CMSC414 Computer and Network Security

Mitigating Memory Safety Vulnerabilities

Yizheng Chen | University of Maryland
surrealyz.github.io

Feb 10, 2026

Announcements

- Project 1, **due in 9 days! Thursday, Feb 19**
- Piazza anonymous post to classmates
- Project 2 will be released this Thursday

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

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

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

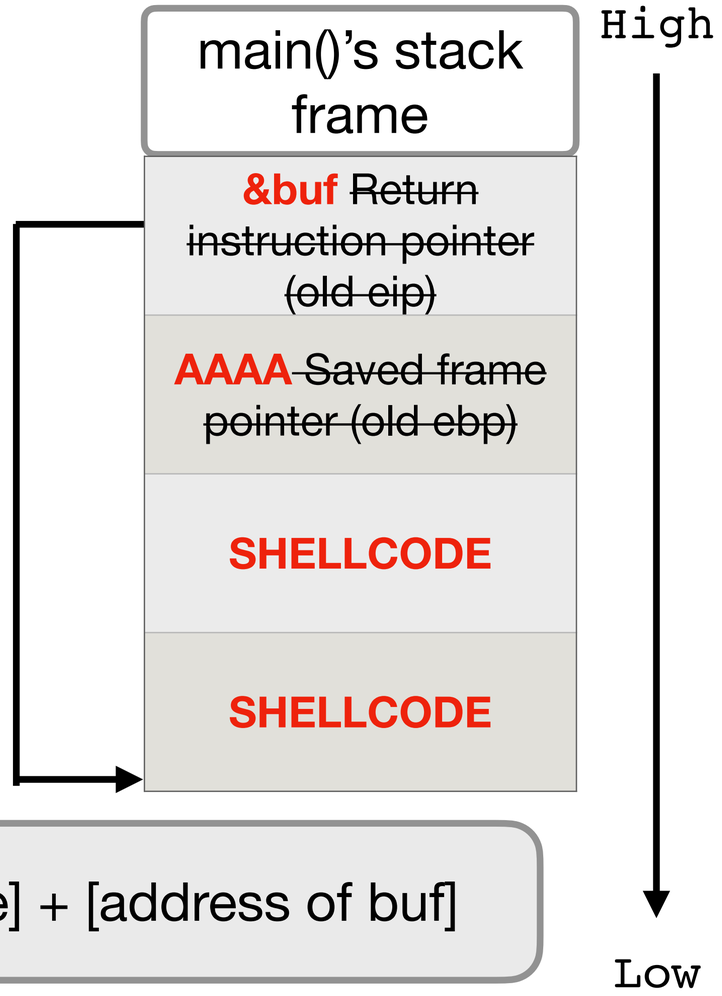
- Exploit mitigations
  - Non-executable pages
  - Stack canaries
  - Pointer authentication
  - Address space layout randomization (ASLR)
- Combining mitigations

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

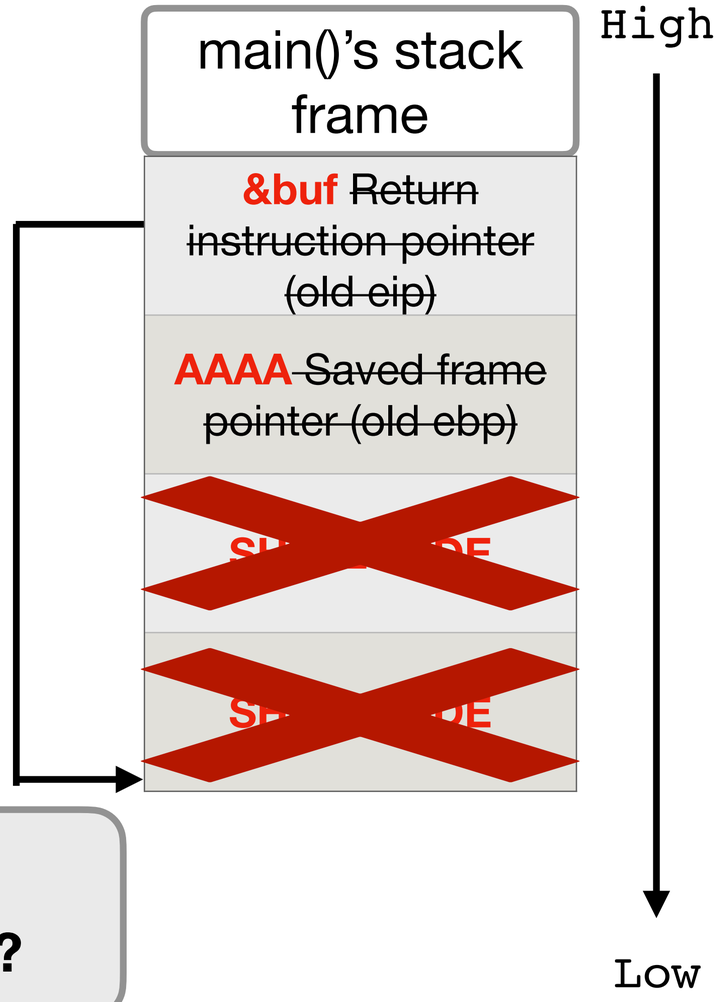
```
void main() {
    vulnerable();
}

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)
    ...
}
```



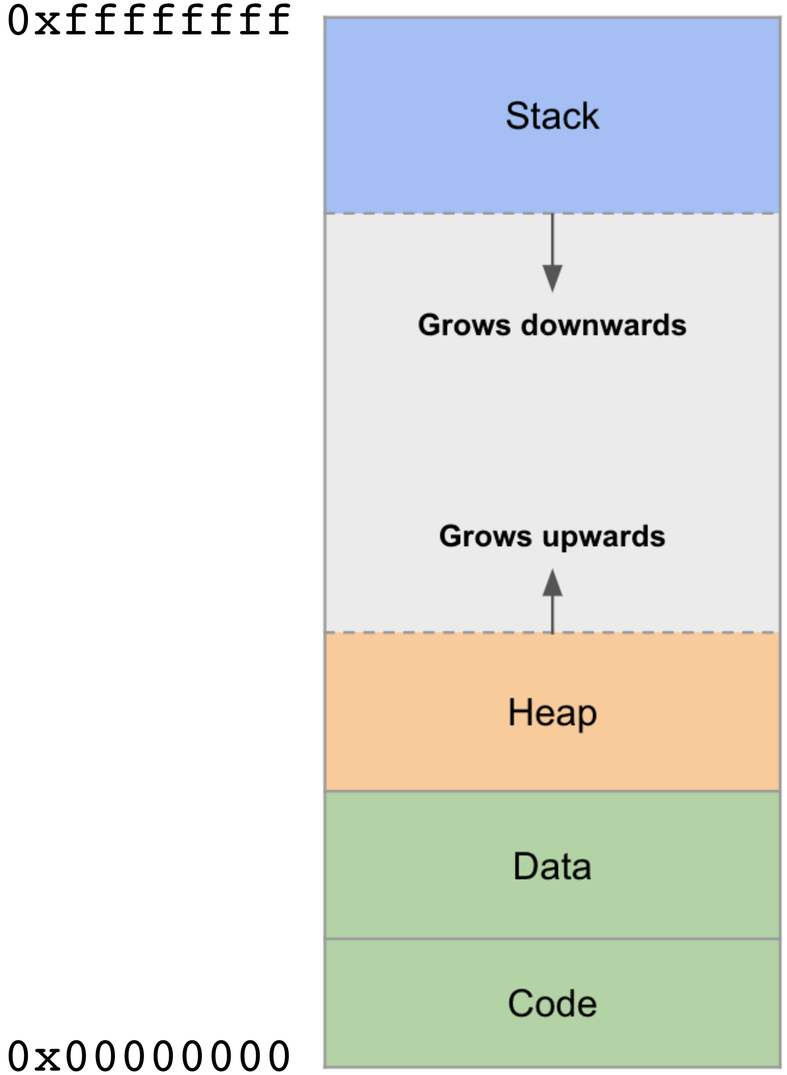
What if shell code cannot be executed?

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

x86 Memory Layout



Local variables and stack frames

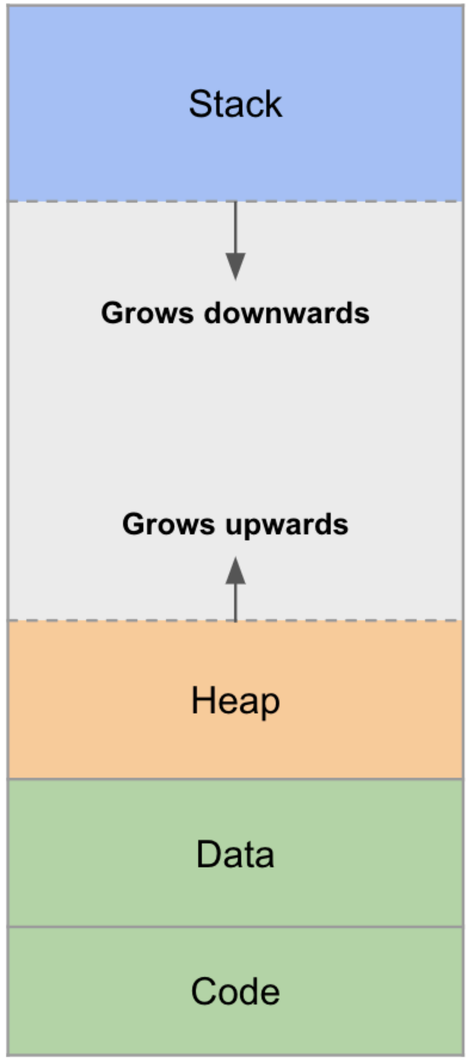
Where buffer overflow happens

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

Static Variables

The program code (also called “text”)

x86 Memory Layout



Writable

Where buffer overflow happens

Writable

Writable

Executable

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

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
- What kind of functions / code?

