CS 241: Systems Programming
Lecture 34. Control Flow Hijacking

Fall 2025

Stephen Checkoway

Slides adapted from Boneh, Miller, Bailey, and Brumley

Today’s Class

An attack that takes advantage of memory unsafety in C (and an example of why languages like Rust are so important)

Control Flow Hijacking - Terms

Control flow is the order in which program instructions are executed

The instruction pointer (also known as the program counter) is a register that stores the address of the next instruction to execute

- All of the instructions in your program are loaded into memory by the loader when you start running it
- When the CPU is ready to execute the next instruction of your program, it will fetch the instruction stored at the address in the instruction pointer

Control Flow Hijacks

Hijacks happen when an attacker gains control of the instruction pointer

Once the attacker has control of the instruction pointer, they can execute any code they want!

- They can put an address that points to arbitrary code in the instruction pointer

Two common methods for hijacking:

- Buffer overflow
- Format string attacks

Stack frames

Variables live in a region of memory called the stack

The stack is organized into **frames**

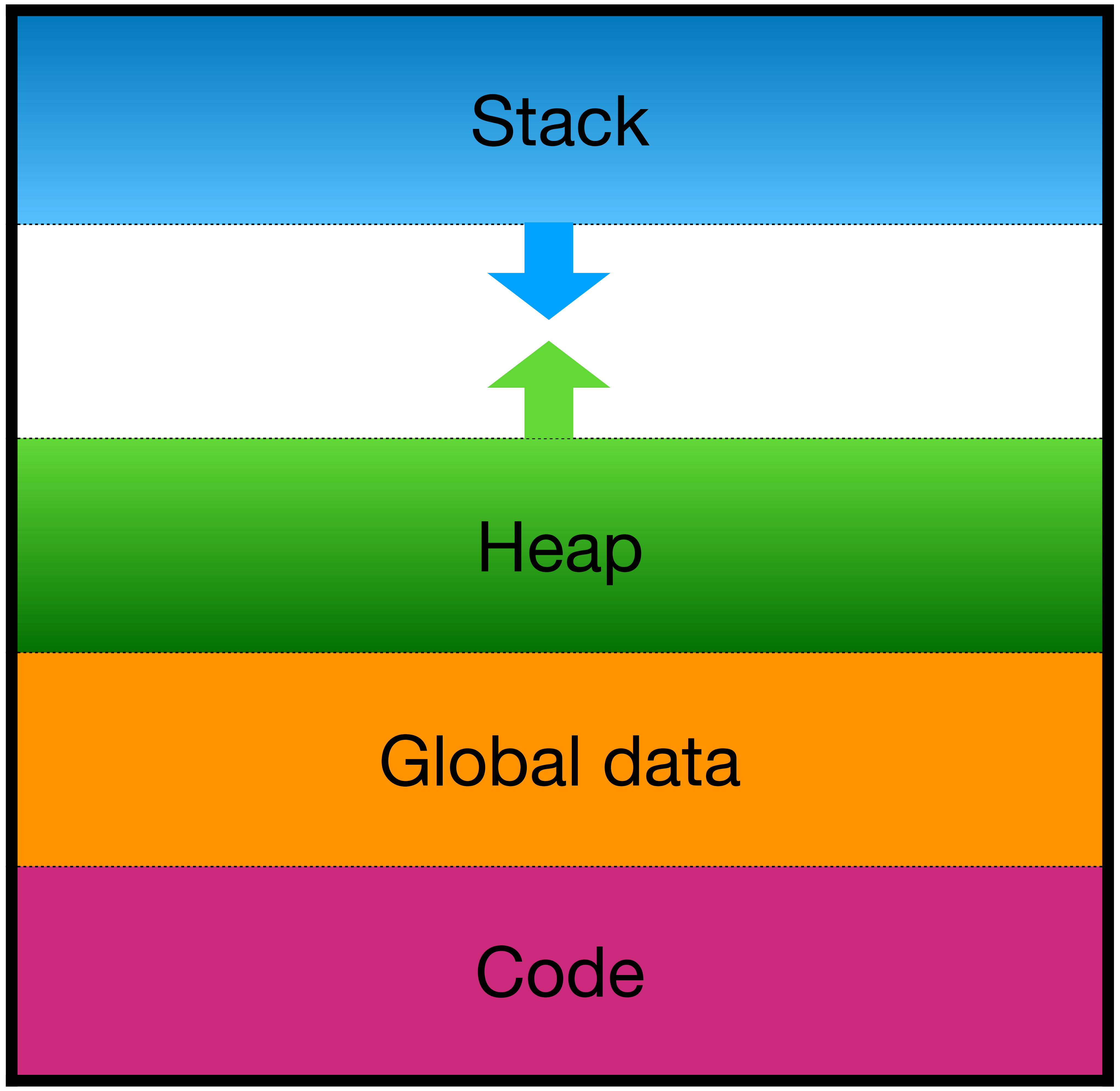
Local variables in functions live in a stack frame

Each function that is called pushes a new frame onto the stack

Each function that returns pops its stack frame off the stack

Memory layout (simplified)

