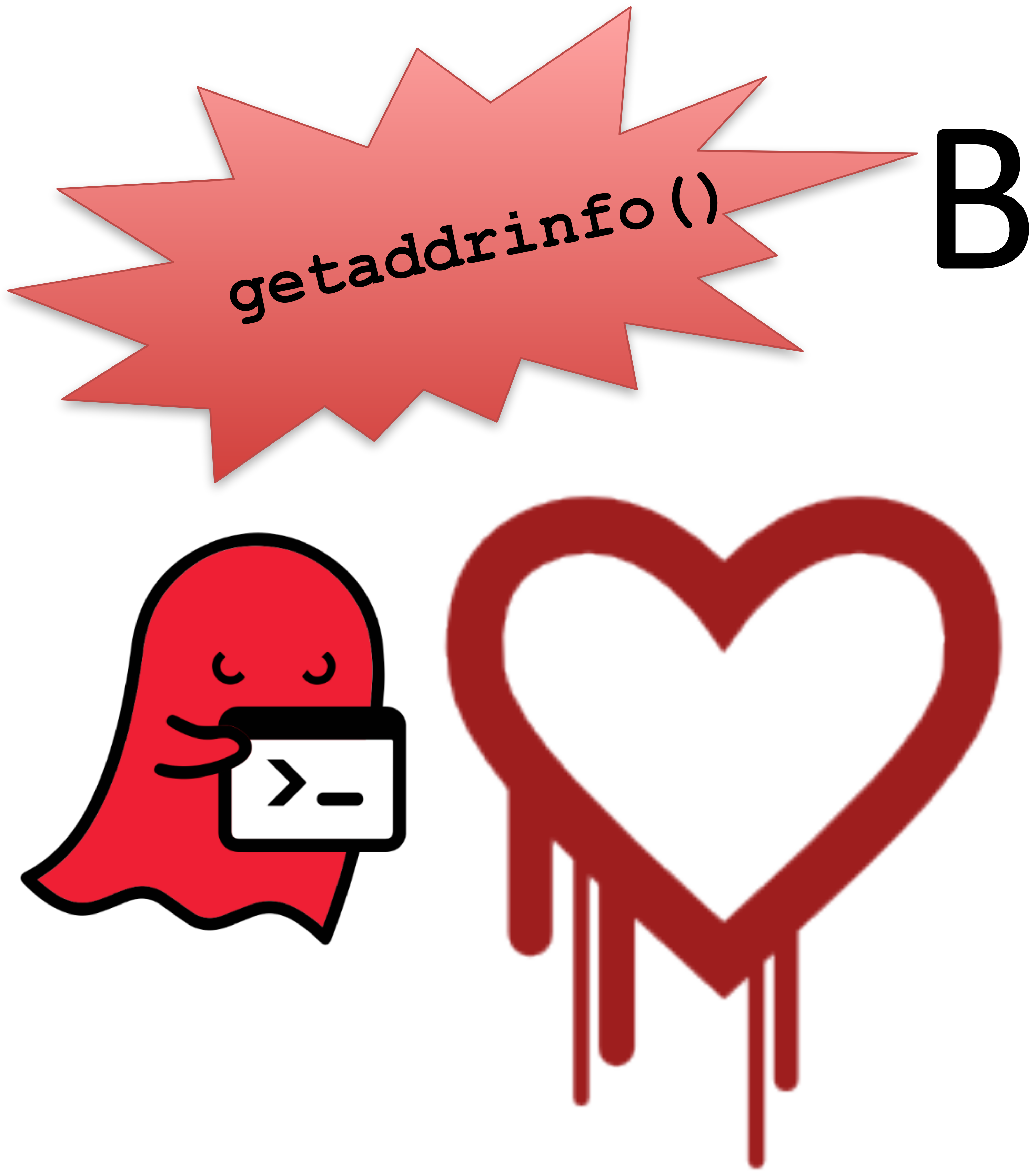


Buffer Overflows

Address space layout,
the stack discipline,
<u>+ C's lack of bounds-checking</u>
= HUGE PROBLEM

FEBRUARY 26, 2024

Statements of Support for Software Measurability and Memory Safety

Read the full report here

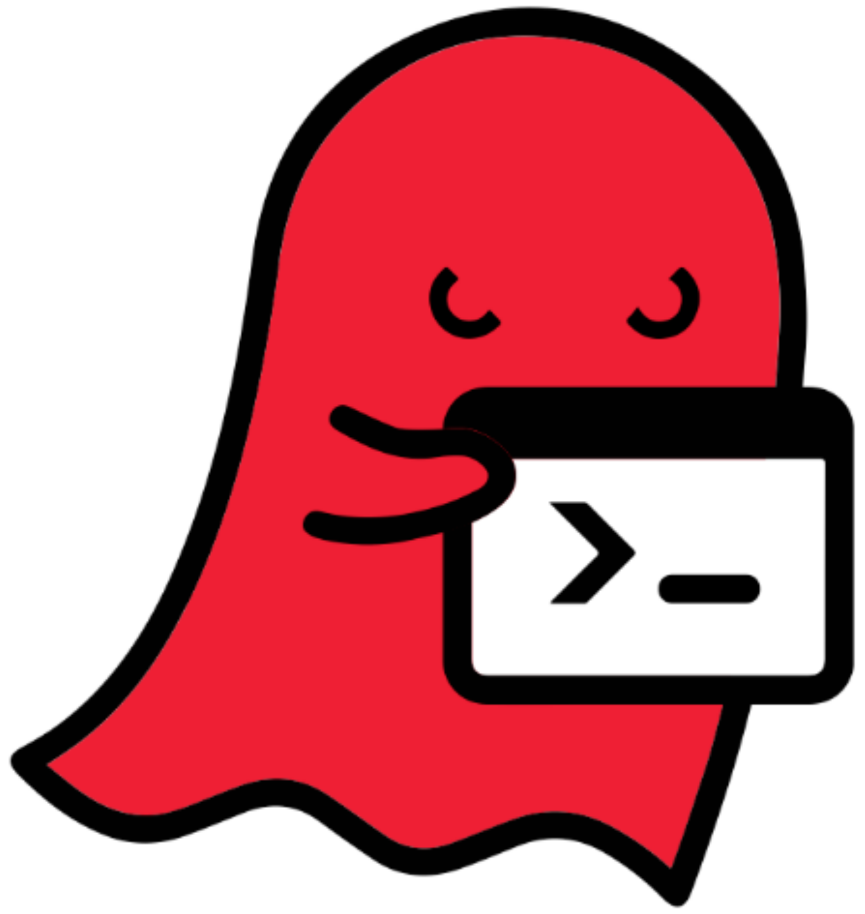
Read the fact sheet here

Today, the Office of the National Cyber Director released a new Technical Report titled *"Back to the Building Blocks: A Path Toward Secure and Measurable Software."* This report builds upon the President's National Cybersecurity Strategy, addressing the technical community to tackle undiscovered vulnerabilities that malicious actors can exploit.