Standard C library (libc)

C stdio Functions

The `<stdio.h>` header provides a variety of functions for input, output and file handling.

- C library:
  - <cassert> (assert.h)
  - <cctype> (ctype.h)
  - <cerrno> (errno.h)
  - <cfenv> (fenv.h) C++11
  - <cfloat> (float.h)
  - <cinttypes> (inttypes.h) C++11
  - <ciso646> (iso646.h)
  - <climits> (limits.h)
  - <clocale> (locale.h)
  - <cmath> (math.h)
  - <csetjmp> (setjmp.h)
  - <csignal> (signal.h)
  - <cstdarg> (stdarg.h)
  - <cstdbool> (stdbool.h) C++11
  - <cstddef> (stddef.h)
  - <cstdint> (stdint.h) C++11
  - <cstdio> (stdio.h)
  - <cstdlib> (stdlib.h)
  - <cstring> (string.h)
  - <ctgmath> (tgmath.h) C++11

Loaded code that already exists in memory

```
 0x5655633f <+94>:    call   0x565560d0 <memset@plt>
 0x56556344 <+99>:    add    $0x10,%esp
 0x56556347 <+102>:   mov    0x4(%esi),%eax
 0x5655634a <+105>:   add    $0x4,%eax
 0x5655634d <+108>:   mov    (%eax),%eax
 0x5655634f <+110>:   sub    $0x4,%esp
 0x56556352 <+113>:   push   $0x63
 0x56556354 <+115>:   push   %eax
 0x56556355 <+116>:   lea    -0x80(%ebp),%eax
 0x56556358 <+119>:   push   %eax
 0x56556359 <+120>:   call   0x565560e0 <strncpy@plt>
 0x5655635e <+125>:   add    $0x10,%esp
 0x56556361 <+128>:   sub    $0x8,%esp
 0x56556364 <+131>:   lea    -0x1f8e(%ebx),%eax
 0x5655636a <+137>:   push   %eax
 0x5655636b <+138>:   lea    -0x80(%ebp),%eax
 0x5655636e <+141>:   push   %eax
Type <RET> for more, q to quit, c to continue without paging--c
 0x5655636f <+142>:   call   0x565560c0 <fopen@plt>
 0x56556374 <+147>:   add    $0x10,%esp
 0x56556377 <+150>:   mov    %eax,-0x1c(%ebp)
 0x5655637a <+153>:   sub    $0xc,%esp
 0x5655637d <+156>:   lea    -0x8a(%ebp),%eax
 0x56556383 <+162>:   push   %eax
 0x56556384 <+163>:   call   0x5655623d <your_fcn>
 0x56556389 <+168>:   add    $0x10,%esp
 0x5655638c <+171>:   sub    $0x8,%esp
 0x5655638f <+174>:   push   -0x1c(%ebp)
 0x56556392 <+177>:   lea    -0x8a(%ebp),%eax
 0x56556398 <+183>:   push   %eax
 0x56556399 <+184>:   call   0x565560f0 <fputs@plt>
```

Subverting Non-Executable Pages

- **Return-to-libc**: An exploit technique that overwrites the RIP to jump to a function 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

x86 function call in assembly

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

void foo(int a, int b) {
    int bar[4];
}
```

```
main:
    push $2
    push $1
    call foo

    # Execution changes to foo now.
After returning from foo:

    add $8, %esp

foo:
    push %ebp
    mov %esp, %ebp
    sub $16, %esp

    # Execute the function (omitted)

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

x86 function call in assembly

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

void foo(int a, int b) {
    int bar[4];
}
```

```
main:
    push $2
    push $1
    call foo

    # Execution changes to foo now.
After returning from foo:

    add $8, %esp

foo:
    push %ebp
    mov %esp, %ebp
    sub $16, %esp

    # Execute the function (omitted)

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

Push arguments on the stack,
Save old eip (rip) on the stack, change eip

Remove arguments from stack

x86 function call in assembly

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

void foo(int a, int b) {
    int bar[4];
}
```

```
main:
    push $2
    push $1
    call foo

    # Execution changes to foo now.
After returning from foo:

    add $8, %esp

foo:
    push %ebp
    mov %esp, %ebp
    sub $16, %esp

    # Execute the function (omitted)

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

Push old ebp (sfp) on the stack,
Move ebp down to esp,
Move esp down

leave
ret

x86 function call in assembly

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

void foo(int a, int b) {
    int bar[4];
}
```

```
main:
    push $2
    push $1
    call foo

    # Execution changes to foo now.
After returning from foo:

    add $8, %esp

foo:
    push %ebp
    mov %esp, %ebp
    sub $16, %esp

    # Execute the function (omitted)

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

Function Prologue

Function Epilogue

How to subvert non-executable pages?

Return-to-libc: **overwrites the RIP** to jump to a function in the standard C library (libc)

Example Function

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: a real call

Caller pushes arguments
Saves eip, moves eip

```
void main() {
    ...
    char cmd[] = "rm -rf /";
    system(cmd);
    ...
}
```

```
main:
    # push arguments of system()
    …
    # Save old eip (rip) on the stack,
change eip
    call system

    # Execution changes to system now.
After returning from system:
    …

system:
    push %ebp
    mov %esp, %ebp
    sub ???, %esp

    # Execute the function (omitted)

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

Return into libc: a real call

Caller pushes arguments
Saves eip, moves eip

```
void main() {
    ...
    char cmd[] = "rm -rf /";
    system(cmd);
    ...
}
```

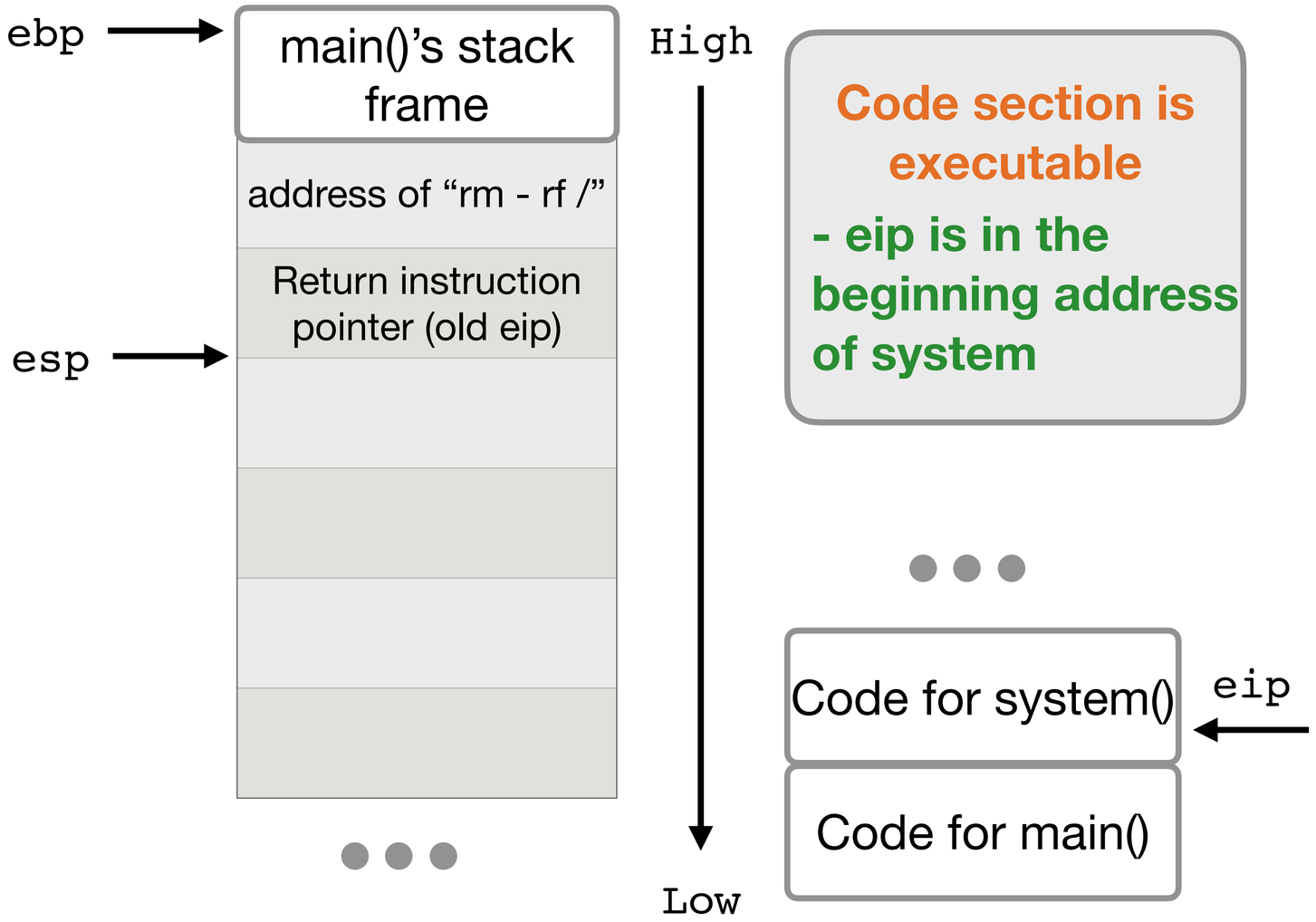
ebp: refer to arg, local var

