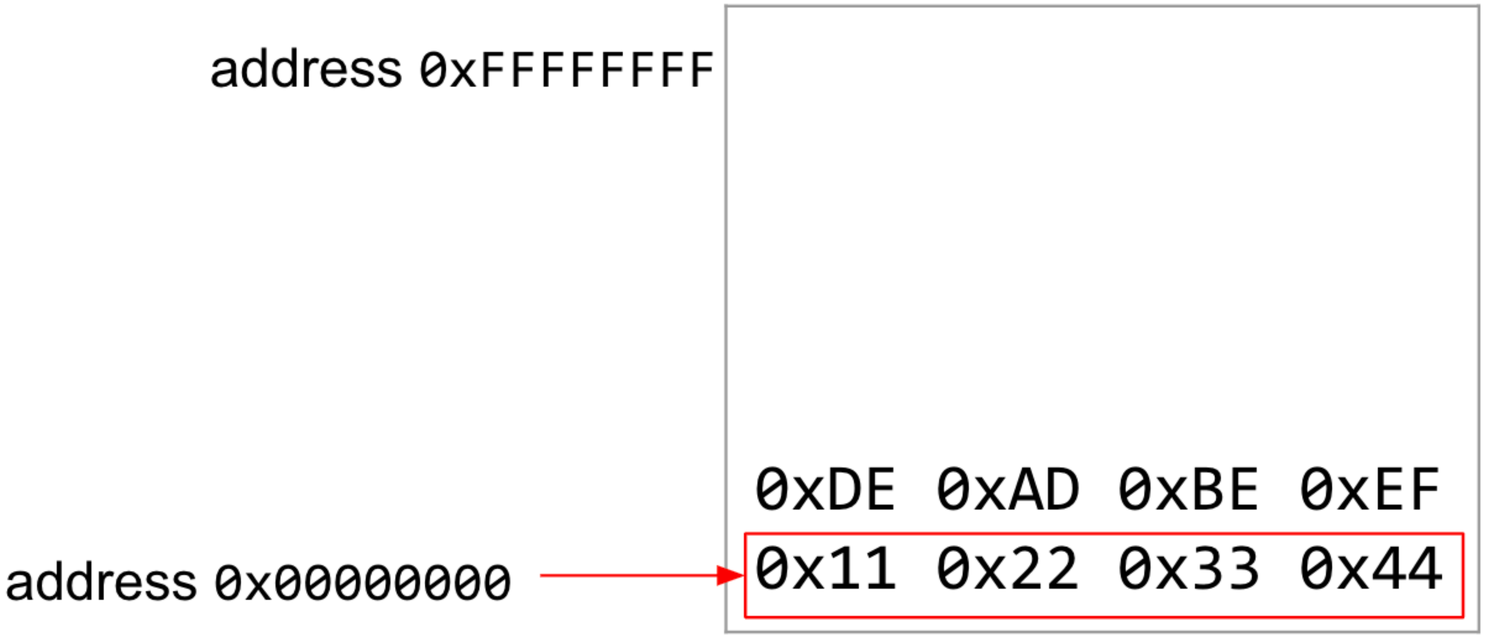


Little-endian words

- One word: 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

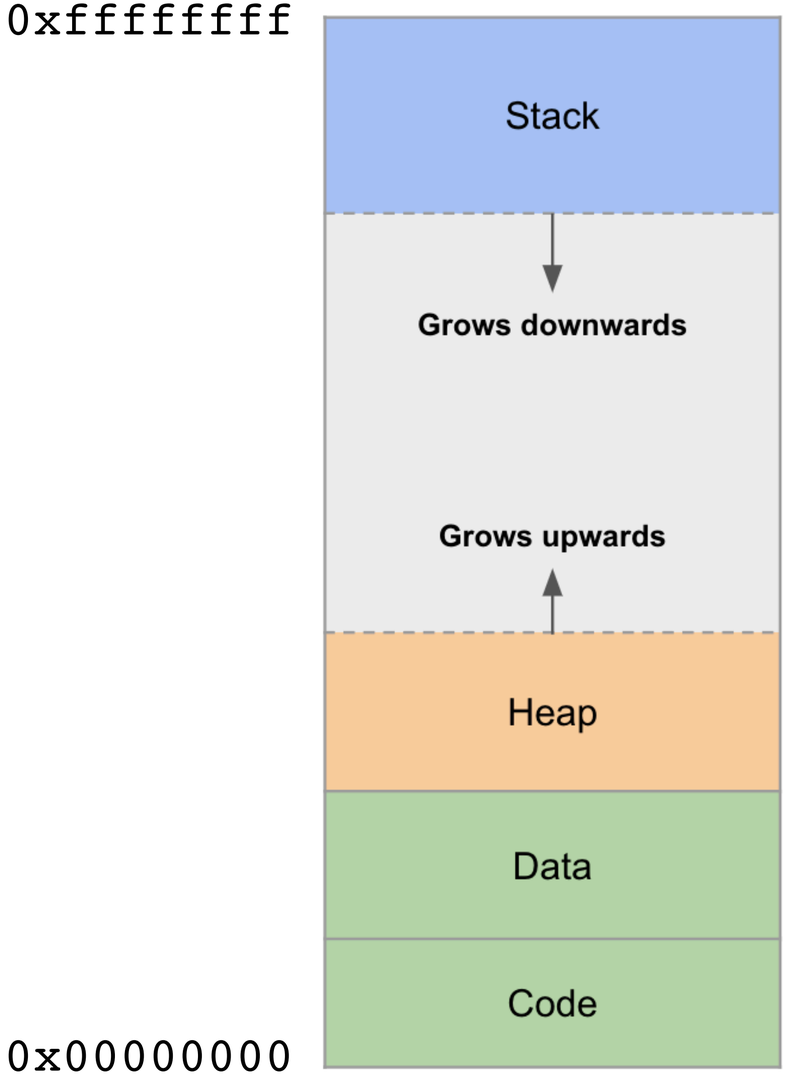


Combine Bytes on a Row to Form a Word



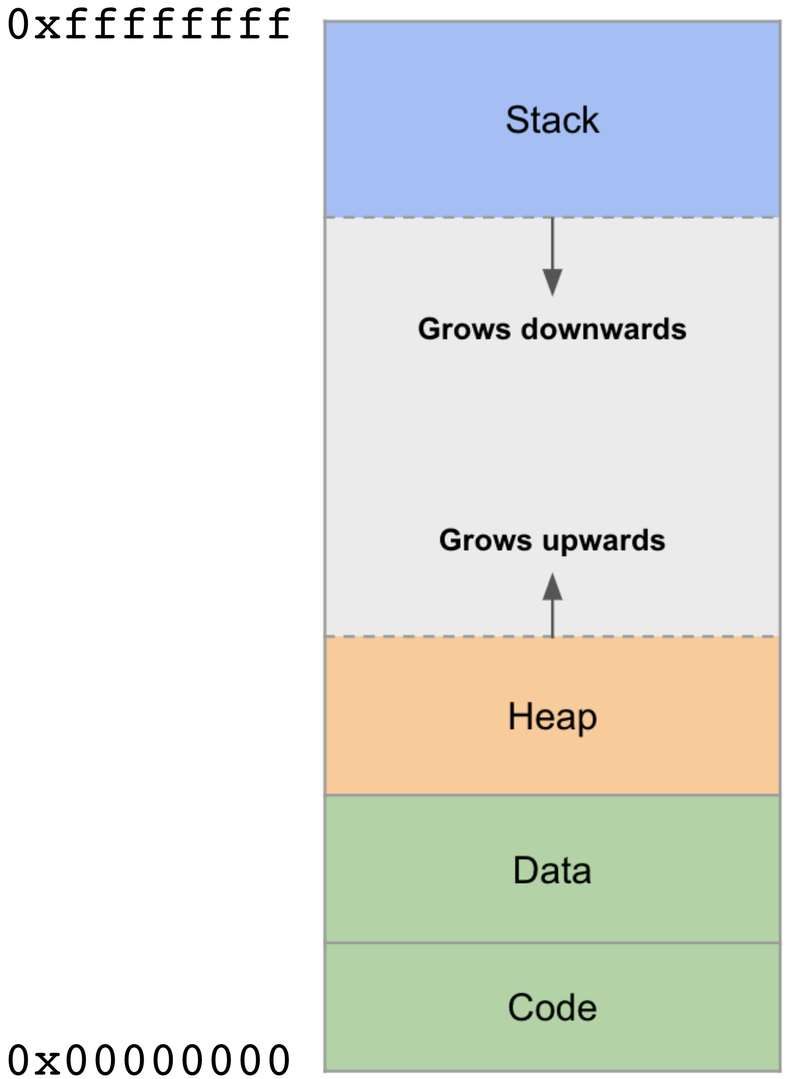
- What is the byte at address 0x00000000? 0x11
- What is the word at address 0x00000000? 0x44332211
- What is the byte / word at address 0x00000004?

