Software Systems

Day 6 - Strings, Buffer Overflow

Error Handling

- We've seen this function in previous classes.
- Without assuming the numbers are positive, you can't tell if the return value is an error or not.

```
int operate(int x, int y, char c) {
  switch(c) {
    case '+':
      return x + y;
    case '*':
      return x * y;
    default:
      return -1;
  }
}
```

Error Handling

- It's not easy to return multiple values in C.
- Solution: use pointers!
  - Pass in a pointer to an int, and put the result there.
  - Have the function return **a value that indicates success or failure**.
- Using a pointer to get results is a common pattern in C.

Error Handling

```
int operate(int x, int y, char c, int* result) {
  switch(c) {
    case '+':
      *result = x + y;
      return 1;
    case '*':
      *result = x * y;
      return 1;
  }
  return 0;
}
```

Error Handling

- Some points to remember:
  - You cannot return Booleans in standard C, just ints.
  - The pointer has to be initialized outside of the function and passed in.
- Return values for success/failure are particularly tricky.
  - Some functions like `main` use 0 for success and nonzero for failure.
  - Other functions might use 1 or nonzero for success.
  - Documentation is important.

Strings

- In Chapter 2.5 of Head First C, you saw how to:
  - Use string.h for various convenient string functions.
  - Store an array of strings as a two-dimensional array or an array of character pointers.
- One of the trickier parts of strings methods in C is how types work.
  - `char* strchr(const char* str, int ch);`
  - Returns an address, and character is an integer.

Strings: Exercise

- Here are some string methods from Python. How would you declare these functions in C? (param/return types)
  - (For all of these, `str` is an actual string, like `"foobarbaz"`.)
  - `str.count(substr)  # "abcabcaba".count("abc") returns 2`
  - `str.startswith(substr)` and `str.endswith(substr)`
  - `# "www.abc.com".strip("cmowz.") returns "abc"`
    `str.strip(chars)`
  - `# "1<>2<>3".split("<>") returns ["1", "2", "3"]`
    `str.split(sep)`

Strings: Exercise

- Remember, header files usually tell the compiler what names (variables and functions) are available and how they are used.
- The implementations may be somewhere else, in which case the linker has to find them.

Strings: Exercise

- Remember, header files usually tell the compiler what names (variables and functions) are available and how they are used.
- The implementations may be somewhere else, in which case the linker has to find them.

Strings

- The string library is pretty low-level.
  - They are small, simple, and fast.
  - Little to no error handling or memory bounds checking.
- This can cause issues with development:
  - You may need to implement your own convenience functions.
  - A lot of subtle bugs can occur.

Functions and the Stack

- Registers and purposes
- Stack
- Calling conventions
- Return addresses

Functions and the Stack: Registers

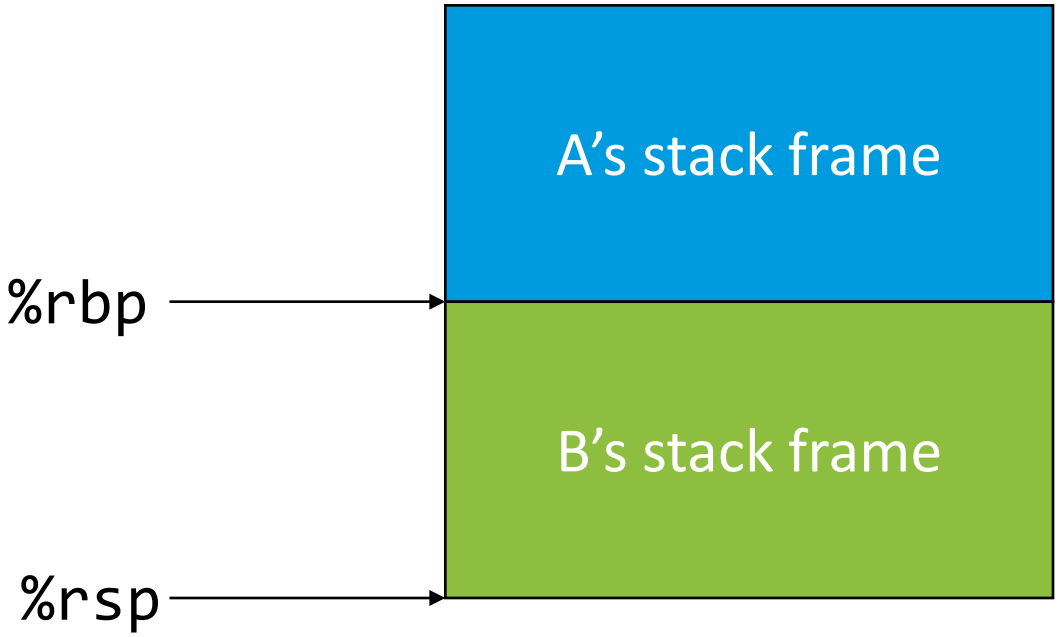
- When your CPU executes instructions of a program, it can access locations in memory.
- It also can access a set of 8-byte *registers*, which it uses for most computation:
  - Named: %rax, %rbx, %rcx, %rdx, %rdi, %rsi, %rbp, %rsp
  - Numbered: %r8-r15
  - EFLAGS
  - Instruction pointer: %rip