High memory address



Low memory address

Stack grows down

Heap grows up

Global data and
Code are fixed size

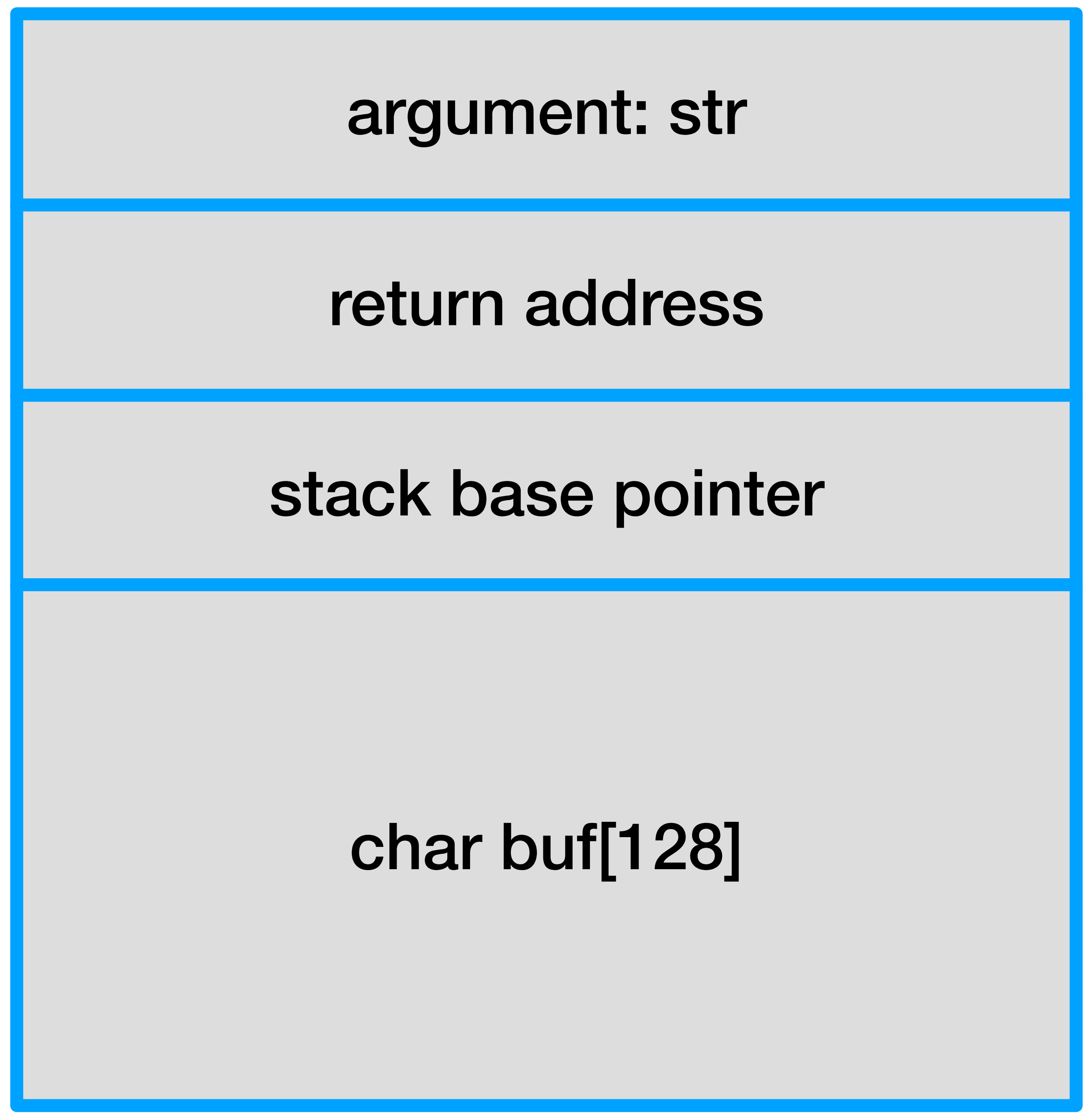
Stack example Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

Return address: the address of the next instruction to execute once the function returns

Stack base pointer: marks the start (base) of a function's stack frame

The stack:



What problems could happen with this code? Take a minute to think and select A when you have an answer.

