CMSC414 Computer and Network Security

Memory Safety Vulnerabilities

Yizheng Chen | University of Maryland
surrealyz.github.io

Feb 1, 2024

Announcements

- Piazza Signup:
  - https://piazza.com/umd/spring2024/cmsc4140102/
- Project 1:
  - Will upload the zip file for project content
- Gitlab:
  - Submissions, backup your work, version control

Agenda

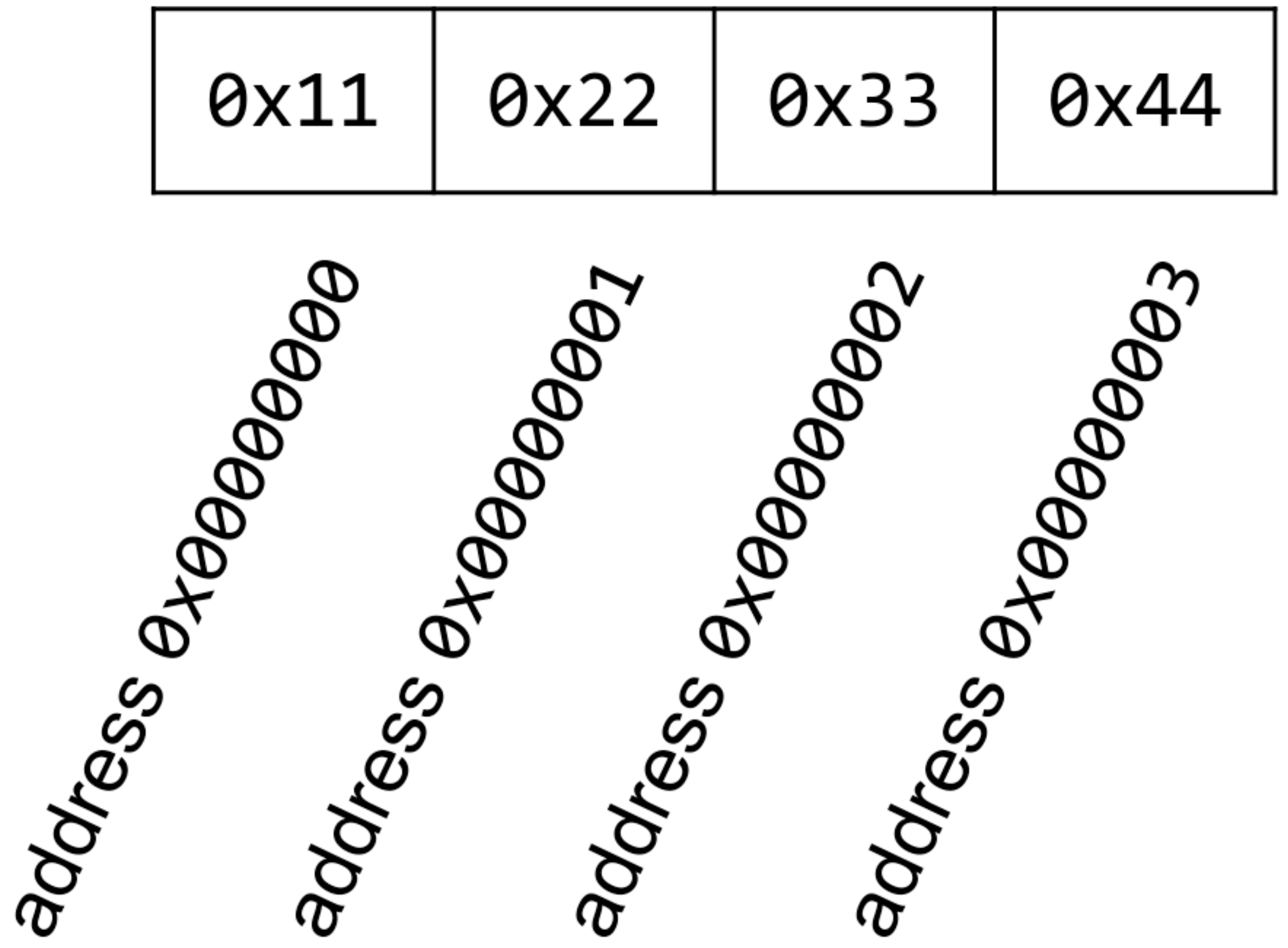
- Recap
- Buffer overflow
- Stack smashing
- Format string vulnerabilities
- Integer conversion vulnerabilities
- Recap
- Off-by-one vulnerabilities

Memory Address vs Content in Memory

- In a 32 bit system, a memory address is 32 bits
  - Could be represented in 8 hex digits
  - esp, ebp, eip store addresses that point to somewhere in memory
  - **eip, instruction pointer, points to the instruction to execute**
- In C, the basic unit of content in the memory is a byte.
- If we just index into a byte, or store a byte, it is read or written as is in memory.

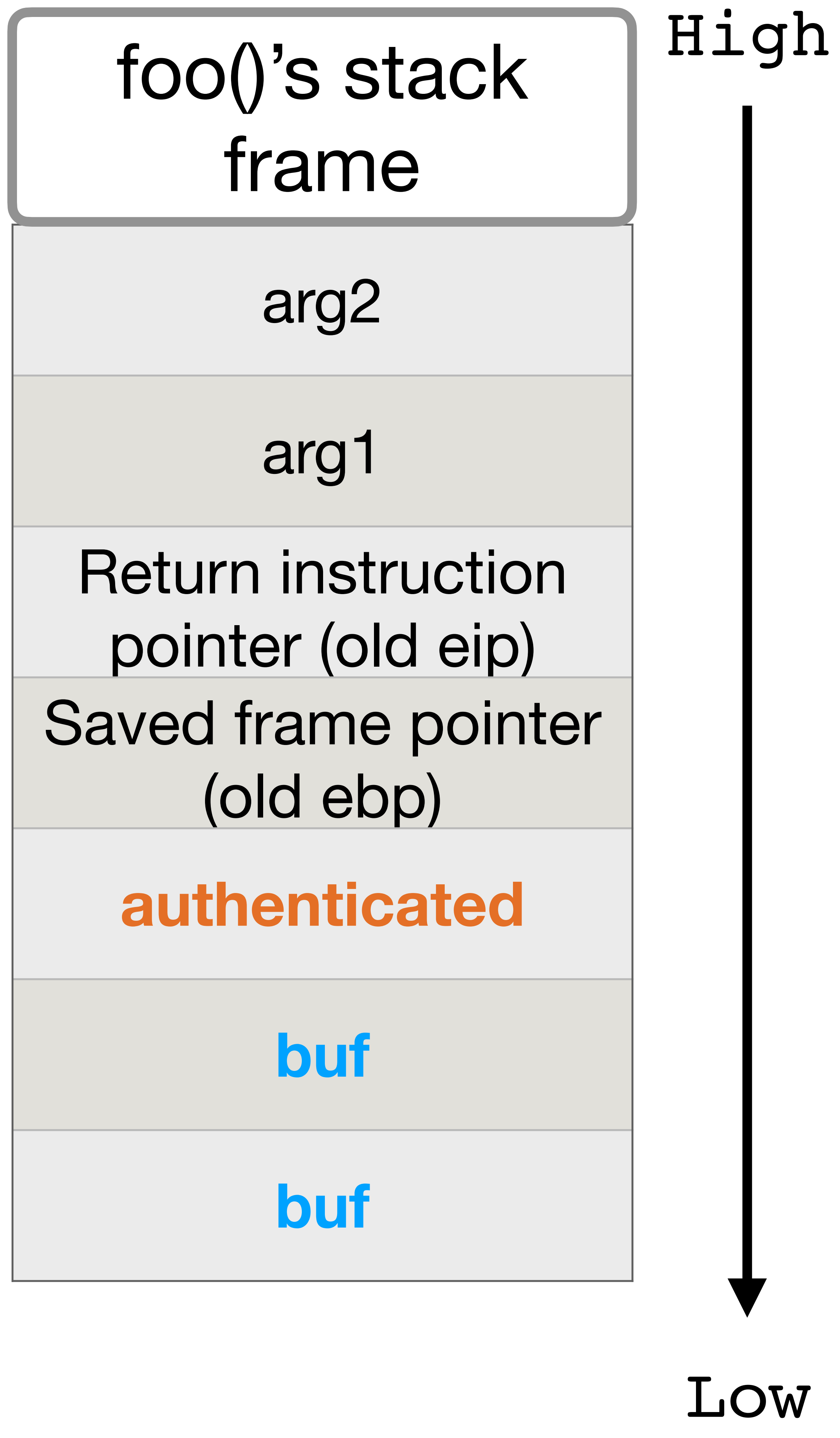
Little-Endian System

- For data types larger than one byte (8 bits)
  - e.g., short, int, long, pointers (to anything)
  - If we refer to a char, it is only one byte, so it is stored as is
- When we **increase an index** for referencing items in an array, or **increase a pointer** by some number of bytes: memory address always goes up



How to change authenticated to 1?