Outline

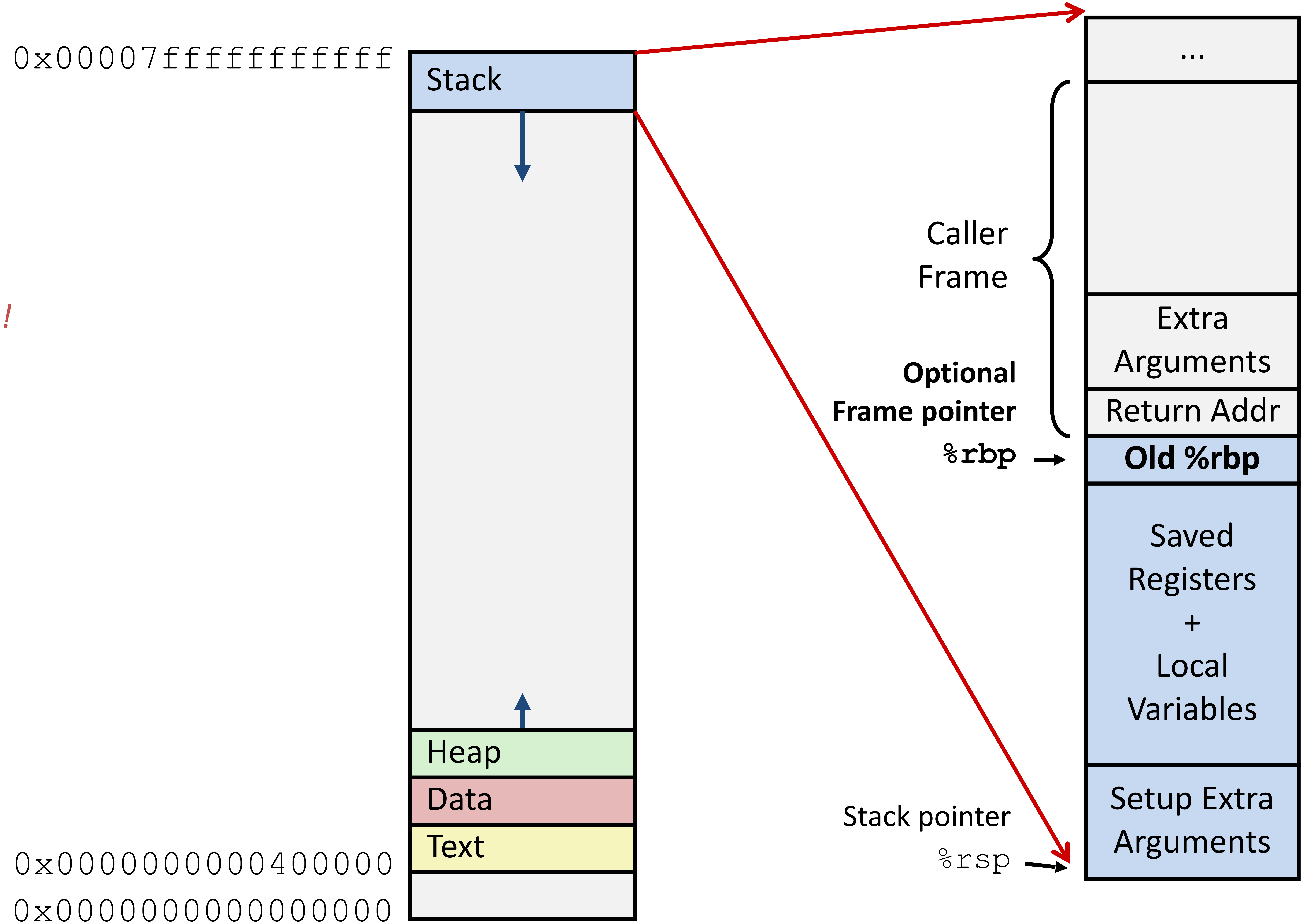
Goal: how the stack + lack of bounds checking make C program vulnerable to a certain (serious!) type of **security vulnerability**

- Understanding buffer overflows
  - Refresher on memory layout
  - C ibrary function: `gets`
  - `gets + echo` buffer overflow example
  - You: simplified security exploit example
- Buffer overflows in the wild
  - When this is a problem
  - Real-world implications



x86-64 Linux **memory layout**

Not drawn to scale!



C: String library code

C standard library function **`gets`**`()`

```c
/* Get string from stdin */
char* gets(char* dest) {
    int c = getchar();
    char* p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

pointer to start of an array

same as:

```c
*p = c;
p = p + 1;
```

What could go wrong when using this code?

What will happen if the input string from stdin is longer than the space allocated at dest?

0

```
/* Get string from stdin */
char* gets(char* dest) {
    int c = getchar();
    char* p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

- The C code will not compile. **(A)**
- An out-of-bounds exception. **(B)**
- It will only get the part of the input. **(C)**
- It depends on the memory layout. **(D)**
- None of the above **(E)**

What will happen if the input string from stdin is longer than the space allocated at dest?

0

```c
/* Get string from stdin */
char* gets(char* dest) {
    int c = getchar();
    char* p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

(A) The C code will not compile.

0%

(B) An out-of-bounds exception.

0%

(C) It will only get the part of the input.

0%

(D) It depends on the memory layout.

0%

(E) None of the above

0%

What will happen if the input string from stdin is longer than the space allocated at dest?

0

```
/* Get string from stdin */
char* gets(char* dest) {
    int c = getchar();
    char* p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

(A) The C code will not compile.

0%

(B) An out-of-bounds exception.

0%

(C) It will only get the part of the input.

0%

(D) It depends on the memory layout.

0%

(E) None of the above

0%

C: String library code

C standard library function **`gets()`**

```
/* Get string from stdin */
char* gets(char* dest) {
    int c = getchar();
    char* p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

pointer to start of an array

same as:

```
*p = c;
p = p + 1;
```

What could go wrong when using this code?

Same problem in many C library functions:

`strcpy`: Copies string of arbitrary length

`scanf`, **`fscanf`**, **`sscanf`**, when given **`%s`** conversion specification

C: Vulnerable buffer code using `gets(...)`

```
/* Echo Line */
void echo() {
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

These two lines of code introduce a vulnerability!

```
int main() {
    printf("Type a string:");
    echo();
    return 0;
}
```

```
$ ./bufdemo
Type a string:123
123
```

```
$ ./bufdemo
Type a string: 0123456789012345678901234
Segmentation Fault
```

```
$ ./bufdemo
Type a string: 012345678901234567890123
012345678901234567890123
```

Vulnerable buffer code using `gets`: disassembled x86

echo code

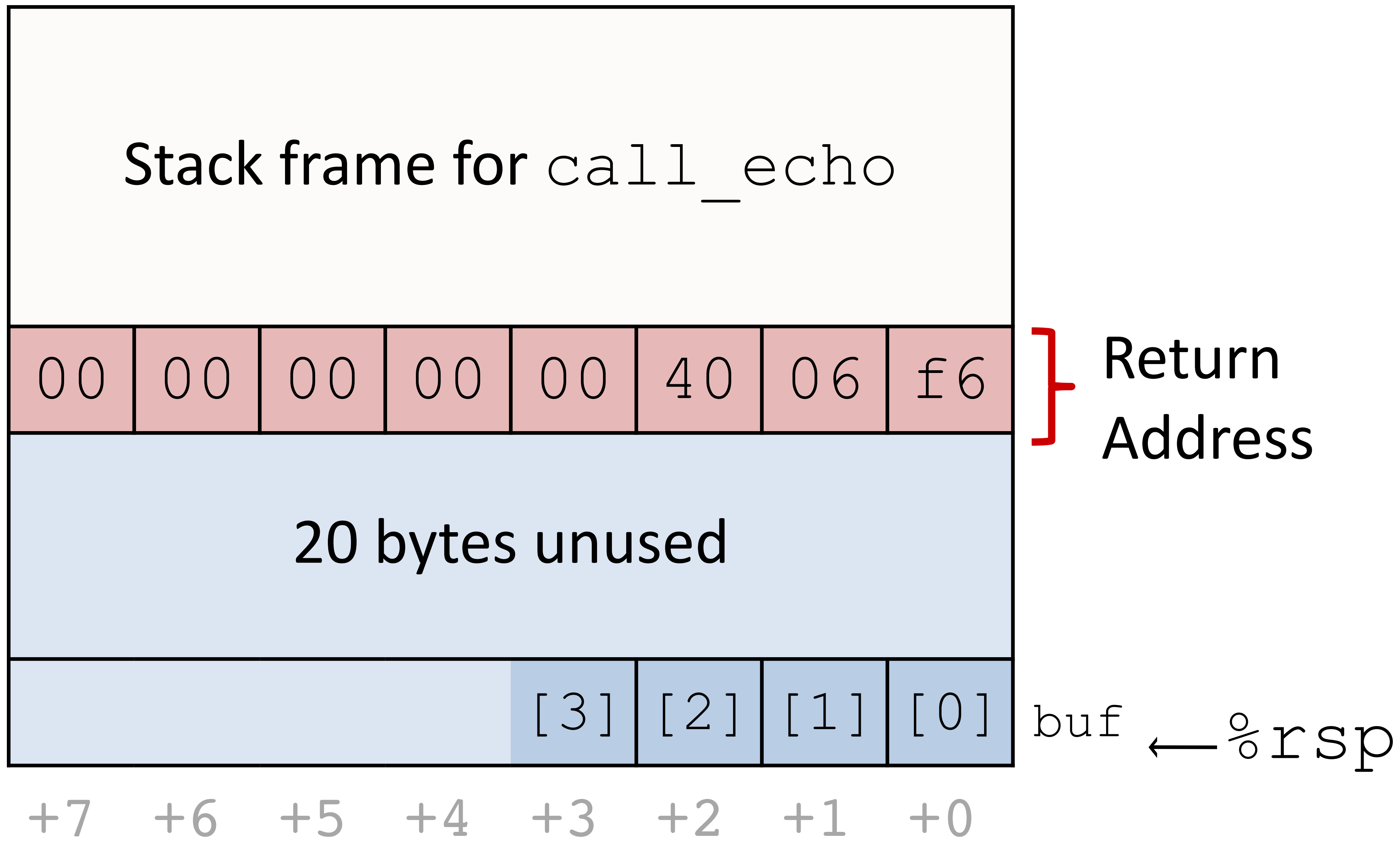
```
00000000004006cf <echo>:
 4006cf:  48 83 ec 18                    sub    $24,%rsp
 4006d3:  48 89 e7                       mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff                 callq  400680 <gets>
 4006db:  48 89 e7                       mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff                 callq  400520 <puts@plt>
 4006e3:  48 83 c4 18                    add    $24,%rsp
 4006e7:  c3                             retq
```