```
main:
    # push arguments of system()
    …
    # Save old eip (rip) on the stack,
change eip
    call system

    # Execution changes to system now.
After returning from system:
    …

system:
    push %ebp
    mov %esp, %ebp
    sub ???, %esp

    # Execute the function (omitted)

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

callee saves ebp
(push local vars, omitted)

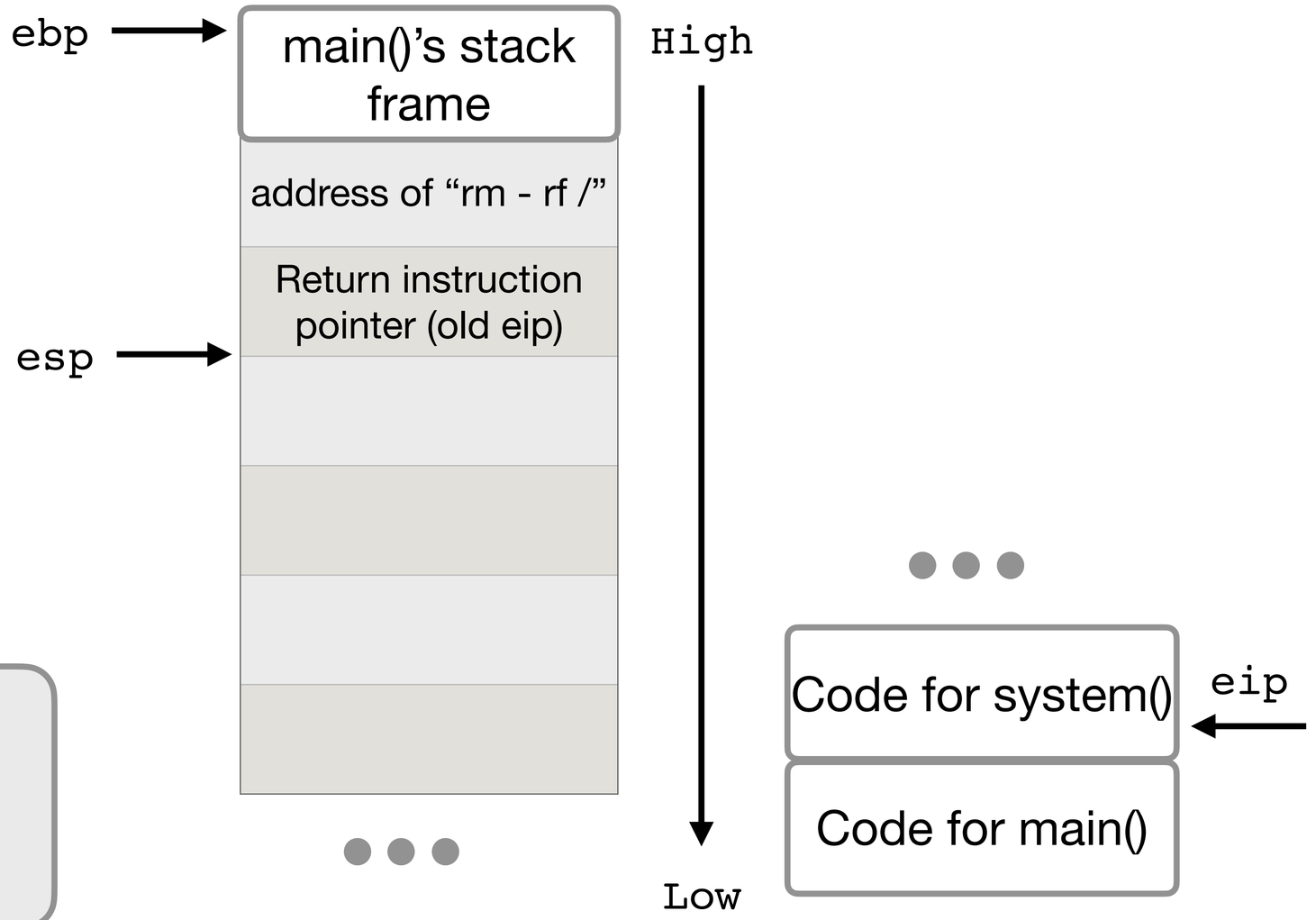
Return into libc: a real call

```
void main() {
    ...
    char cmd[] = "rm -rf /";
    system(cmd);
    ...
}
```



Return into libc: goal of a fake call

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

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



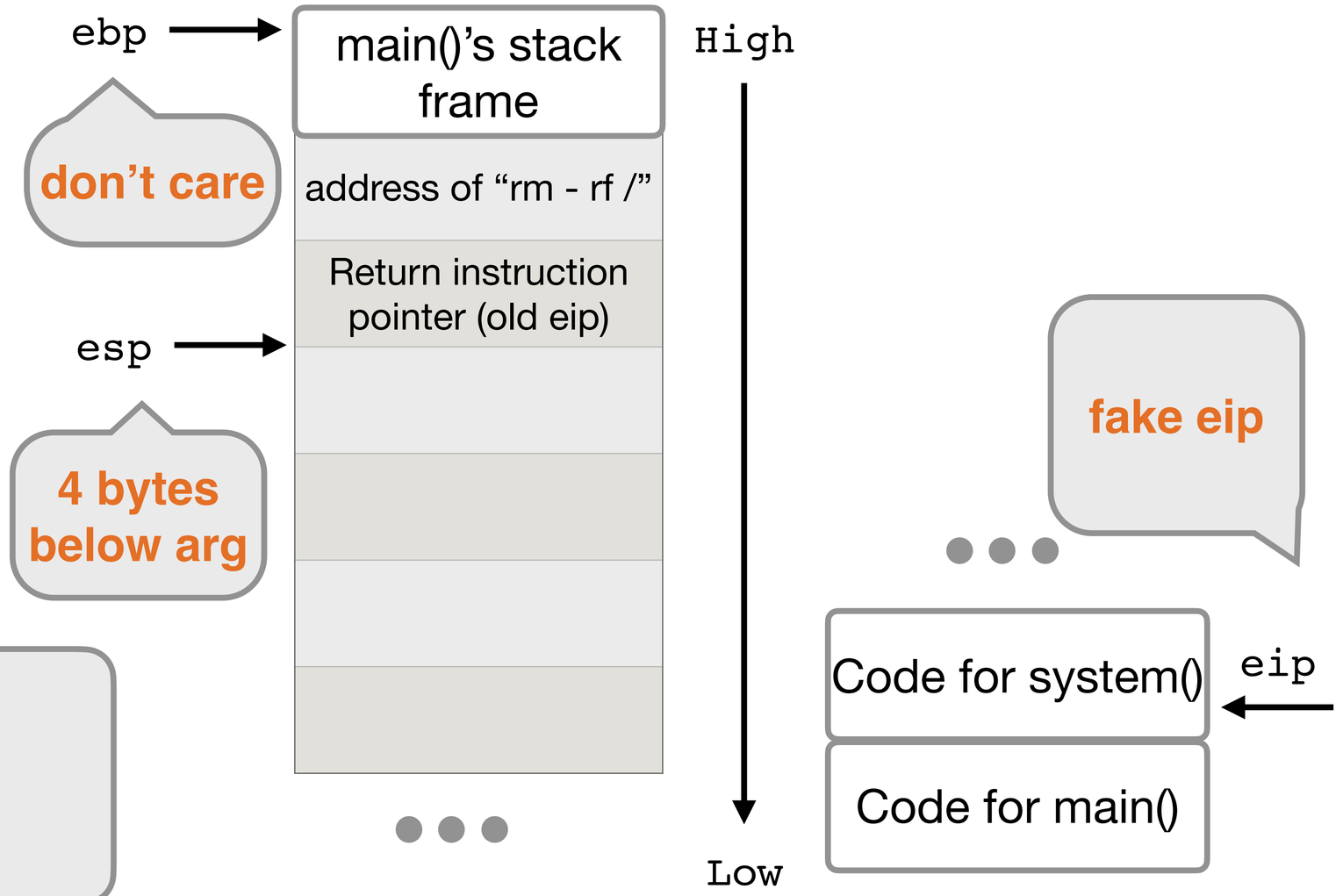
Goal: system("rm -rf /")

Return from vulnerable(),

to a stack like this?

Return into libc: goal of a fake call

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

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



Goal: system("rm -rf /")

Return from vulnerable(),

to a stack like this?

