CMSC414 Computer and Network Security

Mitigating Memory Safety Vulnerabilities

Yizheng Chen | University of Maryland
surrealyz.github.io

Feb 6, 2024

Announcements

- Project 1
  - Gitlab
  - Will add a makefile for part 0
- New TA

Agenda

- Memory-safe languages
- Writing memory-safe code
- Building secure software
- Exploit mitigations
    - Non-executable pages
    - Stack canaries
    - Pointer authentication
    - Address space layout randomization (ASLR)
- Combining mitigations

Memory Safe Language

- Programming languages that include a combination of compile-time and runtime checks that prevent memory errors from occurring, e.g., check bounds, prevent undefined memory access
  - By design, memory-safe languages are not vulnerable to memory safety vulnerabilities
  - Using a memory-safe language is the **only** way to stop 100% of memory safety vulnerabilities
- Examples: Java, Python, C#, Go, Rust
  - Most languages besides C, C++, and Objective C

Why Not Use Memory-Safe Languages?

- Performance
- Comparison of memory allocation performance
  - C and C++ (not memory safe): malloc usually runs in (amortized) constant-time
  - Java (memory safe): The garbage collector may need to run at any arbitrary point in time, adding a 10–100 ms delay as it cleans up memory

The Myth of Performance

- For most applications, the performance difference from using a memory-safe language is insignificant
  - Possible exceptions: Operating systems, high performance games, some embedded systems
- C’s improved performance is not a direct result of its security issues
  - Today, safe alternatives have comparable performance (e.g. Go and Rust)
  - Secure C code (with bounds checking) ends up running as quickly as code in a memory-safe language anyway
  - Have both security and performance

The Real Reason: Legacy Code

- Huge existing code bases are written in C, and building on existing code is easier than starting from scratch
  - If old code is written in {language}, new code will be written in {language}!

Writing Memory Safe Code

- Defensive programming: Always add checks in your code just in case
  - Example: Always check a pointer is not null before dereferencing it, even if you're sure the pointer is going to be valid
  - Relies on programmer discipline
- Use safe libraries
  - Use functions that check bounds
  - Example: Use **`fgets`** instead of **`gets`**
  - Example: Use **`strncpy`** or **`strlcpy`** instead of **`strcpy`**
  - Example: Use **`snprintf`** instead of **`sprintf`**
  - Relies on programmer discipline or tools that check your program

Building Secure Software

- Code Review
  - Hiring someone to look over your code for memory safety errors. Effective but expensive.
- Penetration testing (“pen-testing”)
  - Pay someone to break into your system
- Run-time checks
  - Automatic bounds-checking. Overhead.
  - Crash if the check fails
- Bug-finding tools
  - Static analyzers: heuristic based, e.g., some user inputs affect memory allocation over some program execution paths
  - Fuzz testing: testing with random inputs

Agenda

- Memory-safe languages
- Writing me
- Building se

make it harder for attackers to exploit common vulnerabilities

- Exploit mitigations
  - Non-executable pages
  - Stack canaries
  - Pointer authentication
  - Address space layout randomization (ASLR)
- Combining mitigations

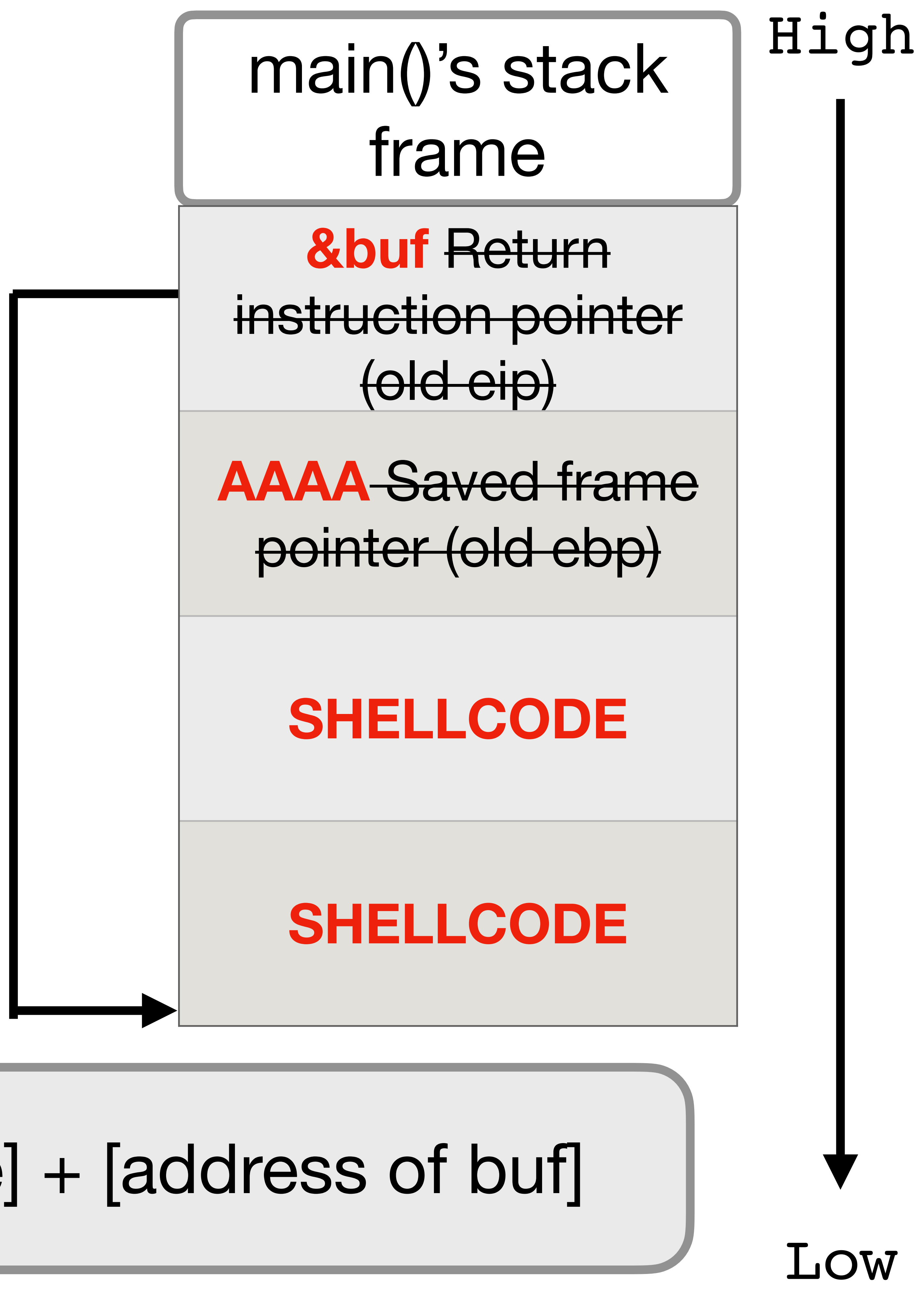
Exploit Mitigations

- Compile and run code with code hardening defenses
  - Compiler and runtime defenses
- Make common exploits harder
- Cause exploits to crash instead of succeeding
- Not foolproof

Recall: if shell code is only 8 bytes

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

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

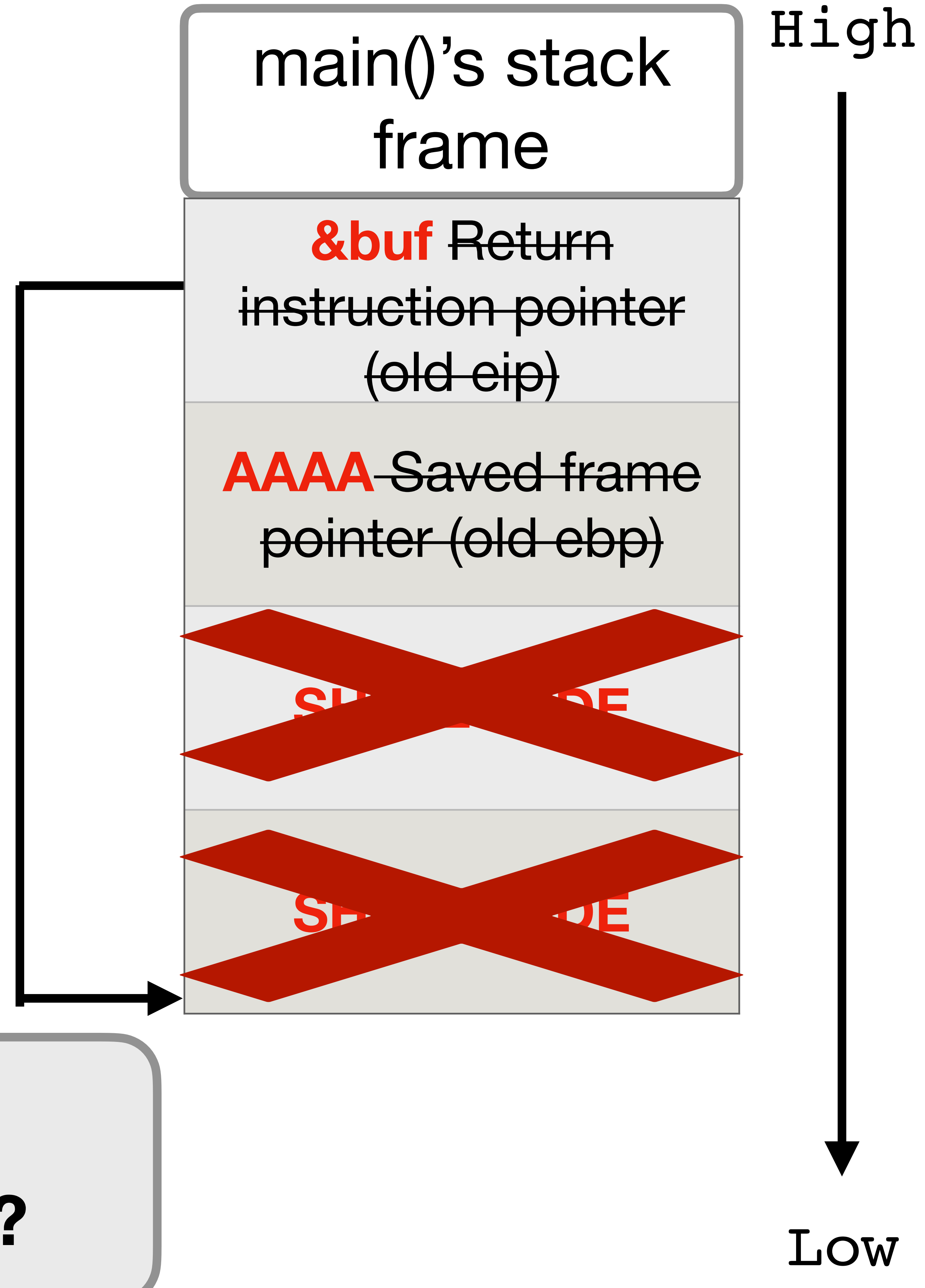


[8 bytes of SHELLCODE] + [4 bytes of garbage] + [address of buf]

Non-Executable Pages

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

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



What if shell code cannot be executed?