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

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

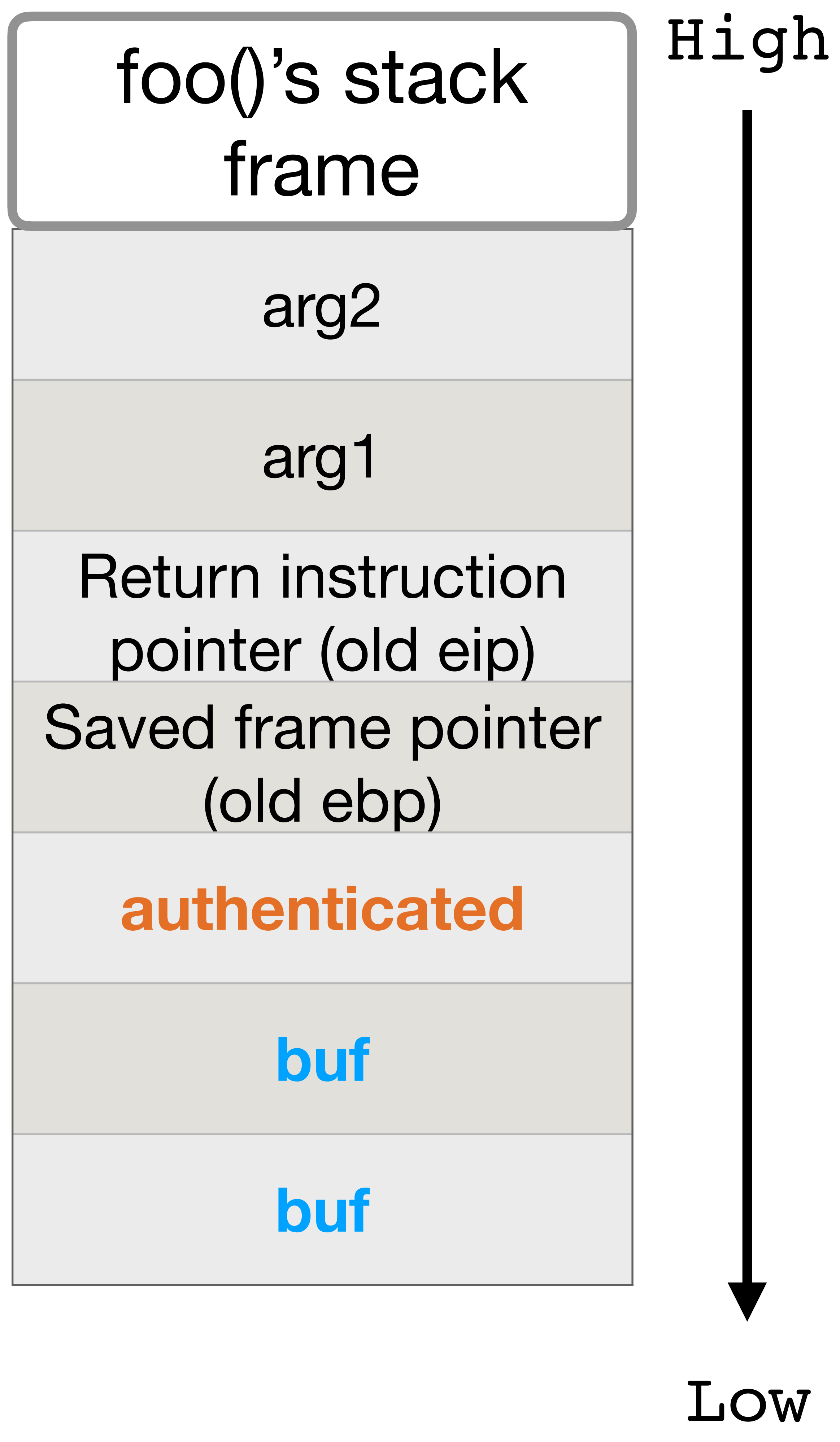


Buffer Overflow

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

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

Set buf[8] to non-zero

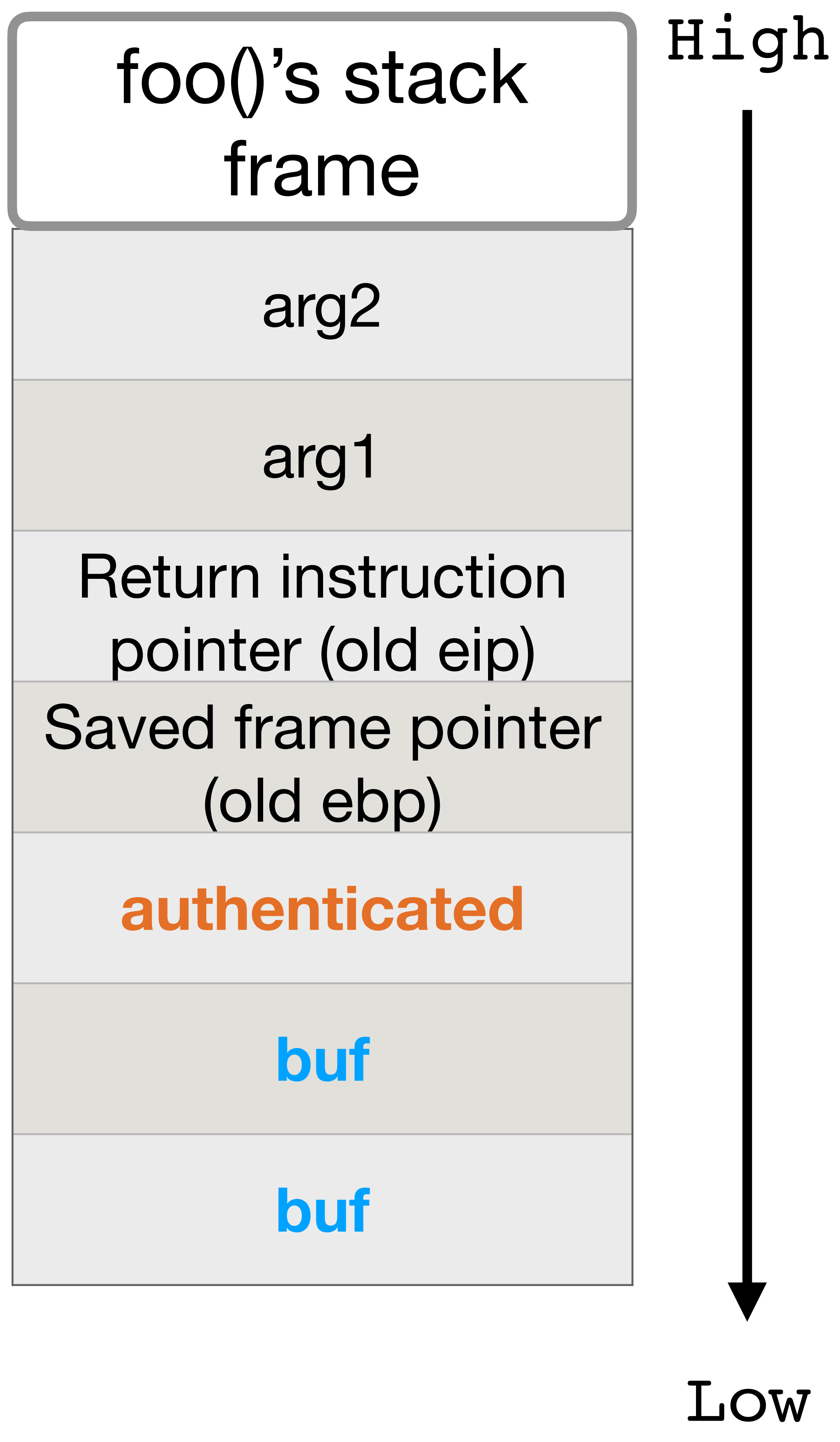
Hint: little-endian



Buffer Overflow

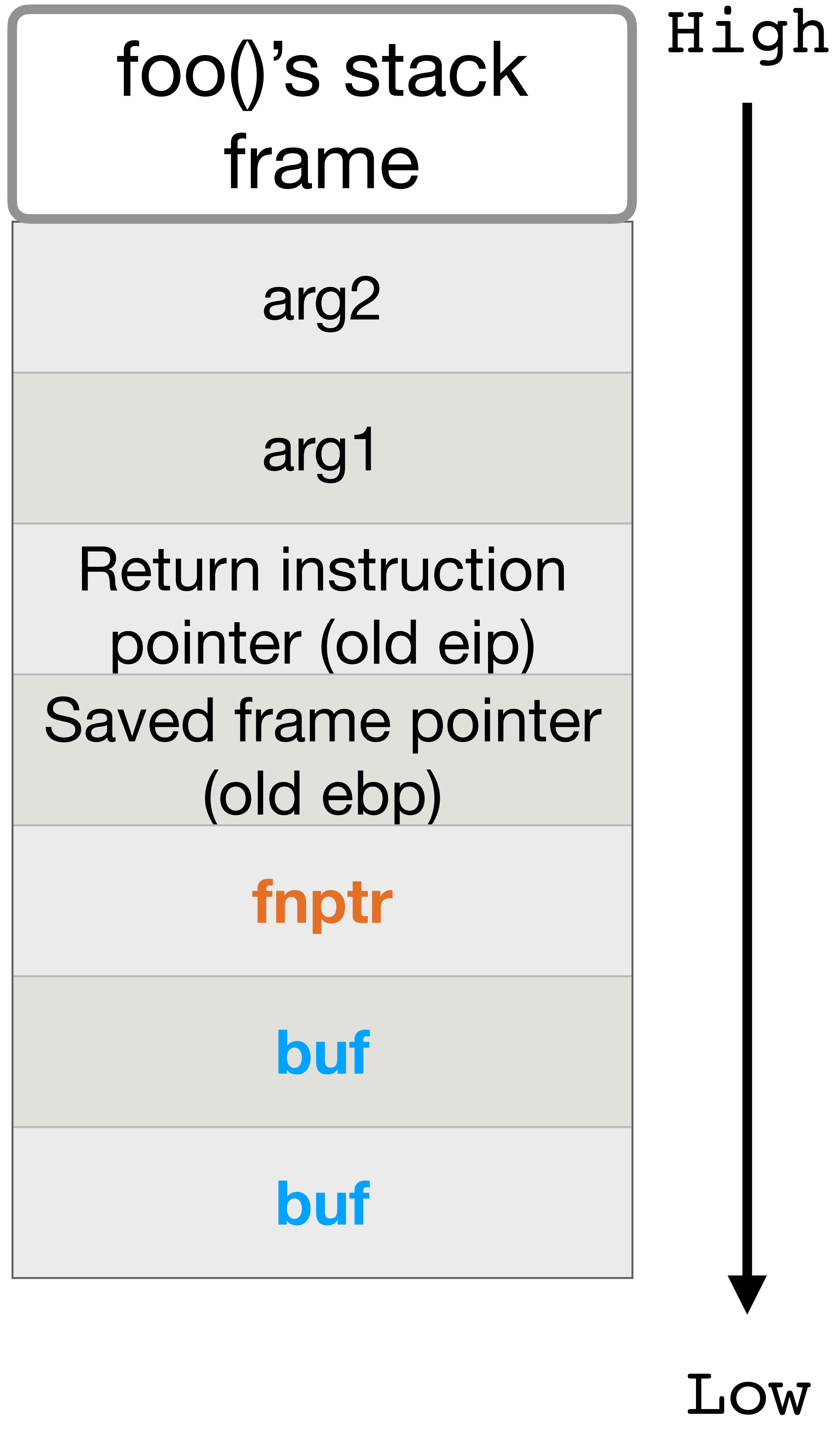
```
void foo() {
    …
    bar(arg1, arg2);
}

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!"**



Special Opportunity: Overwrite Function Pointer

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

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



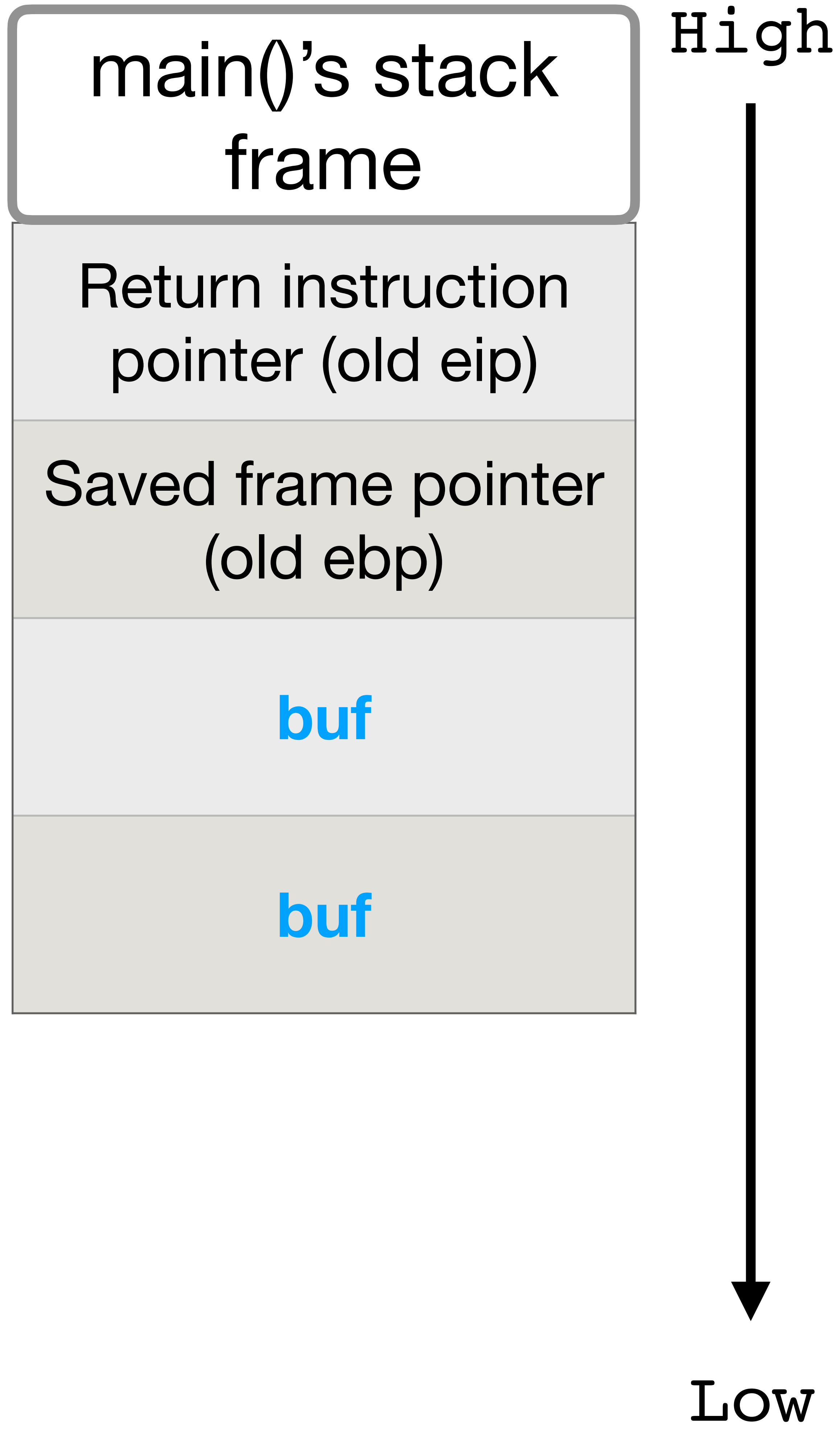
In previous examples, buffer that can be overflowed must be followed in memory by some security-critical data (e.g., a function pointer, or a special flag).

But these conditions rarely occur in practice…

How to inject and execute malicious code in general?

Malicious Code Injection

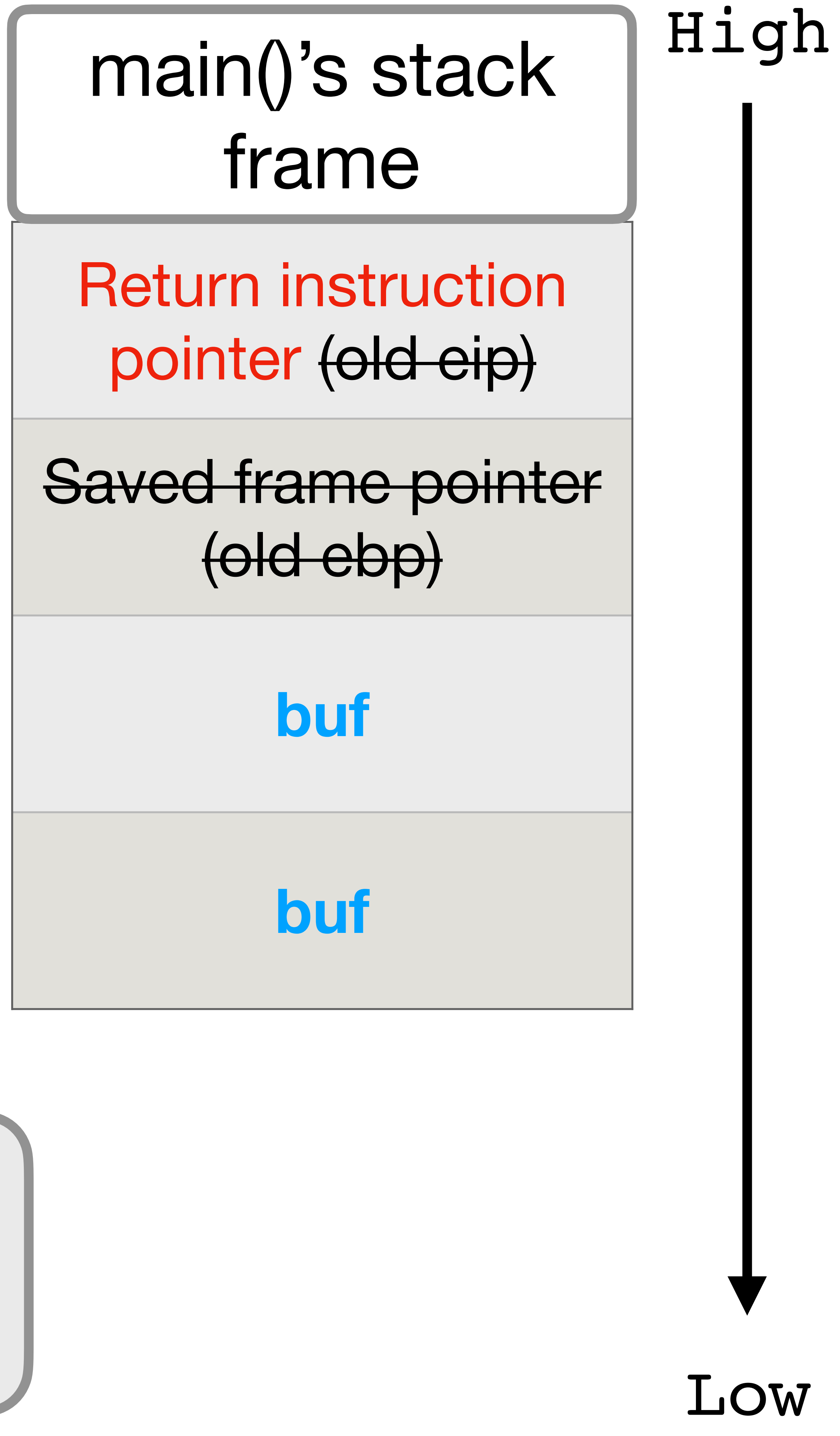
```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[8];
    gets(buf)
    ...
}
```



How to point to malicious code at 0xdeadbeef?

Stack Smashing