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

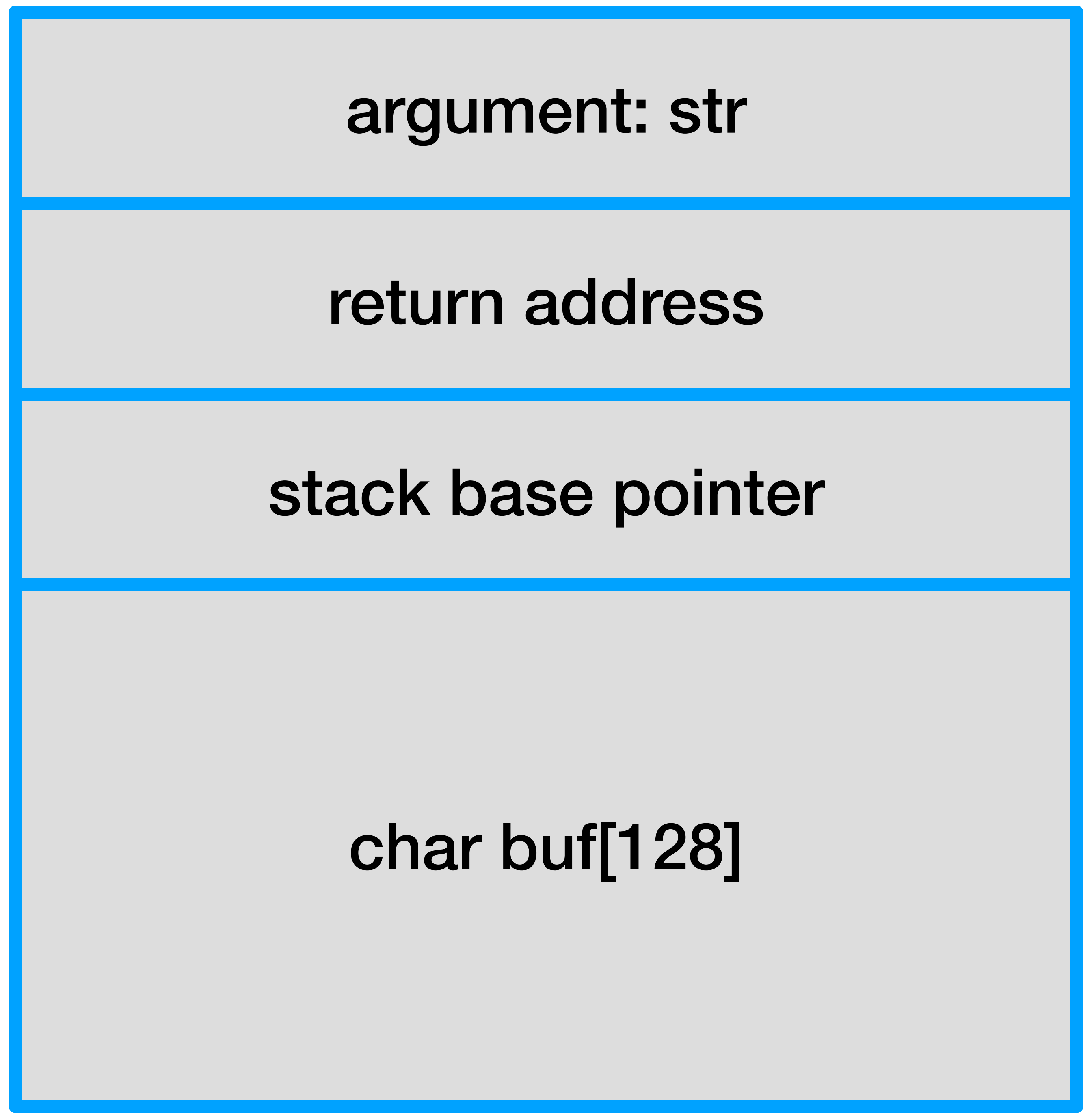
A. Select A when you have an answer

Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

We don’t check the size of str!!!

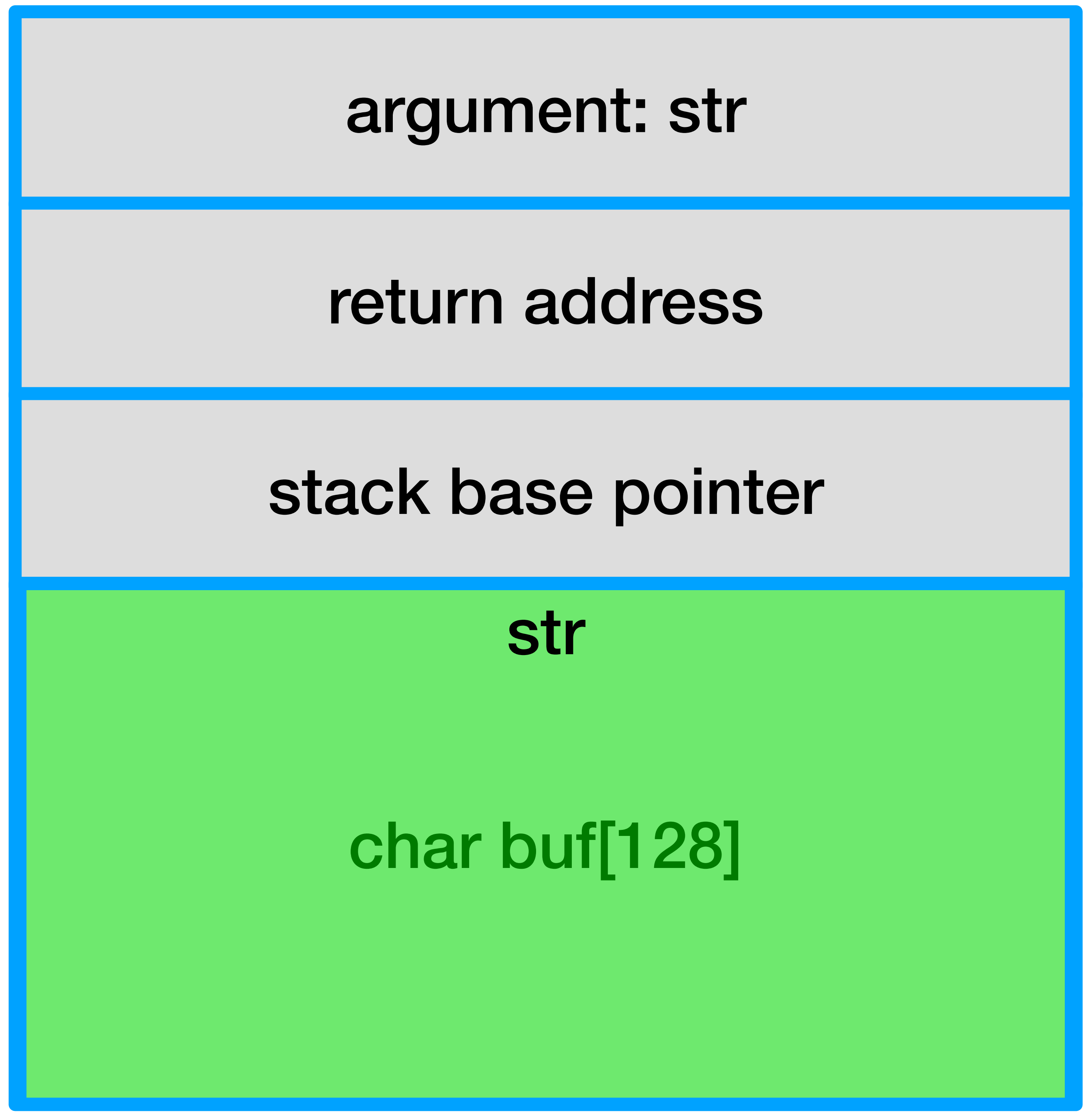
The stack:



Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

The stack:

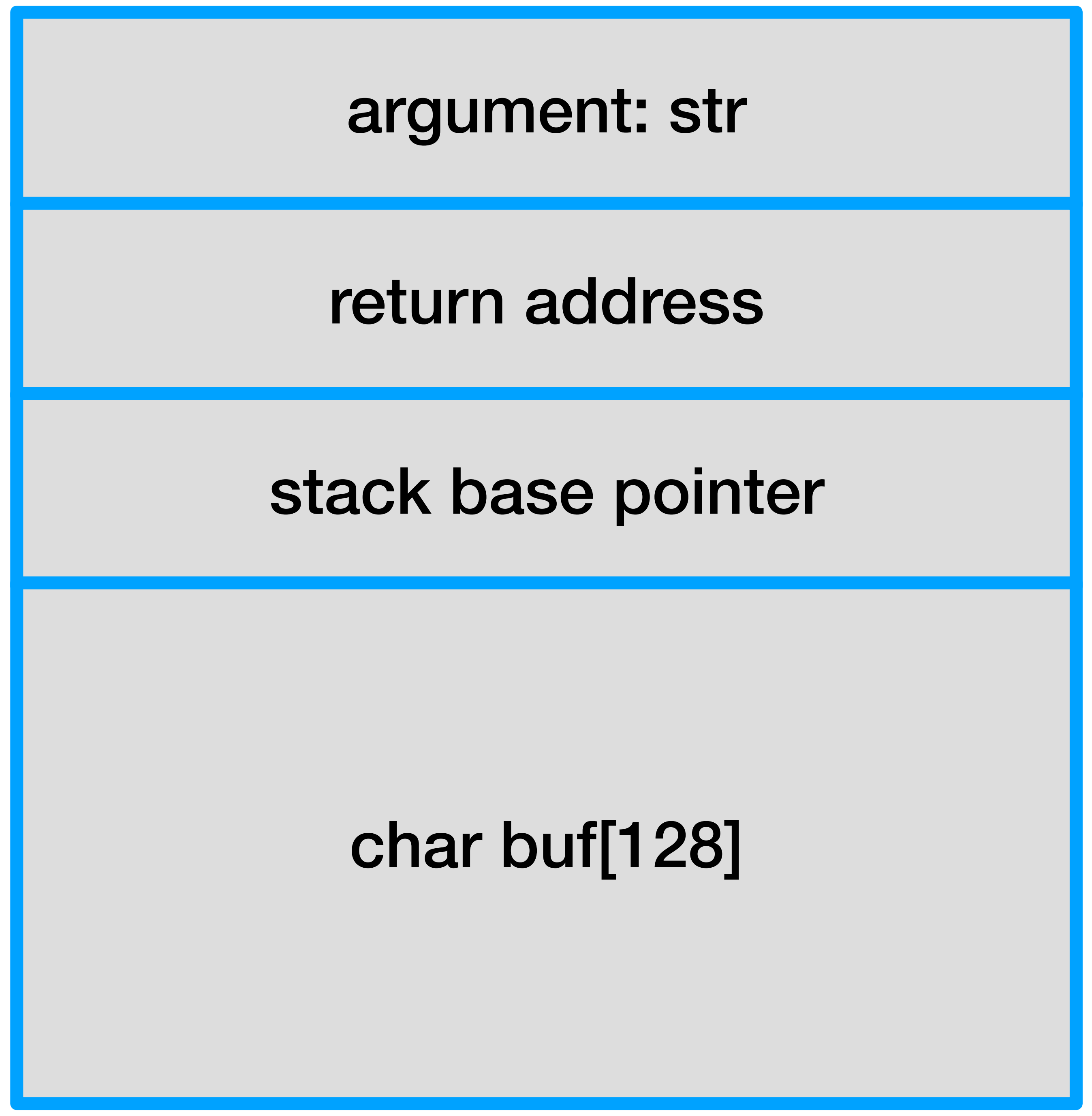


Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

What if str is 144 bytes long?

The stack:

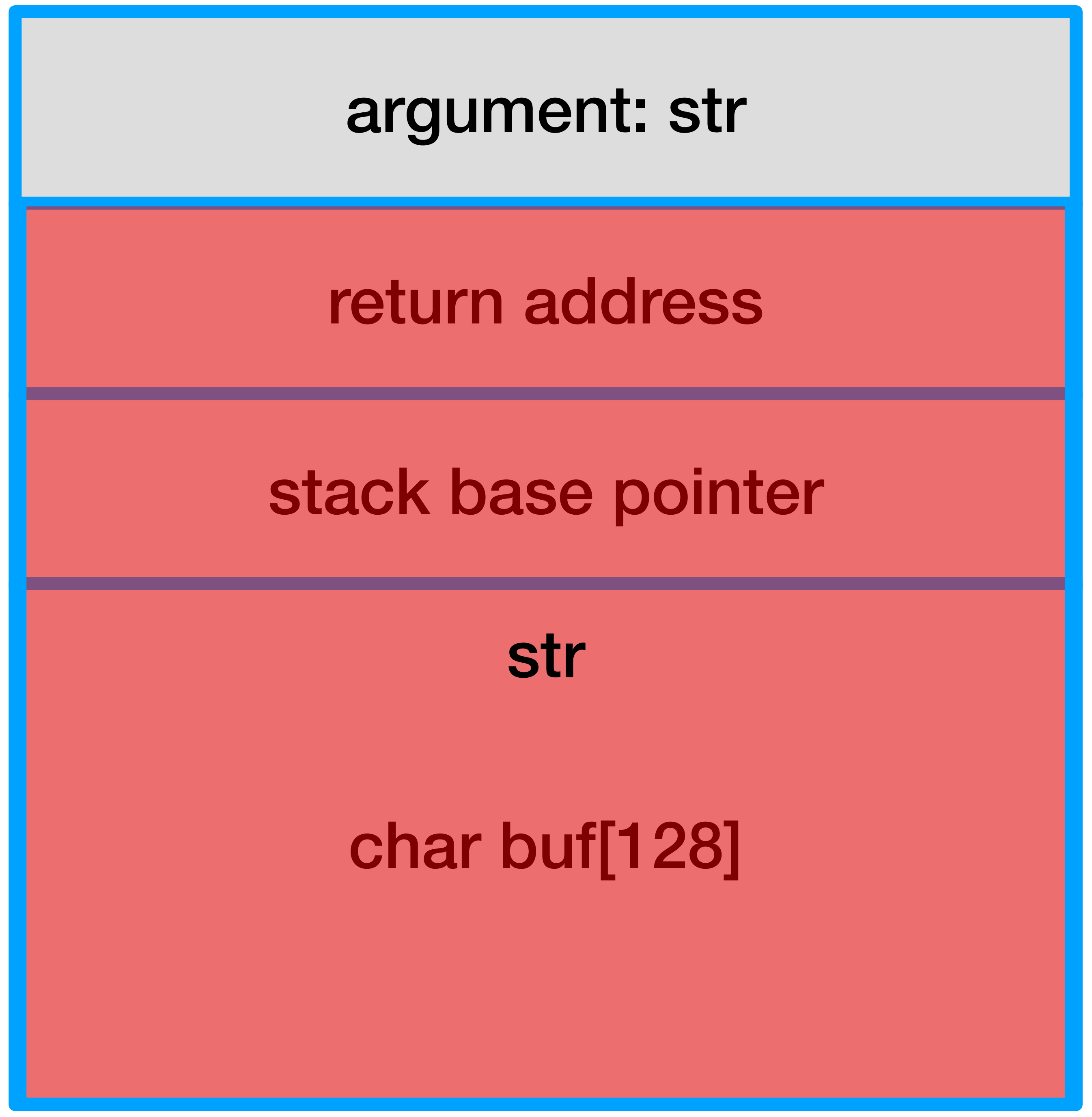


Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

What if str is 144 bytes long?

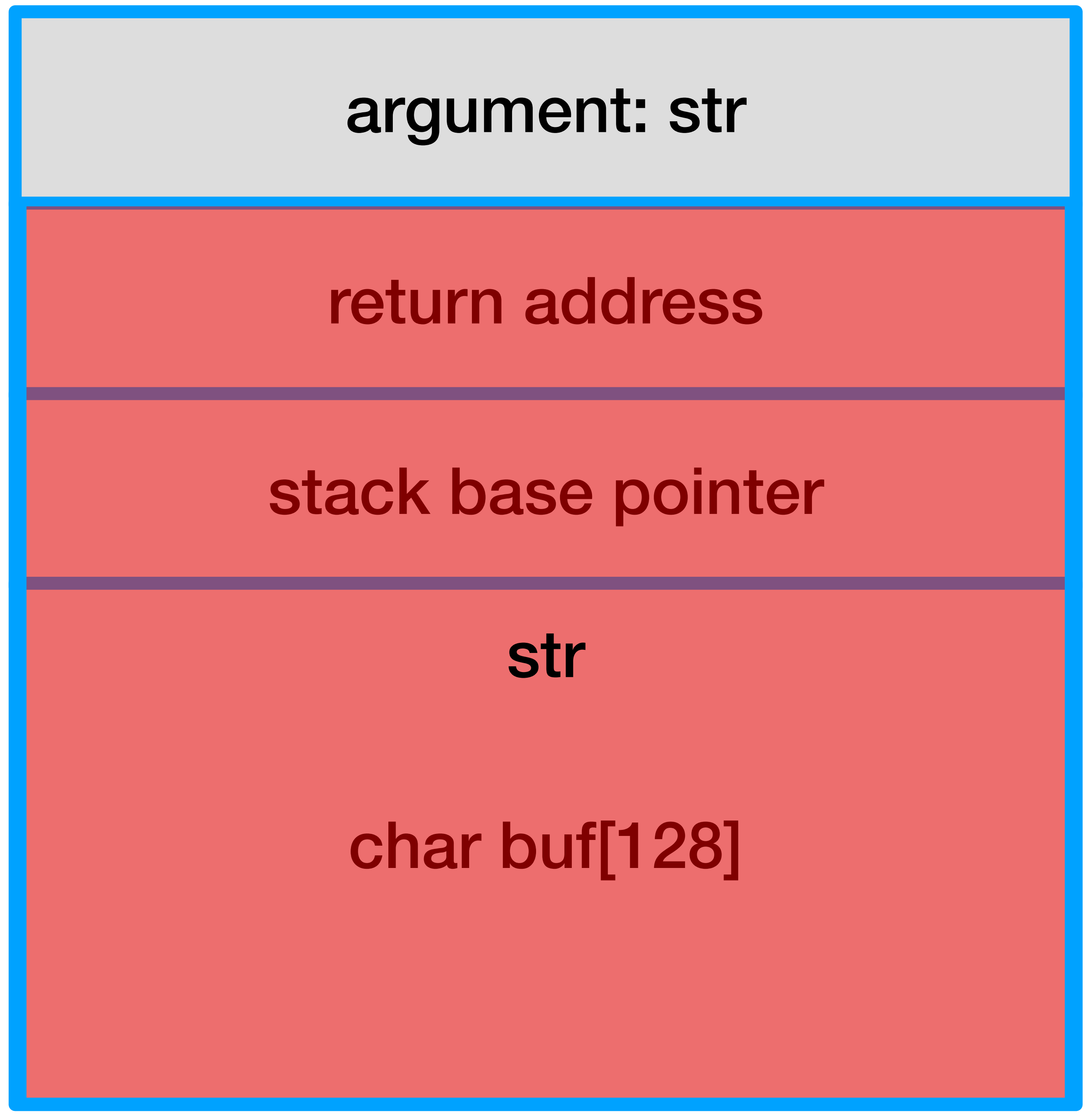
The stack:



Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

The stack:



We've overwritten the return address!

What happens when we overwrite the return address?

A. Nothing, the program would continue to execute normally

B. The program would crash

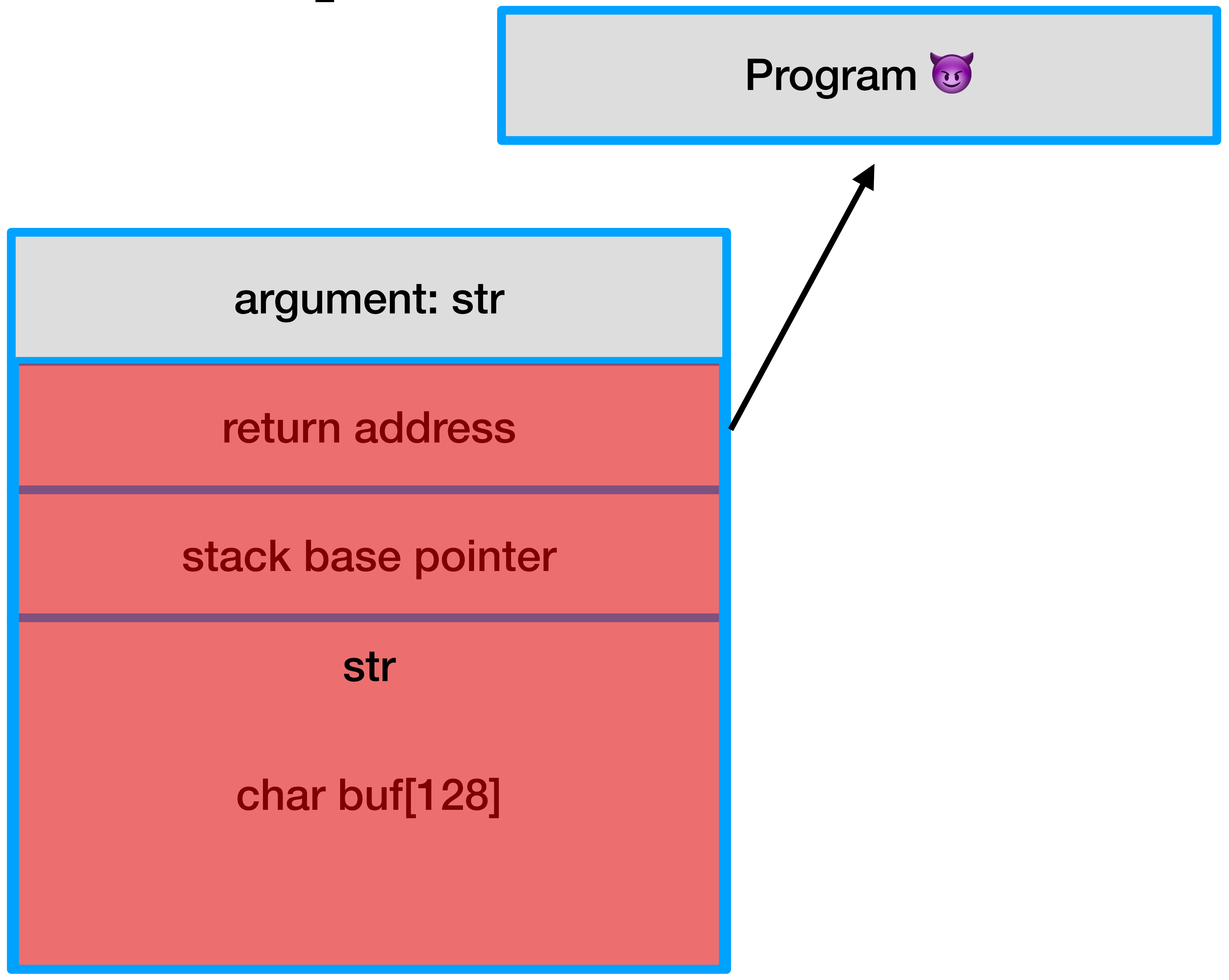
C. We would start executing a different program

D. Not sure/it depends

Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

The stack:



An attacker could construct a str such that the return address gets overwritten and now contains the memory address to a malicious program

What strategies would you use to avoid this problem? Take a minute to think and select A when you have an answer.