What if nothing on the stack can be executed?

Non-Executable Pages

- Idea: Most programs don't need memory that is both written to and executed, so make portions of memory either executable or writable but not both
  - Stack, heap, and static data: Writable but not executable
  - Code: Executable but not writable
- Also known as
  - W^X (write XOR execute)
  - DEP (Data Execution Prevention, name used by Windows)
  - No-execute bit

Subverting Non-Executable Pages

- Issue: Non-executable pages doesn't prevent an attacker from leveraging existing code in memory as part of the exploit
- Most programs have many functions loaded into memory that can be used for malicious behavior
  - **Return-to-libc**: An exploit technique that overwrites the RIP to jump to a functions in the standard C library (libc) or a common operating system function
  - **Return-oriented programming (ROP)**: Constructing custom shellcode using pieces of code that already exist in memory

How to subvert non-executable pages?

How to subvert non-executable pages?

Idea: return to existing code in memory

Return into libc: a real call

Goal: system("rm -rf /")

NAME top

```
system - execute a shell command
```

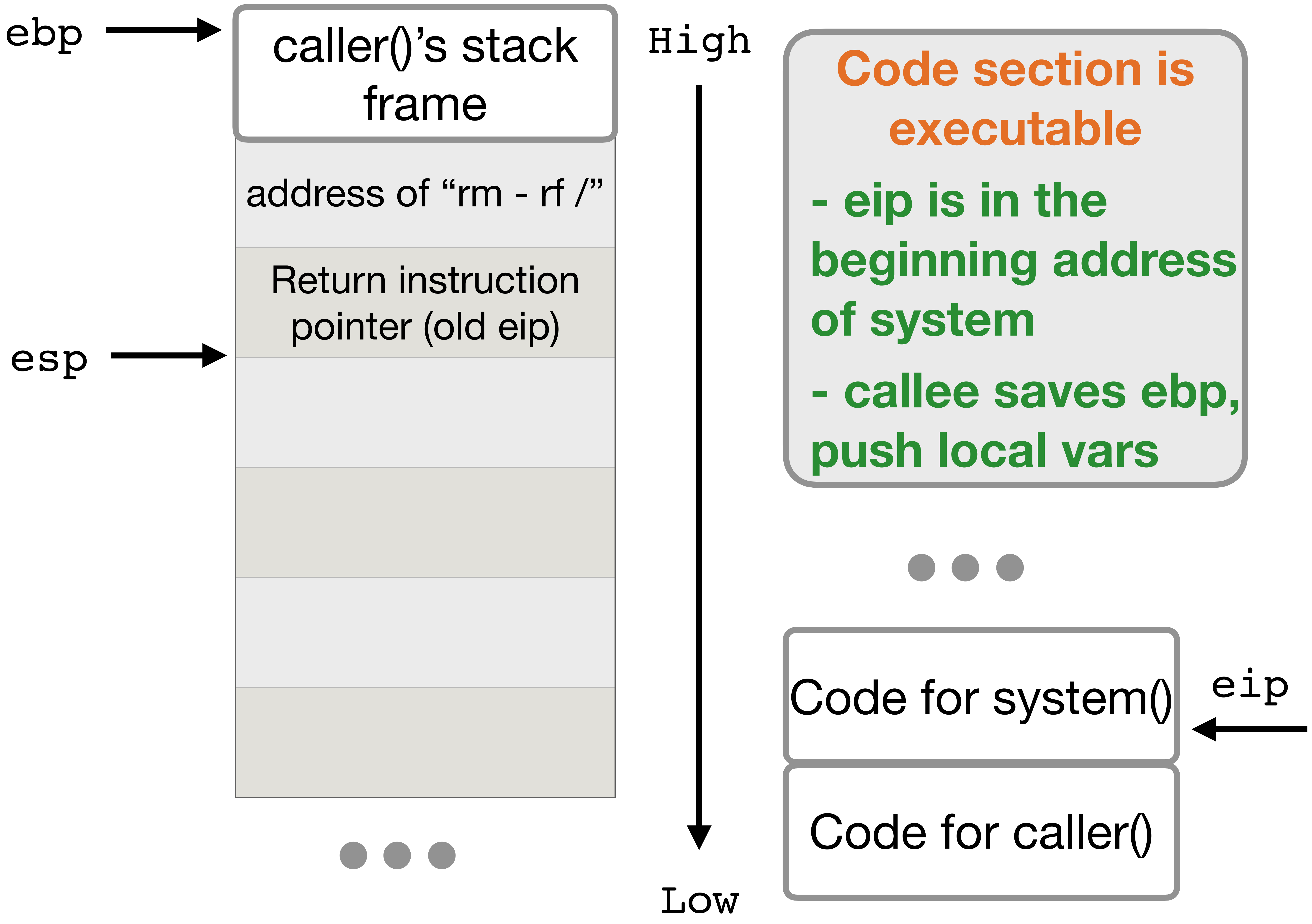
LIBRARY top

```
Standard C library (libc, -lc)
```

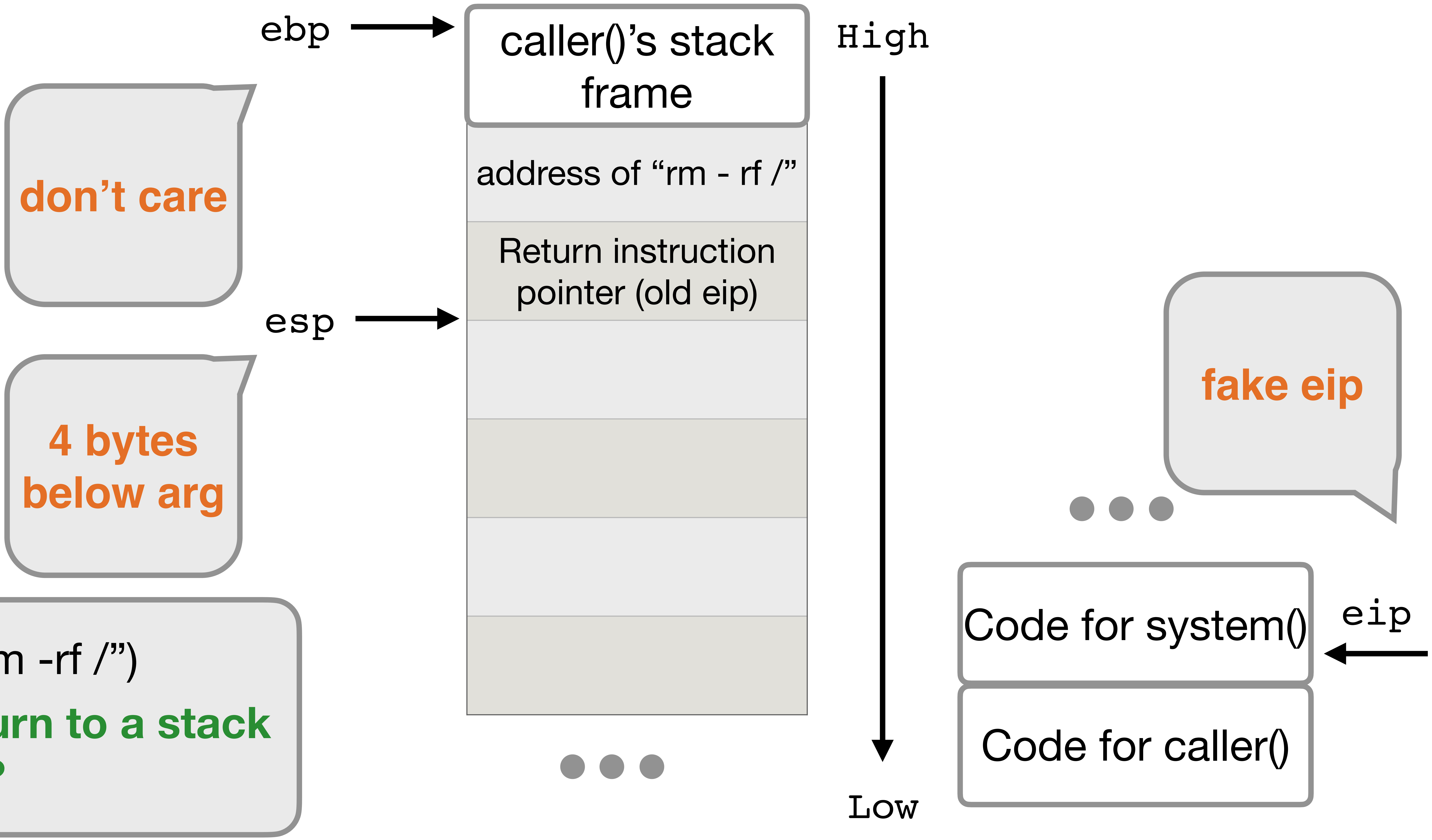
SYNOPSIS top

```
#include <stdlib.h>

int system(const char *command);
```



Return into libc: goal of a fake call



Goal: system("rm -rf /")

Question: can we return to a stack like this?