```c
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[8];
    gets(buf)
    ...
}
```



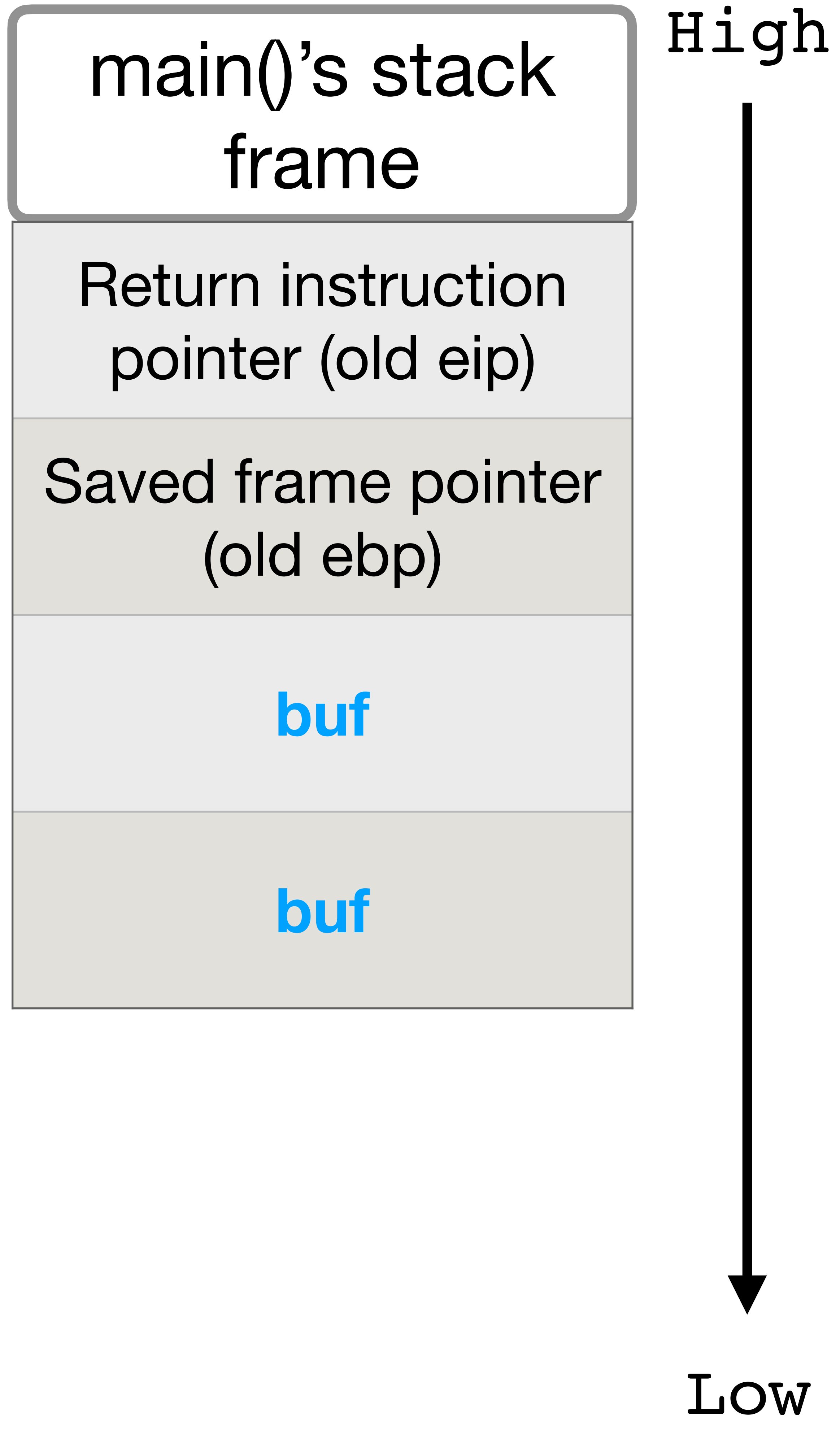
AAAAAAAAAAAA\xef\xbe\xad\xde

Shellcode

- The malicious code is often written to spawn an interactive shell that lets the attacker perform arbitrary actions.

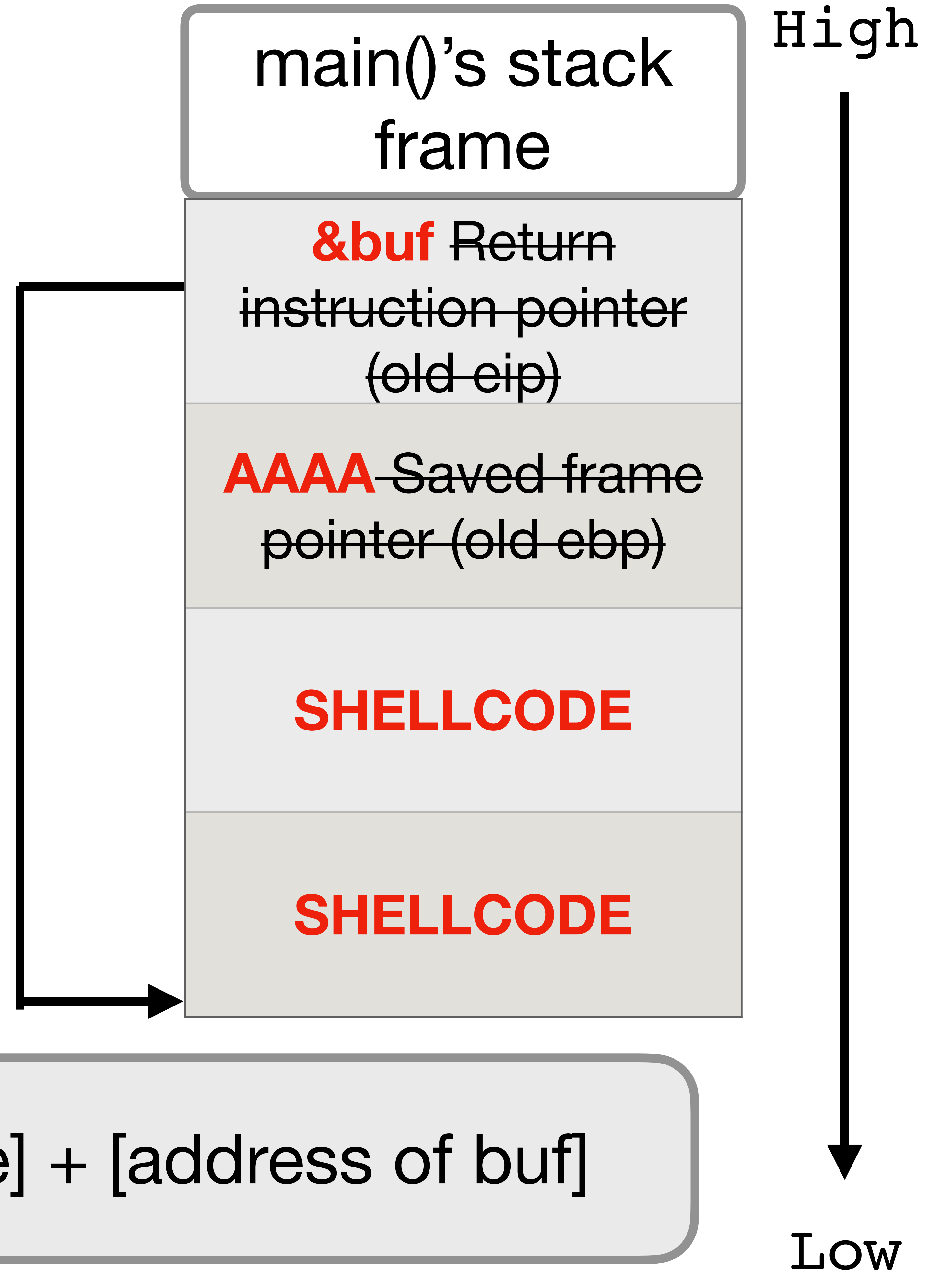
What if malicious code isn't in memory yet?

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[8];
    gets(buf)
    ...
}
```



Simple case: if shell code is only 8 bytes

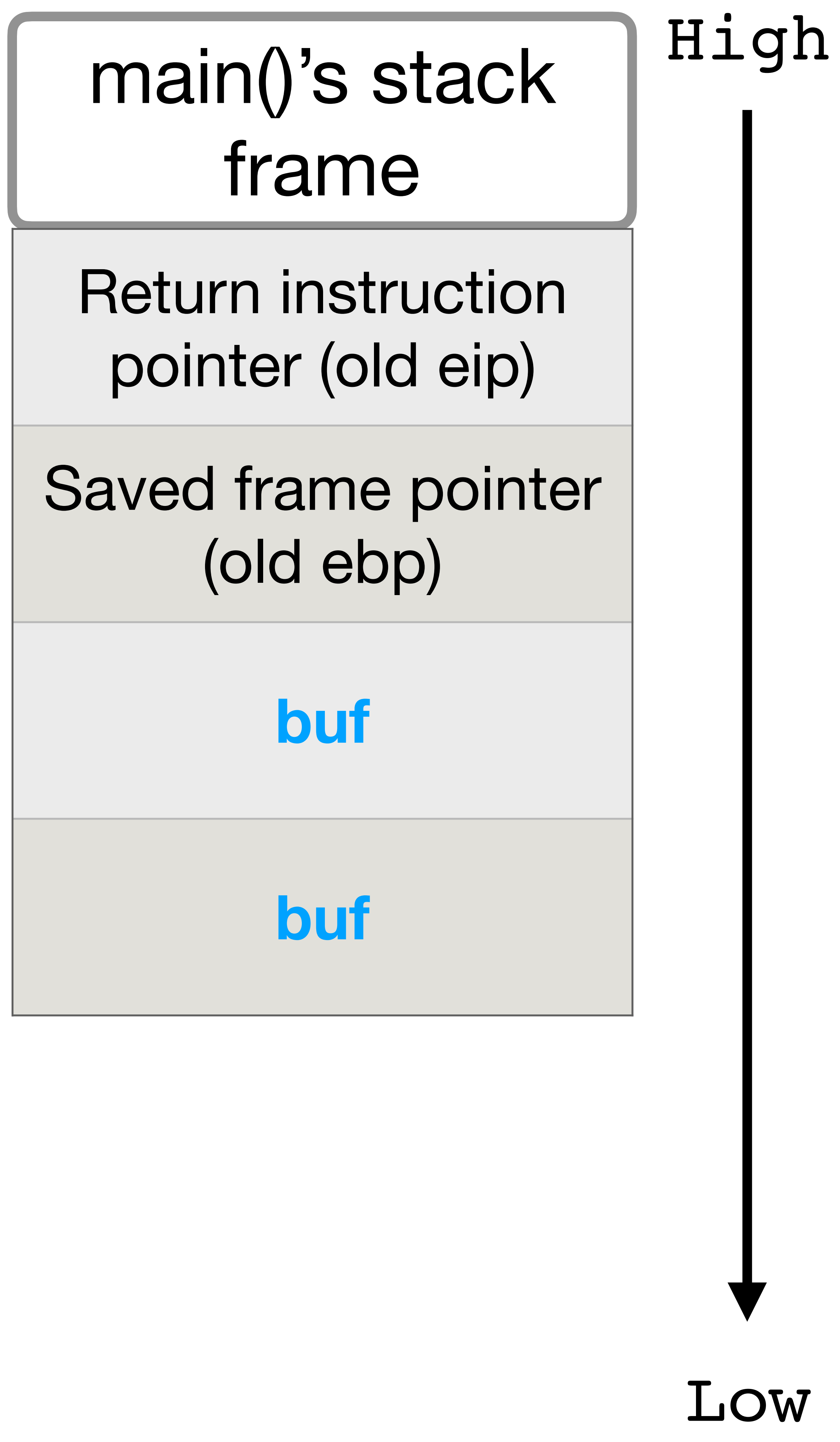
```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[8];
    gets(buf)
    ...
}
```



[8 bytes of SHELLCODE] + [4 bytes of garbage] + [address of buf]

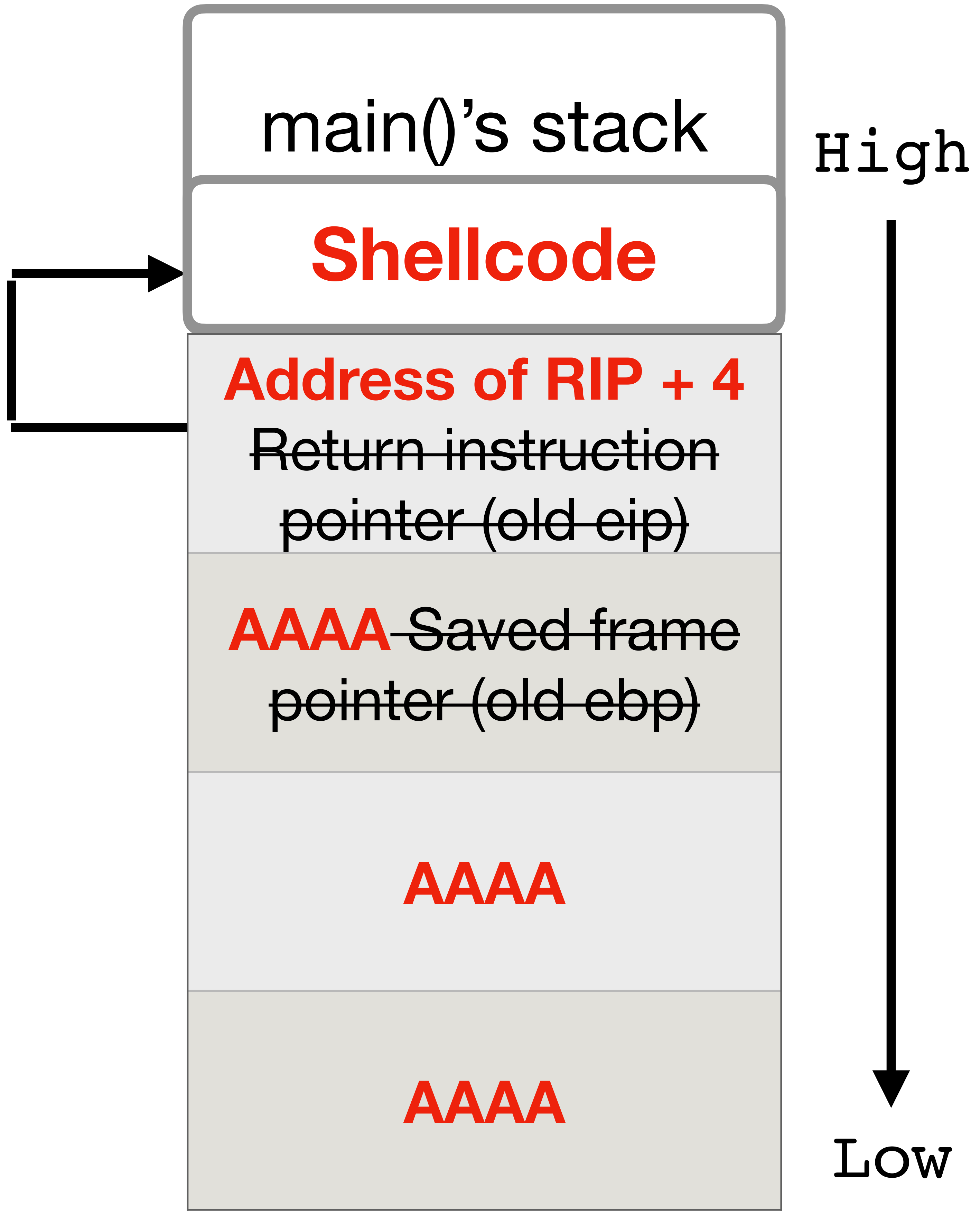
What if shell code is longer than 8 bytes?

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[8];
    gets(buf)
    ...
}
```



What if shell code is longer than 8 bytes?