Functions and the Stack: Registers

- `%rip` is a special pointer: it refers to the address of the next instruction to execute.
- `%rax` is also special on most systems: it stores the return value of a function.
- On Linux, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` are used for function arguments.

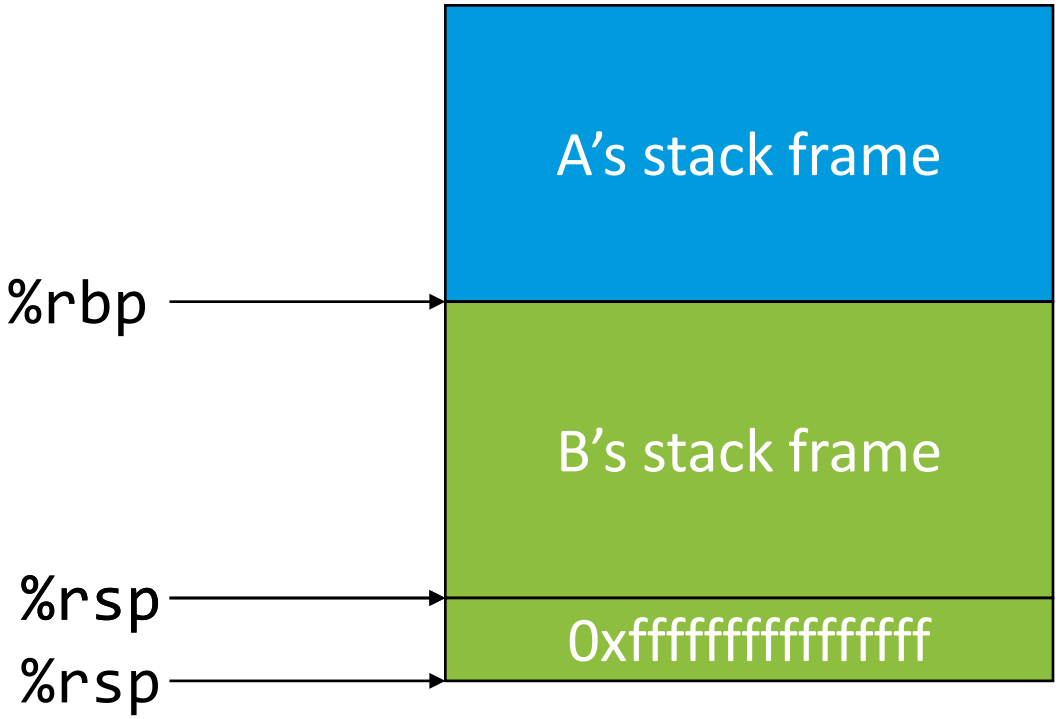
Functions and the Stack: Registers

- Registers `%rbp` and `%rsp` are used for the stack.
- %rsp is the stack pointer, and refers to the "end" of the stack.
- %rbp is the base pointer, and refers to the start of the current stack frame.