Return from a Function

In C

```c
return;
```

In compiled assembly

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

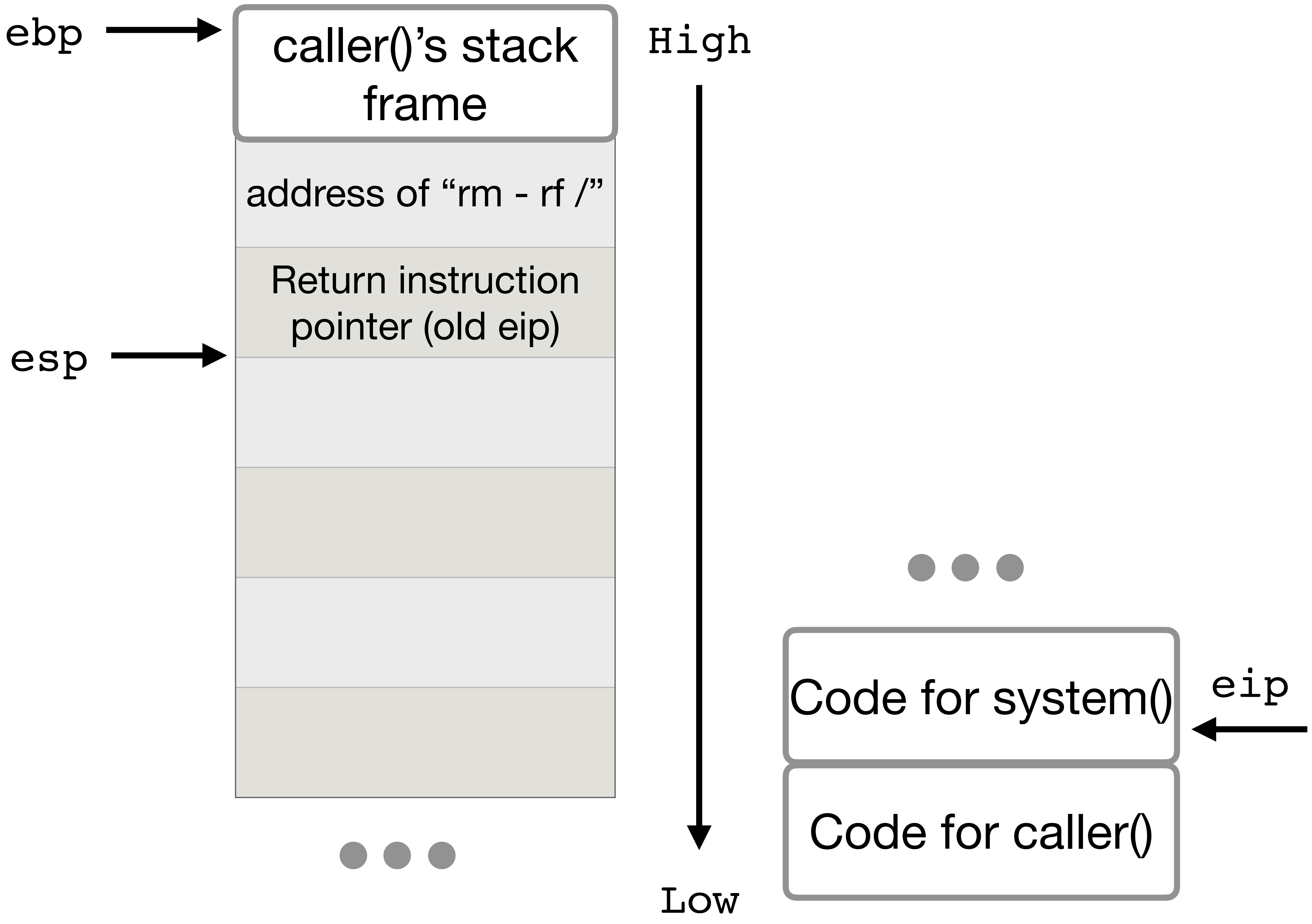
- Leave: leave the stack frame of the callee
  - restore stack pointer (mov %ebp %esp)
  - restore the base pointer (pop %ebp)
- Ret: restore the instruction pointer (pop %eip)

Return into libc: goal of a fake call

Goal: system(“rm -rf /”)

after executing leave ret

- **fake eip**
- **Don’t care what the ebp is**
- **esp is 4 bytes below arg**



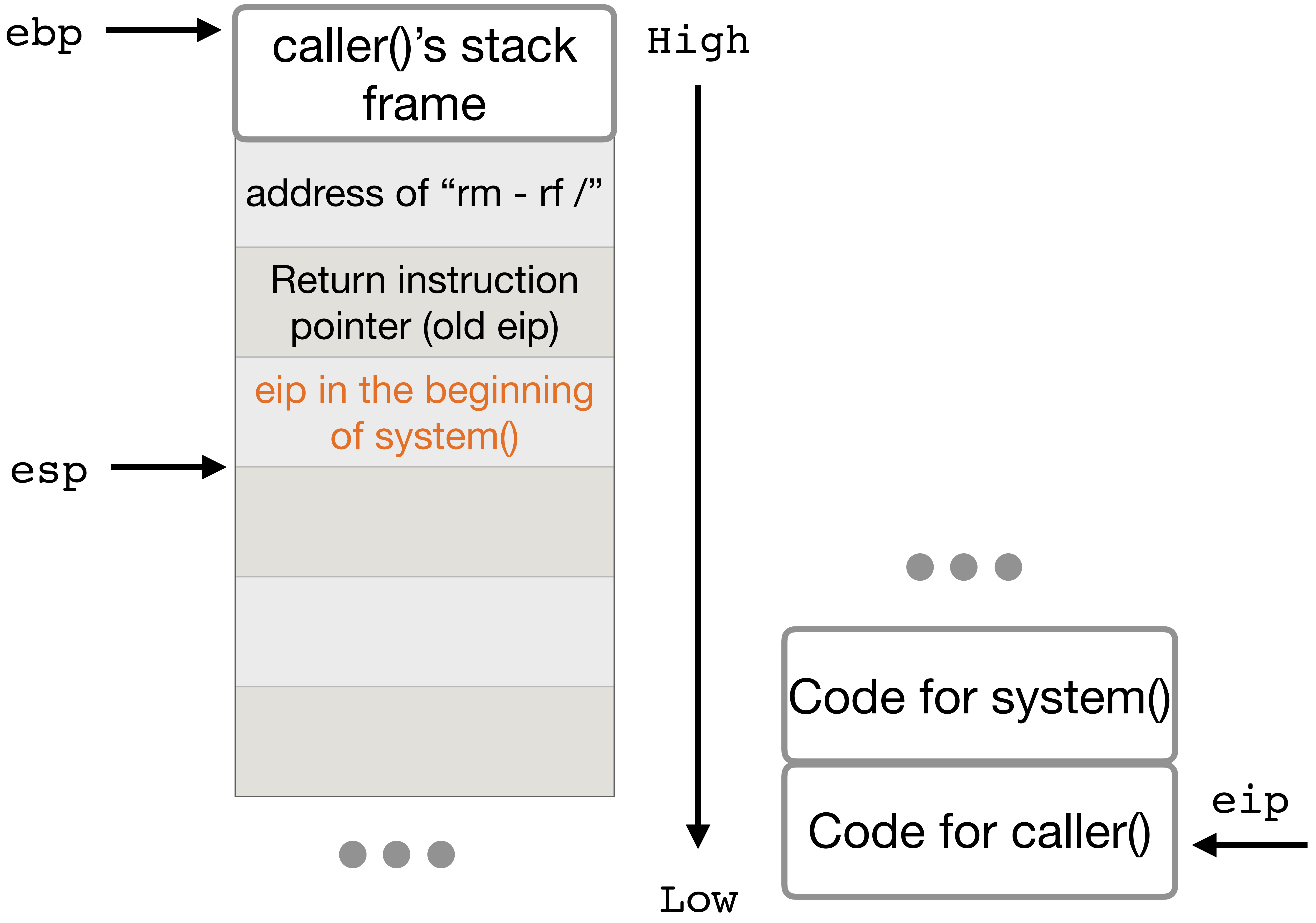
- leave
  - mov %ebp %esp
  - pop %ebp
- ret: pop %eip

Return into libc: goal of a fake call

Goal: system("rm -rf /")

after executing leave ret

- **fake eip**
- **Don't care what the ebp is**
- **esp is 4 bytes below arg**



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

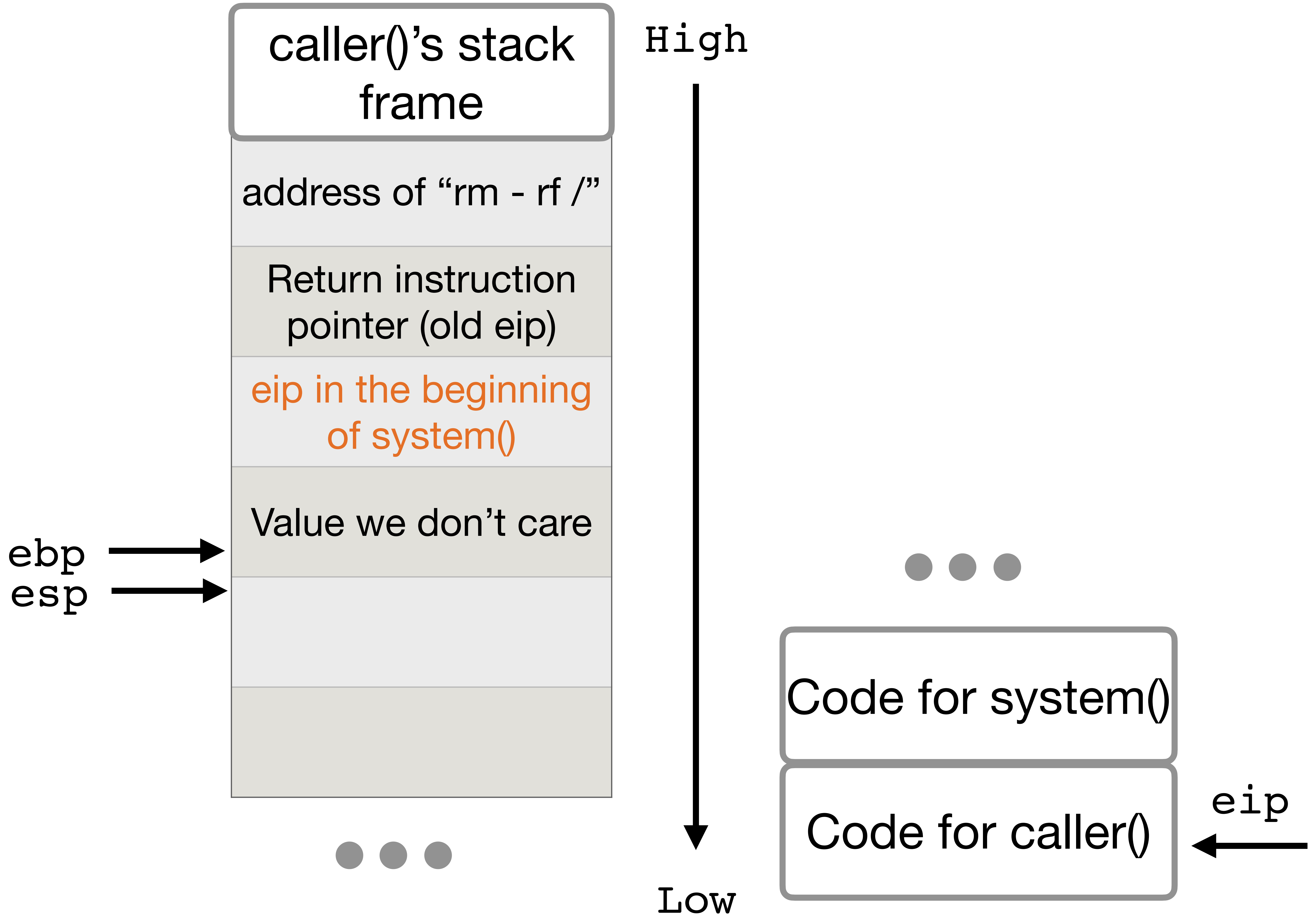
Return into libc: goal of a fake call

Goal: system("rm -rf /")