Return from a Function

In C

```
return;
```

In compiled assembly

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

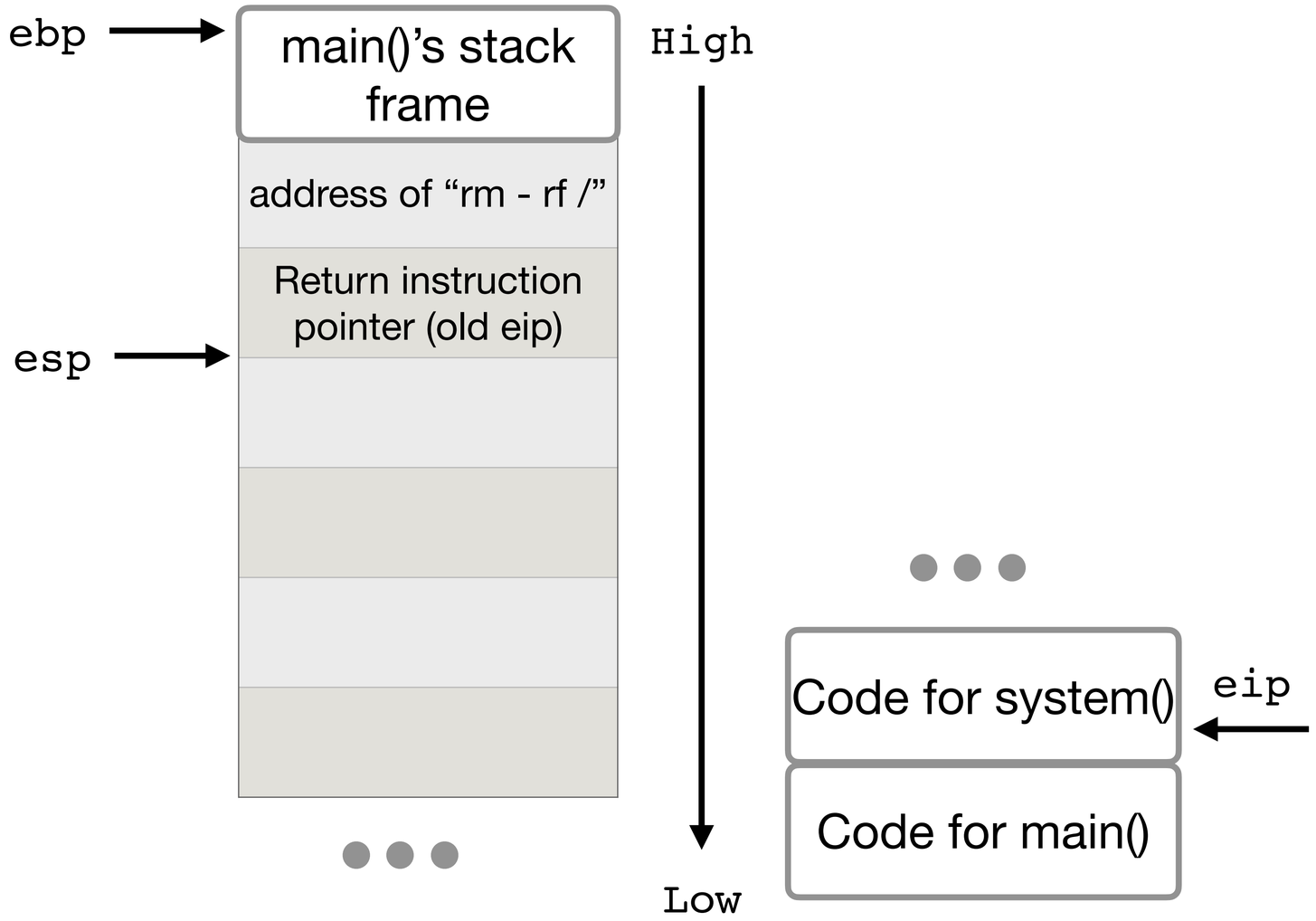
- Leave: leave the stack frame of the callee
  - **restore** the **stack pointer** (mov %ebp %esp)
  - **restore** the **base pointer** (pop %ebp)
- Ret: **restore** the **instruction pointer** (pop %eip)