```c
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[8];
    gets(buf)
    ...
}
```



[12 bytes of garbage] + [address of RIP + 4] + [shellcode]

Hint: need to find the address of Return Instruction Pointer (RIP)

Sophisticated Attacks

- The malicious code is stored at an unknown location.
- The buffer is stored on the heap instead of on the stack.
- The characters that can be written to the buffer are limited (e.g., to only lowercase letters).
- There is no way to introduce *any malicious code* into the program's address space.

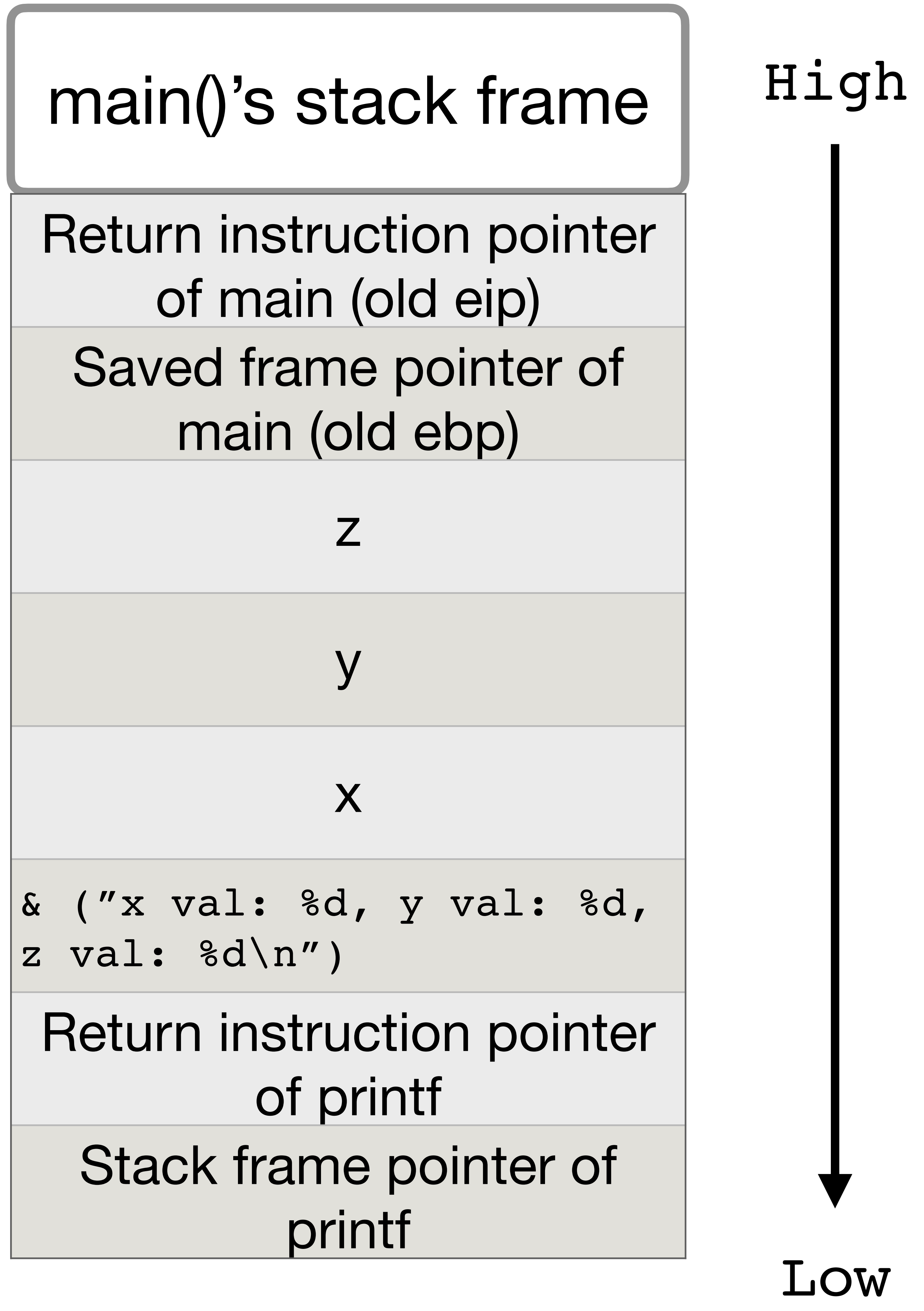
If your program has a buffer overflow bug, you should assume that the bug is exploitable and an attacker can take control of your program.

Agenda

- Recap
- Buffer overflow
- Stack smashing
- Format string vulnerabilities
- Integer conversion vulnerabilities
- Recap
- Off-by-one vulnerabilities

printf

```c
void main() {
    not_vulnerable();
}

void not_vulnerable() {
    printf("x val: %d, y val:
%d, z val: %d\n", x, y, z);
}
```



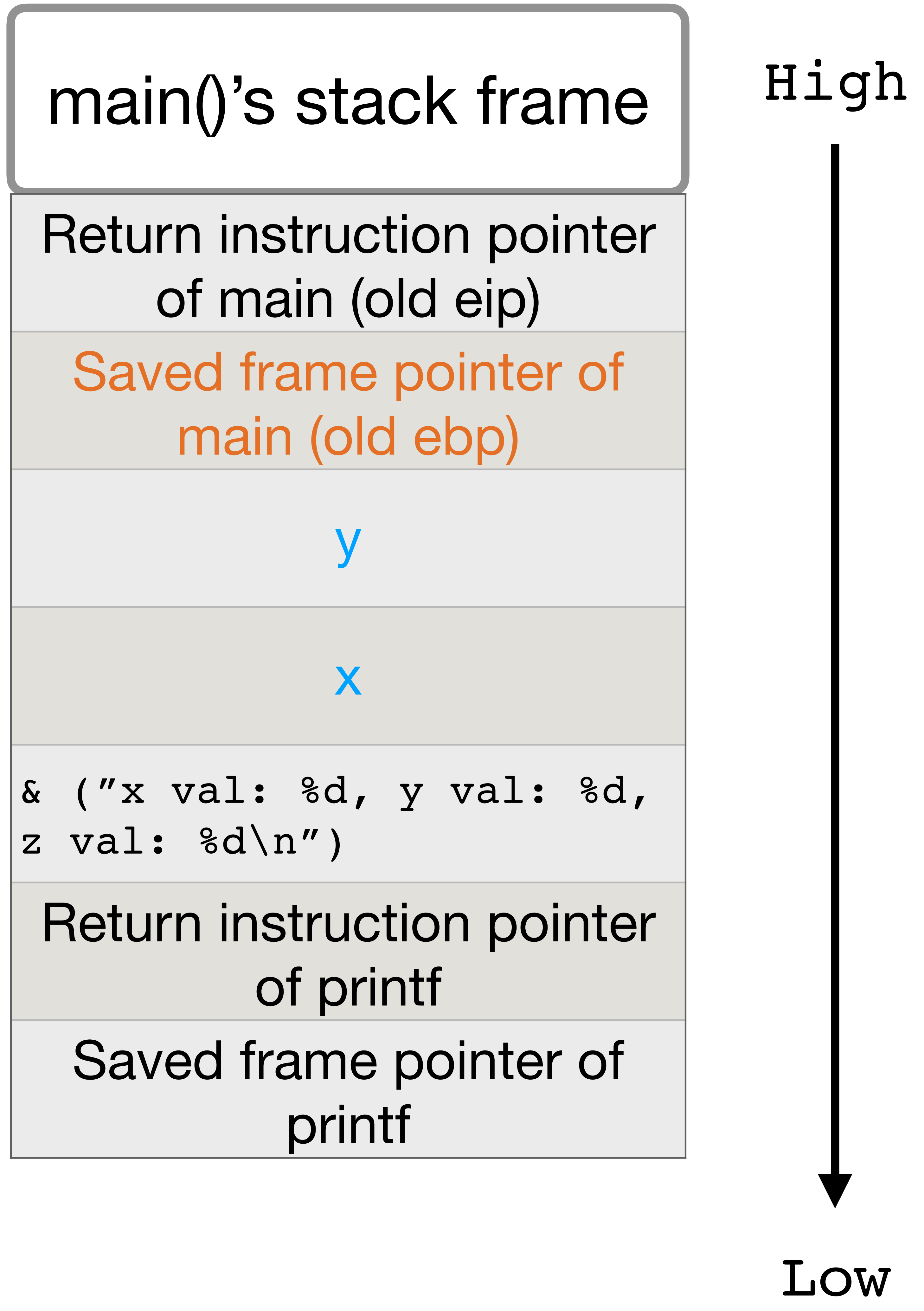
The format string "x val: … %d\n" controls the behavior of printf

Internal pointer in printf looks for content on the stack

Format String Vulnerability

```
void main() {
    vulnerable();
}

void vulnerable() {
    printf("x val: %d, y val:
%d, z val: %d\n", x, y);
}
```

The format string “x val: … %d\n” controls the behavior of printf
Internal pointer in printf looks for content on the stack



Other Formats

- %s → Treat the argument as an address and print the string at that address up until the first null byte
- %n → Treat the argument as an address and write the number of characters that have been printed so far to that address
- %c → Treat the argument as a value and print it out as a character
- %x → Look at the stack and read the first variable after the format string
- %[b]u → Print out [b] bytes starting from the argument

Format string vulnerability: the attacker can learn any value stored in memory and can take control of your program.

What’s wrong with this code?

```
char buf[8];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > 8) {
        error("length too large: bad dog, no cookie for 
you!");
        return;
    }
    memcpy(buf, p, len);
}
```

```
void *memcpy(void *dest, const void *src, size_t n);

typedef unsigned int size_t;
```

Integer Conversion Vulnerabilities

```
char buf[8];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > 8) {
        error("length too large: bad dog, no cookie for
you!");
        return;
    }
    memcpy(buf, p, len);
}
```

If len is a negative integer, casting it to unsigned int could make it a very large int.

=> buffer overflow

Integer Wraparound

- Unsigned int: 0 ~ 2^32 - 1
  - Adding 1 to 2^32 - 1 becomes 0
- If an unsigned integer x = 0, x - 1 = 2^32 - 1

Exercise: What's wrong with this code?

```c
void vulnerable() {
    size_t len;
    char *buf;