A. Select A when you have an answer

Control Flow Hijack Defenses

Bugs are the root cause of hijacks!

- Find bugs with analysis tools
- Prove program correctness
- (Use a memory safe language)

Even if we try to write correct programs, there still could be unknown bugs

- We need better mitigation strategies

Canary Defense

Canary in a coal mine means something is an early warning of danger

- Canaries were used to detect carbon monoxide in mines
- They require a high level of oxygen, so if there was too much CO, they would show signs of distress before humans were affected

Stack Canary Defense

Stack canaries (cookies) are random values inserted into the stack

- If the canary value is different at the end of the function, that’s an indicator that a buffer overflow has occurred and has overwritten the original canary
- If the canary is not the expected value, the program should immediately exit
- (There are several other types of canaries besides random)

The compiler will add the logic for canaries into the program during compilation

Do you think random values for canaries are chosen at compile time or runtime?

A. Compile time

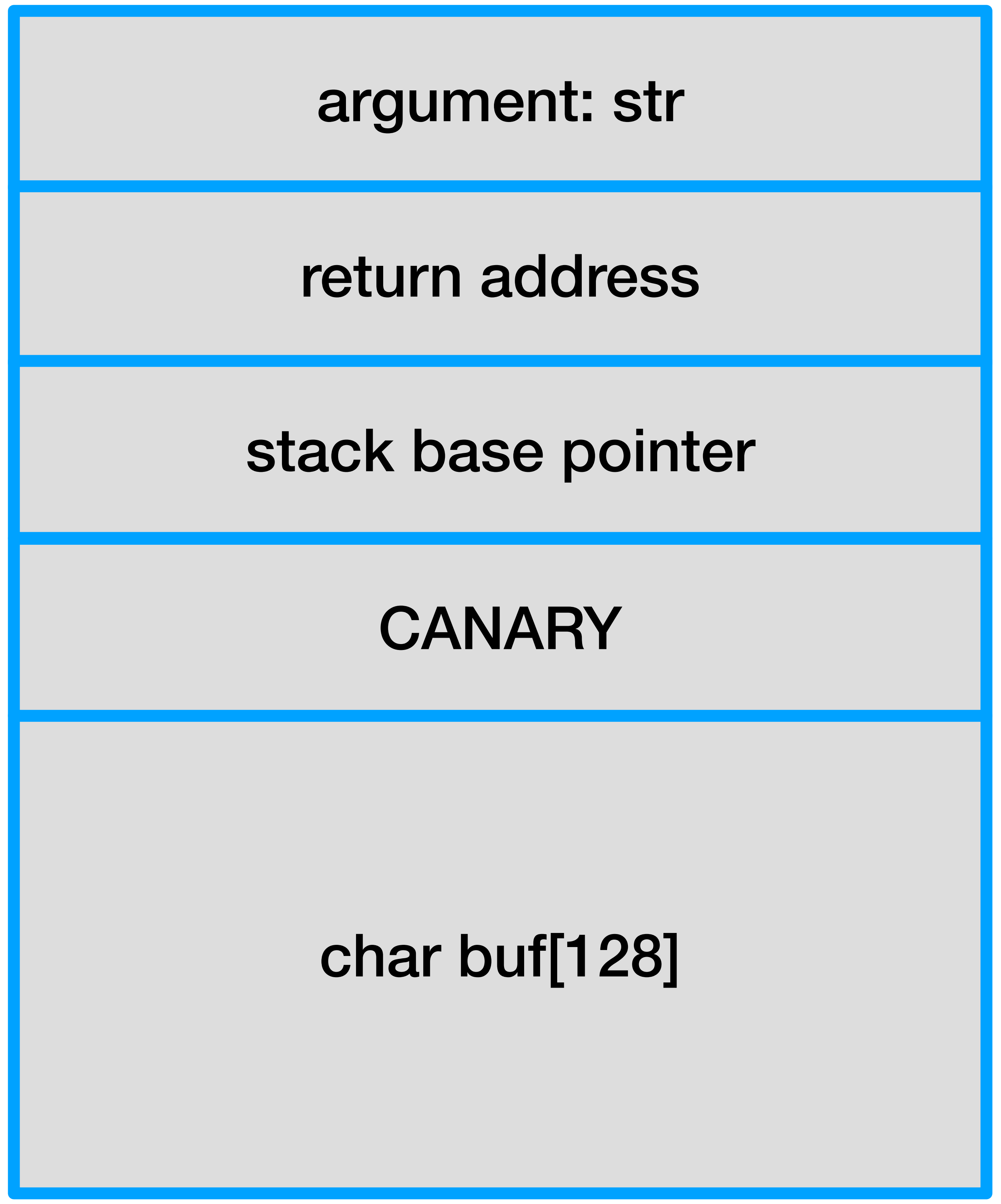
B. Runtime

C. Not sure

Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

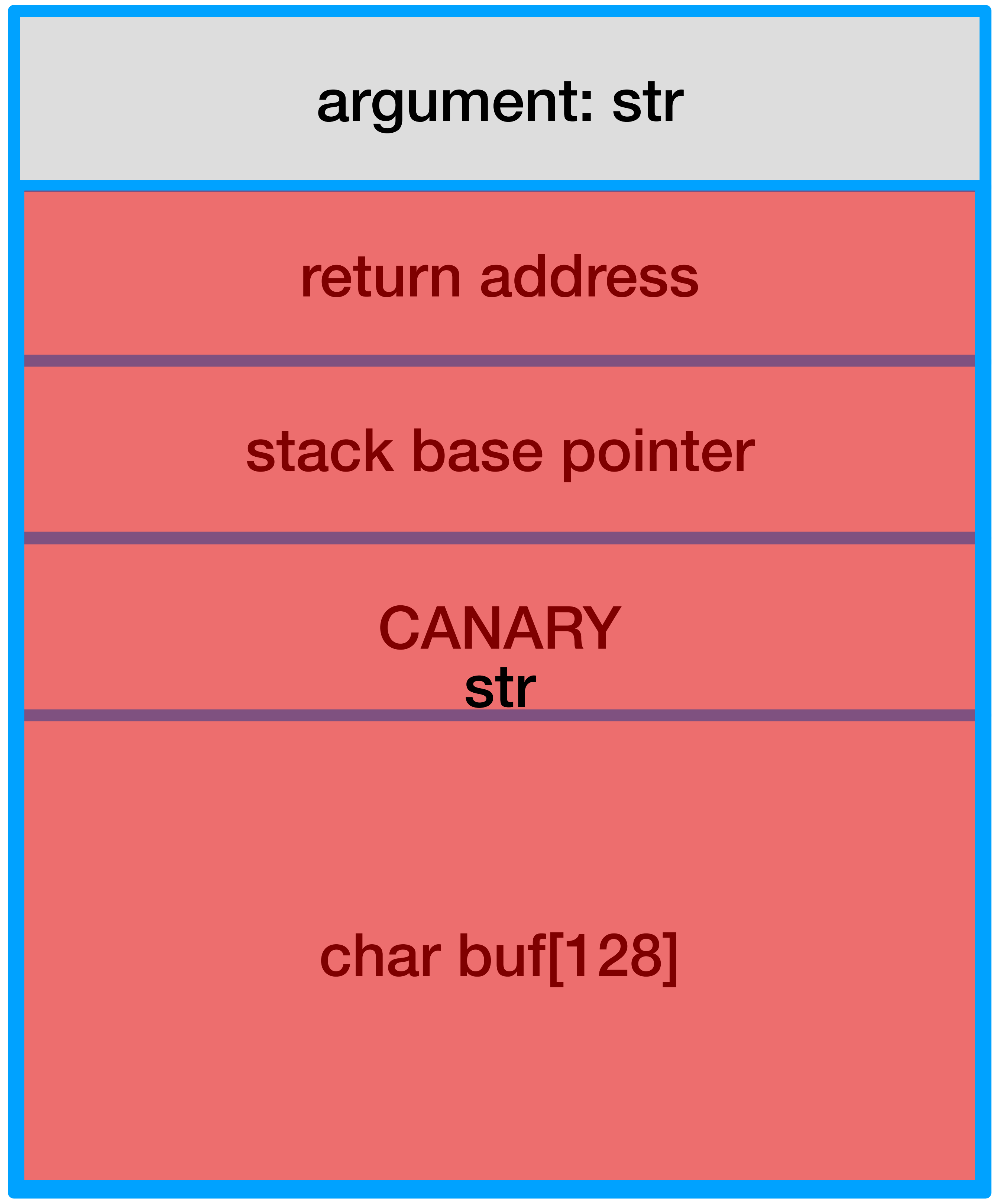
The stack:



Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

The stack:



Check canary at end of function
If canary is incorrect, this means an overflow happened

Canaries aren’t perfect….

There are several ways to bypass canaries, but one way in particular is a memory leak

If the attacker can read the canary value from memory, they can overwrite the stack without changing the canary (they can make sure the value is unchanged)

Format String Attacks

```
void func(char *user) {
  fprintf(stderr, user);
}
```

What if we could construct some string that would force the function to print out the contents of the stack (including the canary)?