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

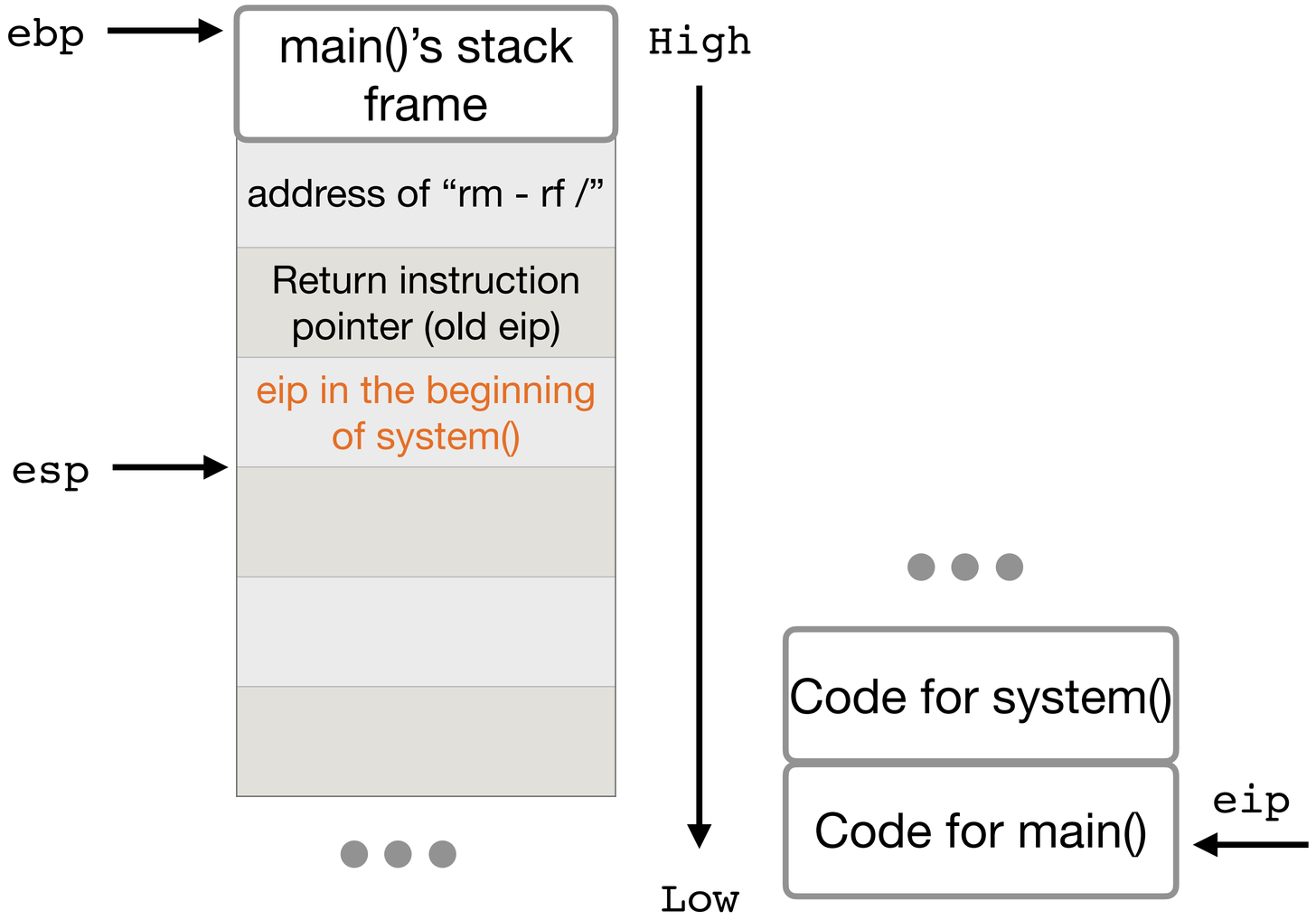
Return into libc: goal of a fake call

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



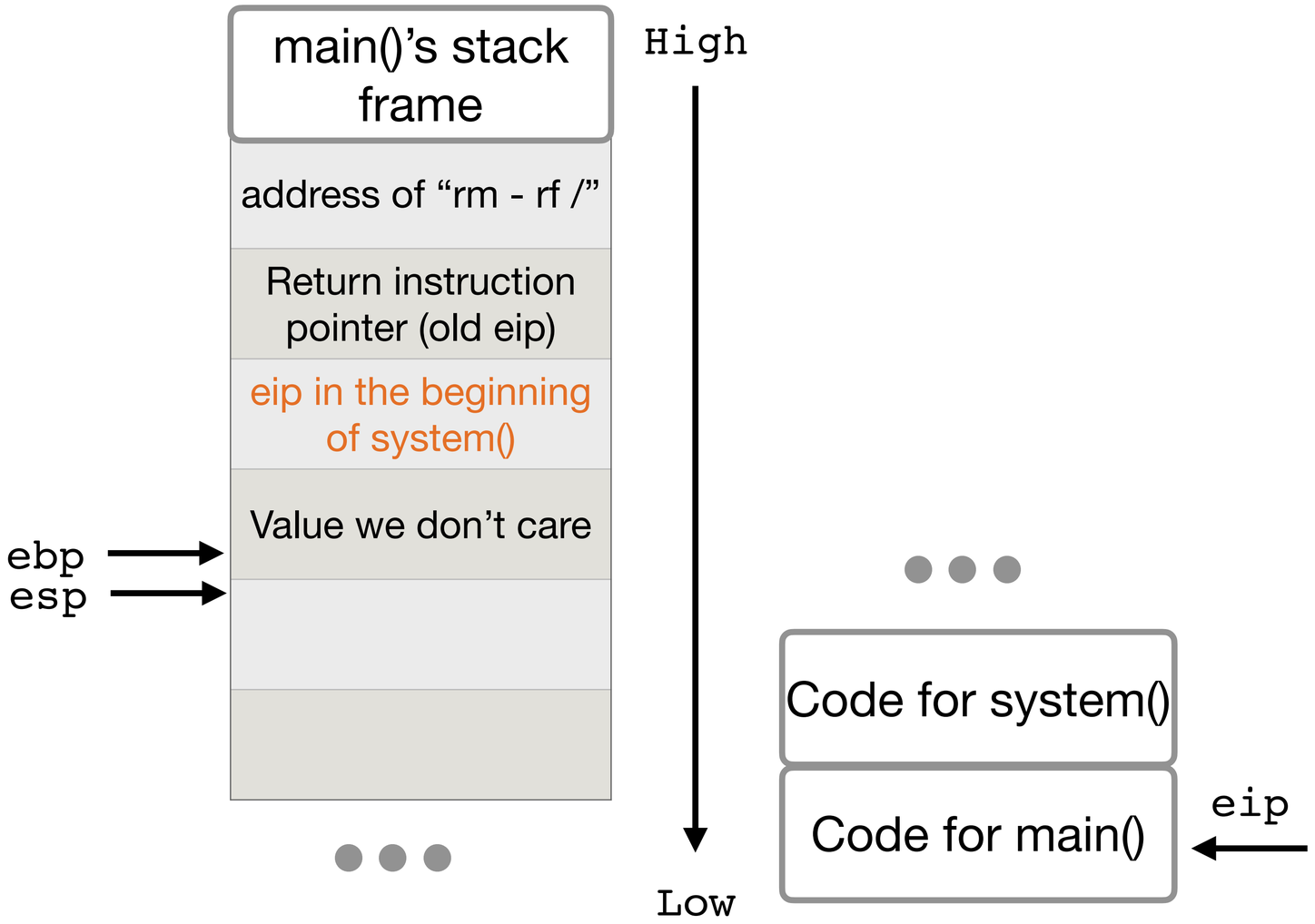
Return into libc: goal of a fake call

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



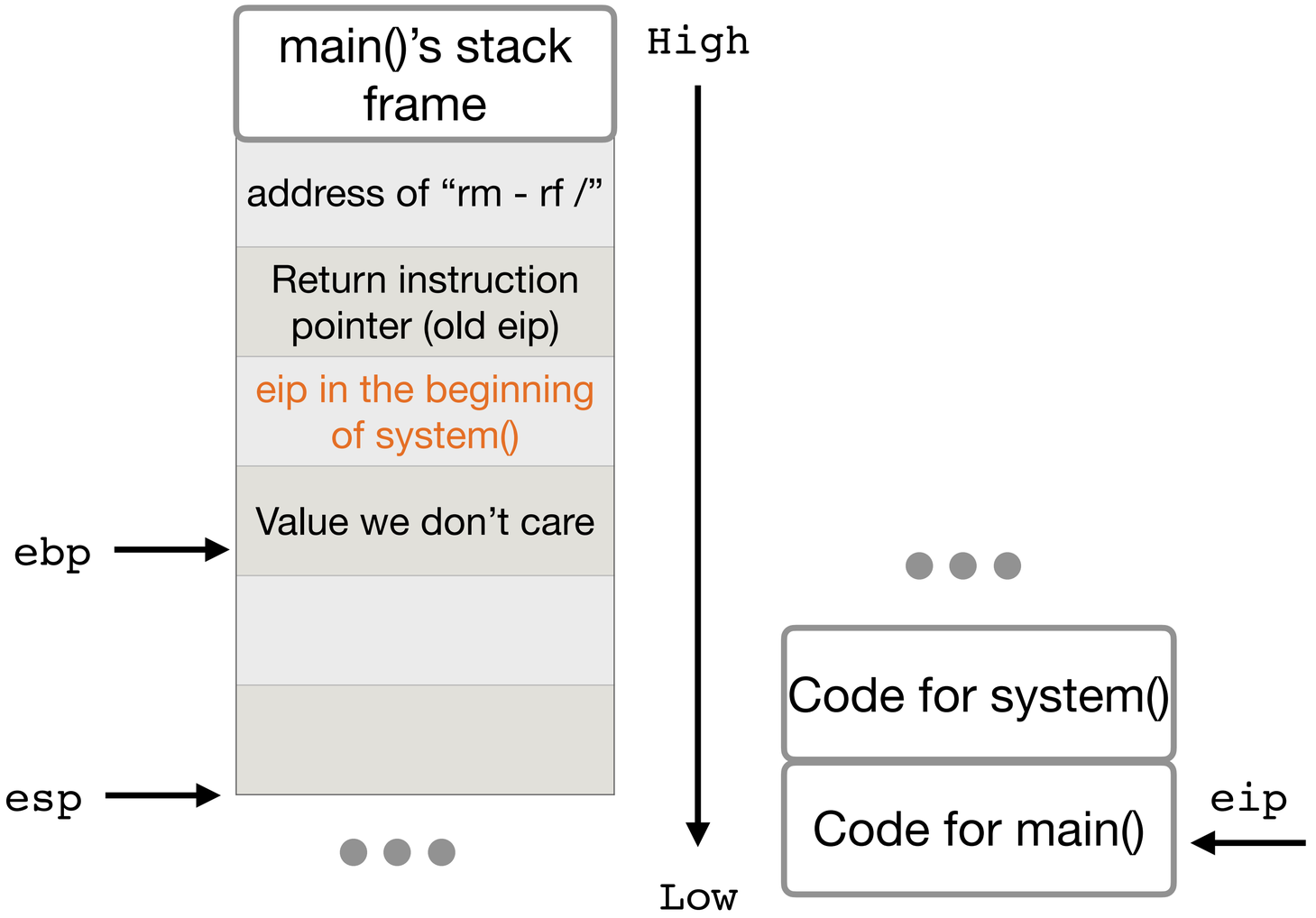
Return into libc: goal of a fake call

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



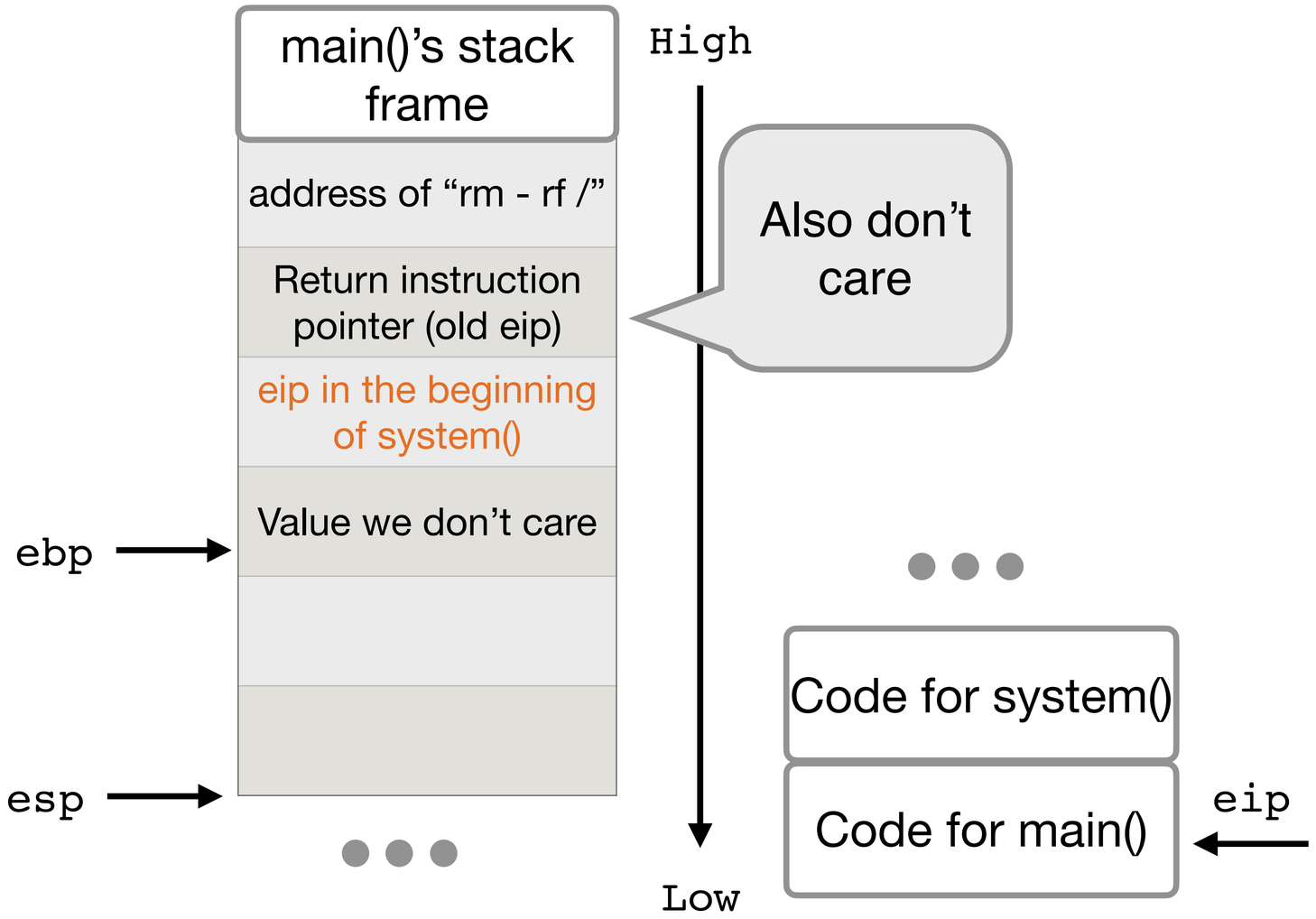
Return into libc: before return

Goal: system(“rm -rf /”)