after executing leave ret

- fake eip
- Don't care what the ebp is
- esp is 4 bytes below arg

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



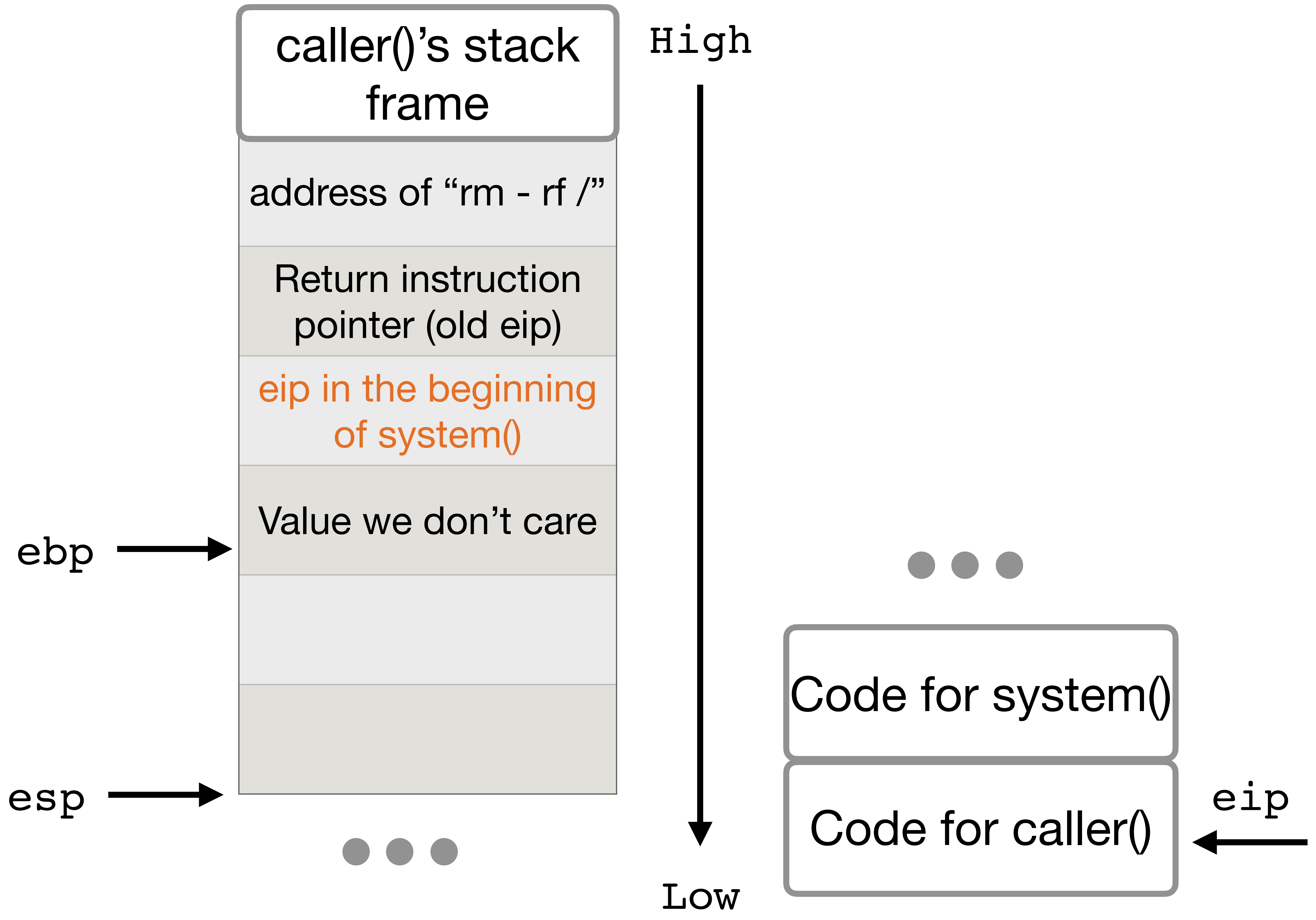
Return into libc: goal of a fake call

Goal: system("rm -rf /")

after executing leave ret

- fake eip
- Don't care what the ebp is
- esp is 4 bytes below arg

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



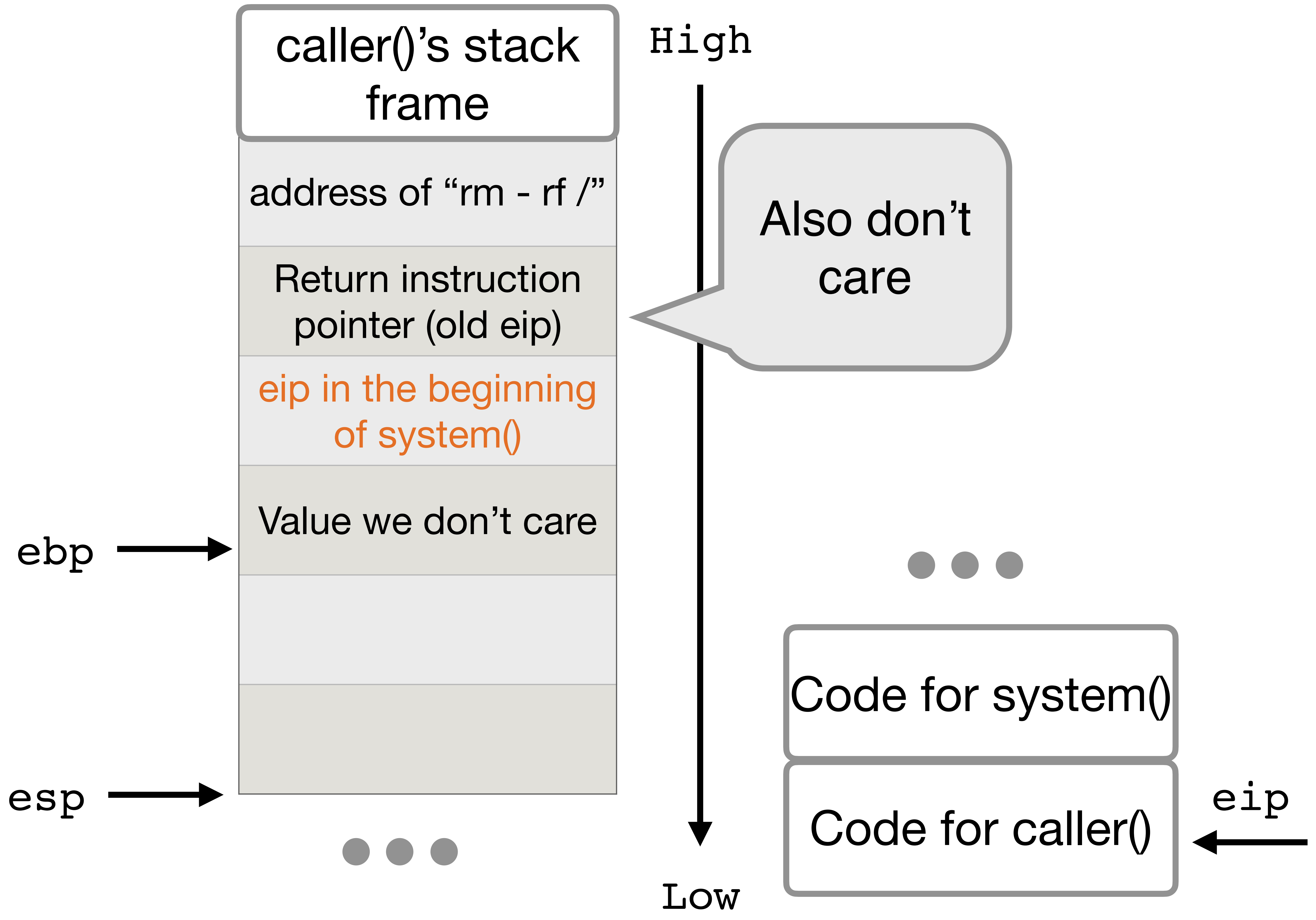
Return into libc: before return

Goal: system(“rm -rf /”)

after executing leave ret

- **fake eip**
- **Don’t care what the ebp is**
- **esp is 4 bytes below arg**

- If we find a buffer…
- Set up the stack like this!
- And return

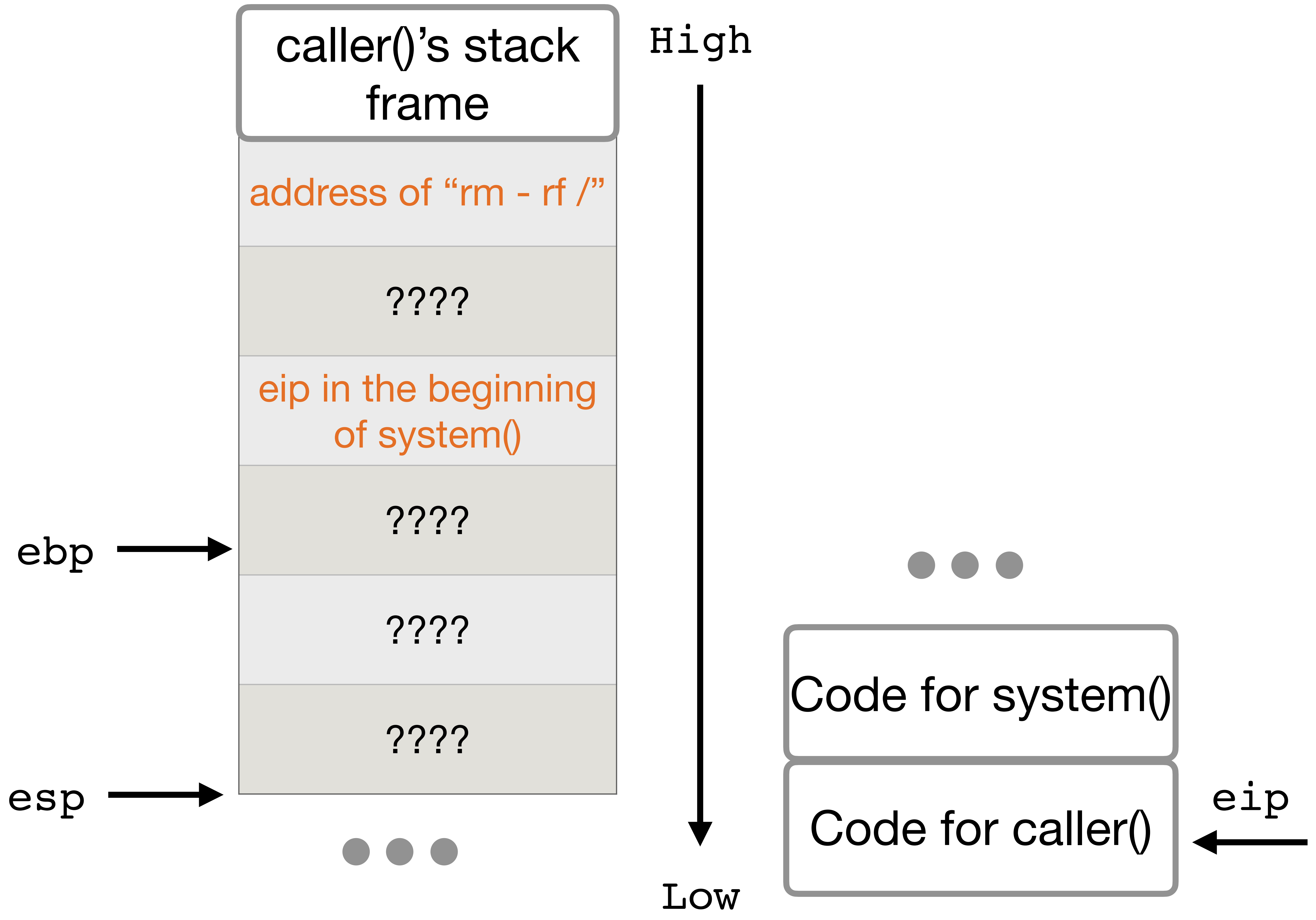


Return into libc: before return

Goal: system("rm -rf /")