x86 Memory Layout



The program code (also called “text”)

x86 Memory Layout

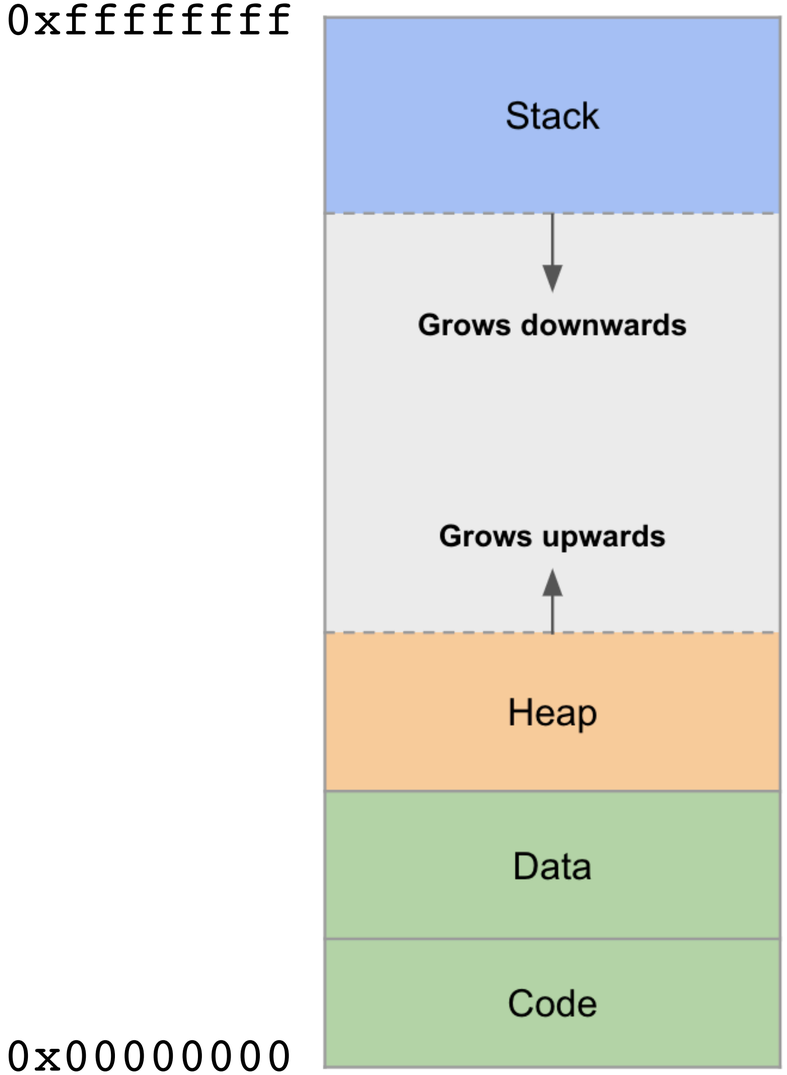


Uninitialized static variables

Initialized static variables

The program code (also called “text”)

x86 Memory Layout



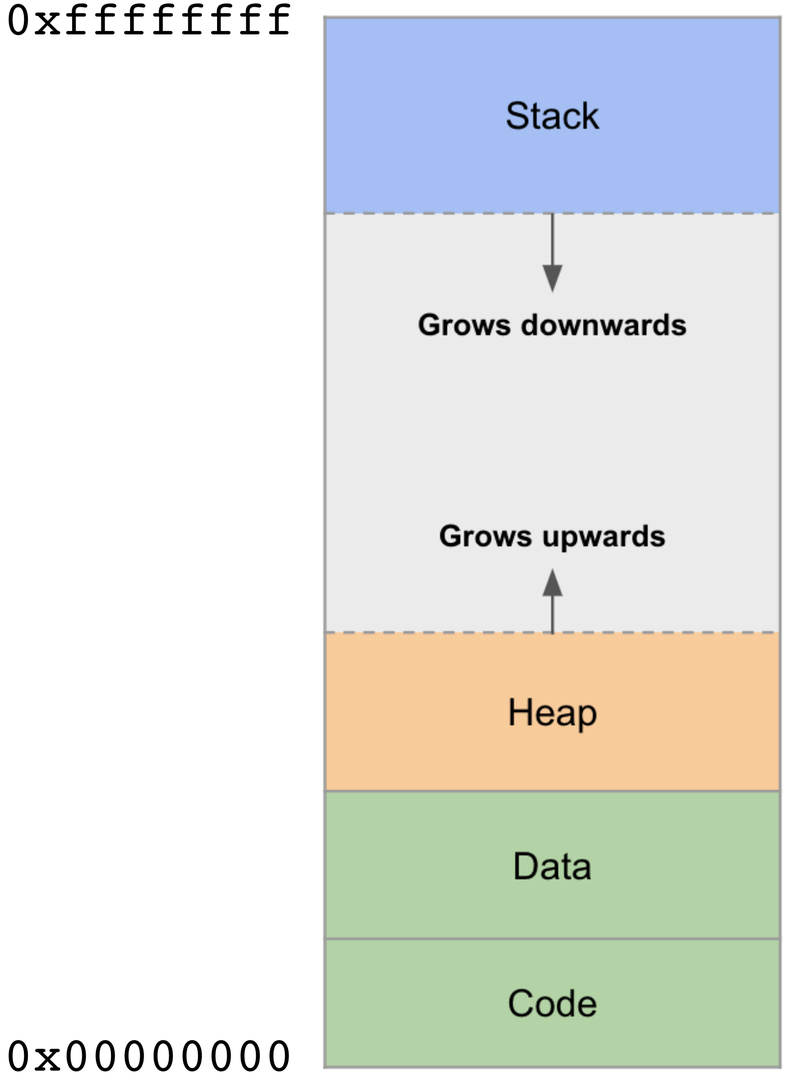
Dynamically allocated memory, e.g., using malloc and free

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

- Little-endian
- Variable-length instructions: 1 to 16 bytes.
- Storage units on CPU: each register can store 1 word (4 bytes)
- 6 general-purpose registers:
  - eax, ebx, ecx, edx, esi, edi
- eip: instruction pointer
- ebp: base pointer (frame pointer)
- esp: stack pointer

x86 instructions

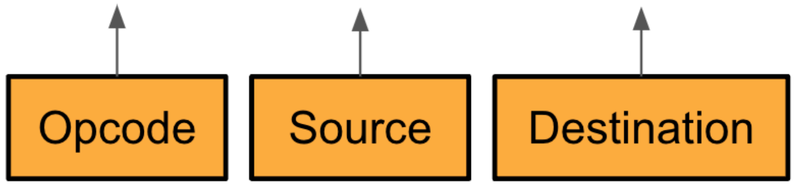
- Little-endian
- Variable-length instructions: 1 to 16 bytes.
- Storage units on CPU: each register can store 1 word (4 bytes)
- 6 general-purpose registers:
  - eax, ebx, ecx, edx, esi, edi
- eip: instruction pointer
- ebp: base pointer (frame pointer)
- esp: stack pointer

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

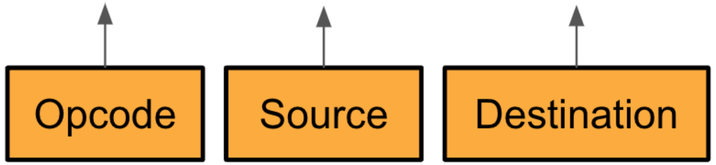
- Instructions are composed of an opcode and zero or more operands.
- `add $0x8, %ebx`