after executing leave ret

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

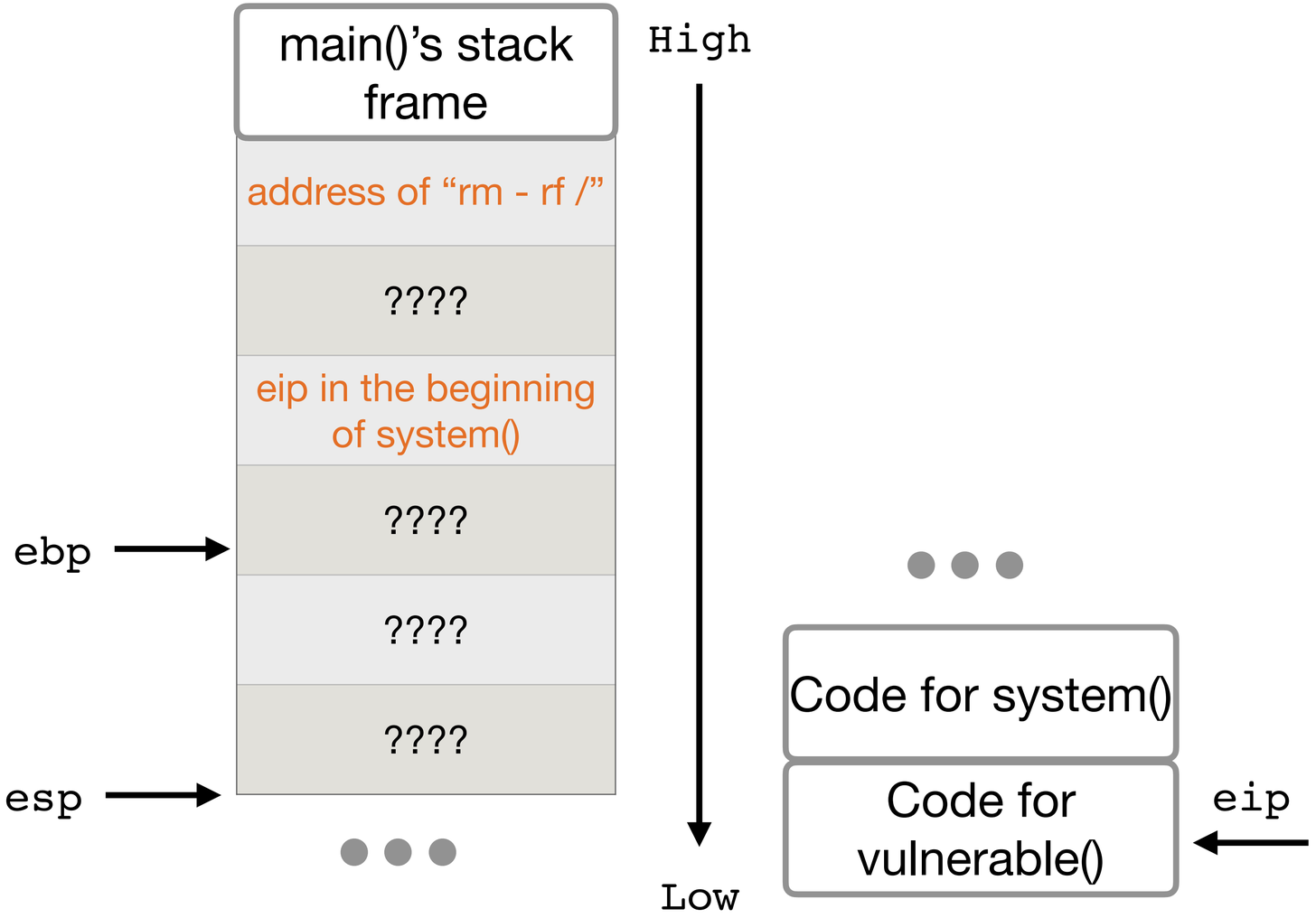
- If we find a buffer…
- Set up the stack like this!
- And return



Return into libc: before return

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

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

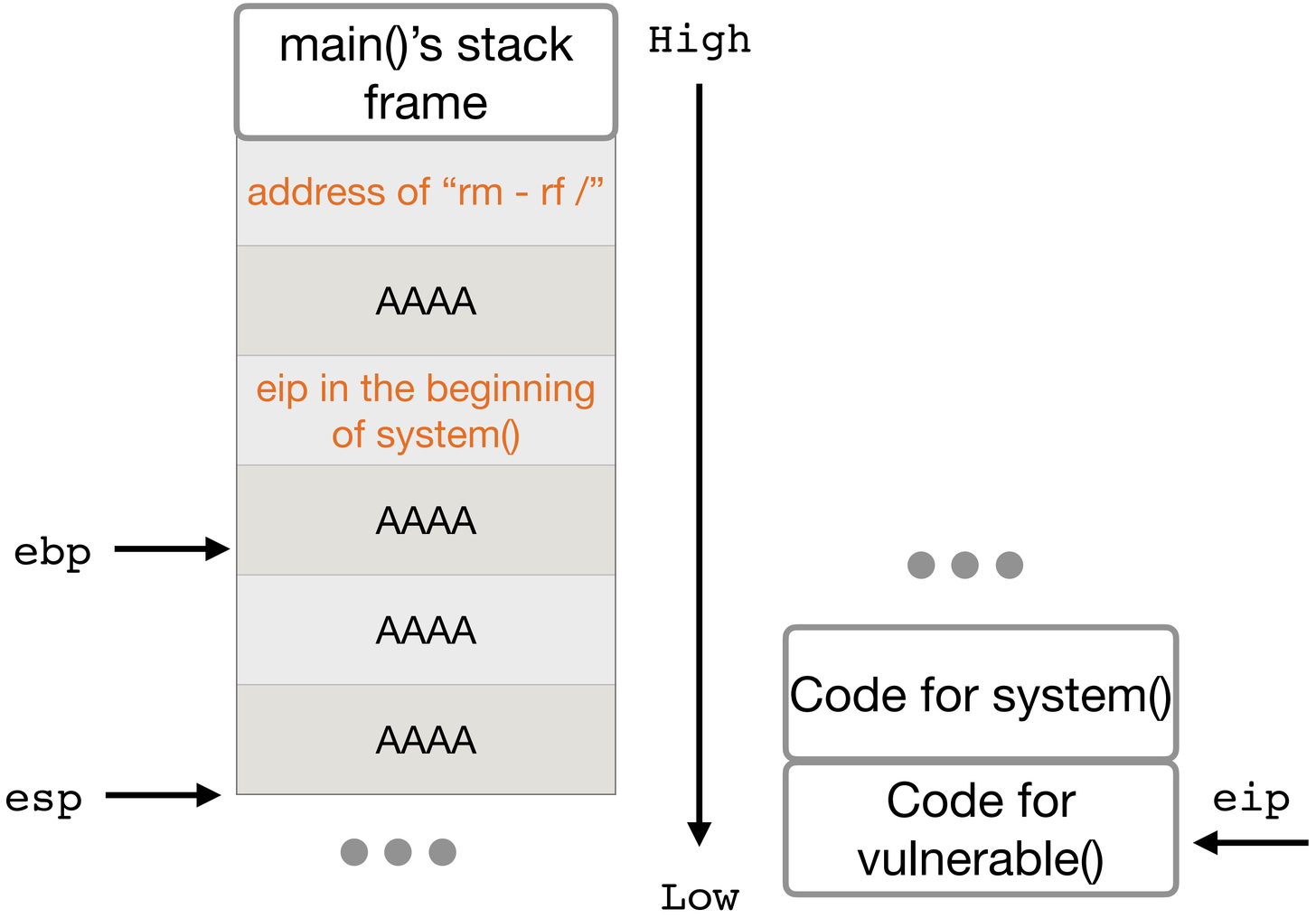
- If we find a buffer…
- Set up the stack like this!
- And return