after executing leave ret

- **fake eip**
- **Don't care what the ebp is**
- **esp is 4 bytes below arg**



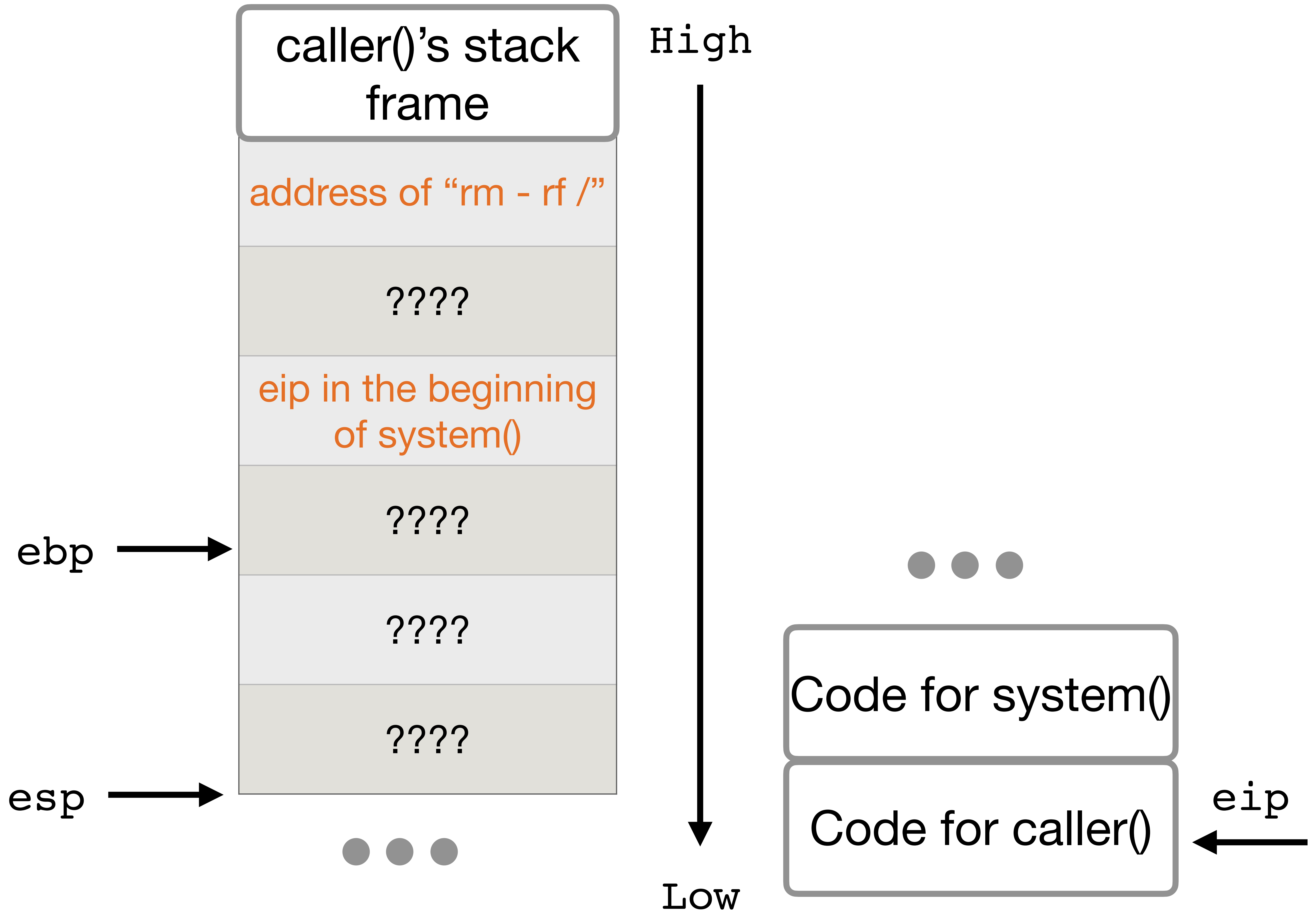
- If we find a buffer…
- Set up the stack like this!
- And return

Exercise: Go through leave return

Check that we care calling system(“rm -rf /”)

after executing leave ret

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



Return Oriented Programming (ROP)

Instead of executing an existing function,

execute different pieces of assembly instructions.

ROP Example

- Execute pieces of assembly code in a chain, among many returns
  - They form the functionality that the attacker wants
- What is a Gadget
- How to chain two gadgets together
- How to start executing the first gadget

ROP Gadget

- Gadget: A small set of assembly instructions that already exist in memory
  - Gadgets usually end in a **`ret`** instruction
  - Gadgets are usually **not** full functions

```
foo:
           ...
<foo+7>   addl $4, %esp
<foo+10>  xorl %eax, %ebx
<foo+12>  ret
```

```
bar:
           ...
<bar+22>  andl $1, %edx
<bar+25>  movl $1, %eax
<bar+30>  ret
```

How to chain two gadgets together

- Supposed our goal is:
  - `movl $1, %eax`
  - `xorl %eax, %ebx`

```
foo:
           ...
<foo+7>   addl $4, %esp
<foo+10>  xorl %eax, %ebx
<foo+12>  ret
```

```
bar:
           ...
<bar+22>  andl $1, %edx
<bar+25>  movl $1, %eax
<bar+30>  ret
```

What to do about ret?

- The following two gadgets allow us to do
  - `<bar+25> movl $1, %eax`
  - `<bar+30> ret`
  - `<foo+10> xorl %eax, %ebx`
  - `<foo+12> ret`

```
foo:
            ...
<foo+7>   addl $4, %esp
<foo+10>  xorl %eax, %ebx
<foo+12>  ret
```

```
bar:
            ...
<bar+22> andl $1, %edx
<bar+25> movl $1, %eax
<bar+30> ret
```

What to do about ret?

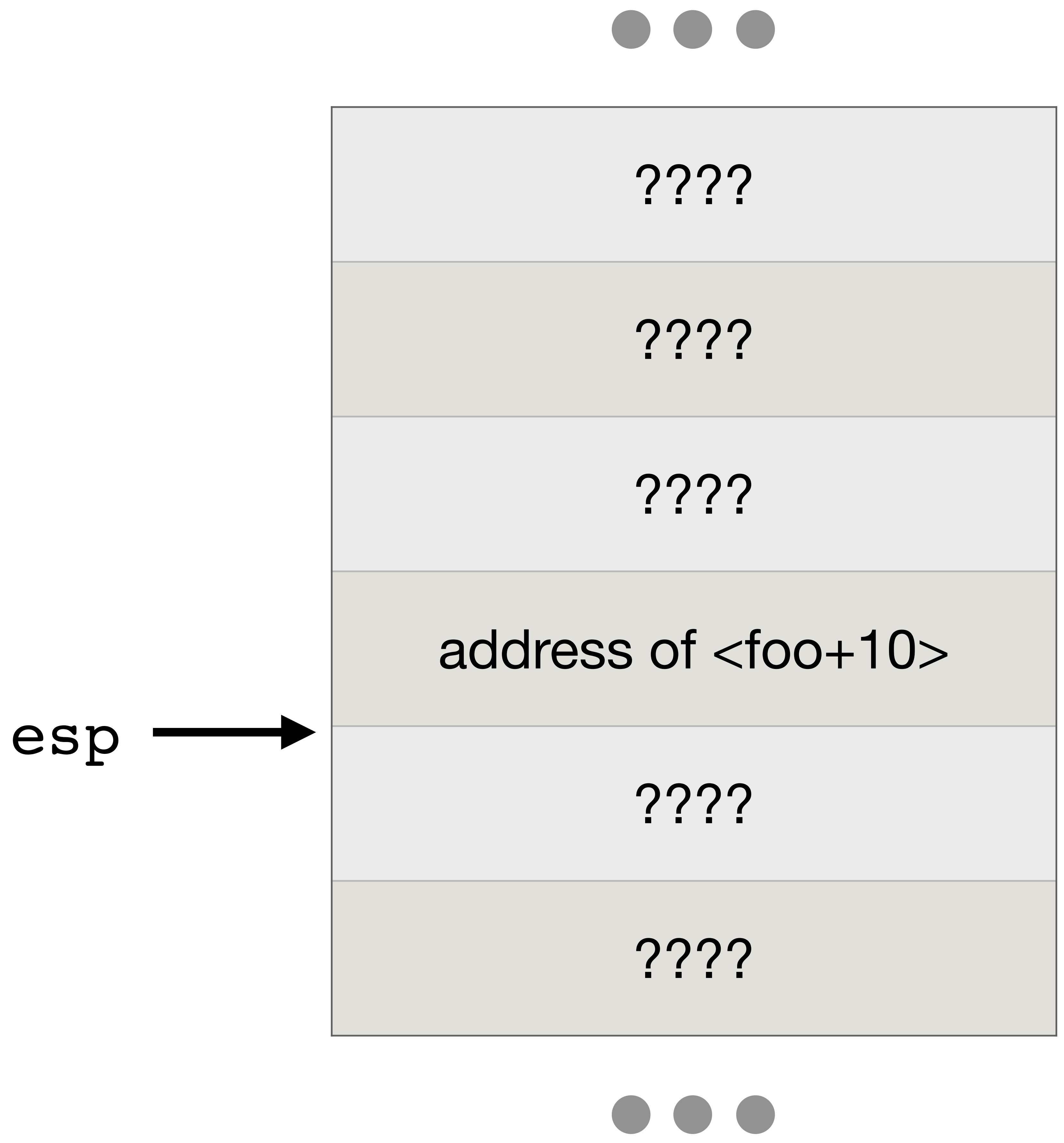
- The following two gadgets allow us to do
  - `<bar+25> movl $1, %eax`
  - `<bar+30> ret`
  - `<foo+10> xorl %eax, %ebx`
  - `<foo+12> ret`