- Pseudocode: `EBX = EBX + 0x8`

x86 Assembly

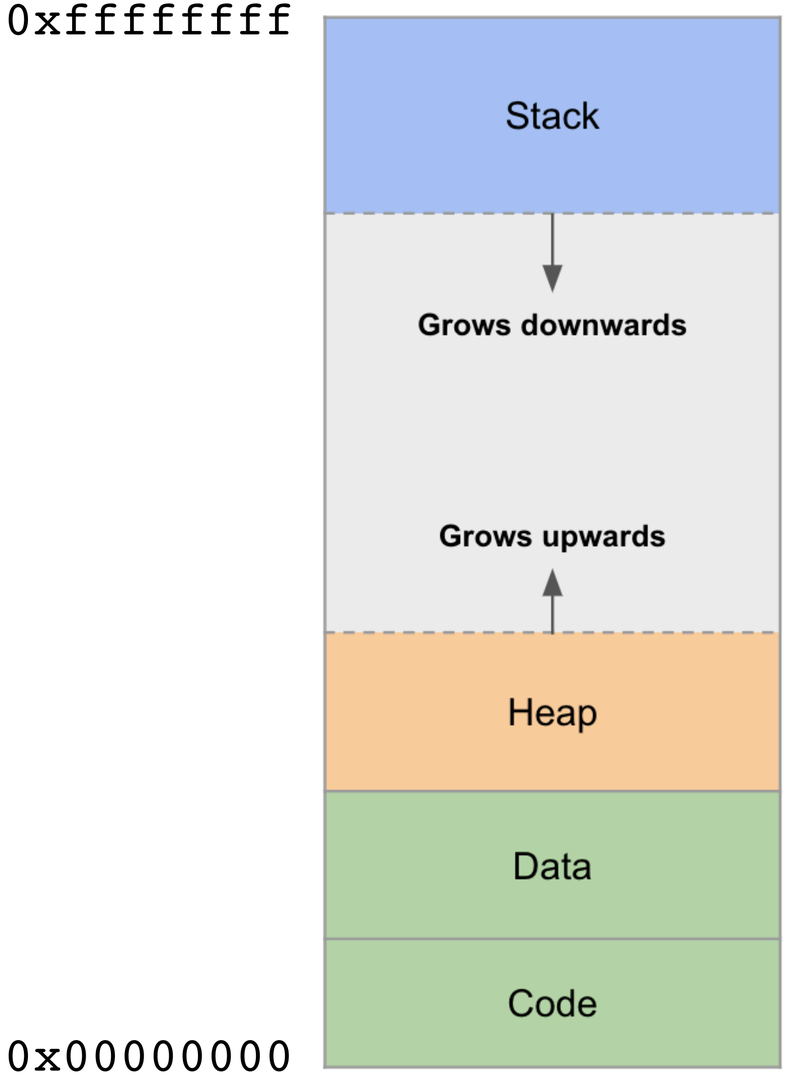
- `xorl 4(%esi), %eax`



- 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"