Opportunity: Buffer

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

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

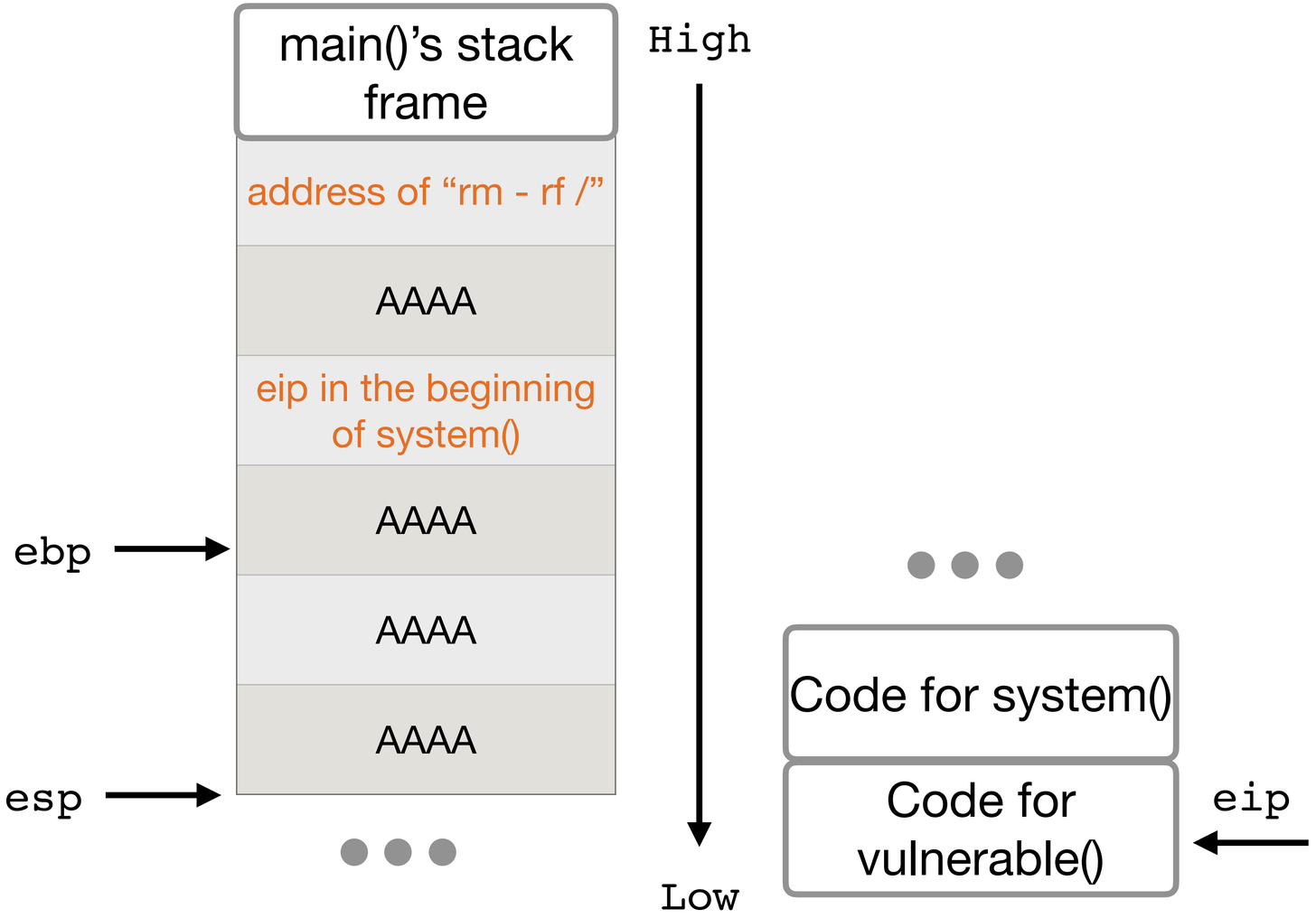
Attacker: find out saved eip, that is 4 bytes from ebp



Exercise: Go through leave return

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

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

- 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?

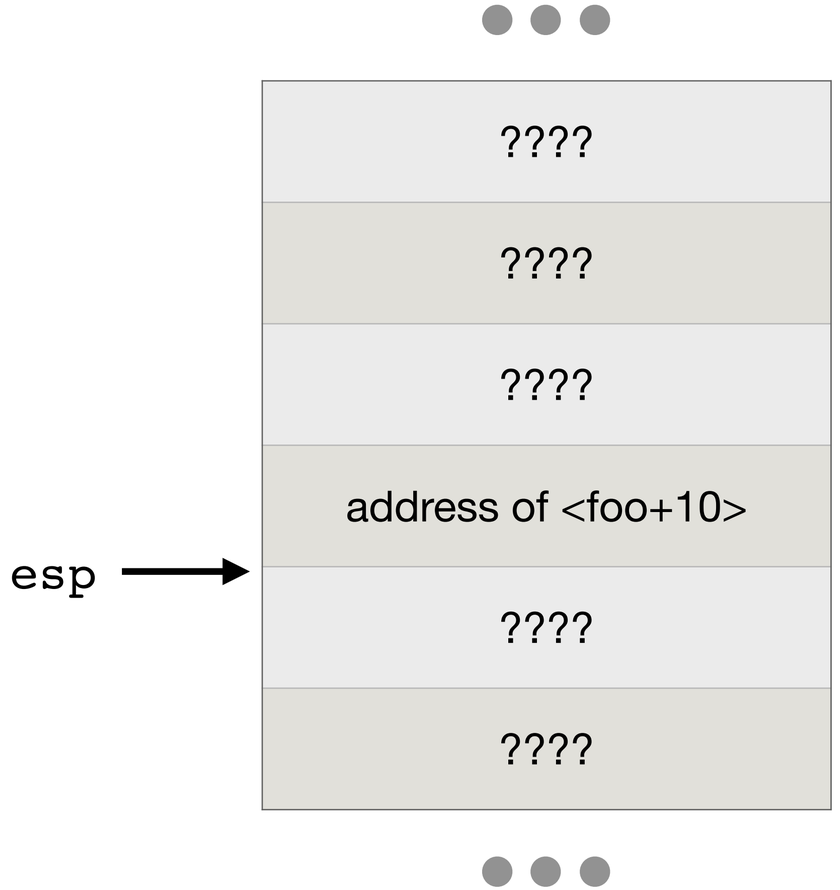
- The following two gadgets allow us to do
  - **<bar+25> movl $1, %eax**
  - **<bar+30> ret**
  - **<foo+10> xorl %eax, %ebx**
  - **<foo+12> ret**

ret: pop %eip

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

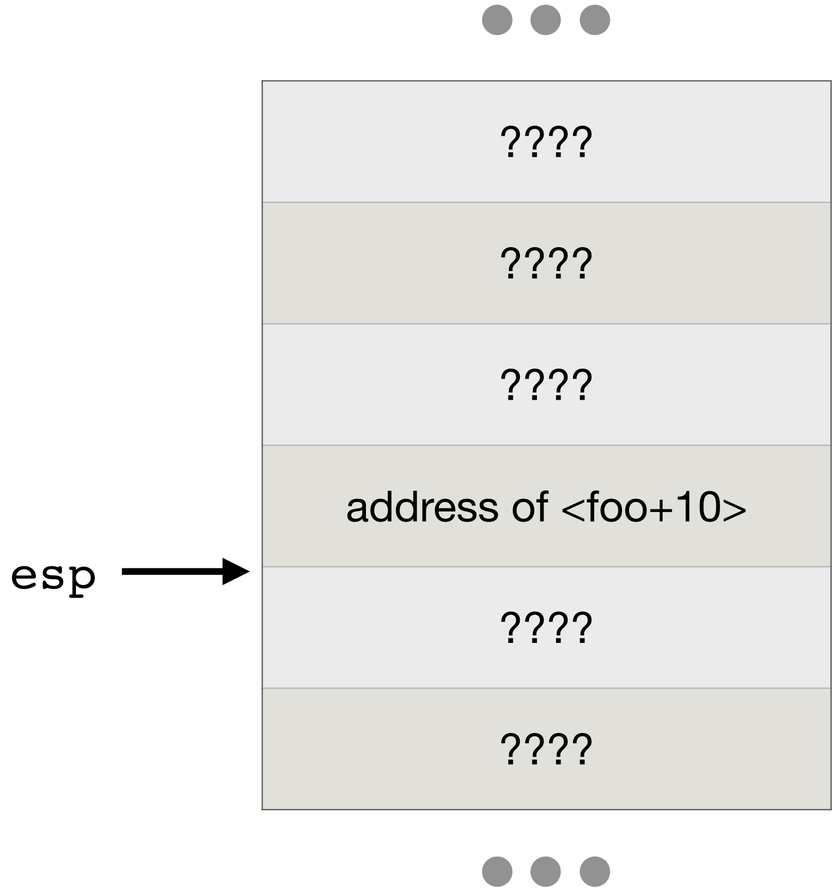
How to chain two gadgets together?