caller code

```
4006e8:   48 83 ec 08                    sub    $0x8,%rsp
 4006ec:  b8 00 00 00 00                 mov    $0x0,%eax
 4006f1:  e8 d9 ff ff ff                 callq  4006cf <echo>
 4006f6:  48 83 c4 08                    add    $0x8,%rsp
 4006fa:  c3                             retq
```

Buffer overflow example: before input

Before call to gets




```
void echo() {
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
 . . .
 4006f1:  callq   4006cf <echo>
 4006f6:  add     $0x8,%rsp
 . . .
```

Buffer overflow example: input #1

After call to gets

Stack frame for `call_echo`								
00	00	00	00	00	40	06	f6	Return Address
00	32	31	30	39	38	37	36	
35	34	33	32	31	30	39	38	
37	36	35	34	33	32	31	30	buf ⟵%rsp
+7	+6	+5	+4	+3	+2	+1	+0	

Null Terminator

```
void echo() {
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

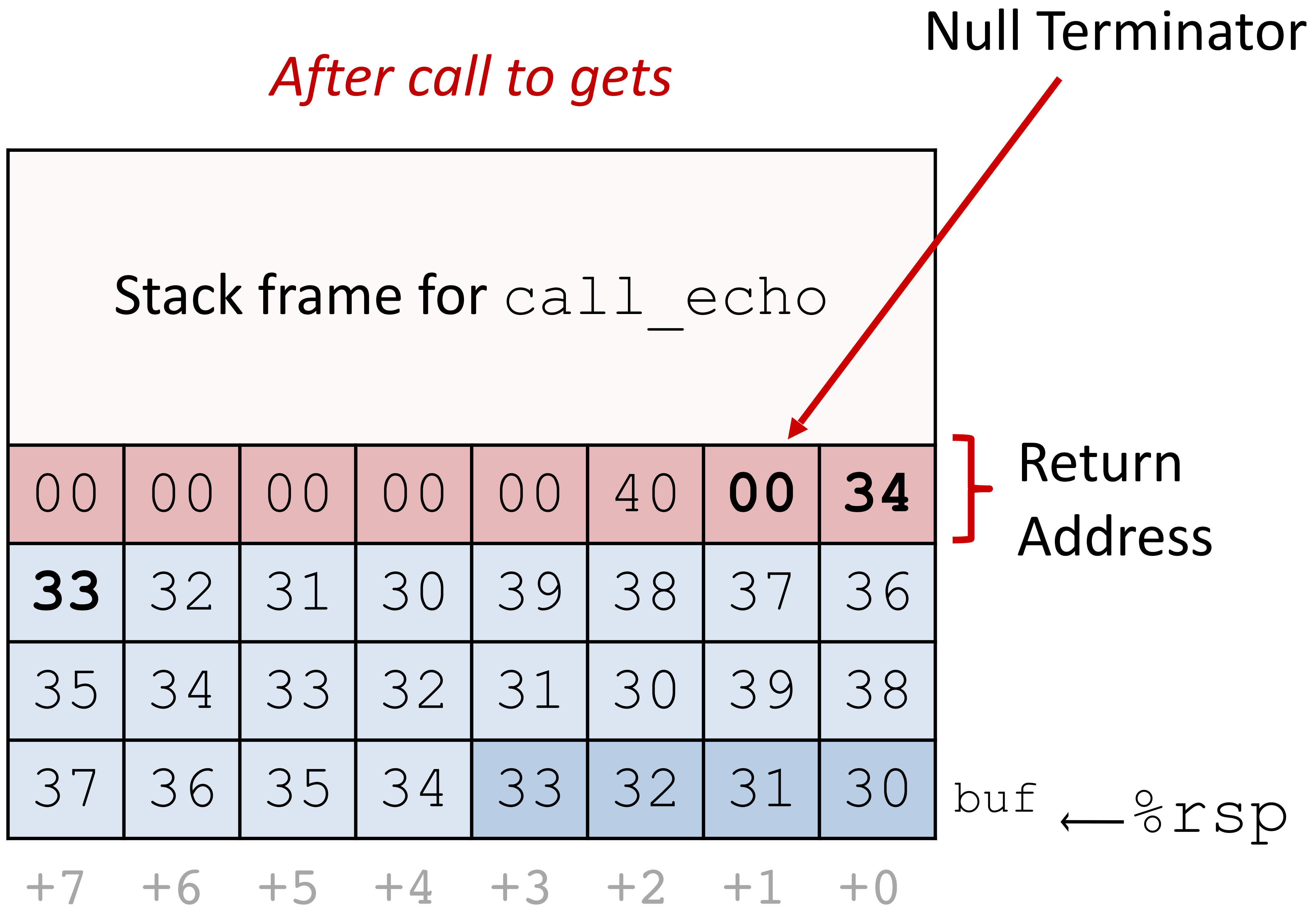
```
  . . .
  4006f1:  callq   4006cf <echo>
  4006f6:  add     $0x8,%rsp
  . . .
```

```
$ ./bufdemo
Type a string: 01234567890123456789012
01234567890123456789012
```

Overflowed buffer, but did not corrupt state

Buffer overflow example: input #2

After call to gets