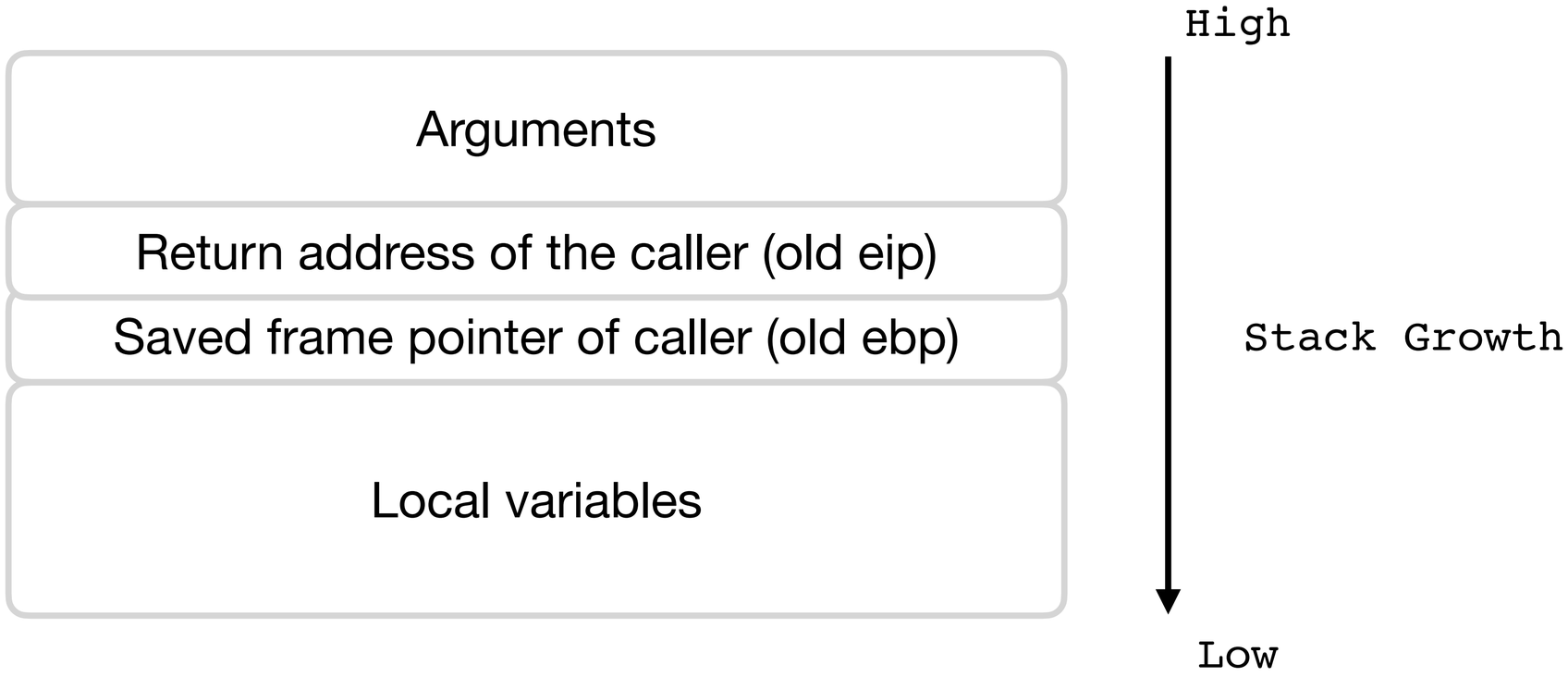
x86 Memory Layout



Local variables and stack frames

Where buffer overflow happens

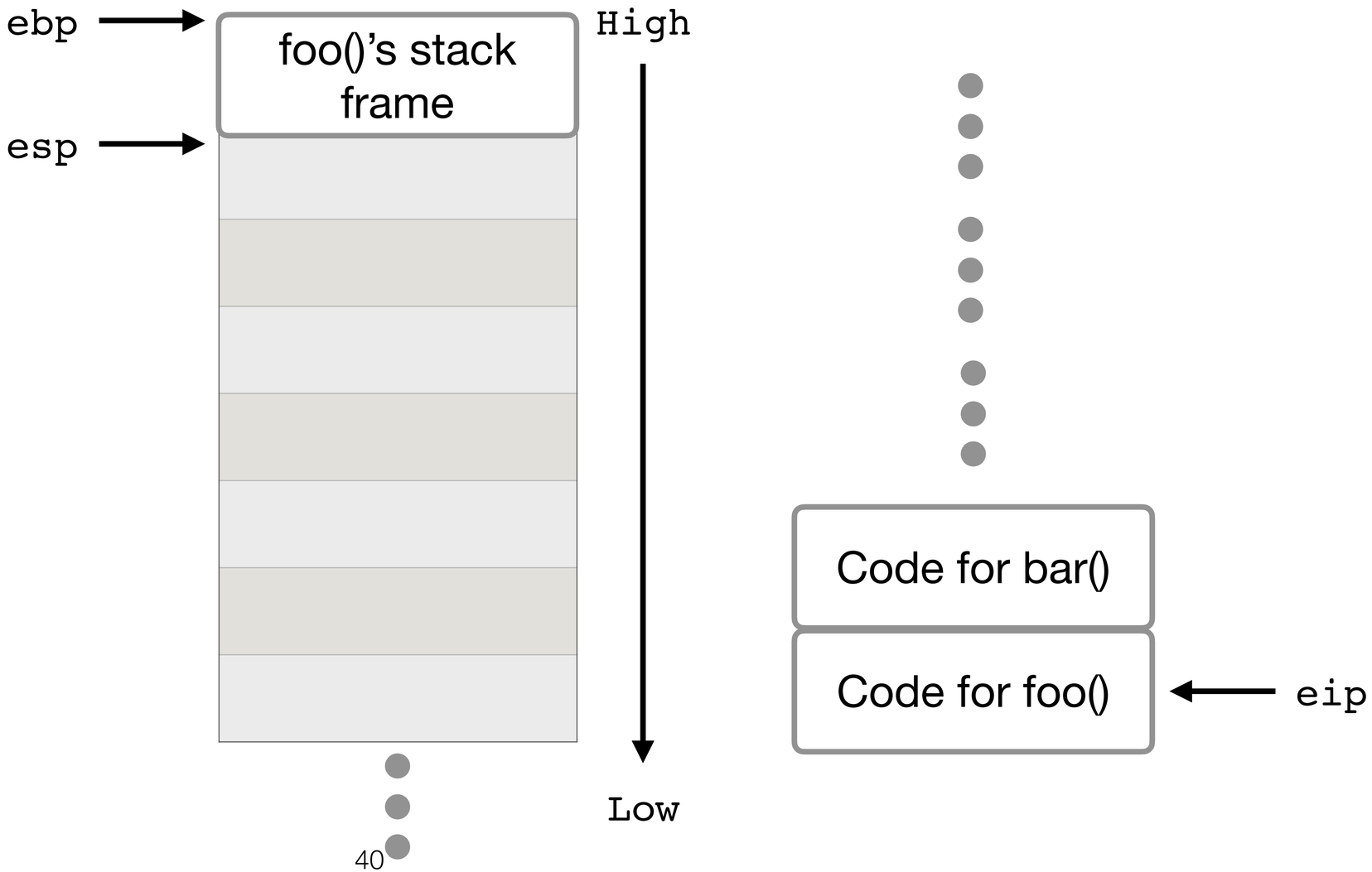
Stack Frame of a Function



Stack Frames: Calling a Function

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

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



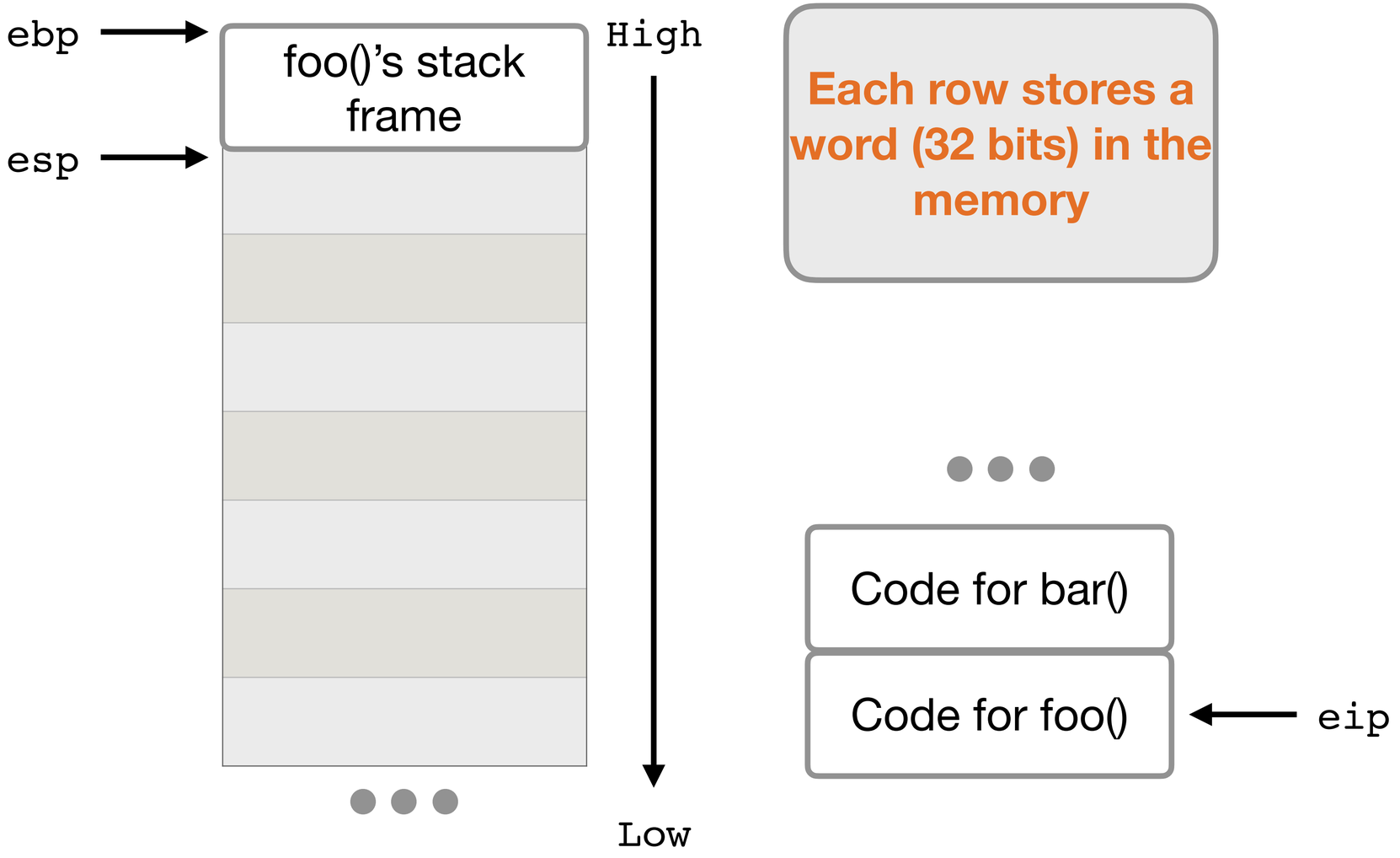
Note:

- 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;
    ...
}
```



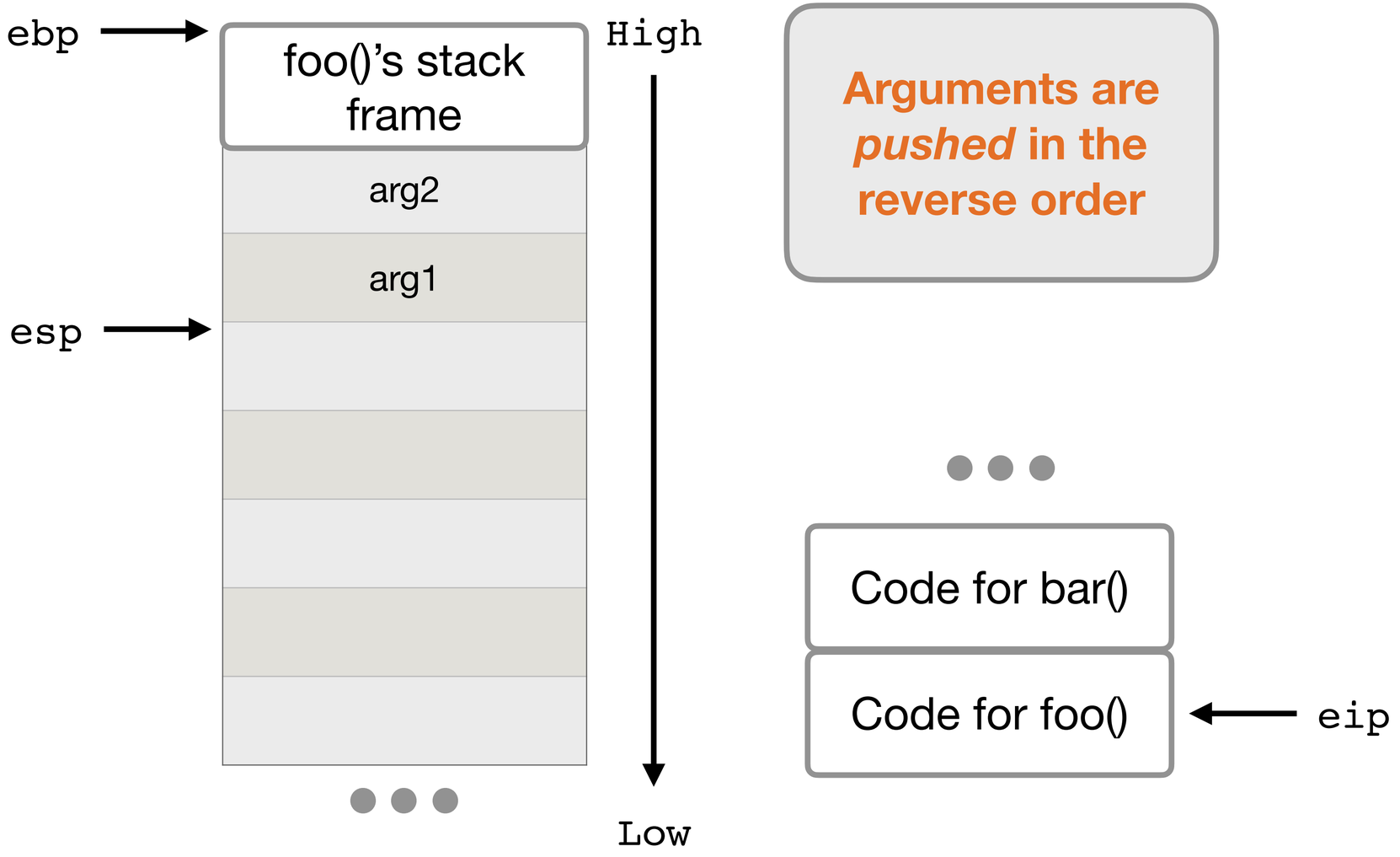
Note:

- 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;
    ...
}
```