ret: pop %eip

Put `<foo+10>` on the stack before we do ret

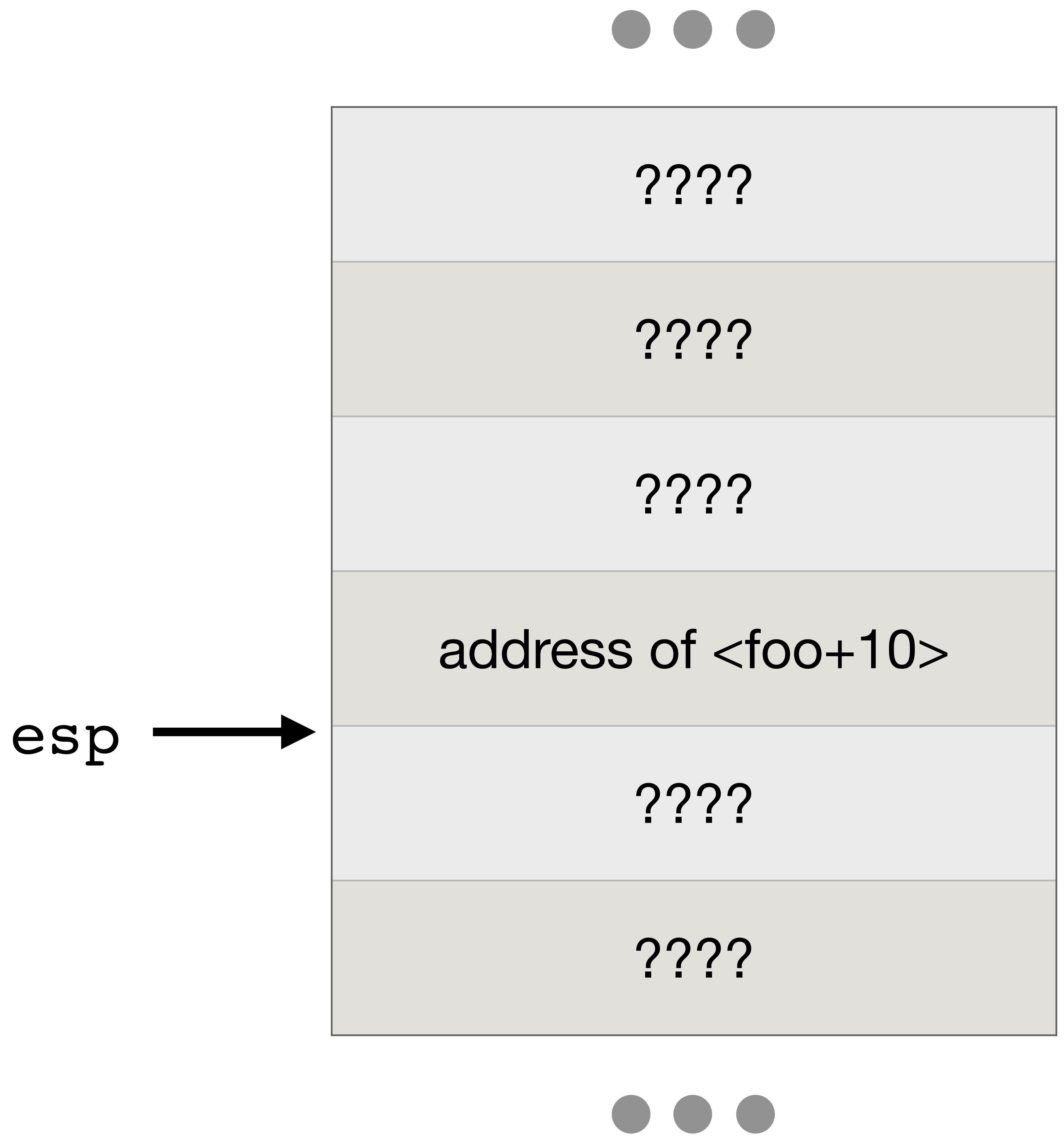
How to chain two gadgets together?

- The following two gadgets allow us to do
  - <bar+25> movl $1, %eax
  - <bar+30> ret
  - <foo+10> xorl %eax, %ebx
  - <foo+12> ret



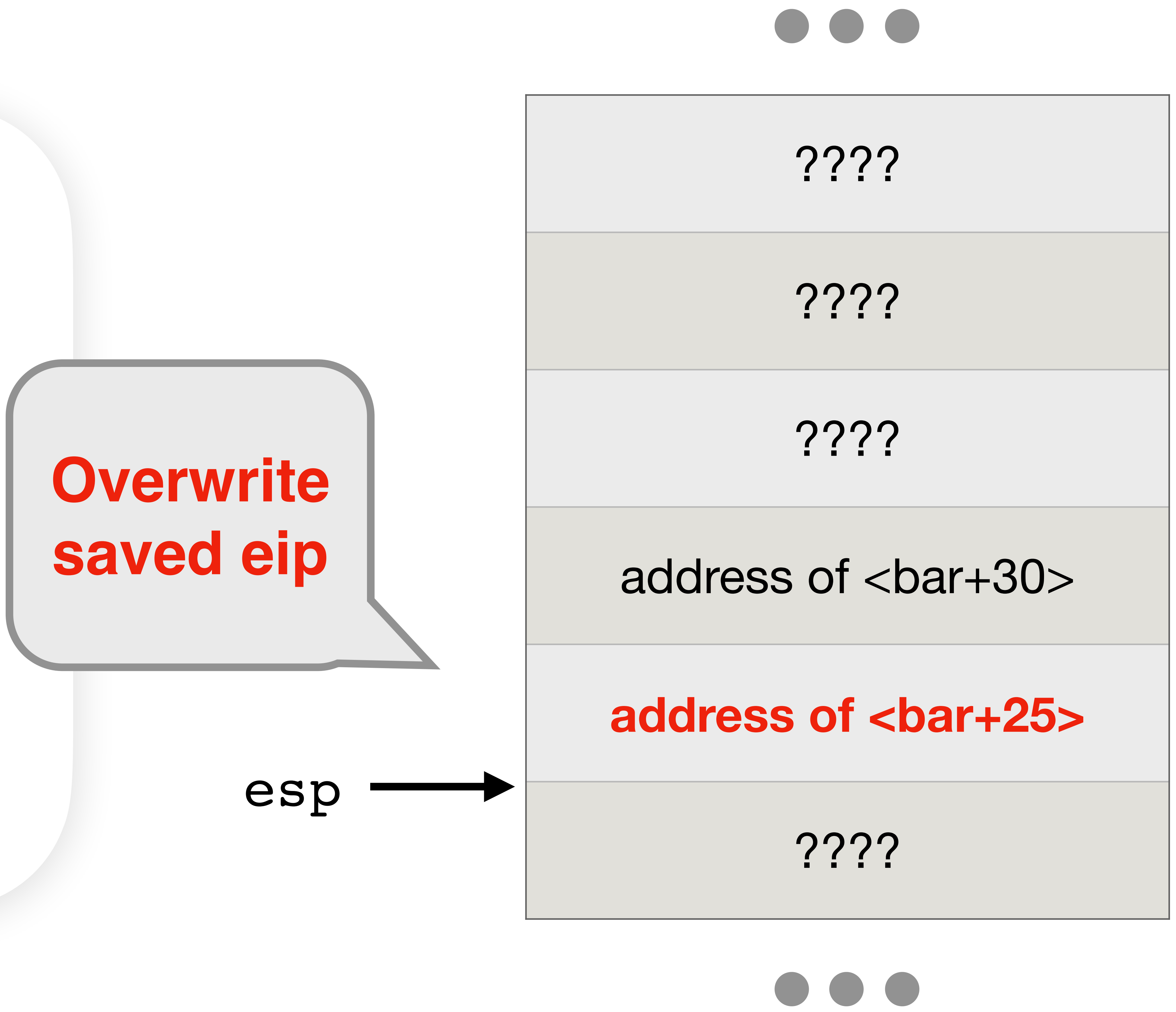
How to start executing?

- The following two gadgets allow us to do
  - `<bar+25> movl $1, %eax`
  - `<bar+30> ret`
  - `<foo+10> xorl %eax, %ebx`
  - `<foo+12> ret`



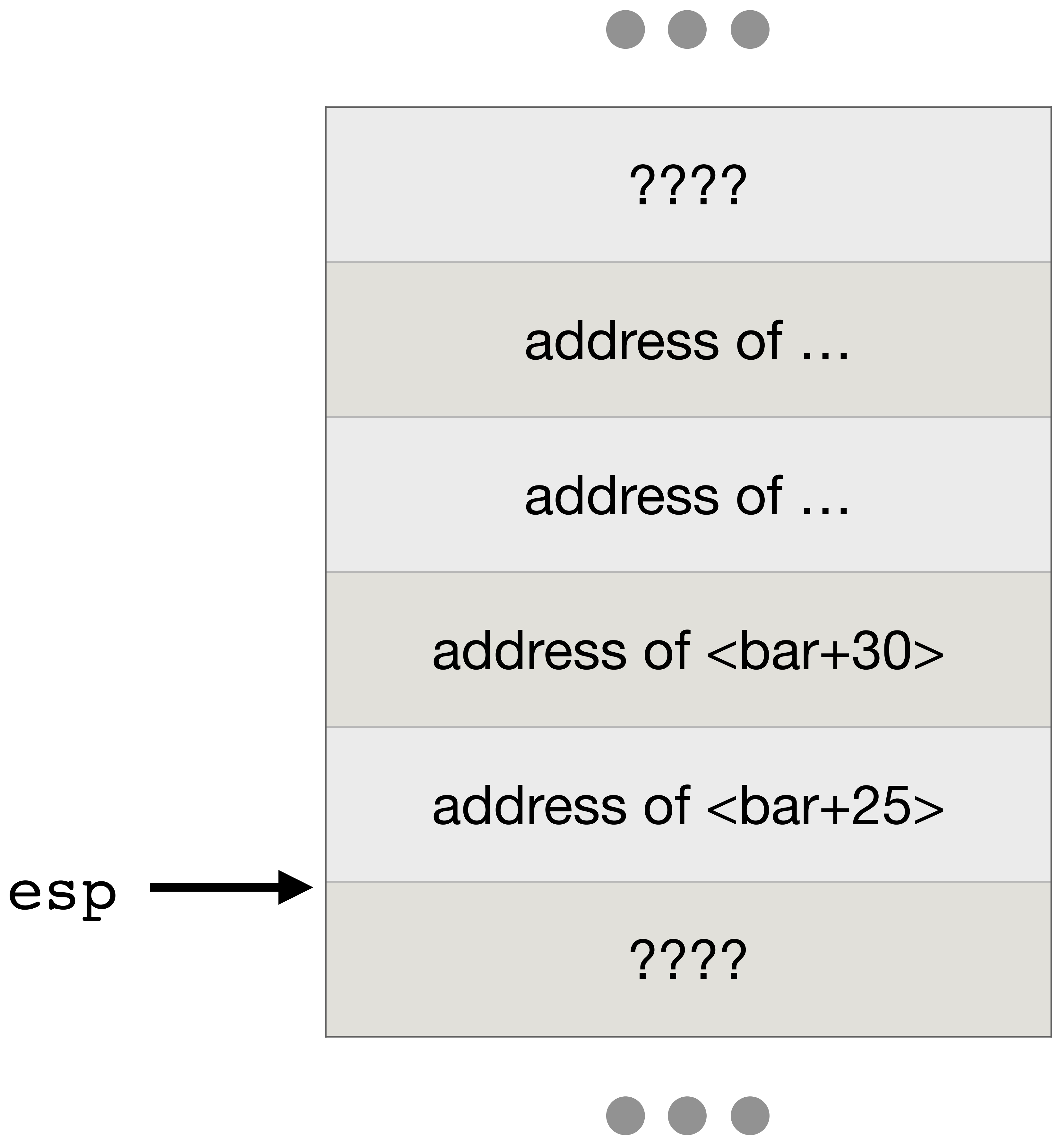
How to start executing?