Turns out, C lets us do this!

C print statements

Like other languages, C allows you to format a string in a print statement

For example, suppose you have some variable `name` you want to print as a string:

- `printf(“Hello %s!”, name);`

The `%s` tells C that you want to format that variable as a string. Other options:

- %c: character
- %d: integer
- %u: unsigned integer
- Etc.

Format String Attack

But what if you don’t include the variable?

- `printf(“Hello %s!”);`

What do you think happens when we don’t include the variable in the print statement?

A. Select A when you have an answer

Format String Attack

But what if you don't include the variable?

- `printf("Hello %s!");`

When `printf` sees `%s`, it expects a `char *` (C's version of a pointer to a string) on the stack

But because we haven't given a variable, it will use whatever is in the stack and print that out

- It can print out our canary!

Format String Attack

```
void func(char *user) {
  fprintf(stderr, user);
}
```

We can construct a string that prints out multiple values in the stack

- `user = “%08x%08x%08x%08x%08x%08x%08x”`
- `%08x` means print an integer in hex, padded to 8 bytes using 0s

It will start printing out memory contents

Format strings can write memory too using %n but that takes more work (take the security class to learn more!)

How would you prevent format string attacks?

A. Select A when you have an answer

Format String Defenses

Better print statements

- `fprintf(stderr, “%s”, user);` instead of `fprintf(stderr, user);`

Listen to the compiler!

- The C compiler will warn you about unsafe format usage (and you can optionally turn those warnings into errors, so your code will not compile until you fix them)

Fuzzing

- Testing technique that runs program with invalid, unexpected, and random input

Use Rust (or some other memory-safe language)!!

- Ultimately, C won’t stop you from writing vulnerable code