```
void echo() {
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

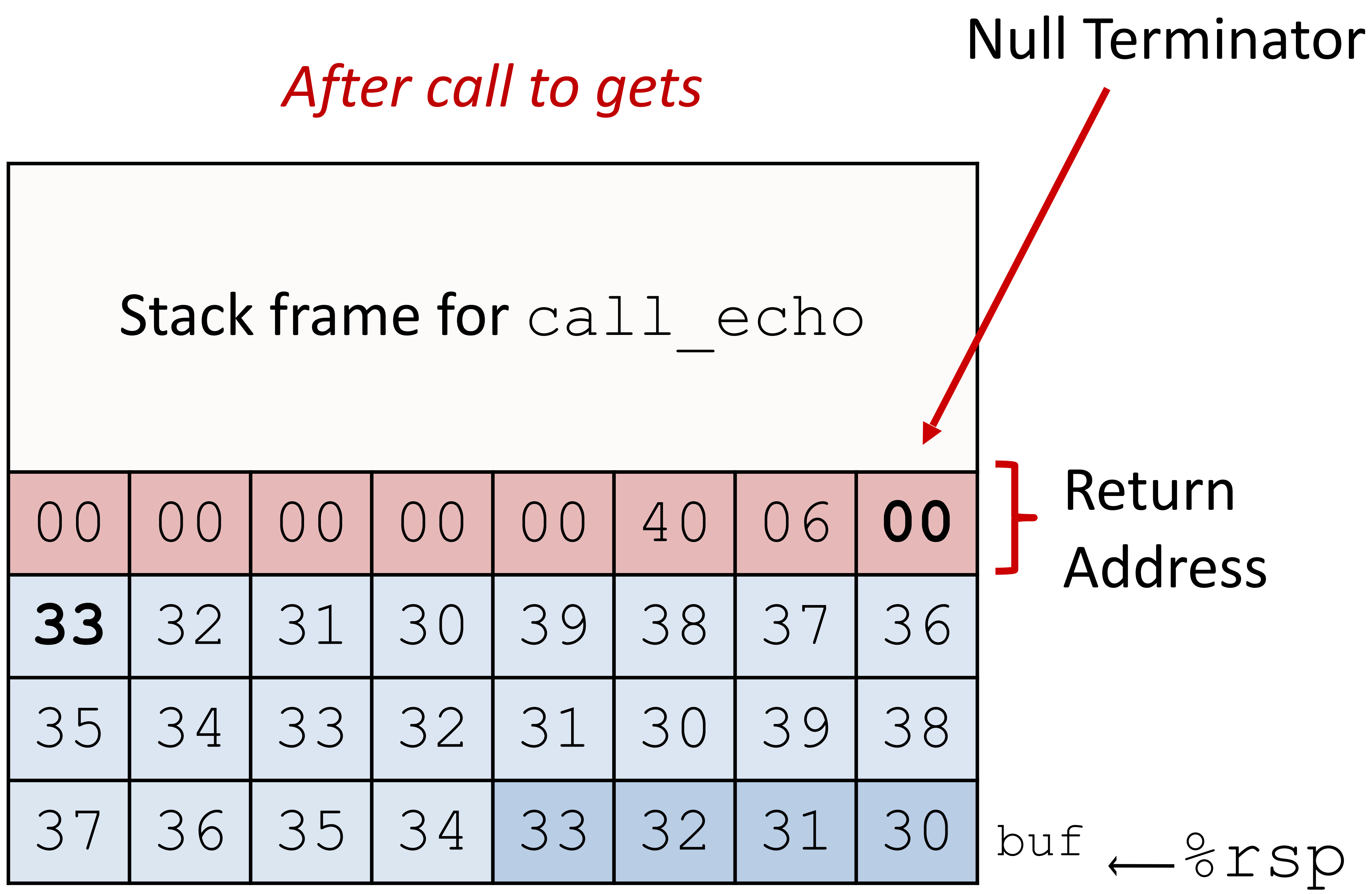
call_echo:

```
  . . .
  4006f1:  callq   4006cf <echo>
  4006f6:  add     $0x8,%rsp
  . . .
```

```
unix> ./bufdemo
Type a string: 01234567890123456789012**34**
Segmentation Fault
```

Overflowed buffer and corrupted return pointer

Buffer overflow example: input #3




```
void echo() {
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

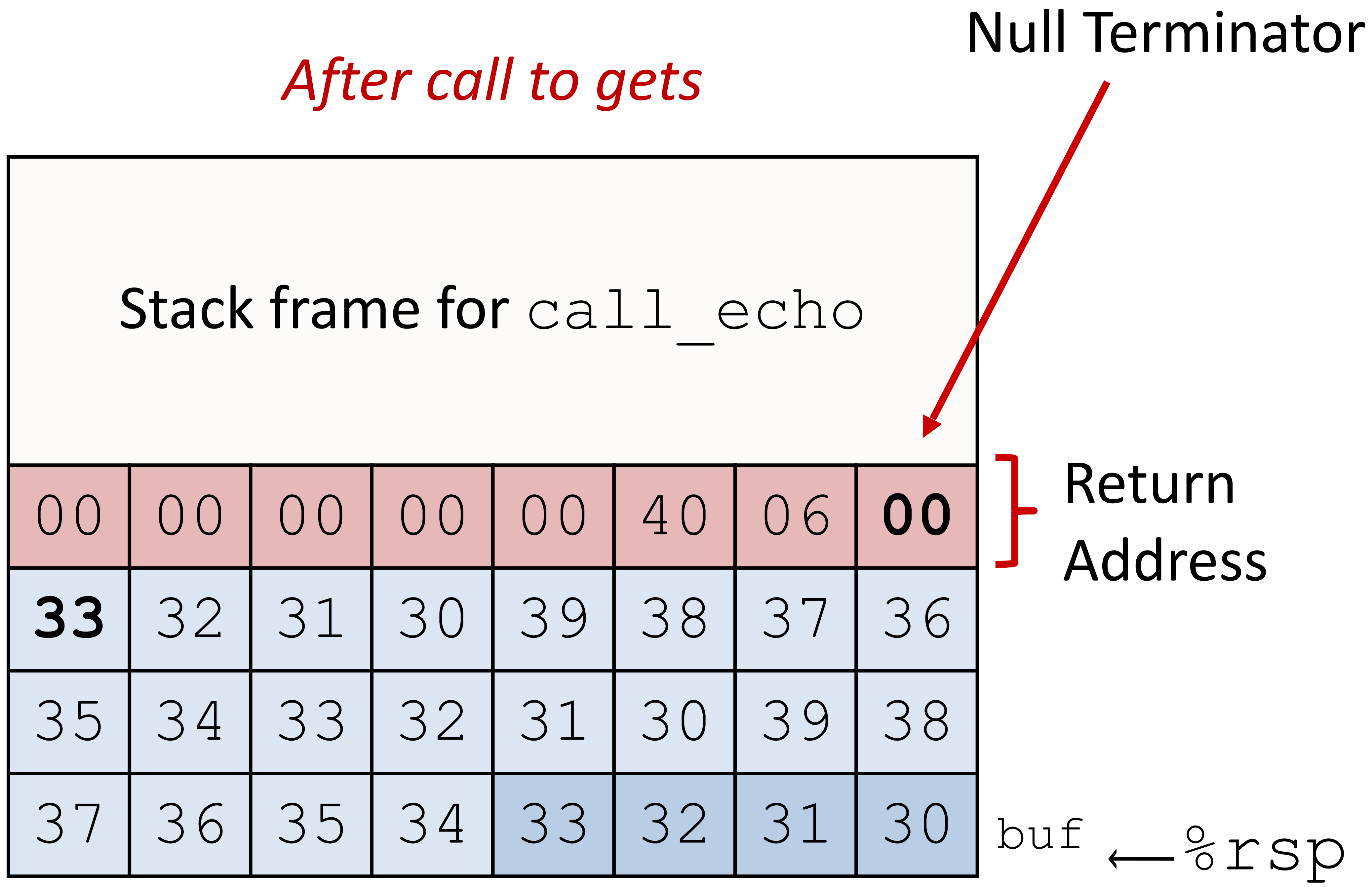
call_echo:

```
  . . .
  4006f1:  callq   4006cf <echo>
  4006f6:  add     $0x8,%rsp
  . . .
```

```
unix> ./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
```

Overflowed buffer, corrupted return pointer, but program seems to work?!

Buffer overflow example: input #3



Some other place in .text

```
   . . .
400600:  mov    %rsp,%rbp
400603:  mov    %rax,%rdx
400606:  shr    $0x3f,%rdx
40060a:  add    %rdx,%rax
40060d:  sar    %rax
400610:  jne    400614
400612:  pop    %rbp
400613:  retq
```

Works because: “Returns” to unrelated code, despite what the C code had!

Lots of things happen, without modifying critical state

Eventually executes `retq` back to `main`

Simplified exploit example (no padding)