- The following two gadgets allow us to do
  - `<bar+25> movl $1, %eax`
  - `<bar+30> ret`
  - `<foo+10> xorl %eax, %ebx`
  - `<foo+12> ret`



ROP

- If we have many gadgets
  - `<bar+25> movl $1, %eax`
  - `<bar+30> ret`
  - `<foo+10> xorl %eax, %ebx`
  - `<foo+12> ret`
  - `<…> …`
  - `<…> ret`
  - `<…> …`
  - `<…> ret`
  - …



ROP

- Gadget: A small set of assembly instructions that already exist in memory
  - Gadgets usually end in a **`ret`** instruction
  - Gadgets are usually **not** full functions
- ROP strategy: We write a chain of return addresses starting at the RIP to achieve the behavior we want
  - Each return address points to a gadget
  - The gadget executes its instructions and ends with a ret instruction
  - The ret instruction jumps to the address of the next gadget on the stack

ROP

- If the code base is big enough (imports enough libraries), there are usually enough gadgets in memory for you to run any shellcode you want

- **ROP compilers** can automatically generate a ROP chain for you based on a target binary and desired malicious code!

- Non-executable pages is not a huge issue for attackers nowadays
  - Having writable and executable pages makes an attacker's life easier, but not *that* much easier

Agenda

- Memory-safe languages
- Writing memory-safe code
- Building secure software
- Exploit mitigations
  - Non-executable pages
  - Stack canaries
  - Pointer authentication
  - Address space layout randomization (ASLR)
- Combining mitigations

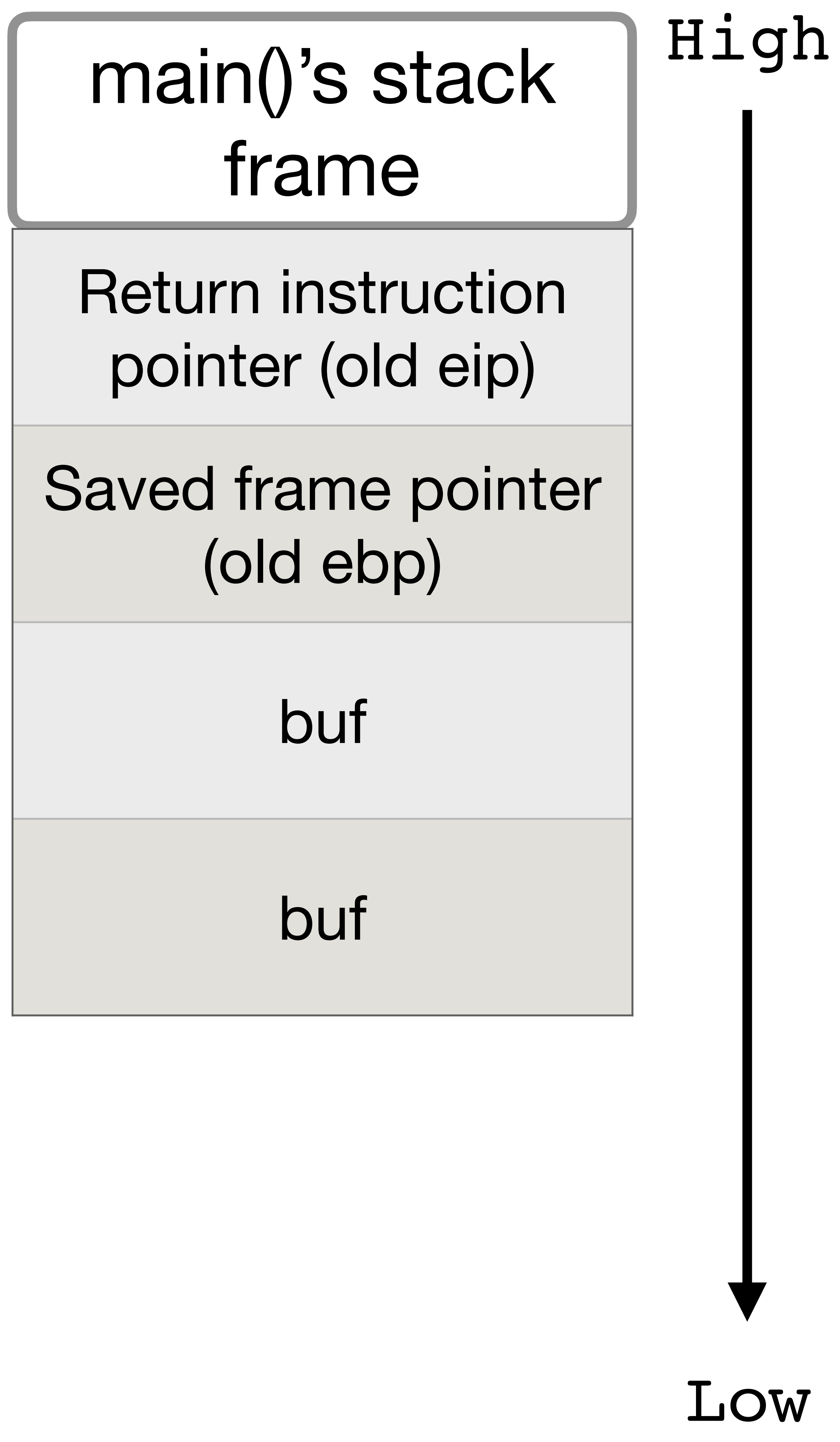
Stack Canaries

https://share.america.gov/english-idiom-canary-coal-mine/

Regular Stack Example

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

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

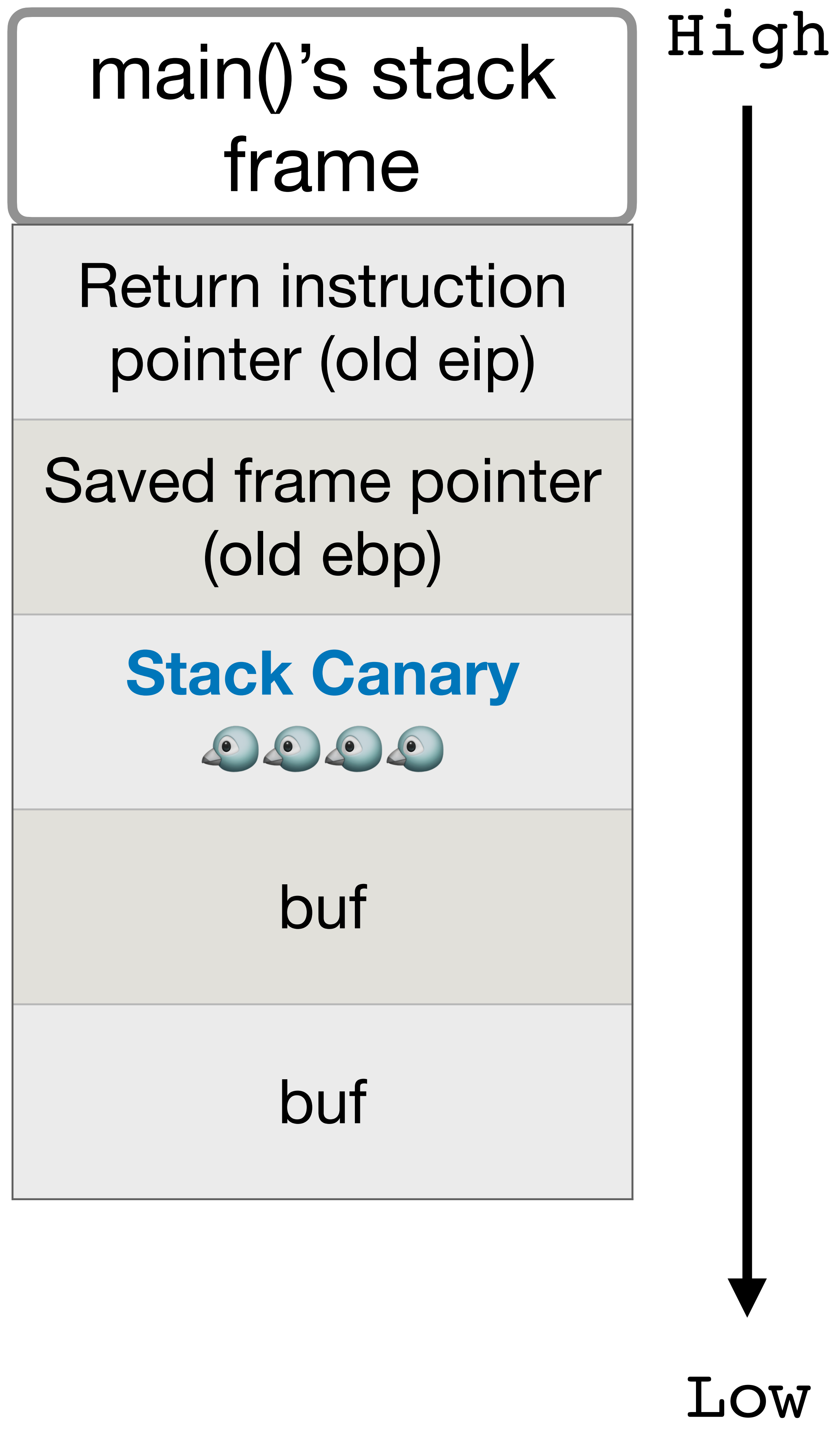


Stack Canaries

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

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

The attack will have to overwrite the **stack canary**



Stack Canaries

- During runtime, generate a **random secret** value and save it in the canary storage
  - In the function prologue, place the canary value on the stack right below the SFP/RIP
  - In the function epilogue, check the value on the stack and compare it against the value in canary storage
  - If the canary value changes, somebody is probably attacking our system!