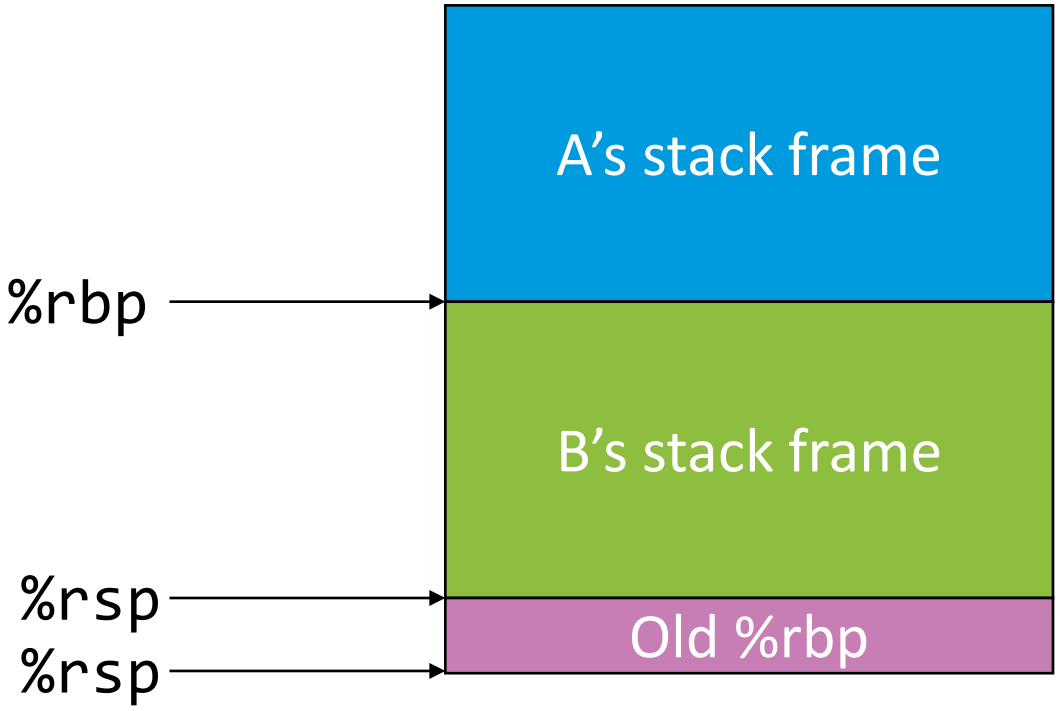
Functions and the Stack: The Stack

- So what happens when we push/pop something on the stack? It depends.
- Remember the stack grows down in addresses.
- To push a piece of data, subtract from the stack pointer and write it to that memory location.
- To pop, add to the stack pointer.