```c
#include <stdio.h>

void delete_all_files() {
    // … users shouldn’t be able to call this
}

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

int main() {
    read_input();
}
```

```
read_input:
401126: subq    $8, %rsp
40112a: leaq    (%rsp), %rdi
40112f: movl    $0, %eax
401134: call    gets
401139: addq    $24, %rsp
40113d: ret
```

```
delete_all_files:
40003e: call    evil
        …
```

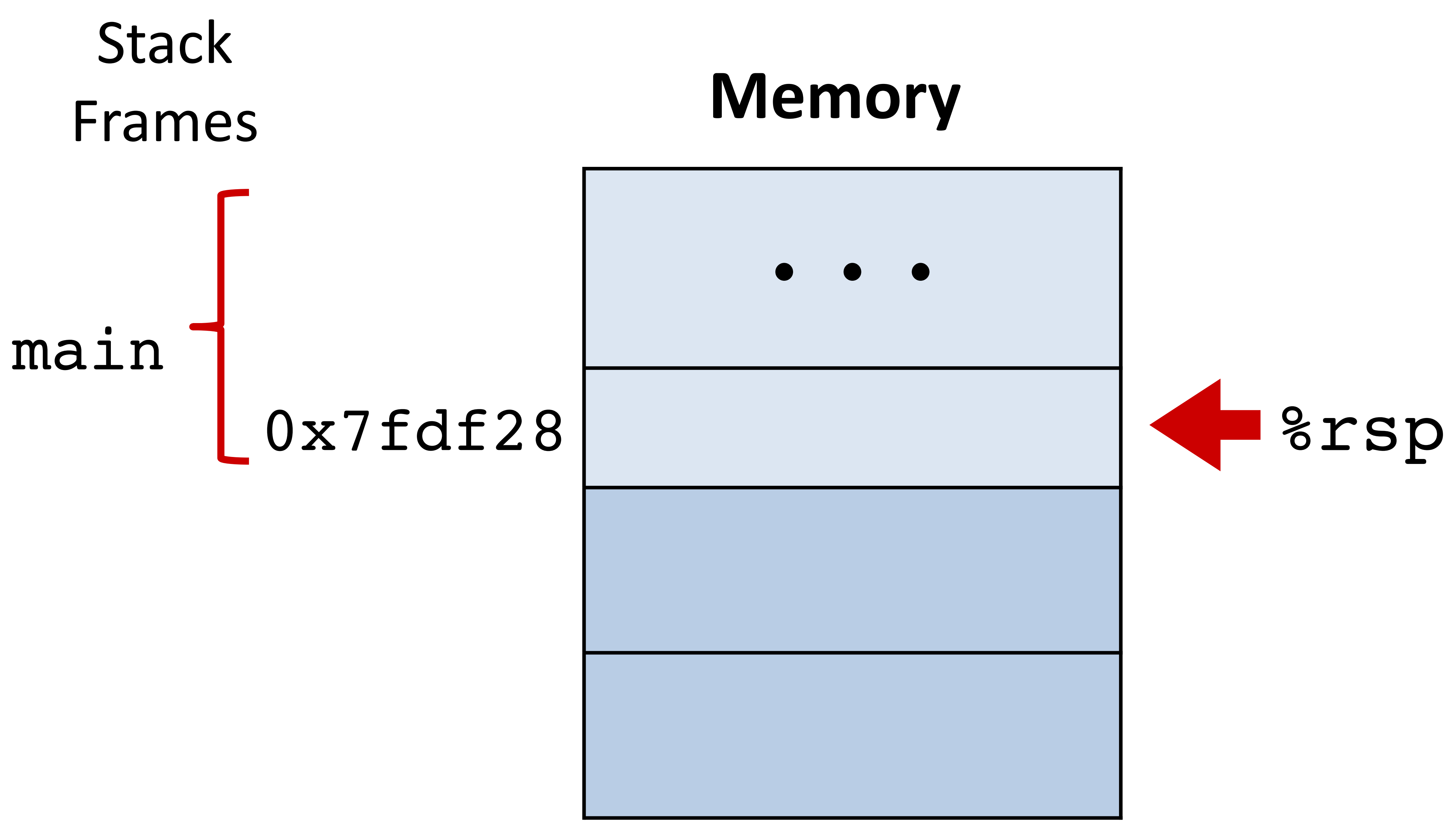
```
main:
            …       …
400048: call    read_input
40004d: addq    $8, %rsp
400051: ret
```

Simplified exploit example (no padding)

```
read_input:
401126: subq    $8, %rsp
40112a: leaq    (%rsp), %rdi
40112f: movl    $0, %eax
401134: call    gets
401139: addq    $24, %rsp
40113d: ret
```

```
delete_all_files:
40003e: call    evil
        …
```

```
main:
           …       …
400048: call    read_input
40004d: addq    $8, %rsp
400051: ret
```



Update the stack and registers diagram to the state at the red line

%rsp

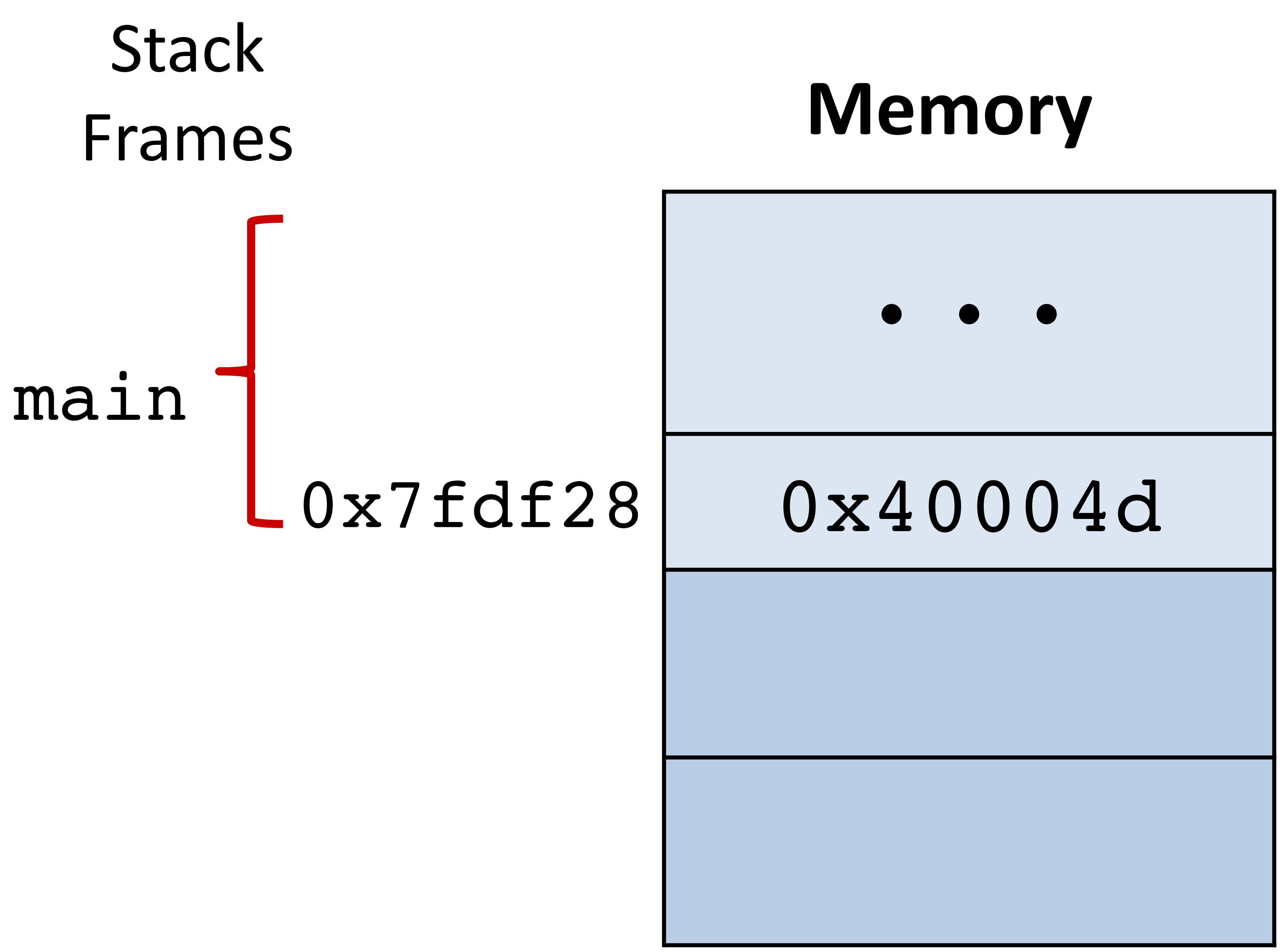
%rip

Simplified exploit example (no padding)

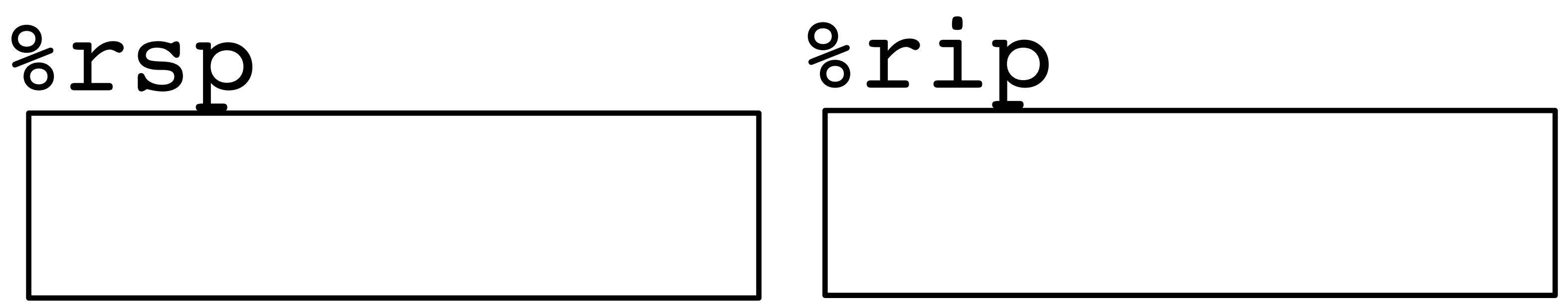
```
read_input:
401126: subq    $8, %rsp
40112a: leaq    (%rsp), %rdi
40112f: movl    $0, %eax
401134: call    gets
401139: addq    $24, %rsp
40113d: ret
```