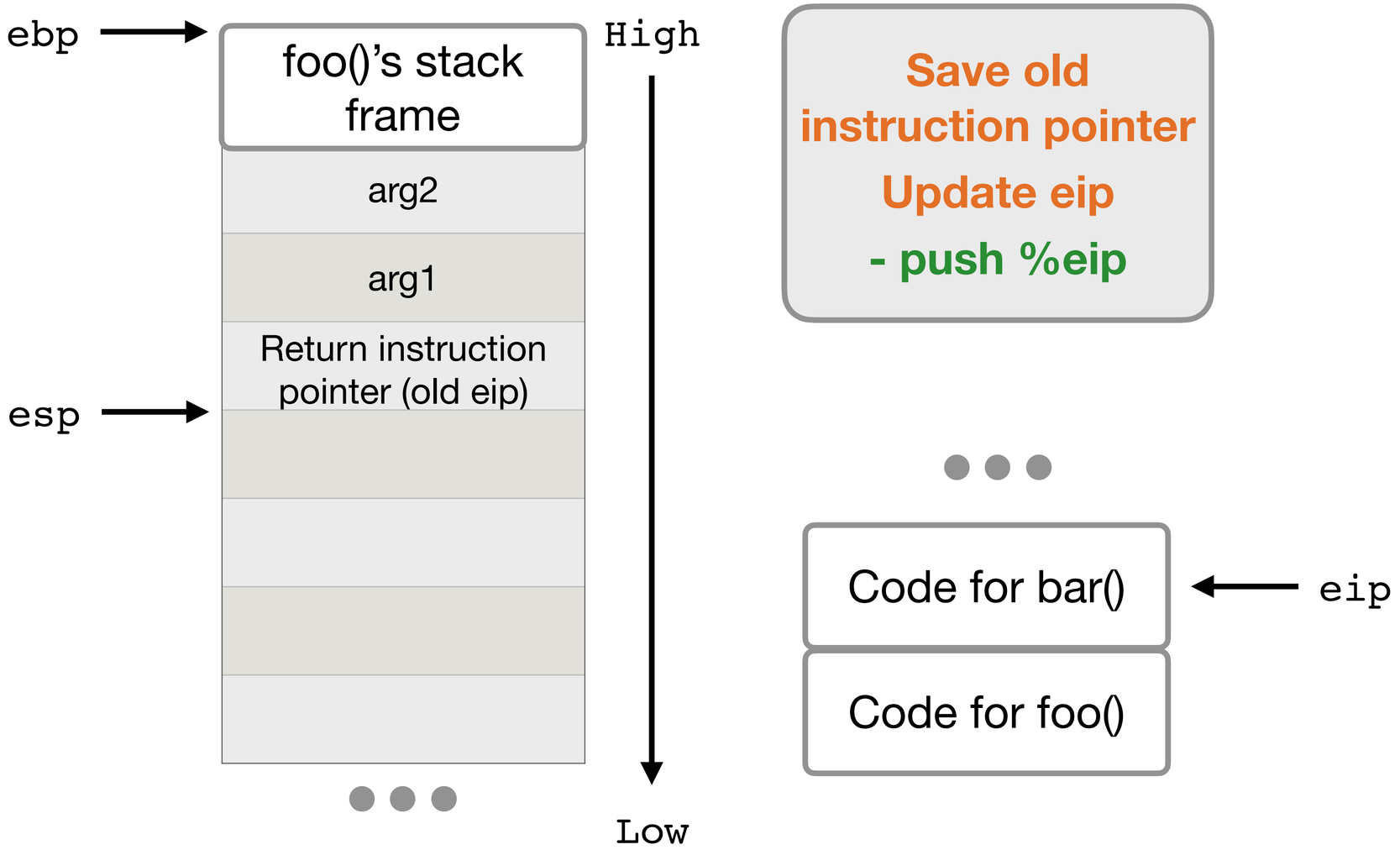
Note:

- 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;
    ...
}
```



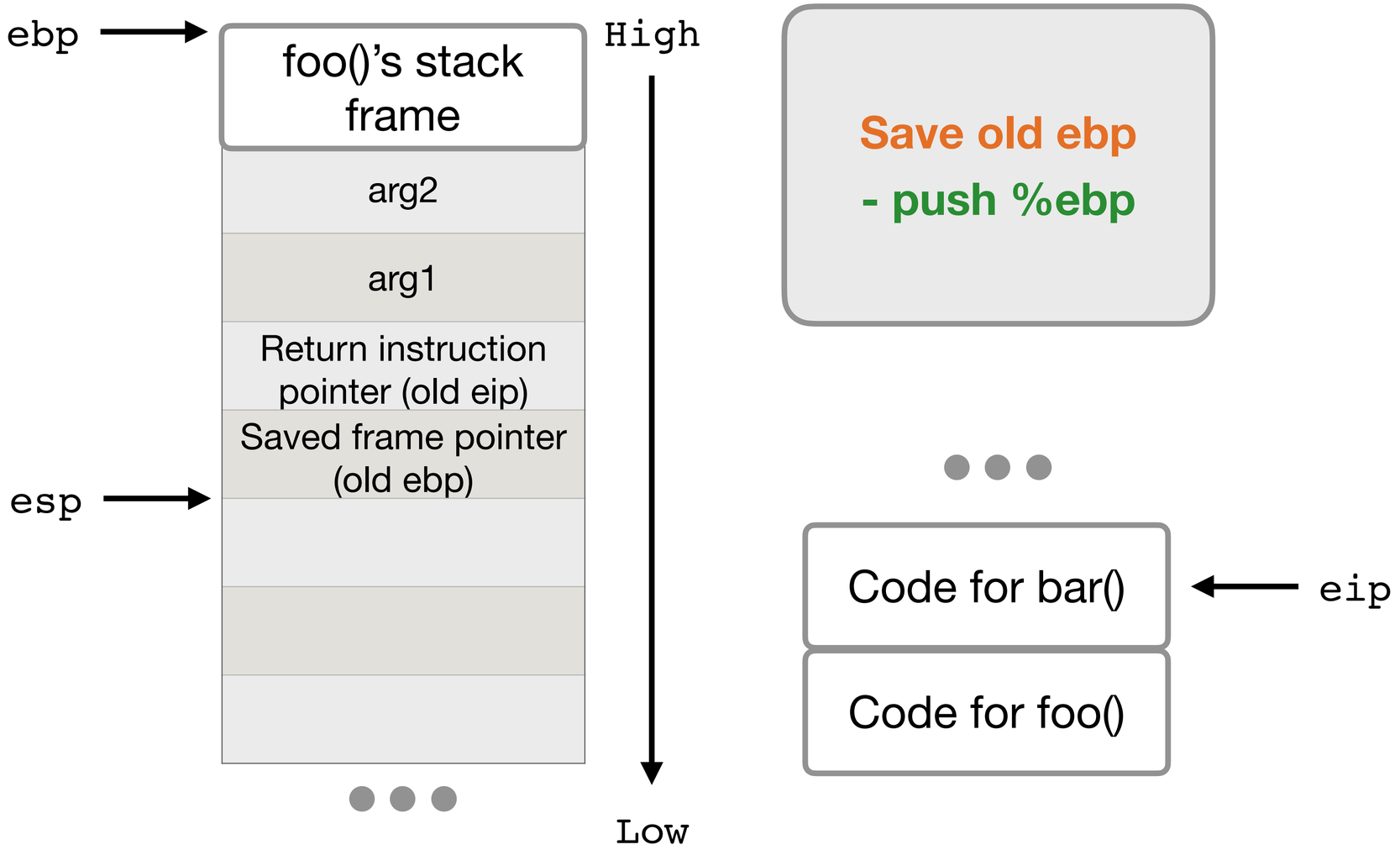
Note:

- 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;
    ...
}
```