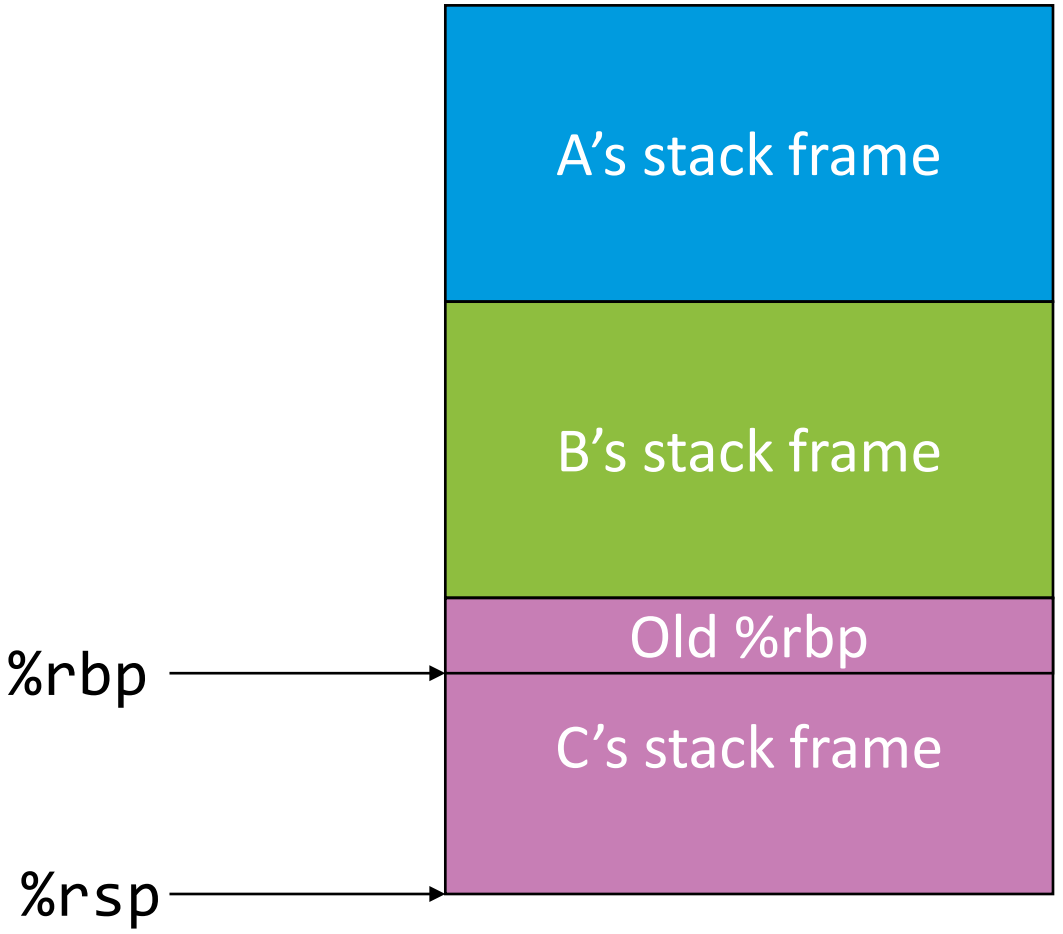
Functions and the Stack: The Stack

- What about a stack frame?
- When a function is called, the first thing it does is save the base pointer (%rbp).
- Then it sets the rest of the frame.
- When it finishes, it pops the old %rbp back into the register.