```
delete_all_files:
40003e: call    evil
        …
```

```
main:
           …       …
400048: call    read_input
40004d: addq    $8, %rsp
400051: ret
```



Update the stack and registers diagram to the state at the red line

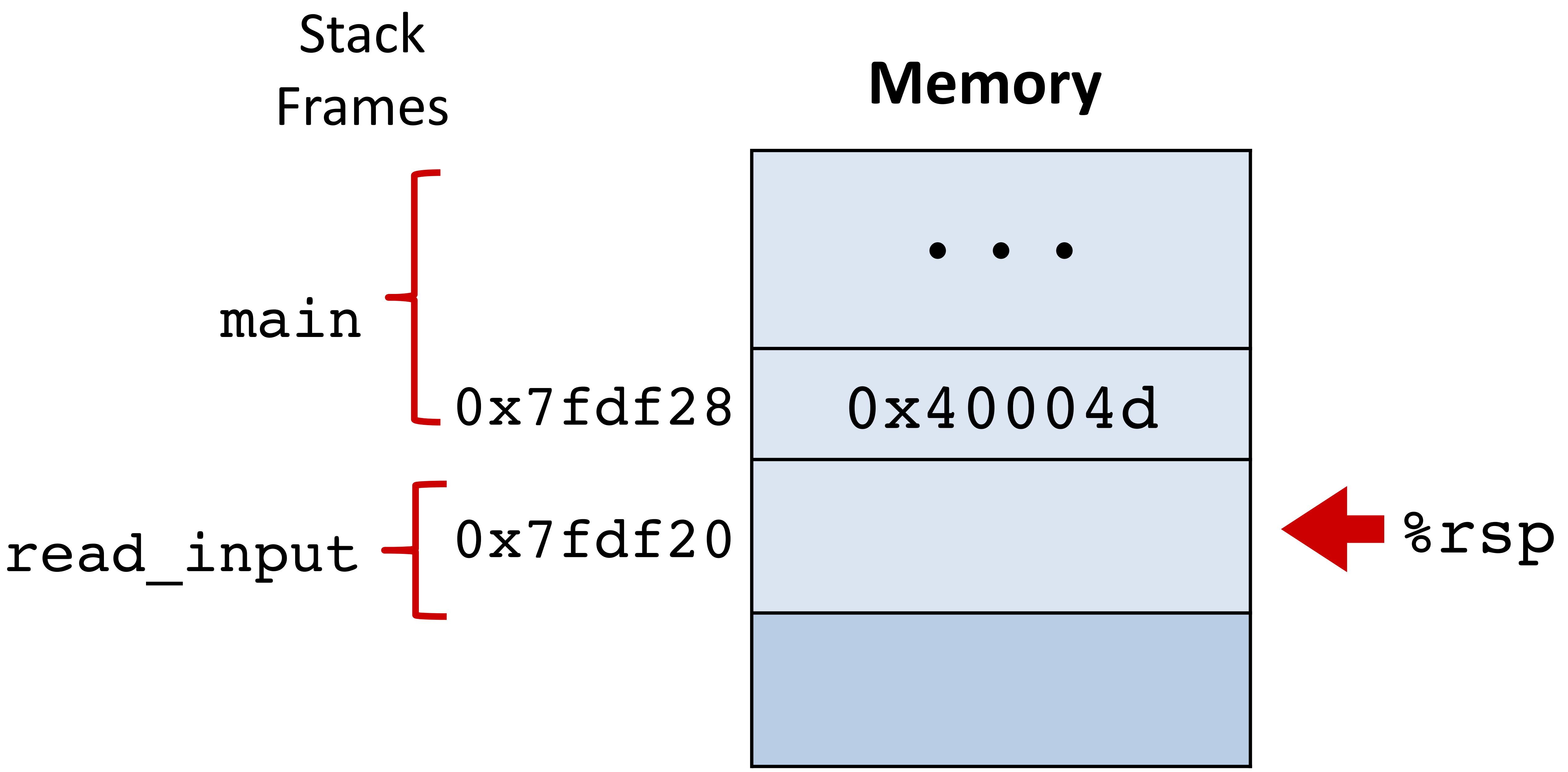


Simplified exploit example (no padding)

```
read_input:
401126: subq    $8, %rsp
40112a: leaq    (%rsp), %rdi
40112f: movl    $0, %eax
401134: call    gets
401139: addq    $24, %rsp
40113d: ret
```