Note:

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

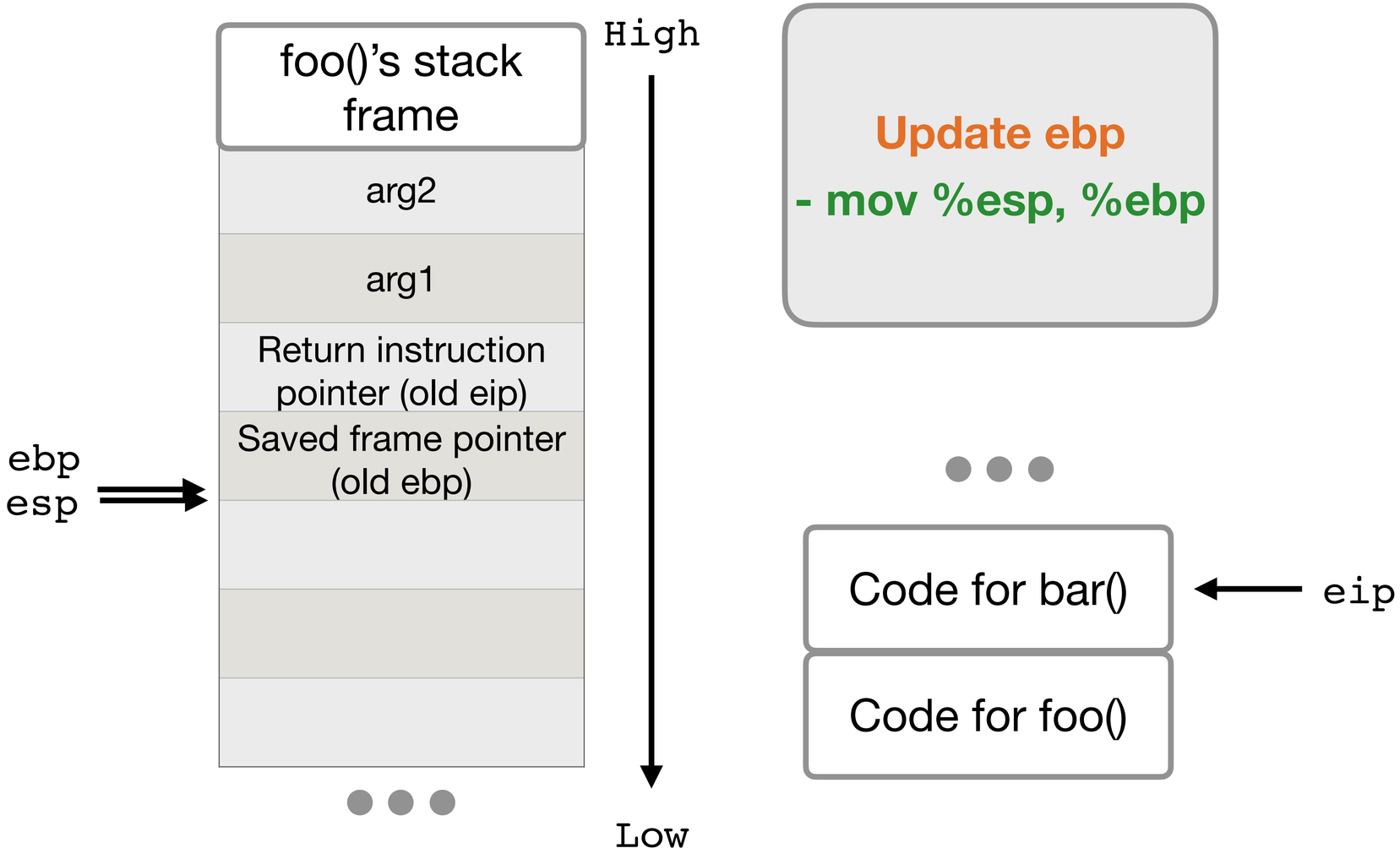
Stack Frames: Calling a Function

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

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

Note:

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



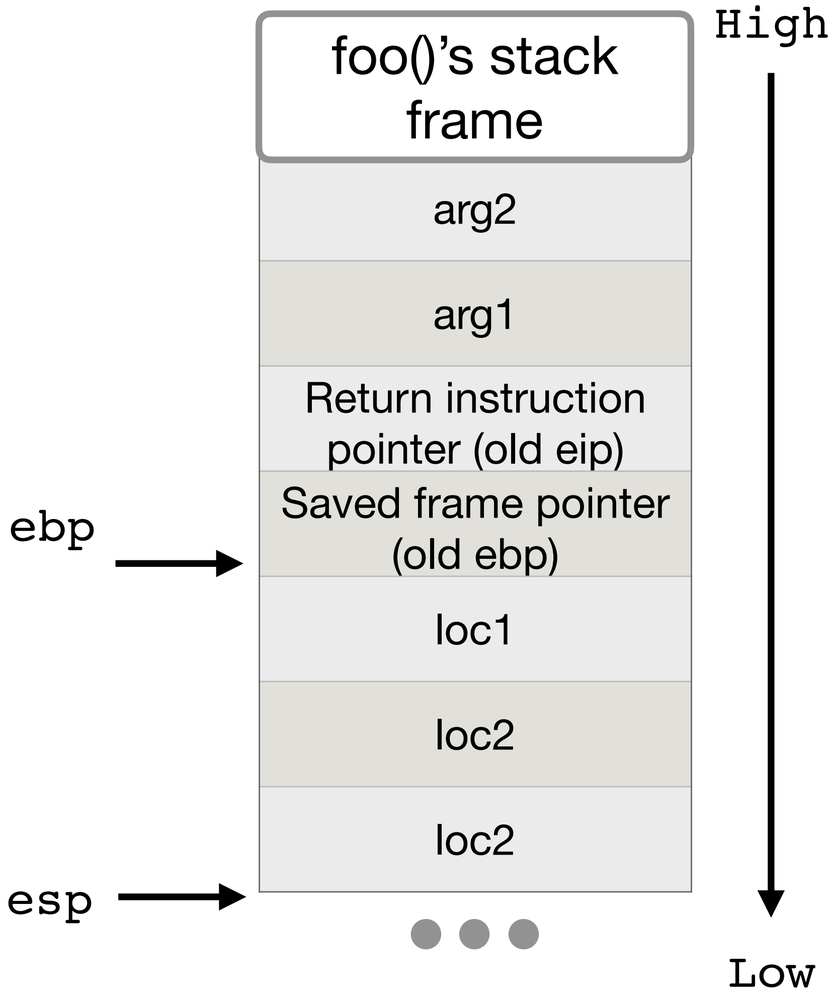
Stack Frames: Calling a Function

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

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

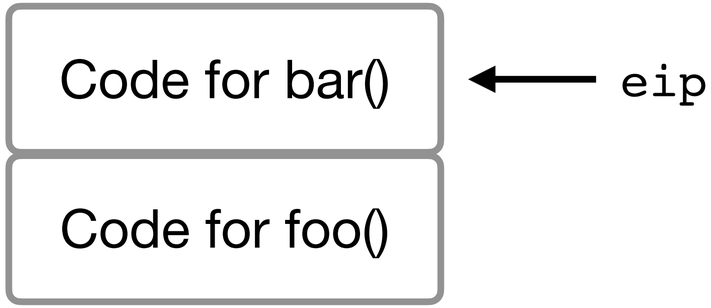
Note:

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



***Push* local variables in the same order as they appear in the code**

- **loc2 is 2 words**
- **little-endian**



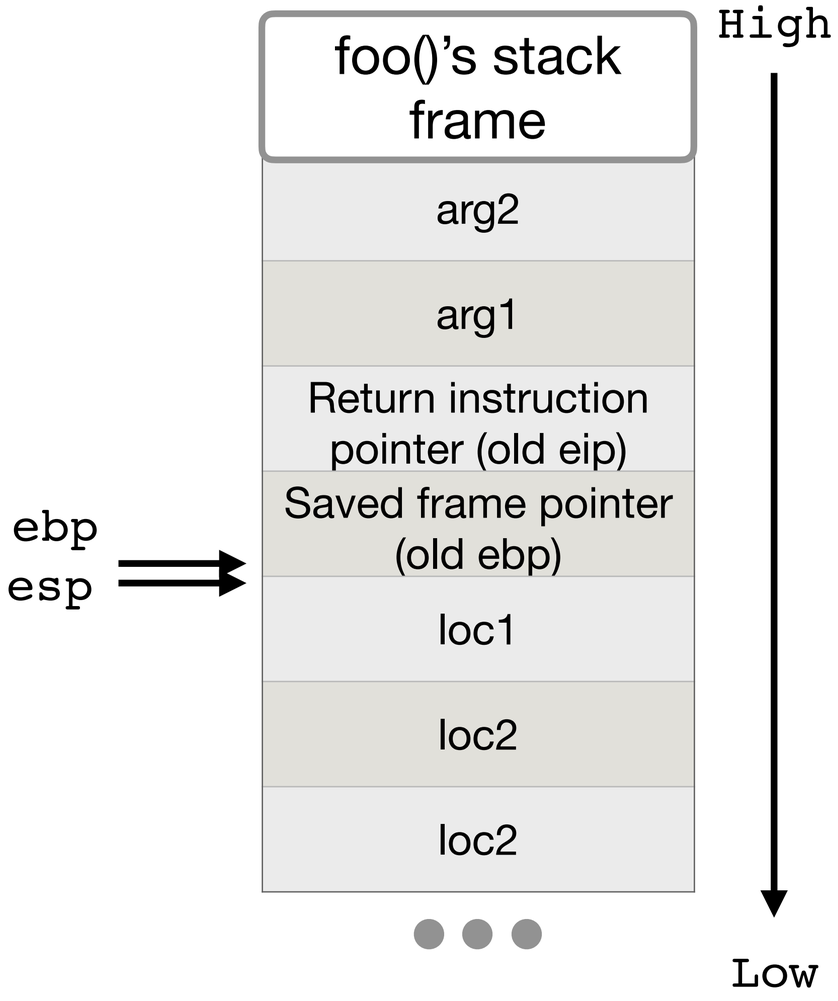
Stack Frames: Return from a Function

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

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

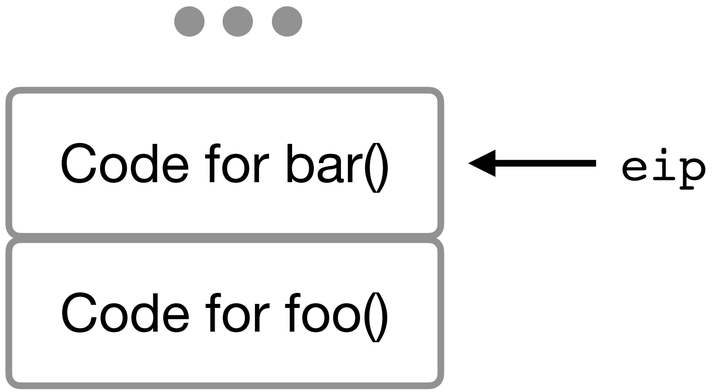
Note:

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



Restore the stack pointer

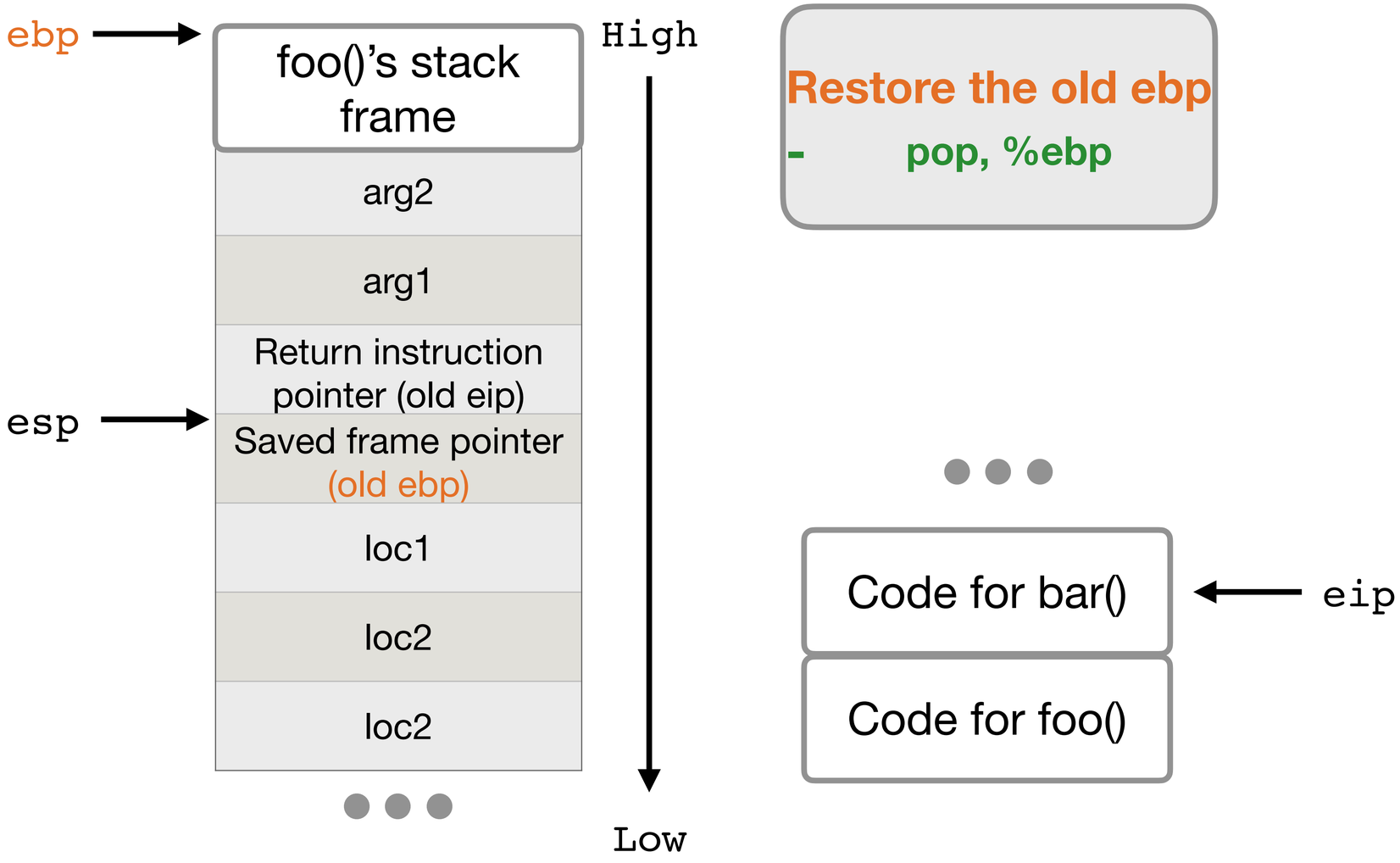
- mov %ebp, %esp
- This does not wipe off loc1 and loc2 from the memory
- They are undefined