Stack Canaries

- A canary value is unique every time the program runs but the **same for all functions within a run**
- A canary value uses a NULL byte as the first byte to mitigate string-based attacks (since it terminates any string before it)
    - Example: A format string vulnerability with %s might try to print everything on the stack
    - The null byte in the canary will mitigate the damage by stopping the print earlier.
- Overhead: compiler inserts a few extra instructions, bust mostly low overhead

Subverting Stack Canaries

- **Leak** the value of the canary: Overwrite the canary with itself
- **Bypass** the value of the canary: Use a random write, not a sequential write
- **Guess** the value of the canary: Brute-force

Guess the Canary

- The first byte (8 bits) is always a NULL byte
- On 32-bit systems: 24 bits to guess
  - 32 - 8 = 24
  - 2^24 possibilities (~16 million), can be brute-forced, depending on the setting
- On 64-bit systems: 56 bits to guess

Pointer Authentication

- **Stack Canaries:** place some secret value below pointers (return instruction pointer and saved frame pointer)
- **Pointer Authentication:** place some secret value in the pointers

Pointer Authentication

- **Stack Canaries:** place some secret value below pointers (return instruction pointer and saved frame pointer)
- **Pointer Authentication:** place some secret value in the pointers
  - In a 64 bit system, 42 bits are ~4TB of memory, 22 bits are unused
  - Put the secret **(PAC, pointer authentication code)** in unused bits

Pointer Authentication

- **Stack Canaries:** place some secret value below pointers (return instruction pointer and saved frame pointer)
- **Pointer Authentication:** place some secret value in the pointers
  - In a 64 bit system, 42 bits are ~4TB of memory, 22 bits are unused
  - Put the secret **(PAC, pointer authentication code)** in unused bits
  - Before using the pointer in memory, check if the PAC is still valid
    - Invalid: crash the program
    - Valid: restore unused bits, use the address normally

Properties of PAC

- Each possible address has its own PAC
- Message Authentication Code (MAC) in the cryptography lectures
- Only someone who knows the CPU’s master secret can generate a PAC for an address
- The CPU’s master secret is not accessible to the program
    - Leaking program memory will not leak the master secret

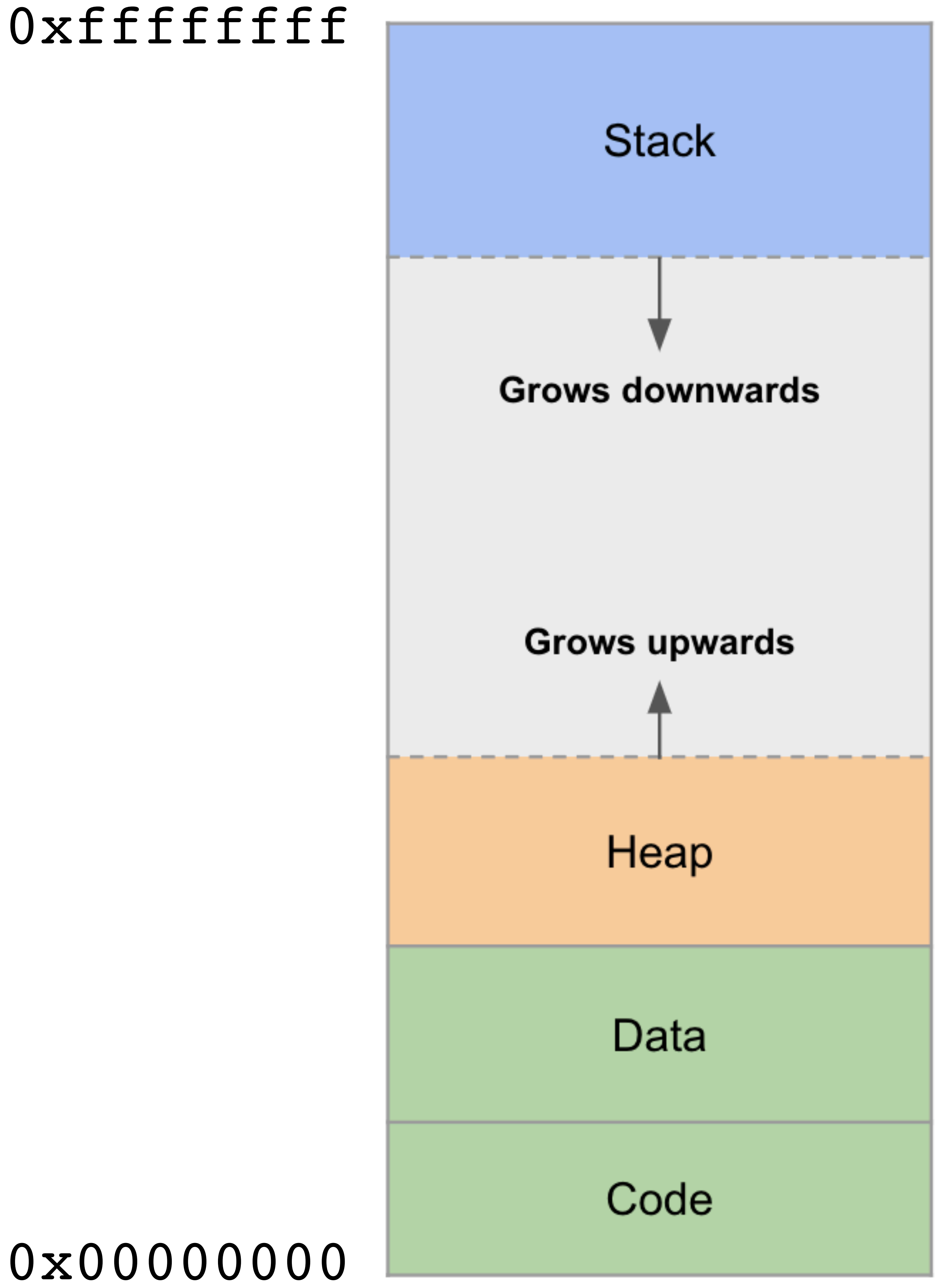
Subverting Pointer Authentication

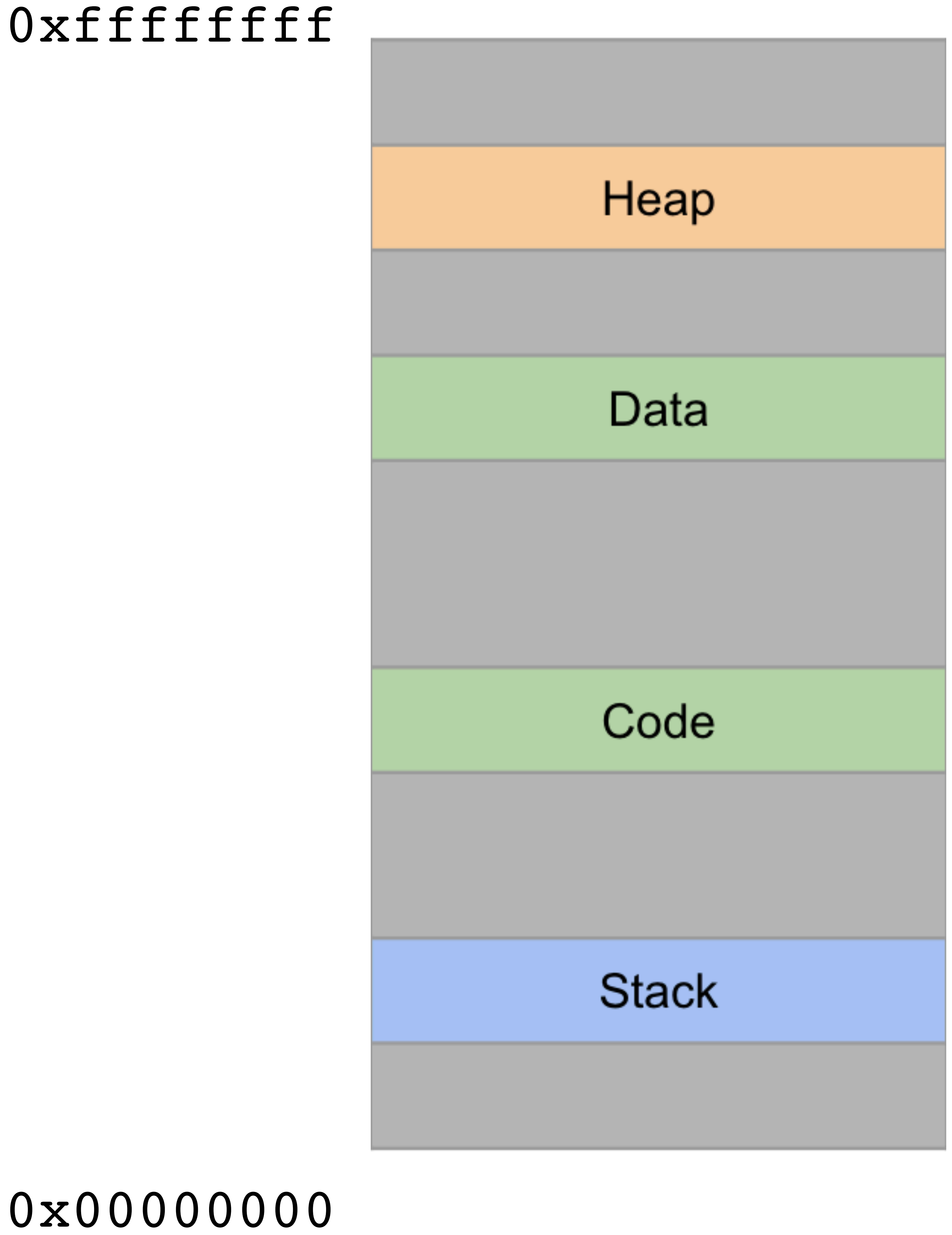
- Find a vulnerability to trick the program to generating a PAC for any address
- Learn the master secret
  - Vulnerability in the OS
- Guess a PAC: Brute-force
- Pointer reuse

Address Space Layout Randomization

- Goal: make it hard for attackers to place shell code on the stack, on the heap, or find out the address of the code
- Randomize the addresses of code, data, heap, stack
- Theoretically, very hard to know the addresses, so we can mitigate the attacks

Address Space Layout Randomization





Address Space Layout Randomization

- **Address space layout randomization (ASLR):** Put each segment of memory in a different location each time the program is run
  - Programs are dynamically linked at runtime, so ASLR has almost no overhead

Address Space Layout Randomization

- **Address space layout randomization (ASLR):** Put each segment of memory in a different location each time the program is run
  - Programs are dynamically linked at runtime, so ASLR has almost no overhead
- However…
- Within each segment of memory, relative addresses are the same (e.g. the RIP is always 4 bytes above the SFP)
  - Leak the address of a pointer, whose address relative to your shellcode is known (stack pointer, RIP)
  - Guess the address of your shellcode: Brute-force

Combining Mitigations

- **Defense in depth**
- Example: Combining ASLR and non-executable pages
- To defeat ASLR and non-executable pages, the attacker needs to find two vulnerabilities
  - First, find a way to leak memory and reveal the address randomization (defeat ASLR)
  - Second, find a way to write to memory and write a ROP chain (defeat non-executable pages)