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



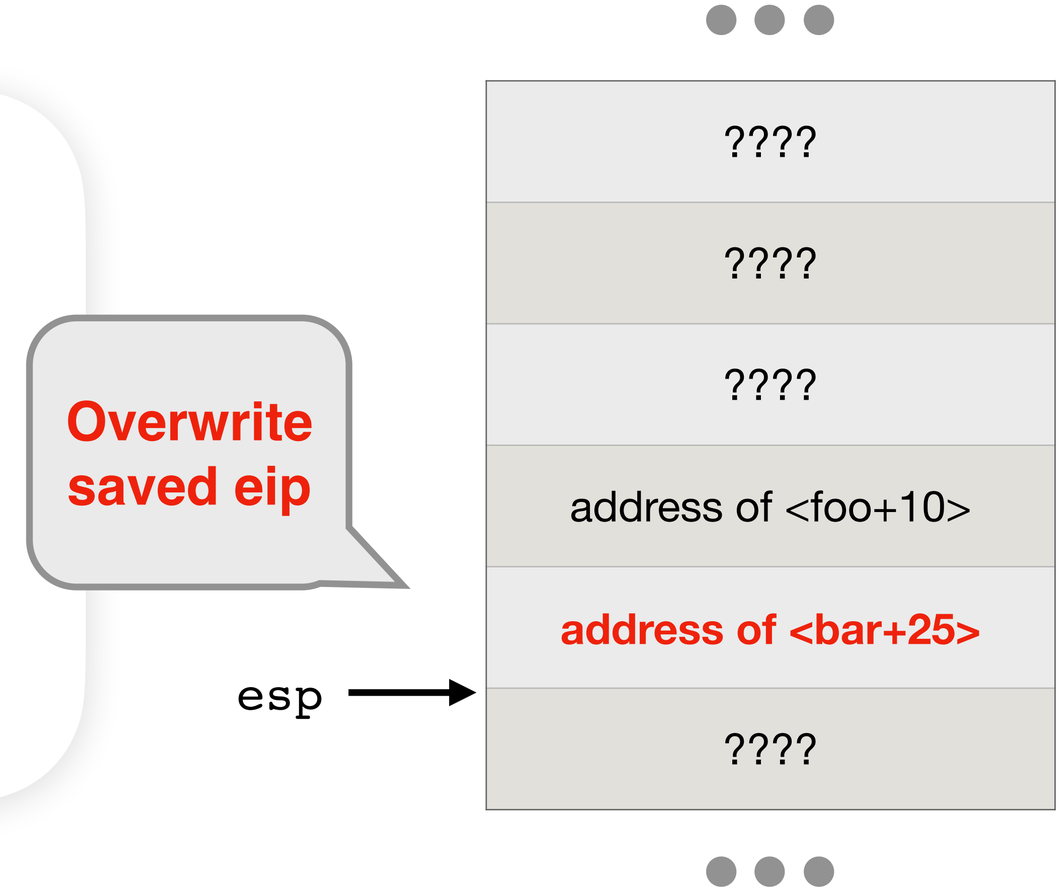
How to start executing?

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



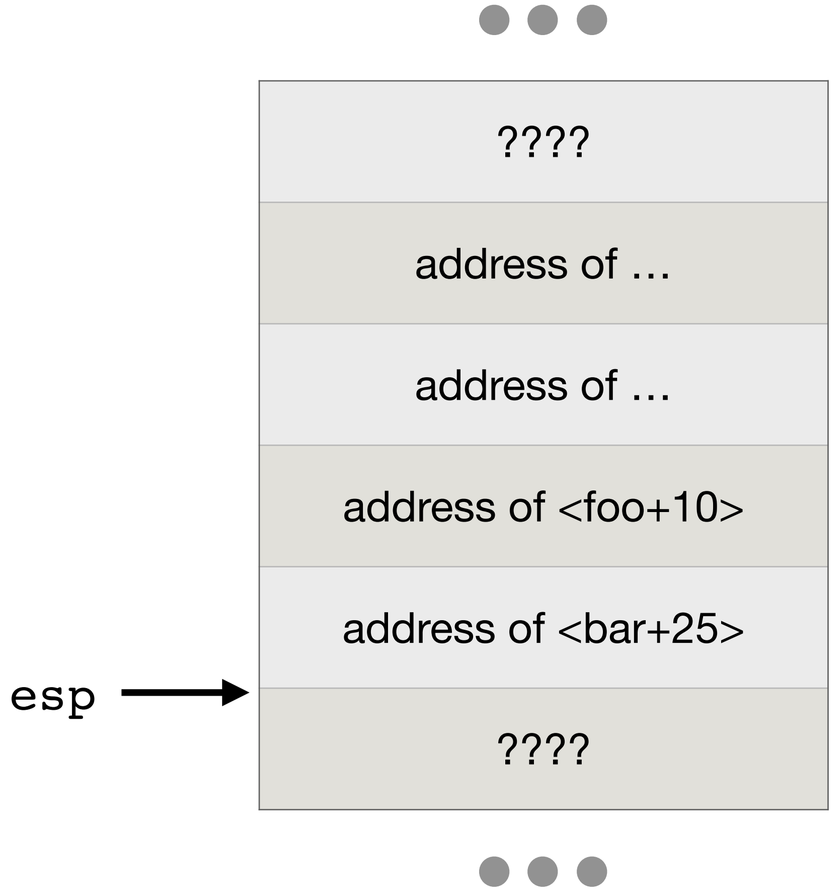
How to start executing?

- 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

Recall: Putting Together an Attack

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
4. Return from the function
5. Begin executing malicious shellcode

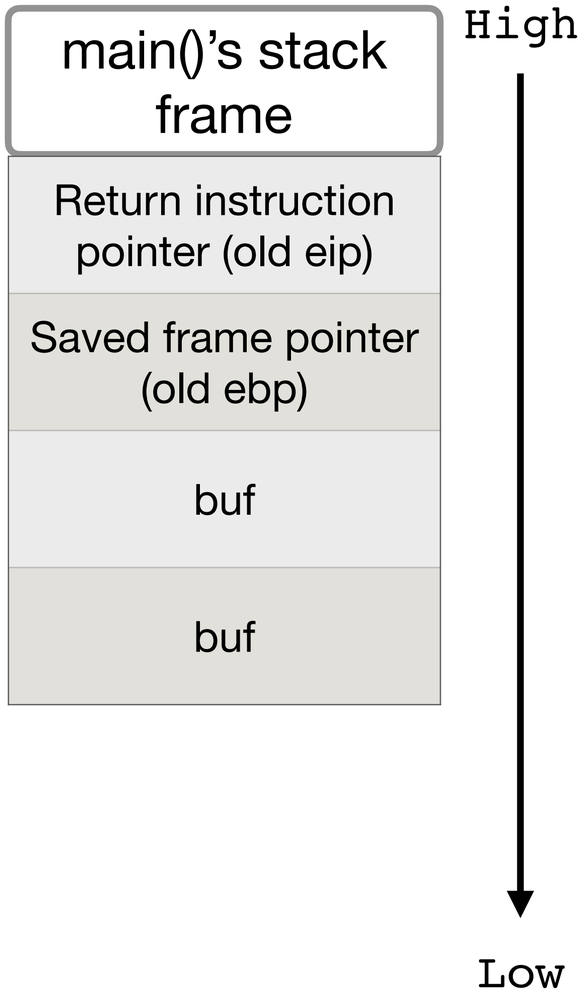
We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!

Stack Canaries

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

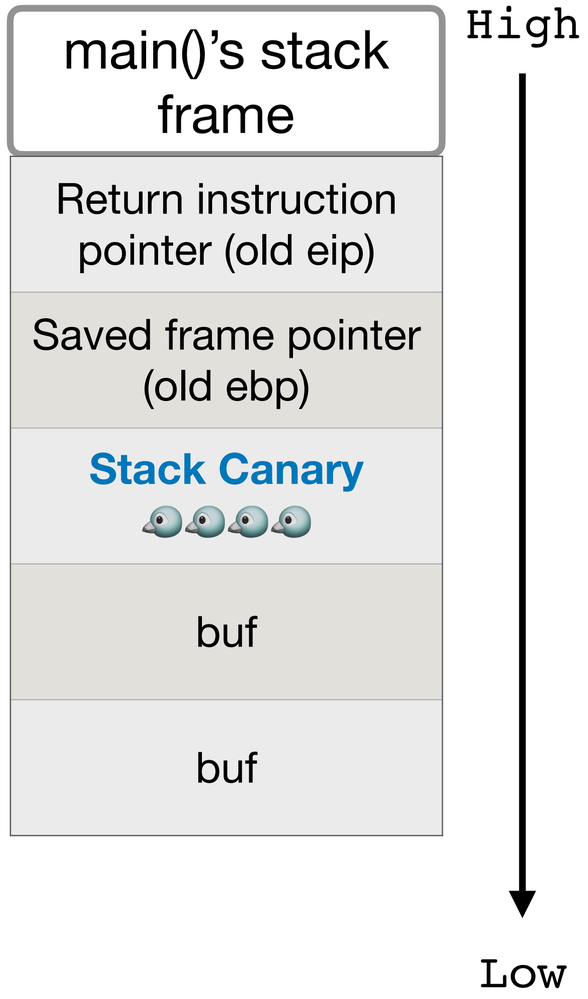
Regular Stack Example

```
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**

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
  - 1 try/sec, 100 days
- On 64-bit systems: 56 bits to guess
  - 1,000 tries/sec, 2 million years