    len = read_int_from_network();
    buf = malloc(len+5);
    read(fd, buf, len);
    ...
}
```

Agenda

- Recap
- Buffer overflow
- Stack smashing
- Format string vul
- Integer conversio
- Recap
- Off-by-one vulnerabilities

Off by one byte can lead to malicious code injection

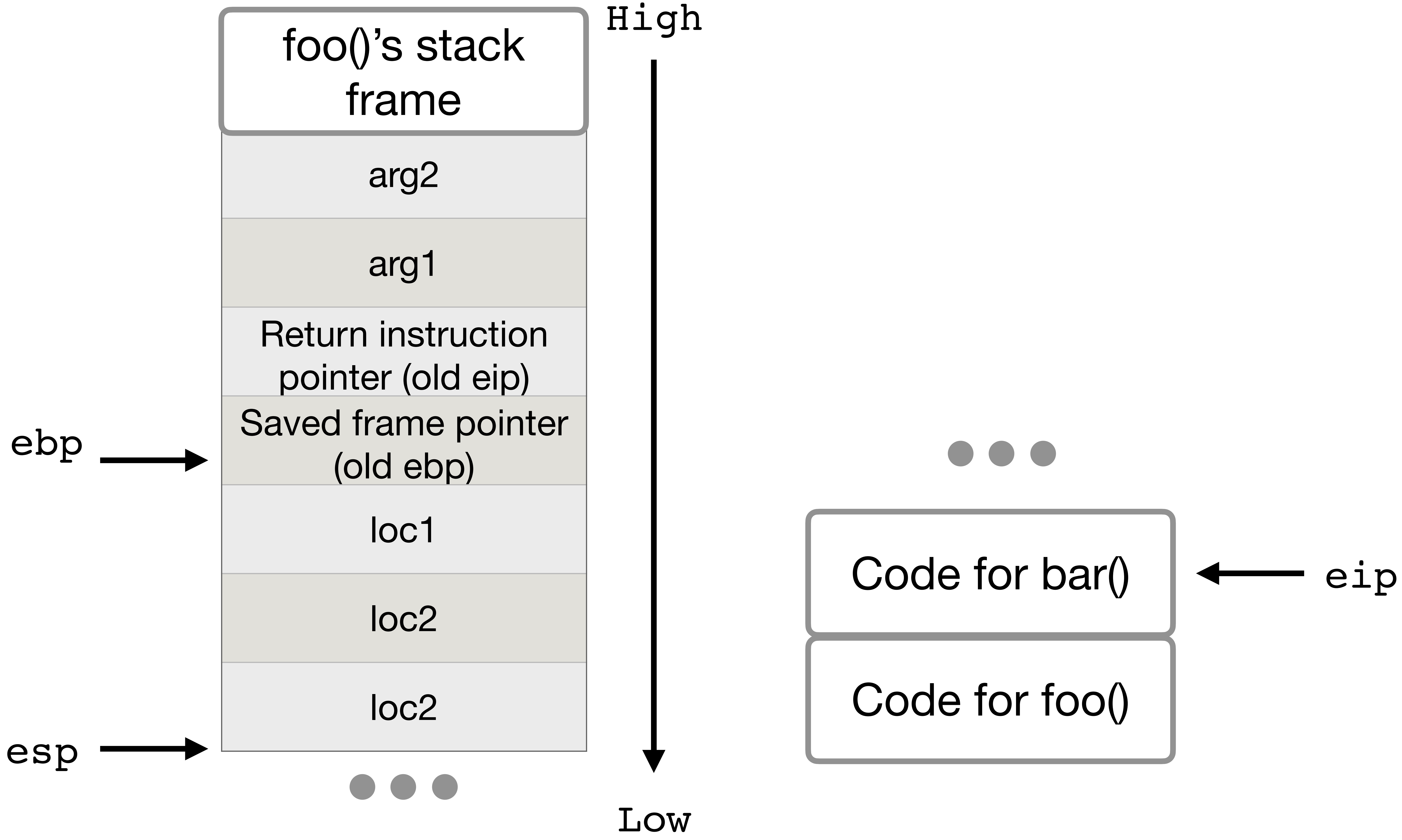
- < vs <=
- i = 0 vs i = 1

x86 Assembly Syntax

- This class follows the AT&T x86 syntax, what gdb uses
- mov instruction: the source is the first argument, and the destination is the second argument
  - mov %esp, %ebp
  - take the value in esp and put it in ebp
- Note: if you do research online, you may read Intel syntax where source and destination is reversed

Stack Frames: Return from a Function

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

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

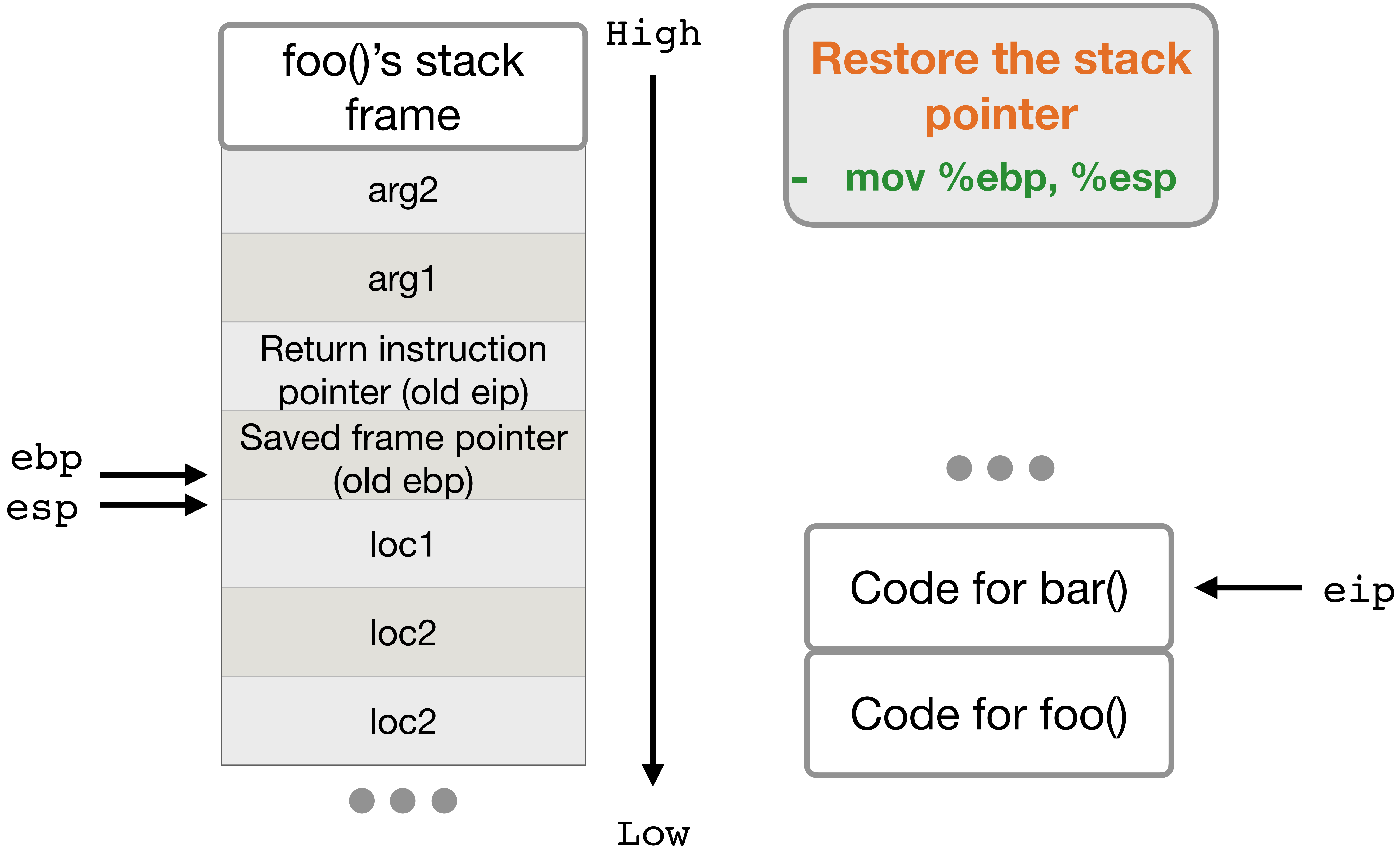


Note:

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

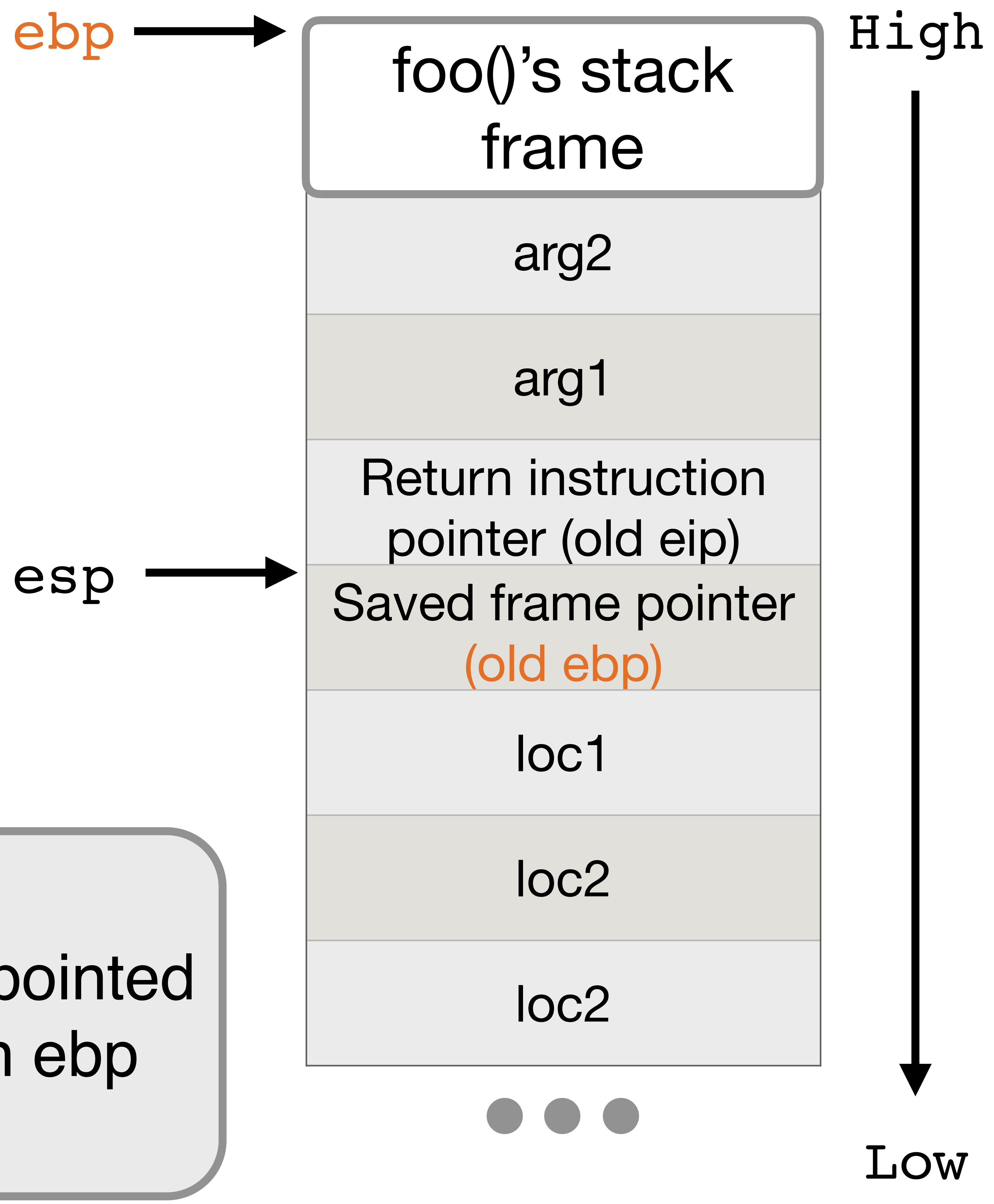
Stack Frames: Return from a Function

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

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



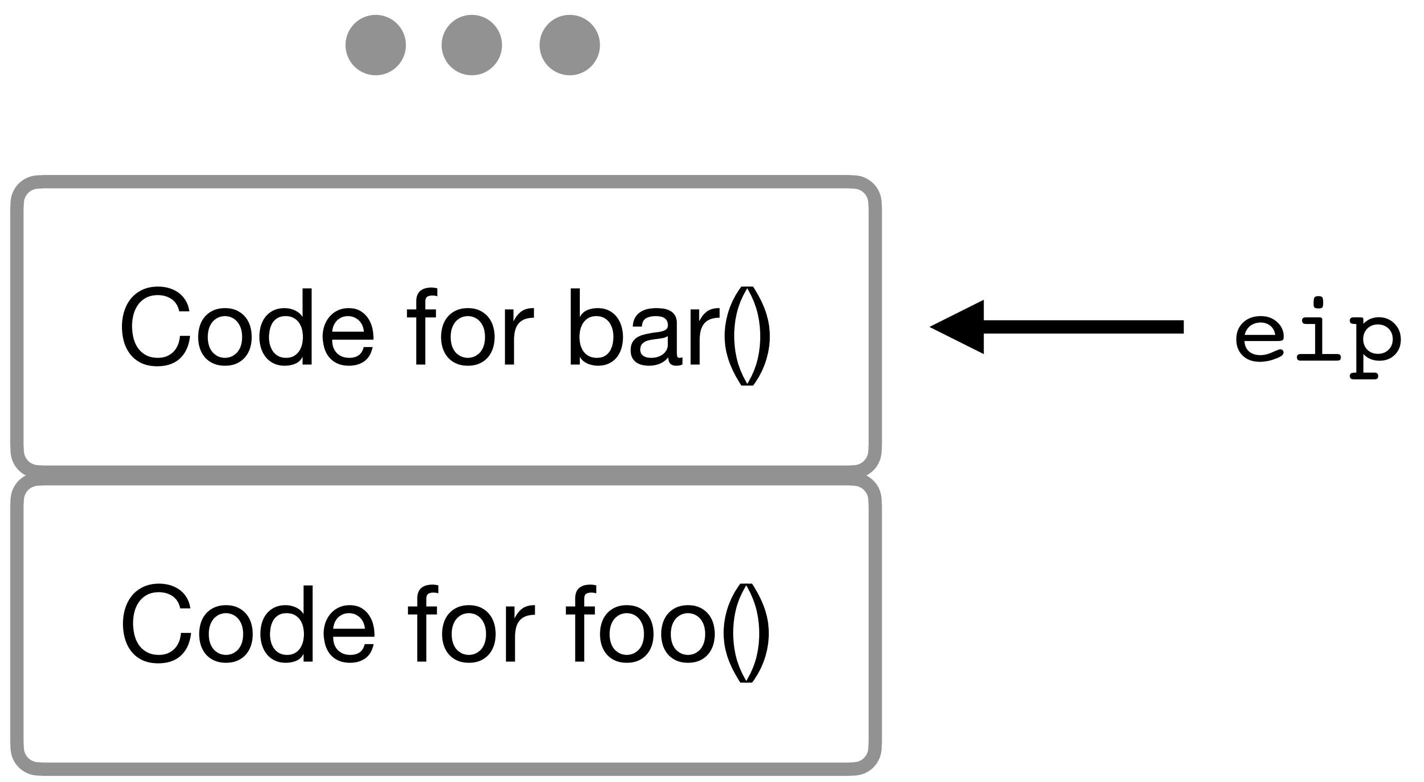
Stack Frames: Return from a Function

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

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



Restore the old ebp

- **pop, %ebp**

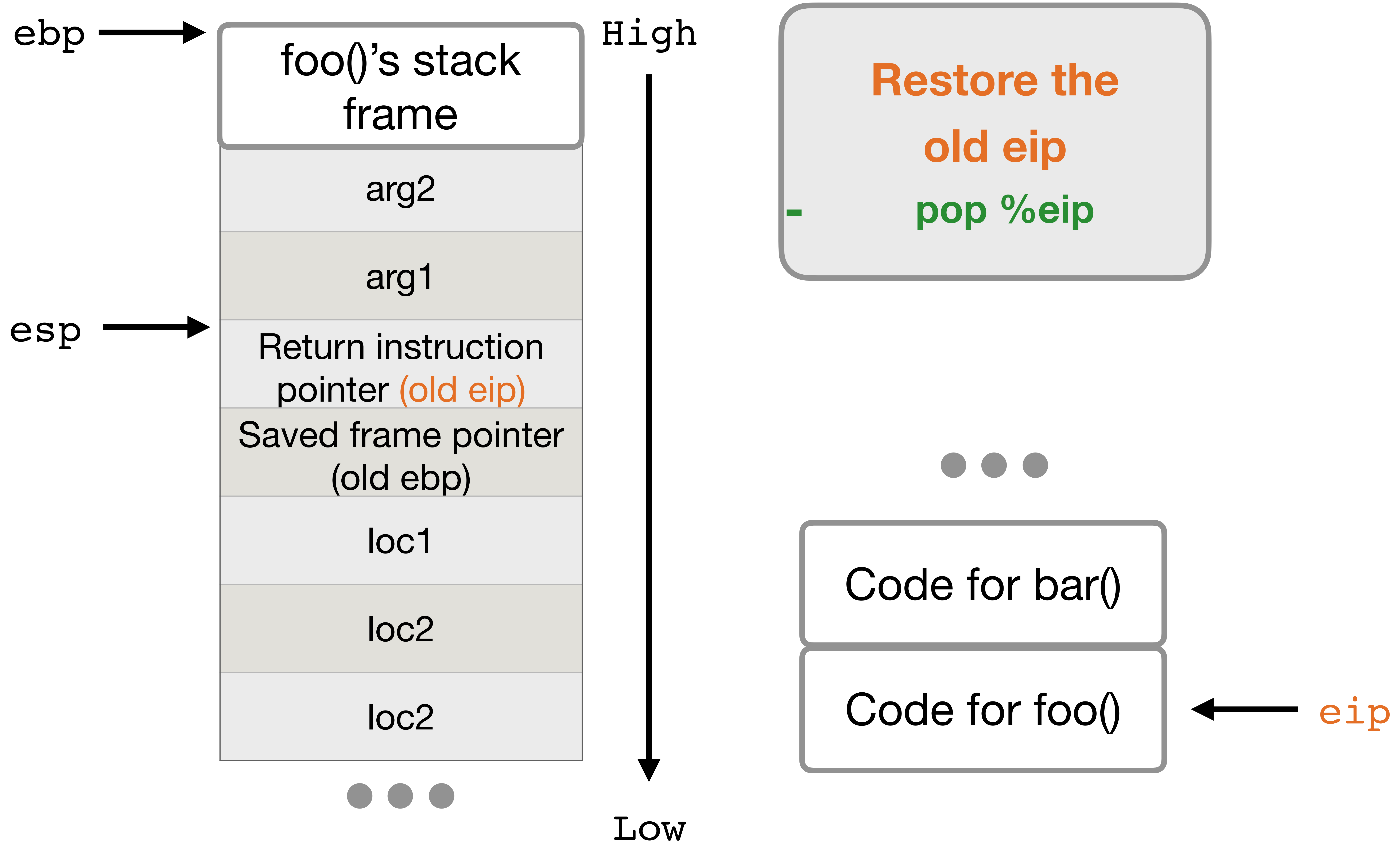


pop %ebp

- Takes the next value on stack (pointed to by esp), store it at destination ebp
- Move esp up by 4 bytes

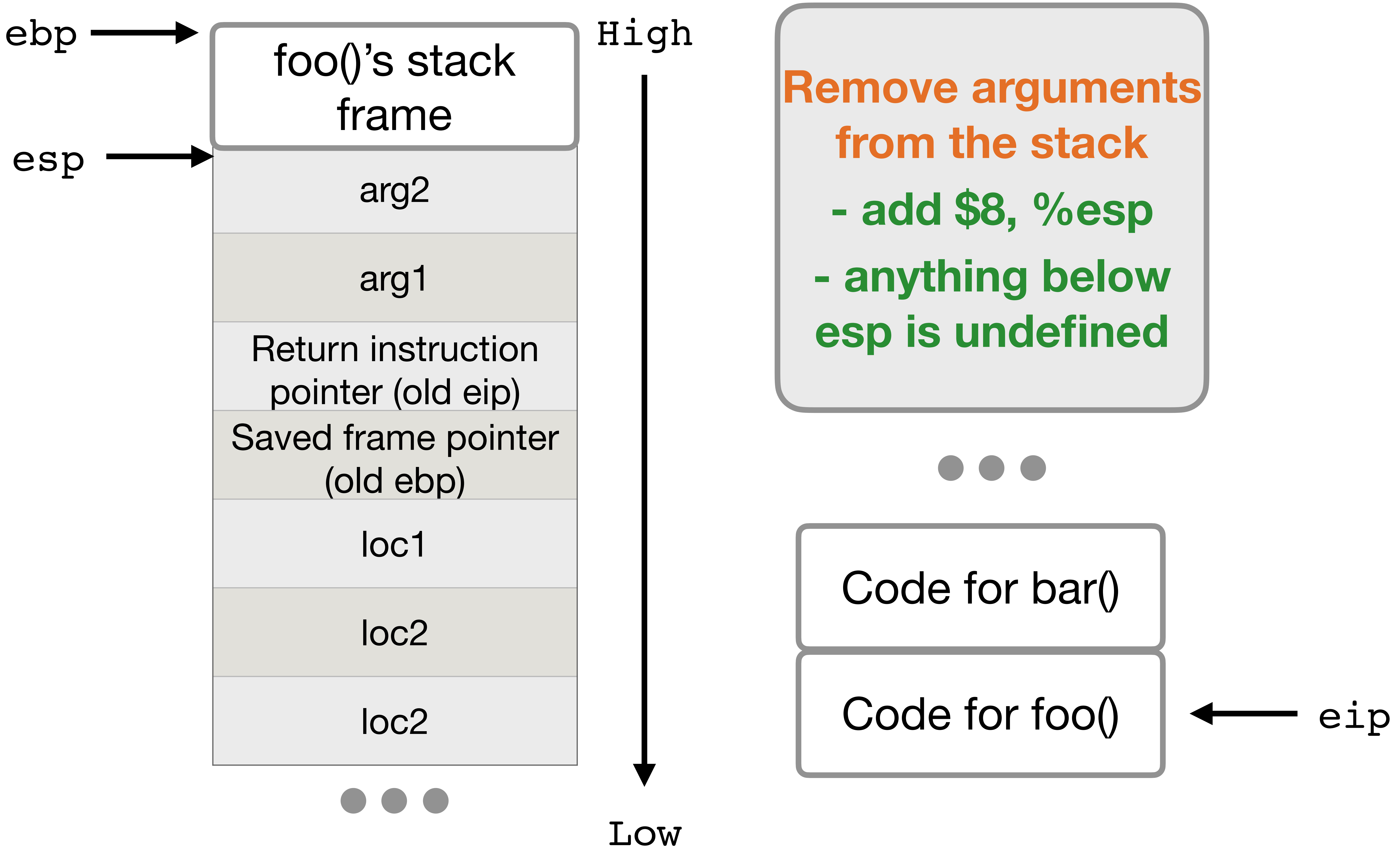
Stack Frames: Return from a Function

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

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



Stack Frames: Return from a Function

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

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



Return from a Function

In C