Stack Frames: Return from a Function

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

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



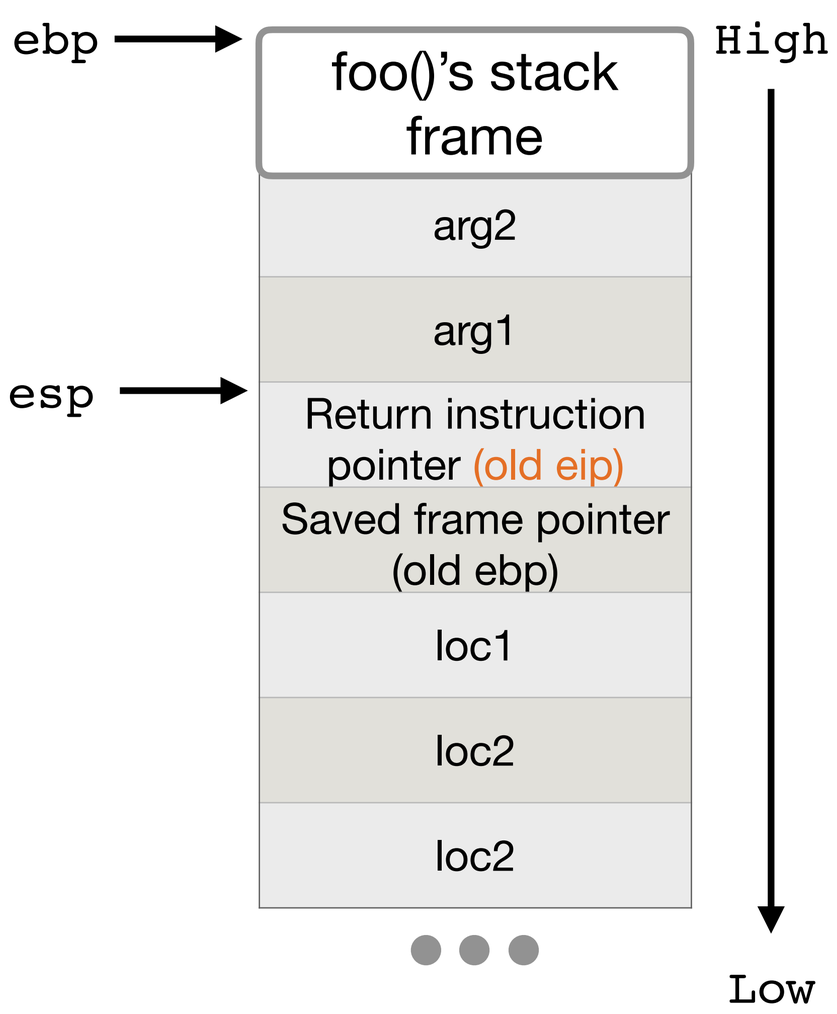
Note:

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

Stack Frames: Return from a Function

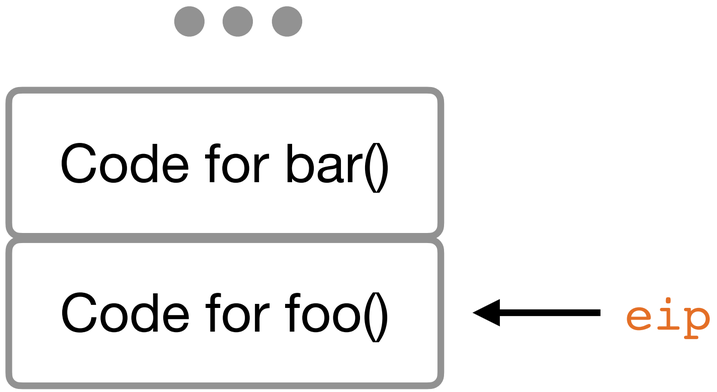
```
void foo() {
    …
    bar(arg1, arg2);
}

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



Restore the old eip

\- pop %eip



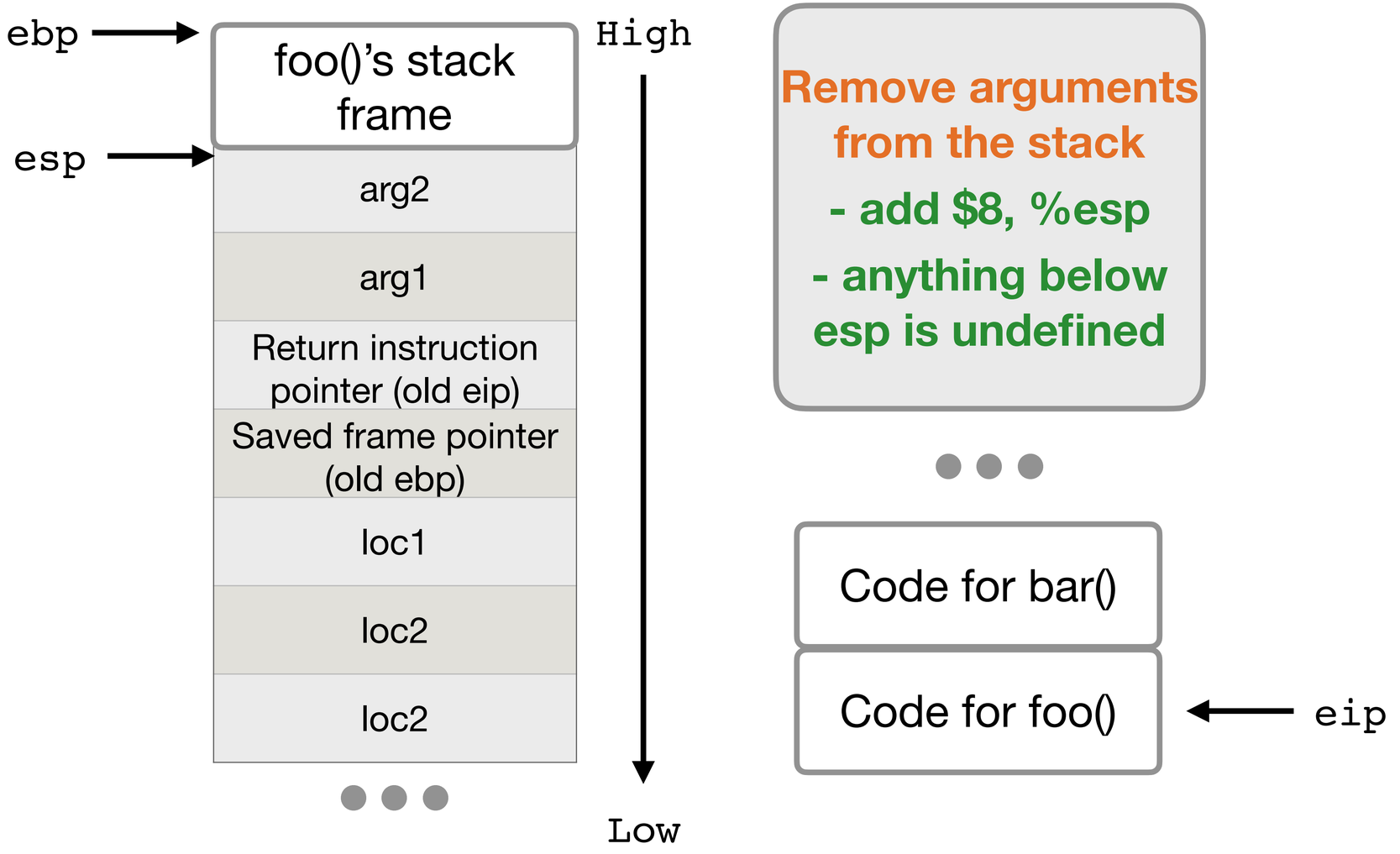
Note:

- 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

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 (frame pointer)
  - **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

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

Buffer Overflow

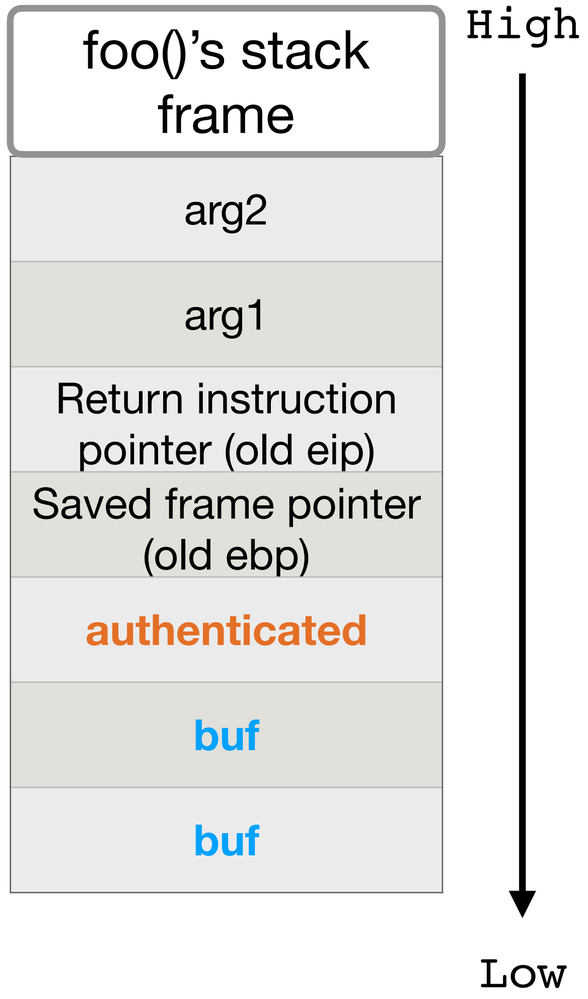
```
char name [2];
name[3] = 'a';
```

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