```
delete_all_files:
40003e: call    evil
        …
```

```
main:
           …       …
400048: call    read_input
40004d: addq    $8, %rsp
400051: ret
```



Discuss: what address would we want to appear, and where, to have our exploit delete all files?

Draw out the bytes (in hex) of the exploit string.

%rsp

%rip

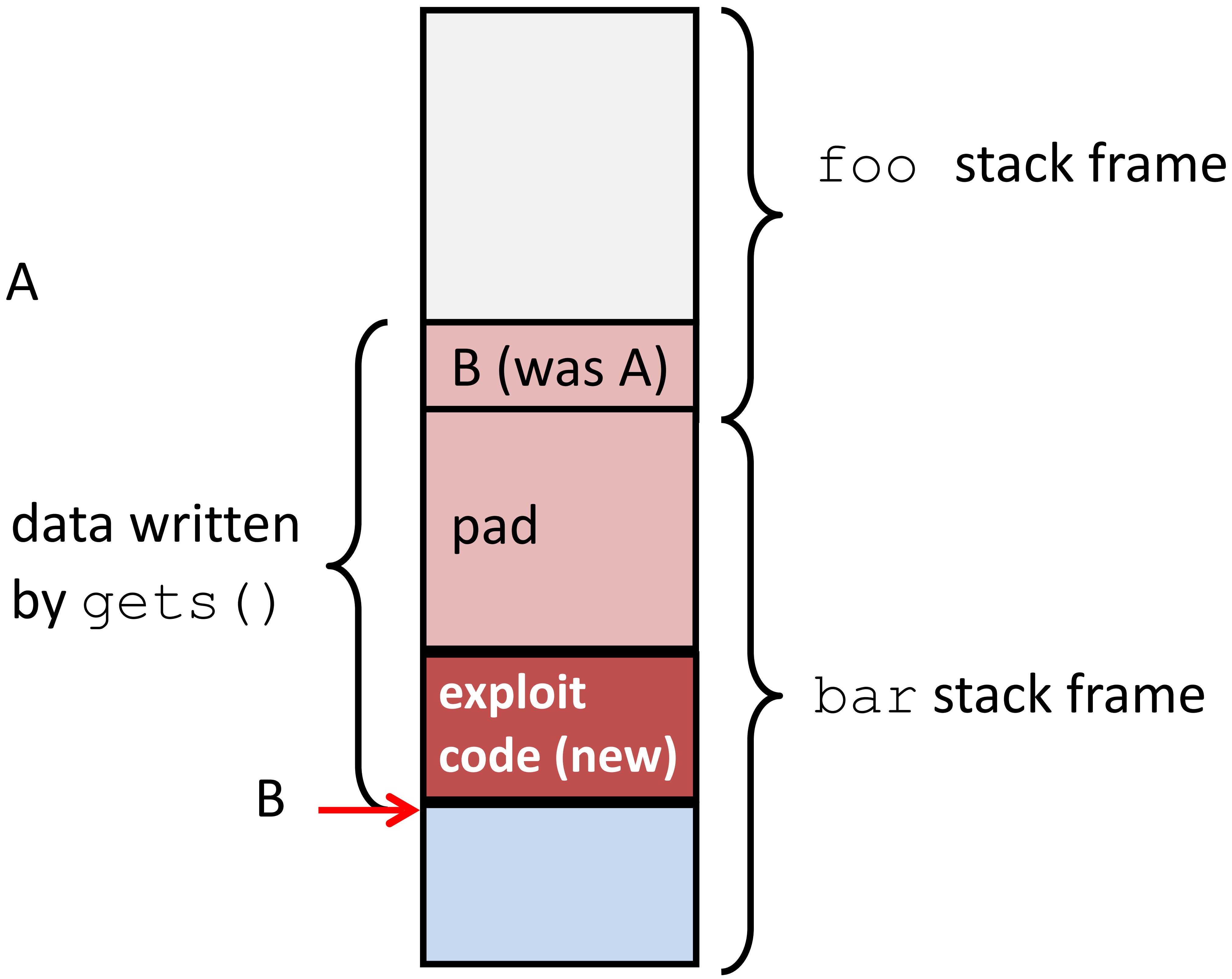
Exploiting buffer overflows: arbitrary code

```
void foo(){
  bar();
  ...
}
```

return address A

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

Stack after call to `gets()`

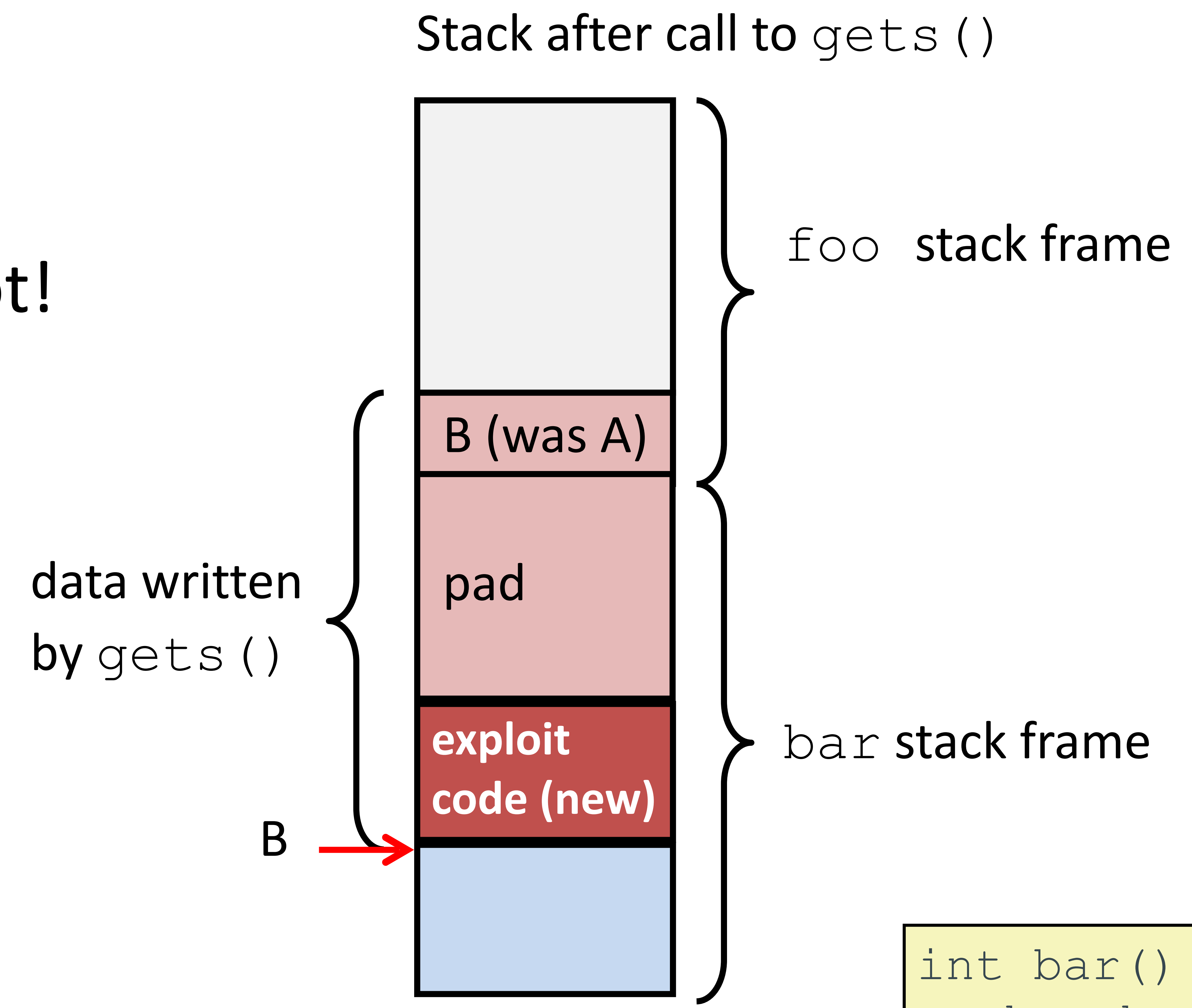


Exploiting buffer overflows: when is this a problem?

We could construct x86 code to mess up our own programs call stack

But, we trust our own code to not!

The problem: allowing
user input (untrusted source)
to potentially corrupt the stack




```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

Combination of: untrusted input, code that does not enforce bounds

`gets(input); strcpy(input, …); scanf(input, …); etc`

Exploits in the wild

Buffer overflow bugs allow remote attackers to execute arbitrary code on machines running vulnerable software.

1988: Internet worm

Early versions of the finger server daemon (fingerd) used **`gets()`** to read the argument sent by the client:

```
finger somebody@cs.wellesley.edu
```

commandline facebook of the 80s!