```
return;
```

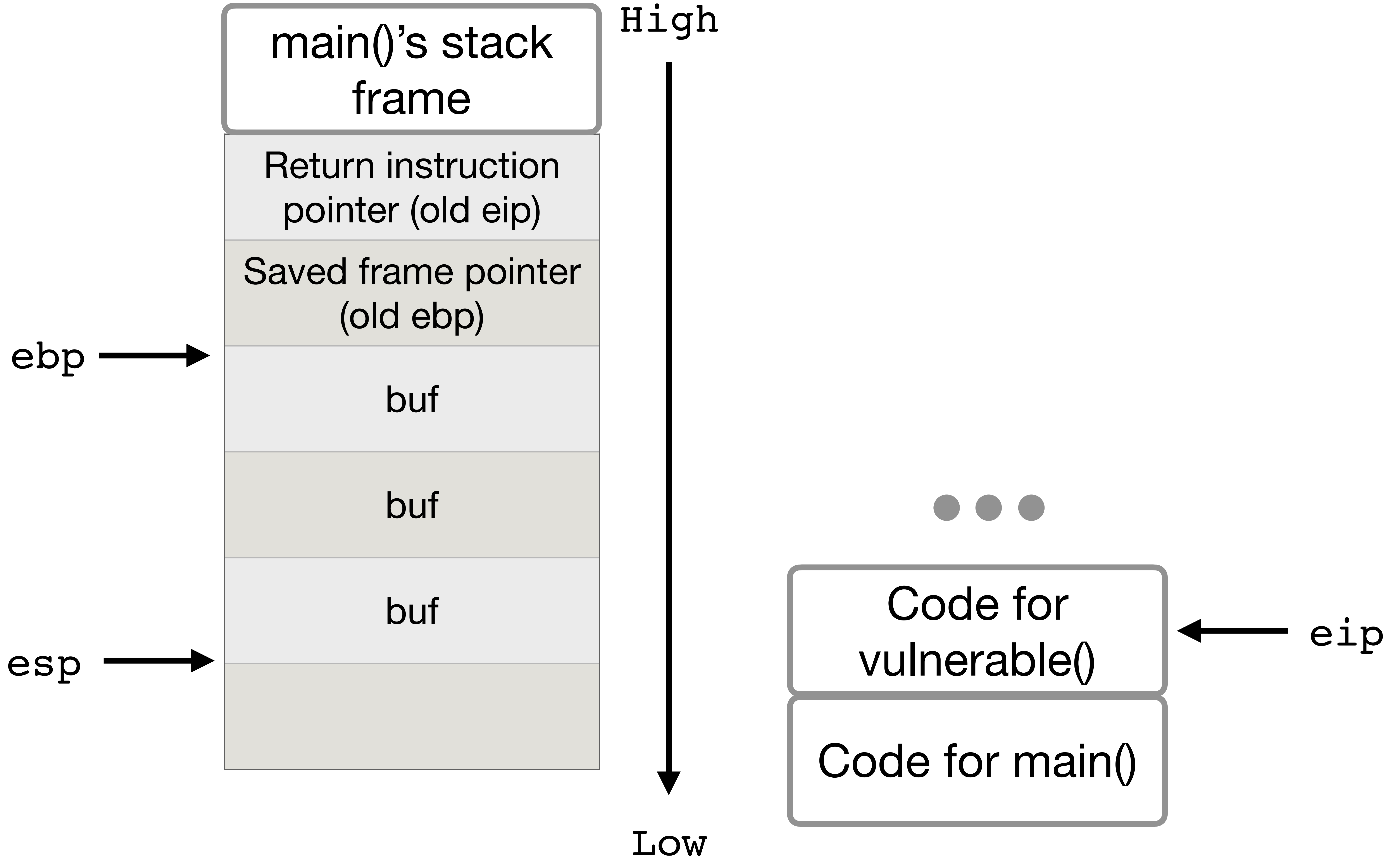
In compiled assembly

```
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)

What if we only overwrite buf by one byte?

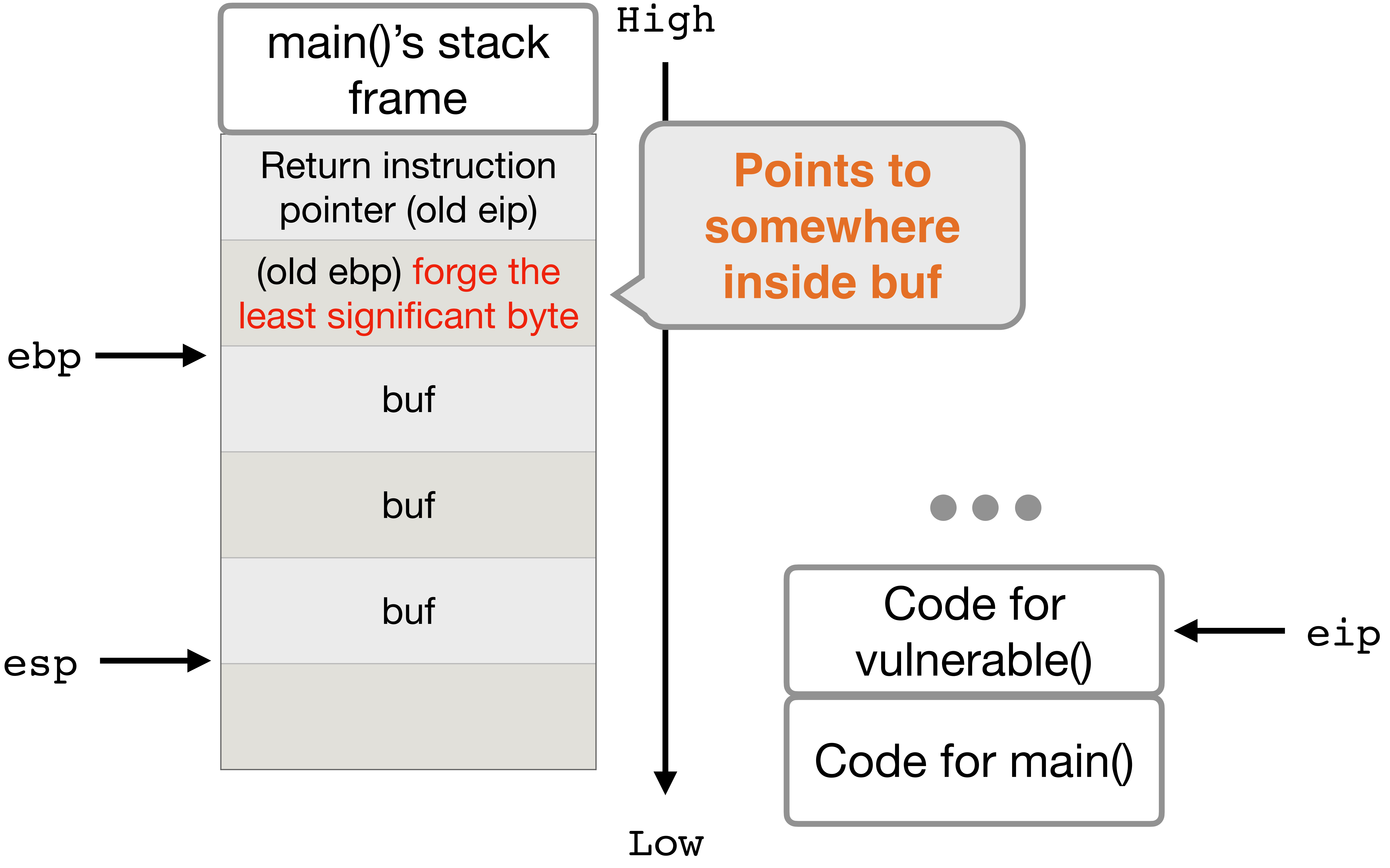
```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[12];
    gets(buf)
}
```



What if we only overwrite buf by one byte?

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[12];
    gets(buf)
}
```

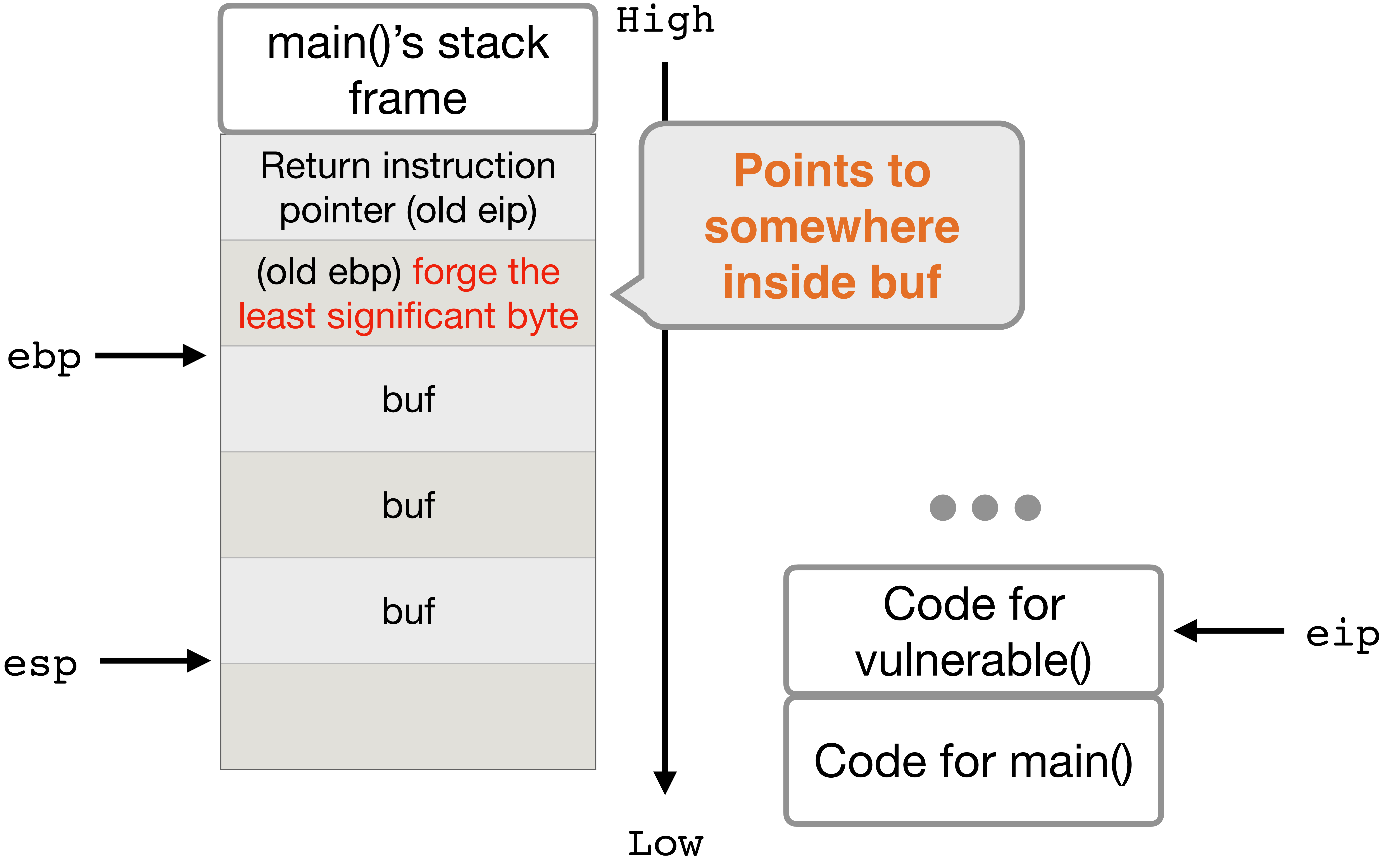
- leave
  - mov %ebp %esp
  - pop %ebp
- ret: pop %eip



What if we only overwrite buf by one byte?

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[12];
    gets(buf)
}
```

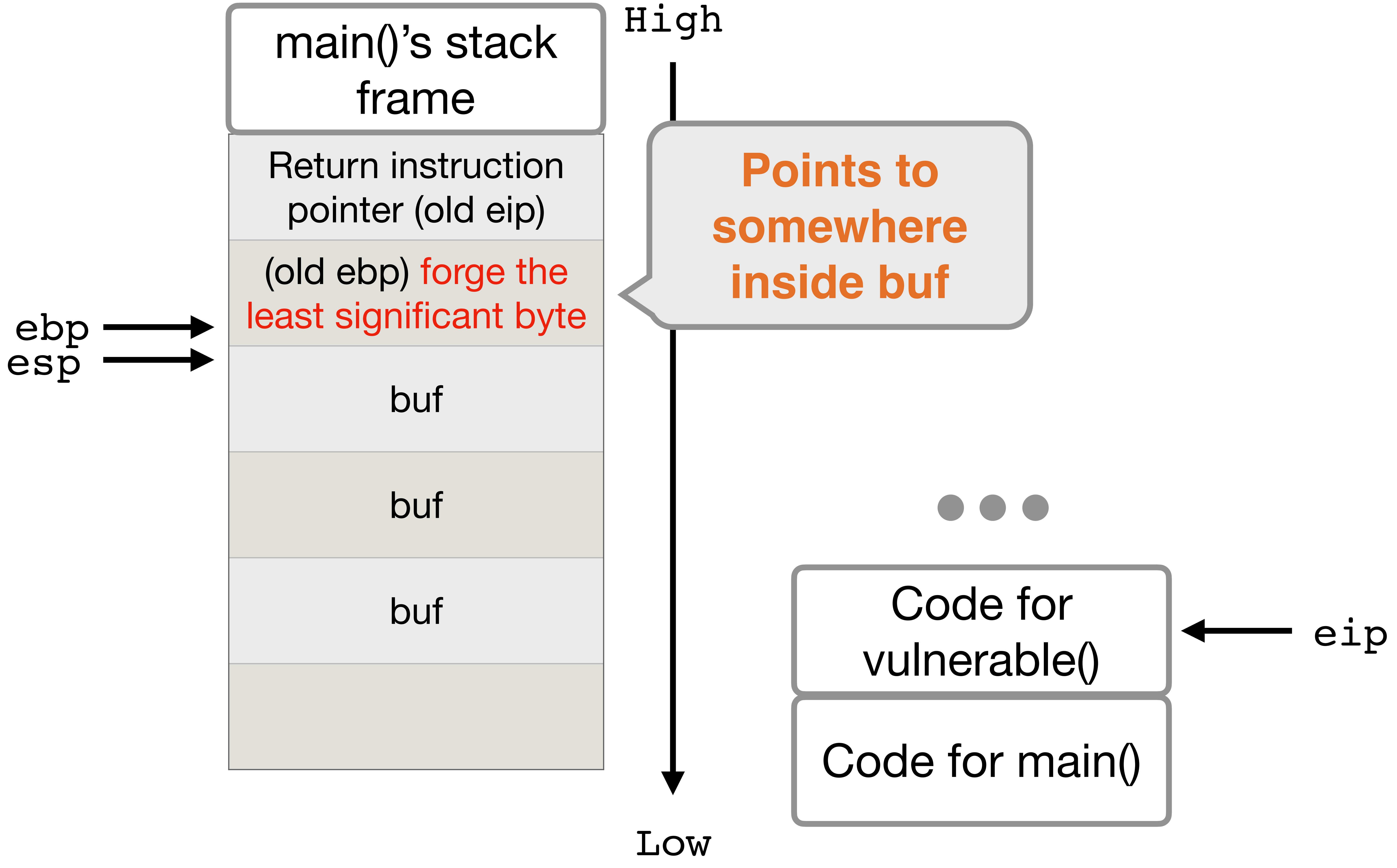
- leave
  - mov %ebp %esp
  - pop %ebp
- ret: pop %eip



Function Return

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[12];
    gets(buf)
}
```

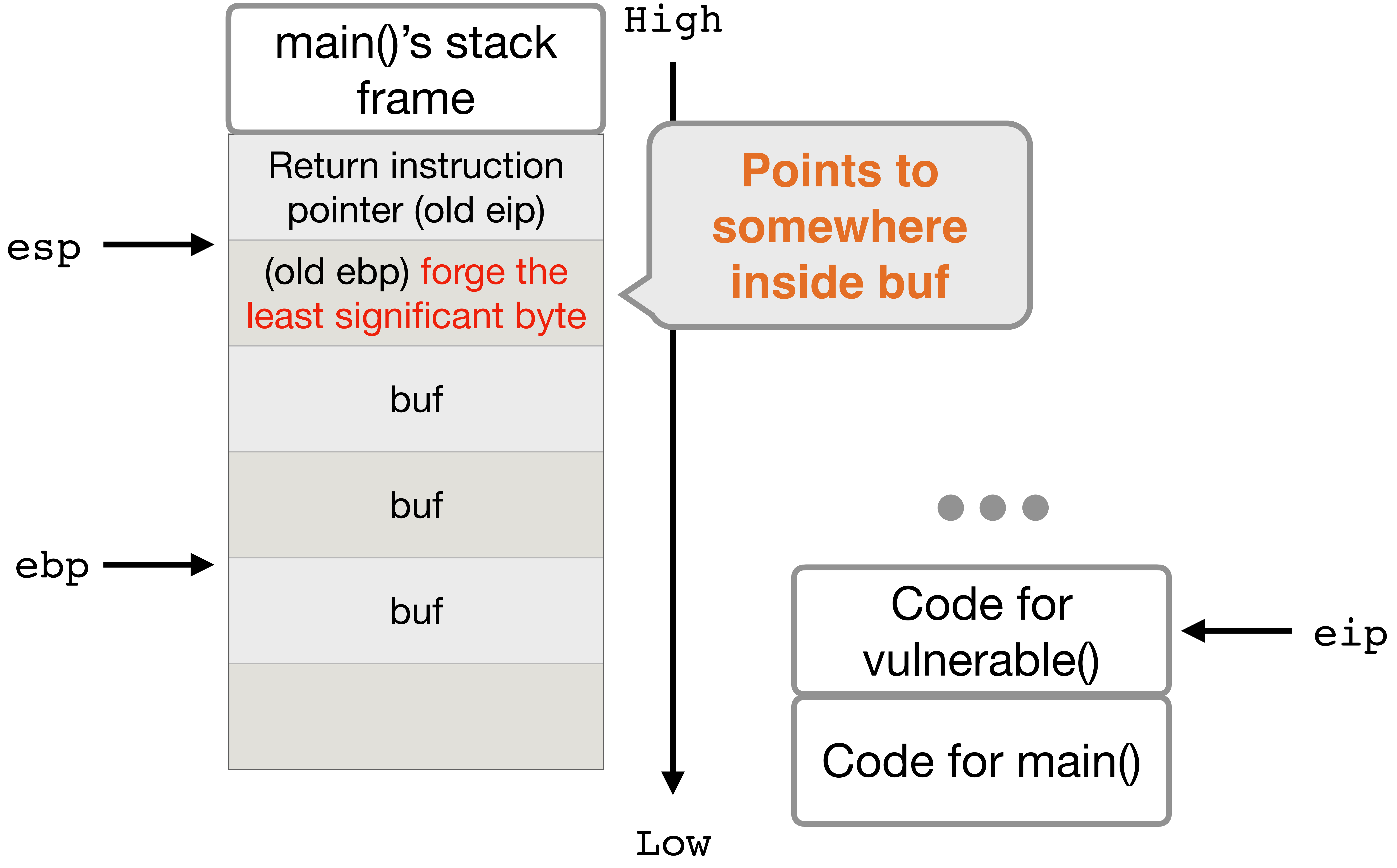
- leave
    - mov %ebp %esp
    - pop %ebp
- ret: pop %eip



Function Return

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[12];
    gets(buf)
}
```

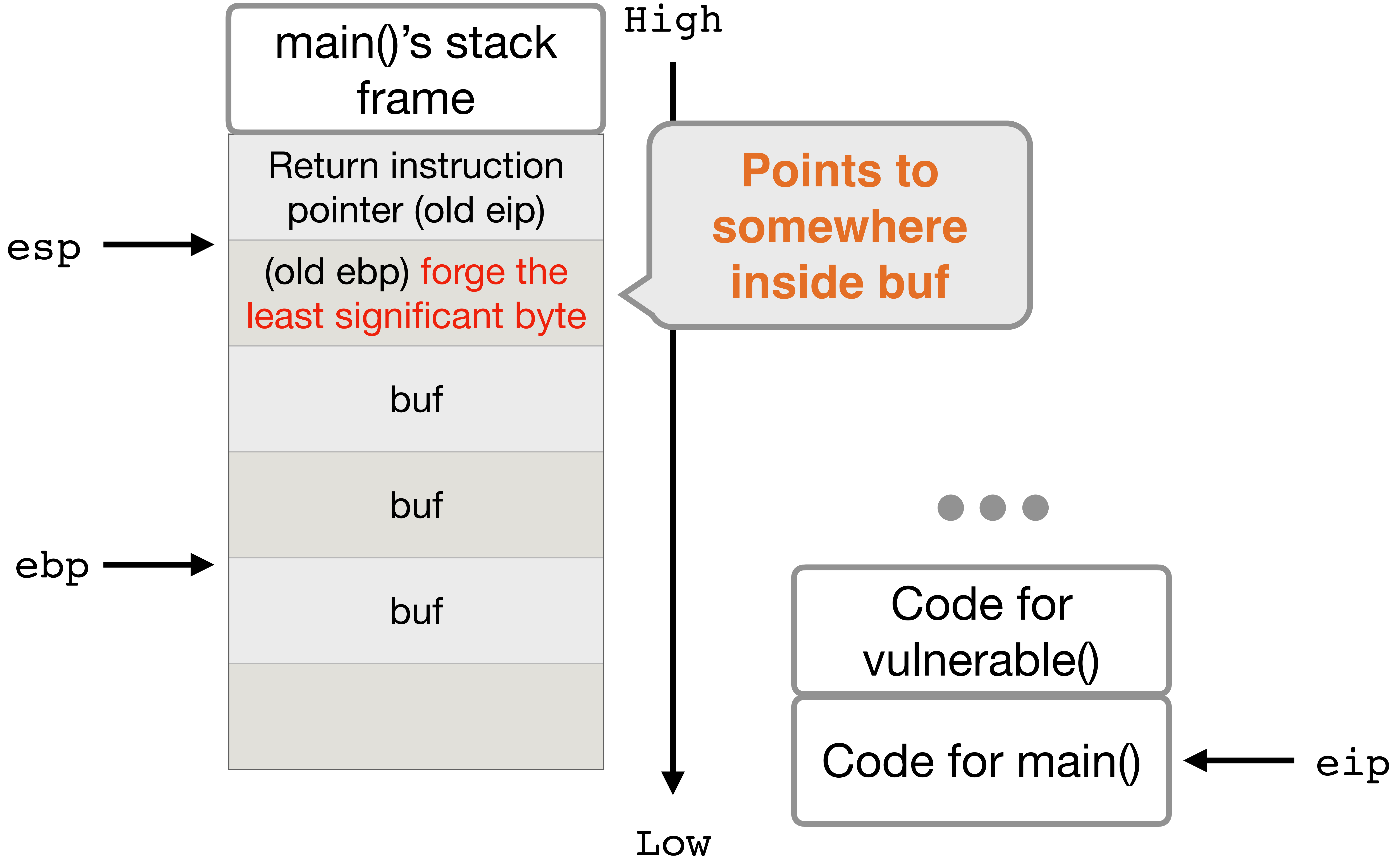
- leave
  - mov %ebp %esp
  - pop %ebp
- ret: pop %eip



Function Return

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[12];
    gets(buf)
}
```

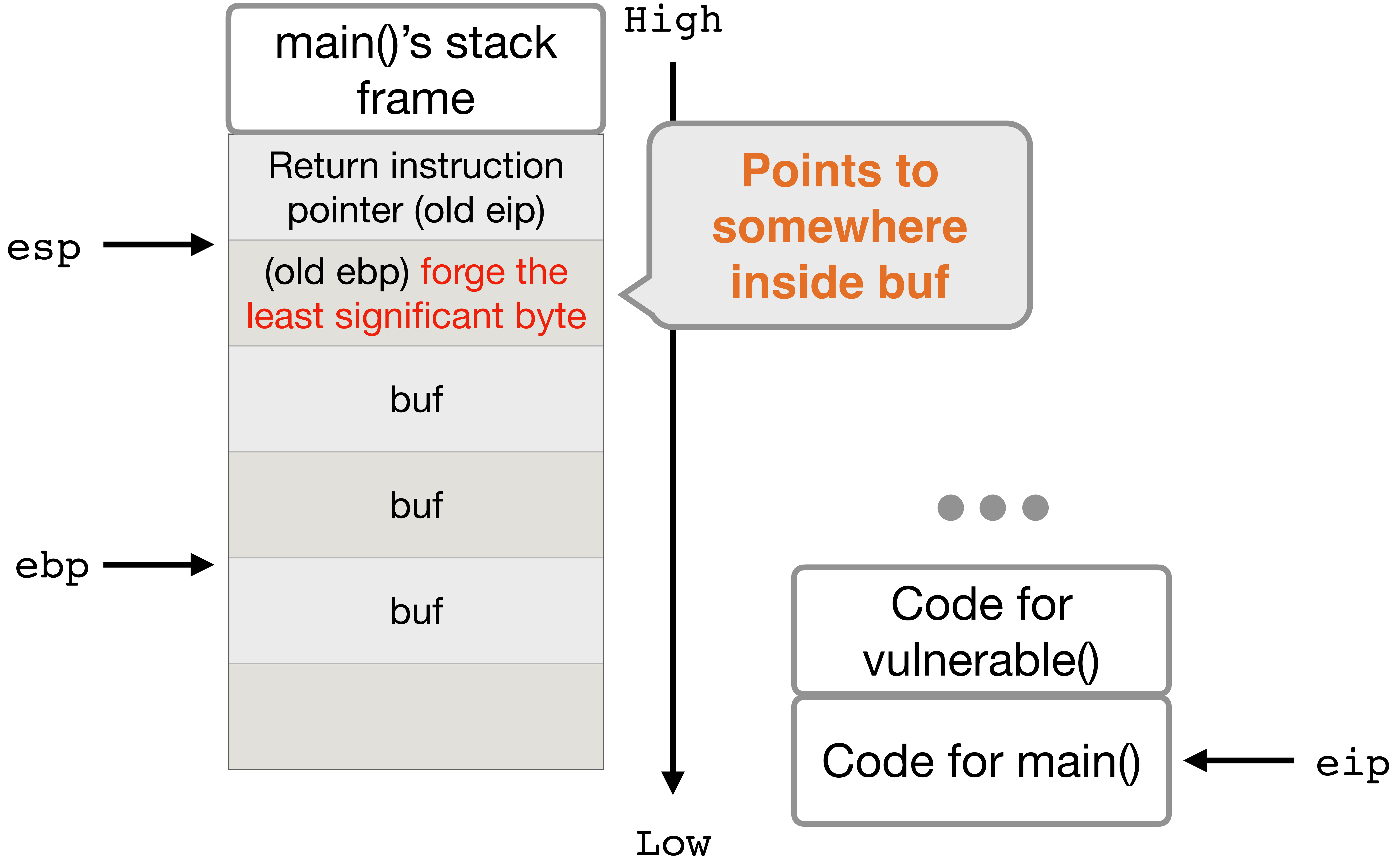
- leave
  - mov %ebp %esp
  - pop %ebp
- ret: pop %eip



A Second Function Return

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[12];
    gets(buf)
}
```