Attack by sending phony argument:

`finger "exploit-code padding new-return-address"`

...

Still happening

"Ghost:" 2015

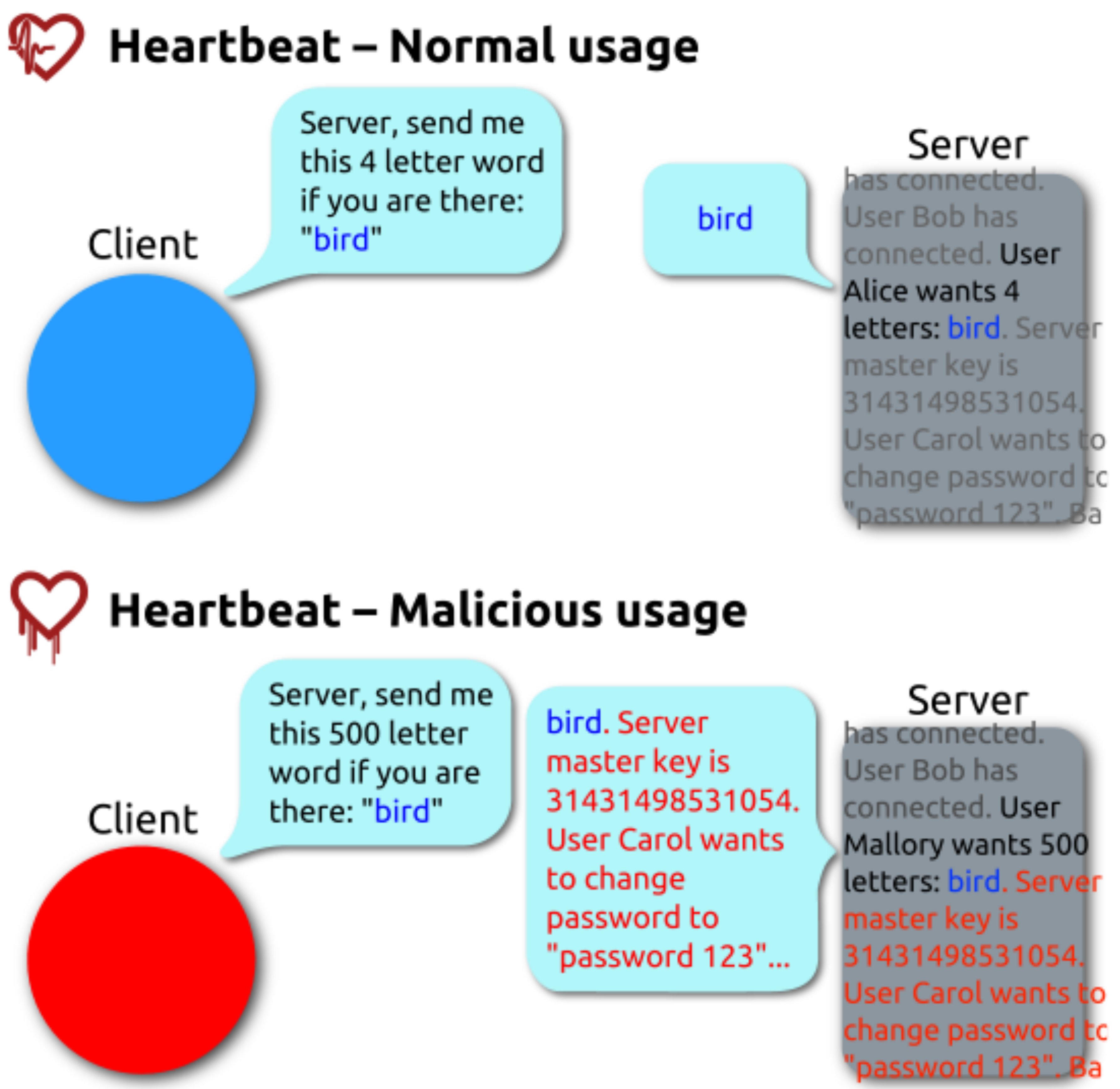
`gethostname()`

`getaddrinfo()`
Feb. 2016

Heartbleed (2014)

optional

- Buffer over-read in OpenSSL
  - Widely used encryption library (https)
- “Heartbeat” packet
  - Specifies length of message
  - Server echoes that much back
  - Library just “trusted” this length
  - Allowed attackers to read contents of memory anywhere they wanted
- ~17% of Internet affected
  - “Catastrophic”
  - Github, Yahoo,
Stack Overflow, Amazon AWS, ...



Avoiding overrun vulnerabilities

1. Use a memory-safe language (not C)!

2. If you have to use C, use library functions that limit string lengths.

   `fgets` instead of **`gets`**

```
/* Echo Line */
void echo() {
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

   `strncpy` instead of **`strcpy`**

   Don't use **`scanf`** with **`%s`** conversion specification

   - Use **`fgets`** to read the string
   - Or use **`%ns`** where **`n`** is a suitable integer

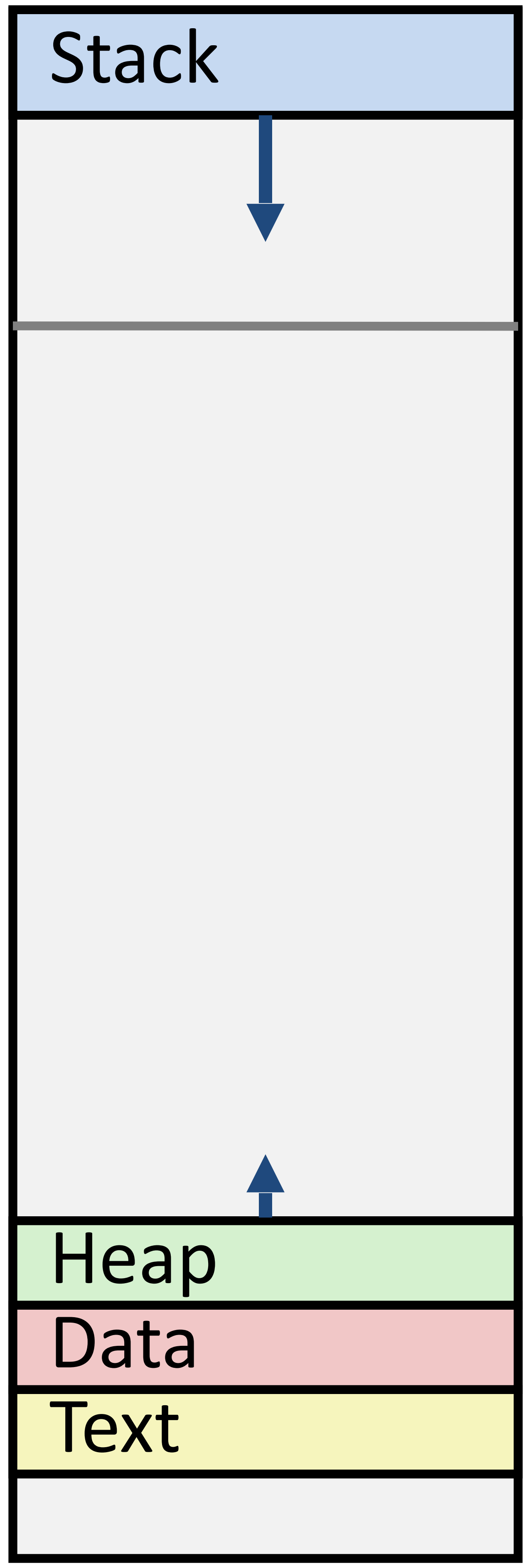
Other ideas?

System-level protections

Available in modern OSs/compilers/hardware

(We disabled these for buffer assignment.)

not drawn to scale



1. Randomize stack base, maybe frame padding

2. Detect stack corruption

   save and check stack "canary" values

3. Non-executable memory segments

   stack, heap, data, … everything except text

   hardware support

Helpful, not foolproof!

   Return-oriented programming, over-reads, etc.