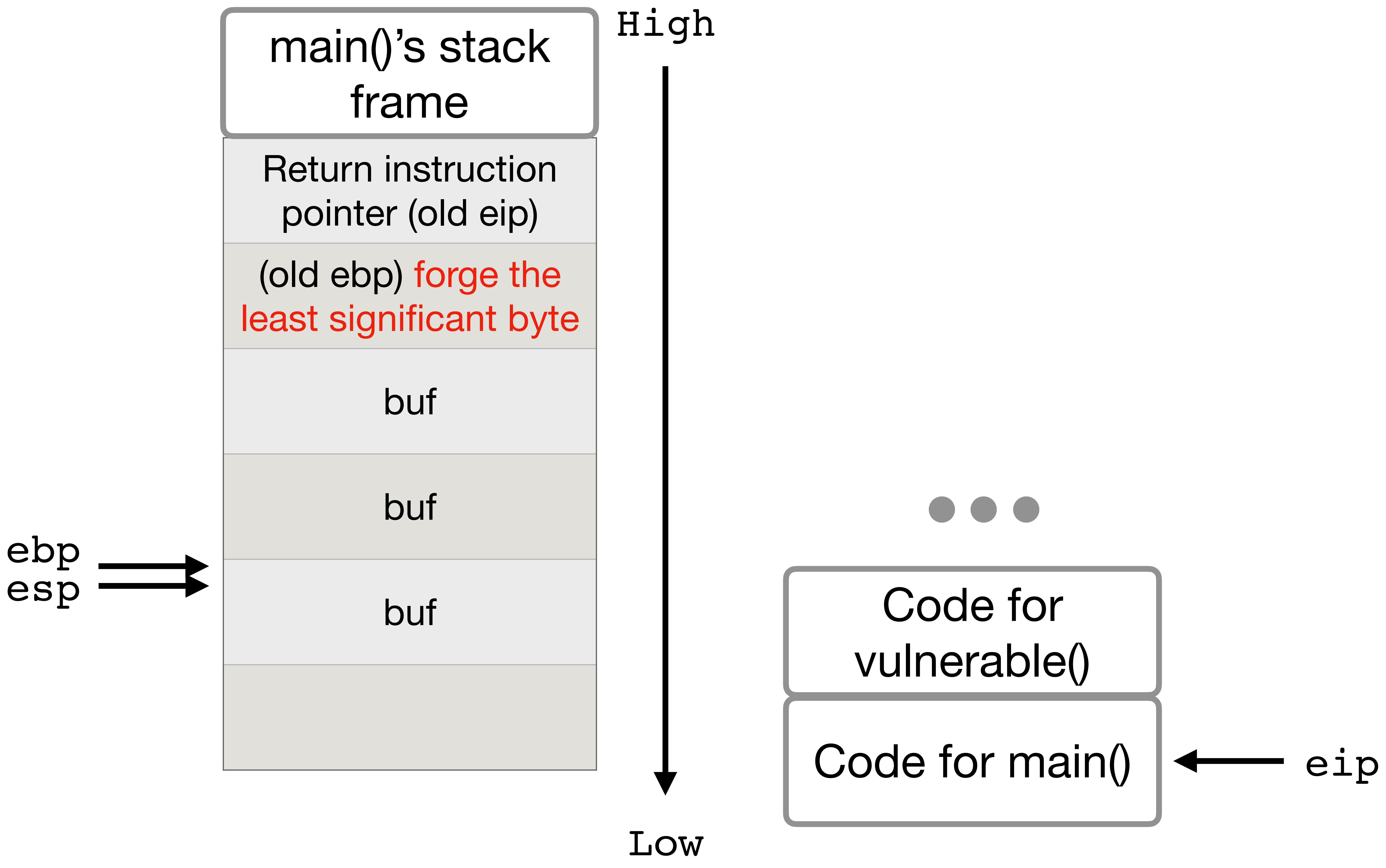
- leave
  - mov %ebp %esp
  - pop %ebp
- ret: pop %eip



A Second Function Return

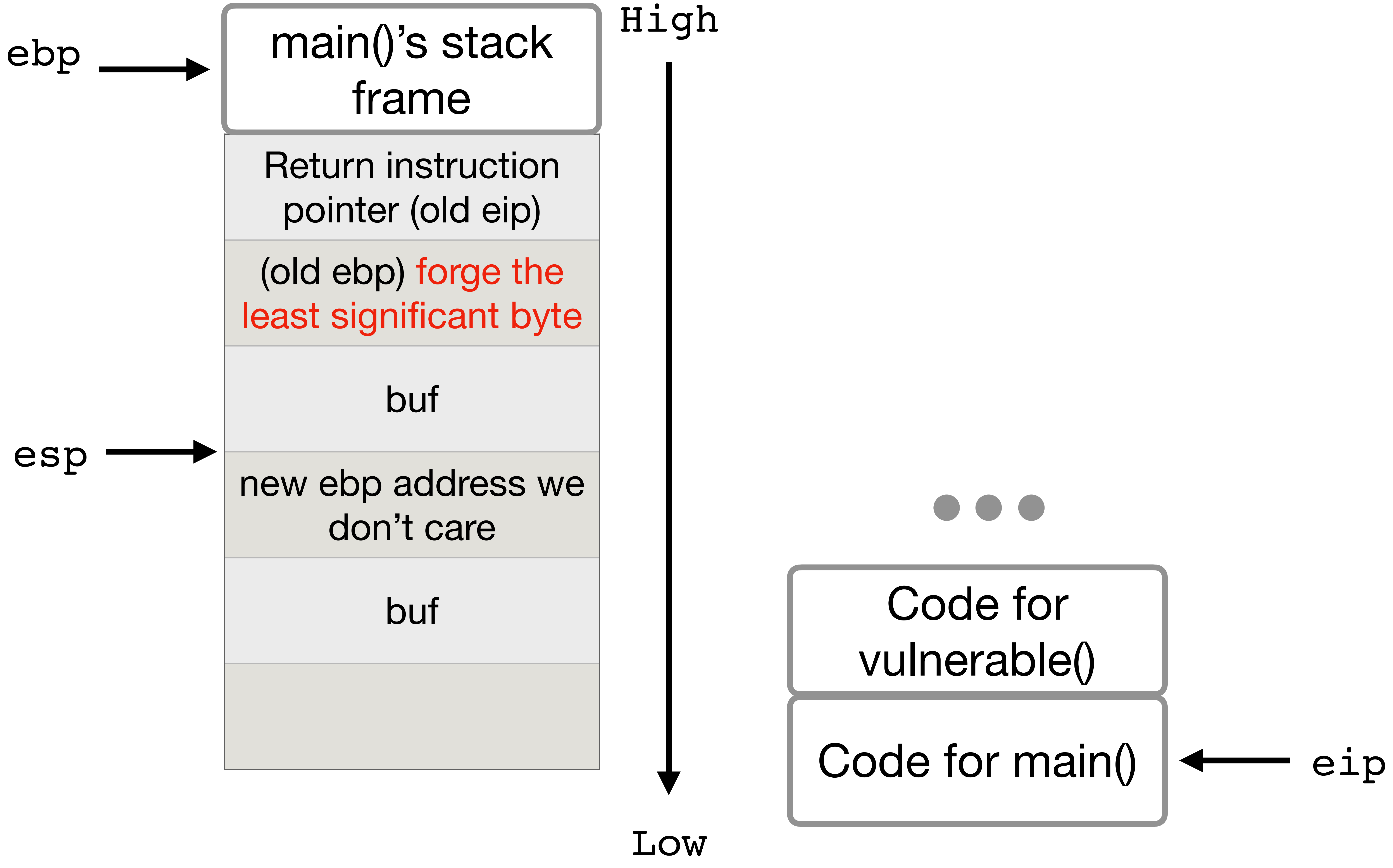
```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[12];
    gets(buf)
}
```

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



A Second Function Return

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[12];
    gets(buf)
}
```

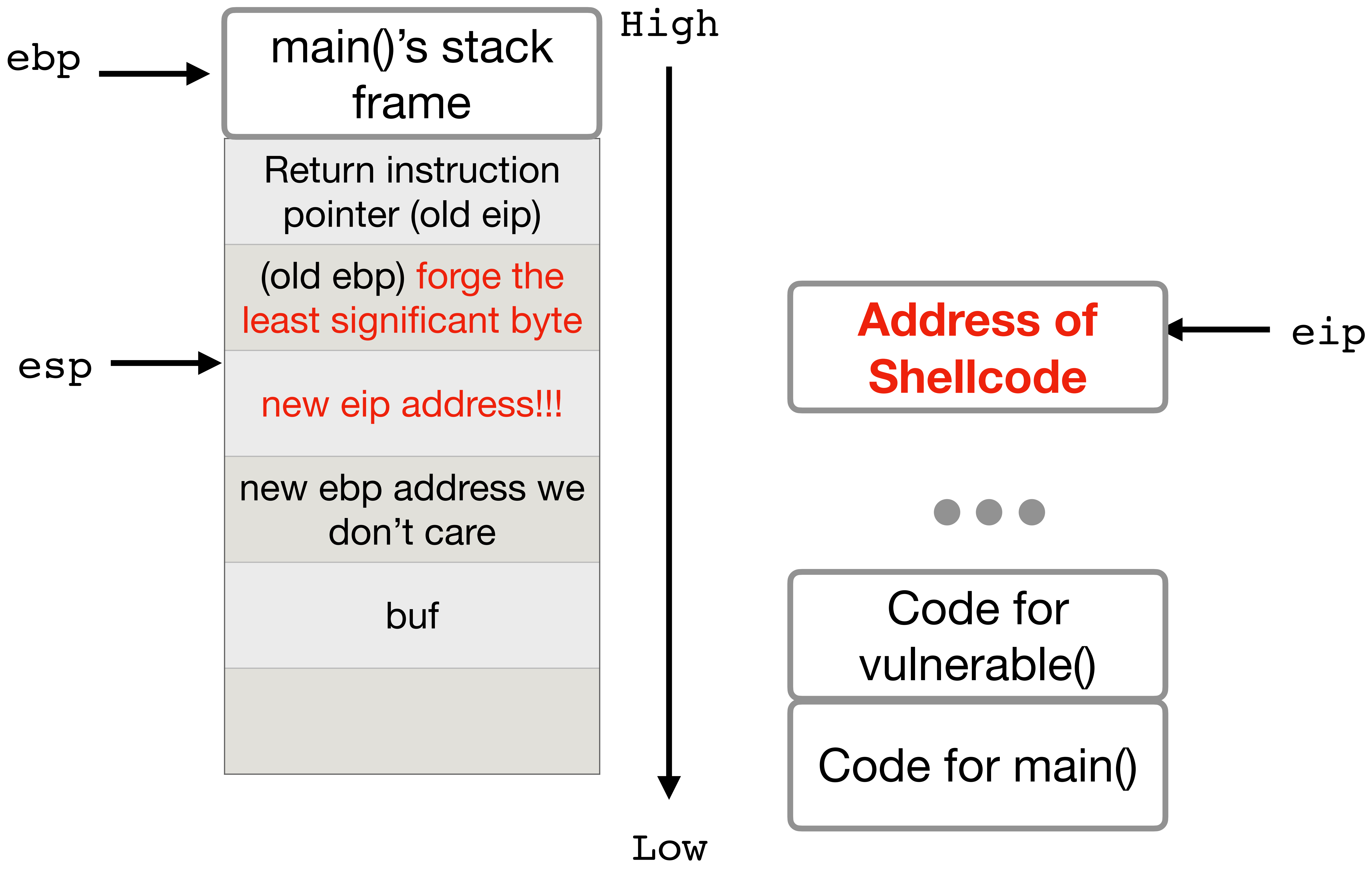


- leave
  - mov %ebp %esp
  - pop %ebp
- ret: pop %eip

A Second Function Return

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[12];
    gets(buf)
}
```

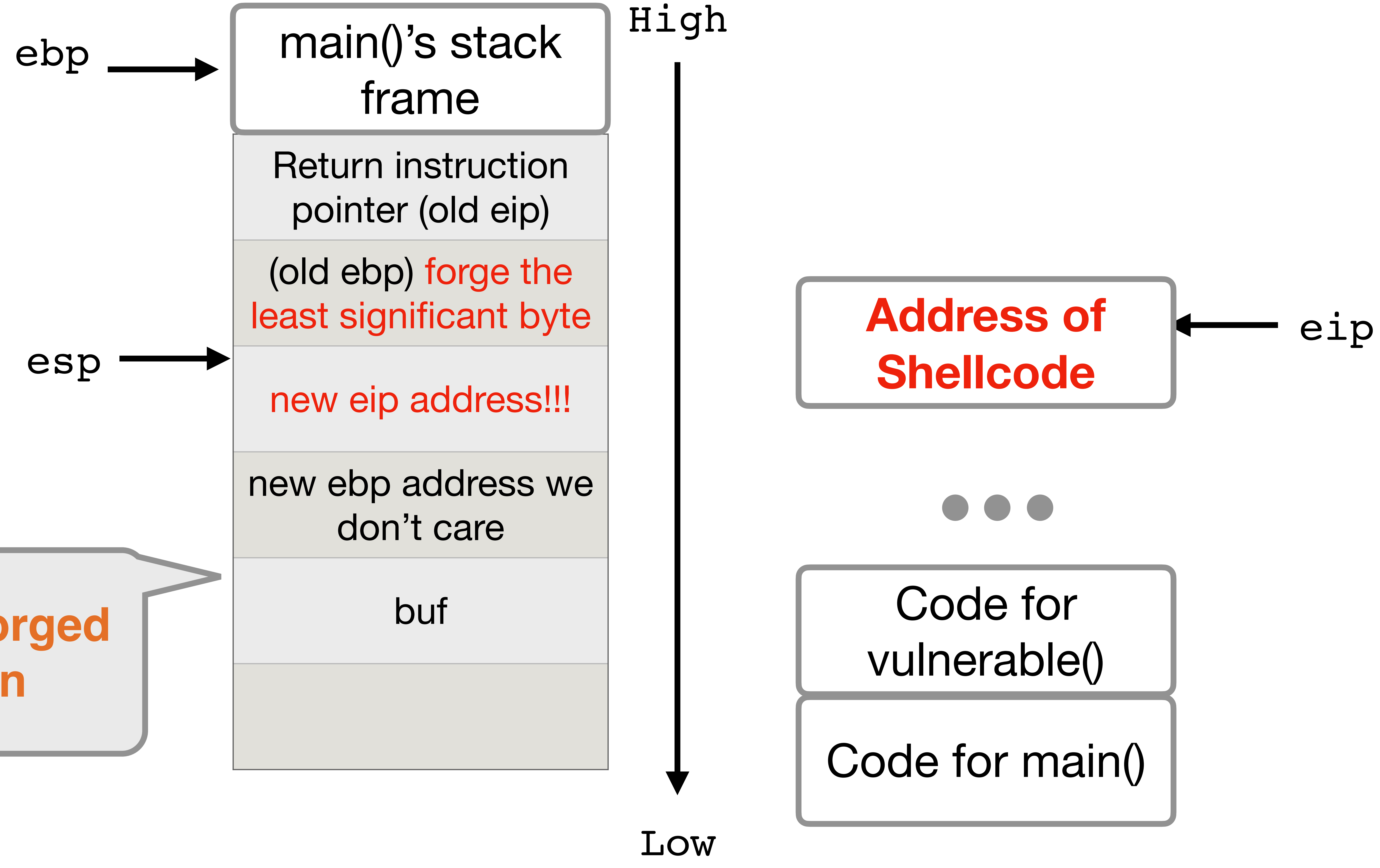
- leave
  - mov %ebp %esp
  - pop %ebp
- ret: pop %eip



A Second Function Return

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[12];
    gets(buf)
}
```

4 bytes offset from the forged frame pointer location



Other Memory Safety Vulnerabilities

- Use after free
- Heap overflow
- …

2023 CWE Top 25 Most Dangerous Software Weaknesses

Top 25 Home | Share via: | View in table format | Key Insights | Methodology

1 Out-of-bounds Write
CWE-787 | CVEs in KEV: 70 | Rank Last Year: 1

2 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-79 | CVEs in KEV: 4 | Rank Last Year: 2

3 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-89 | CVEs in KEV: 6 | Rank Last Year: 3

4 Use After Free
CWE-416 | CVEs in KEV: 44 | Rank Last Year: 7 (up 3) ▲

5 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
CWE-78 | CVEs in KEV: 23 | Rank Last Year: 6 (up 1) ▲

6 Improper Input Validation
CWE-20 | CVEs in KEV: 35 | Rank Last Year: 4 (down 2) ▼

7 Out-of-bounds Read
CWE-125 | CVEs in KEV: 2 | Rank Last Year: 5 (down 2) ▼

8 Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-22 | CVEs in KEV: 16 | Rank Last Year: 8

https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html