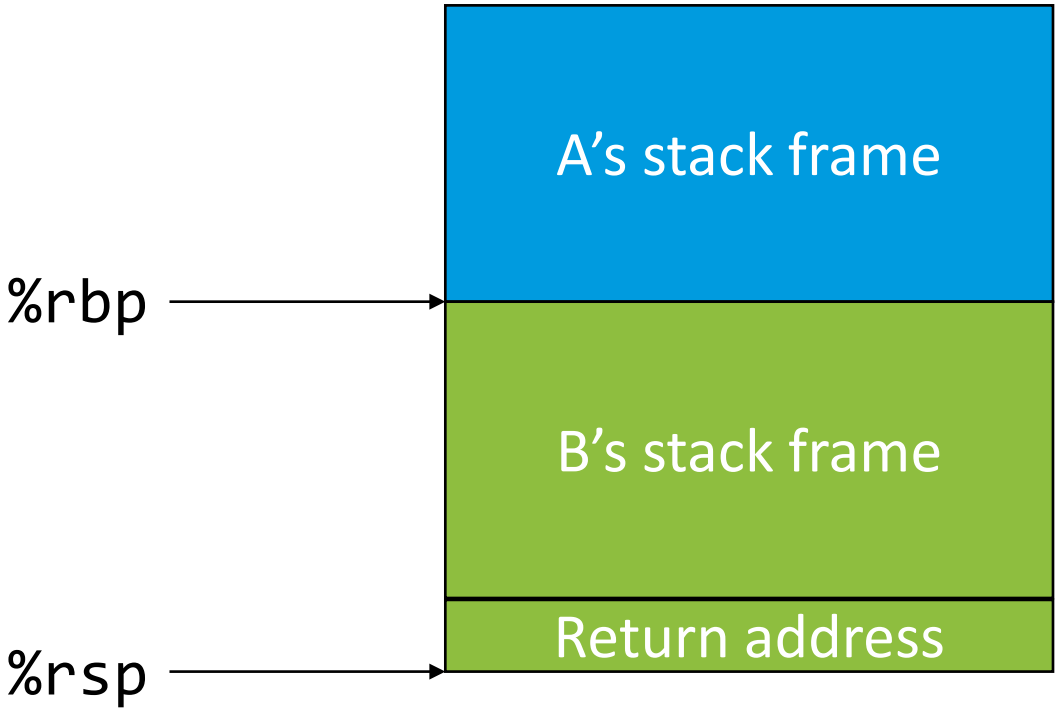
Functions and the Stack: The Stack

- What about a stack frame?
- When a function is called, the first thing it does is save the base pointer (%rbp).
- Then it sets the rest of the frame.
- When it finishes, it pops the old %rbp back into the register.



Functions and the Stack: Calling Functions

- When a function is called, it has to know where to go back to once it's done.
- A function first loads up all of its arguments into the specified registers: %rdi, %rsi, etc.
- Then it pushes its return address onto the stack.

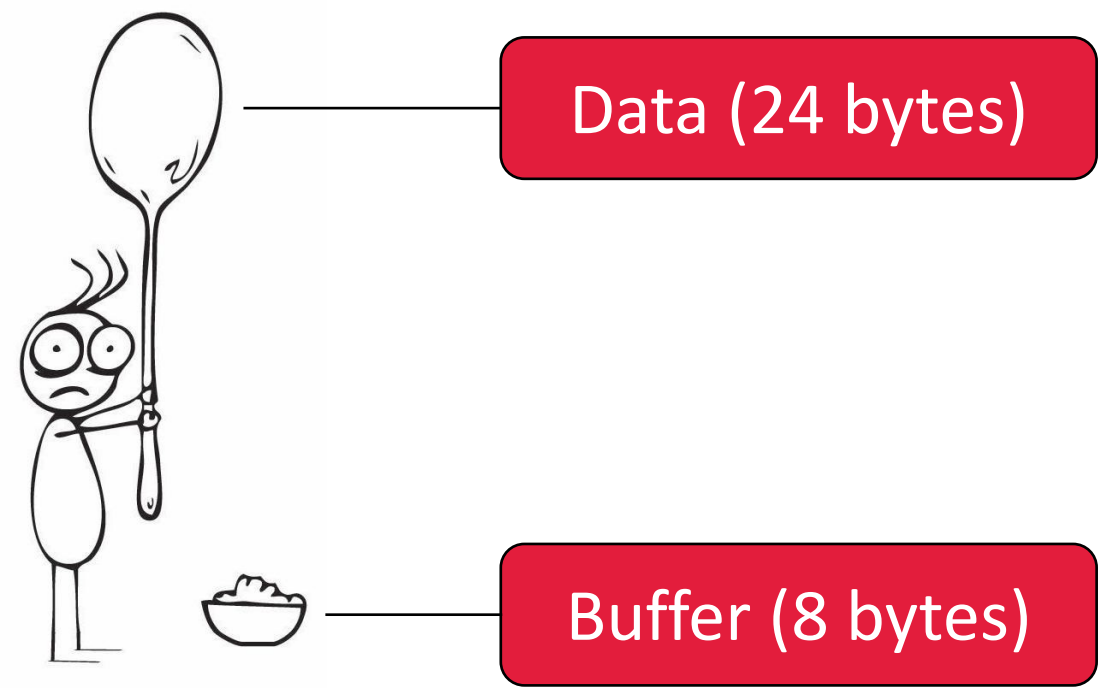


Buffer Overflow

- You can use a string function to copy one string into another.
- If the destination buffer (character array) is too small, it doesn't matter - the function will plow ahead with copying.
- This can cause issues - but what exactly?

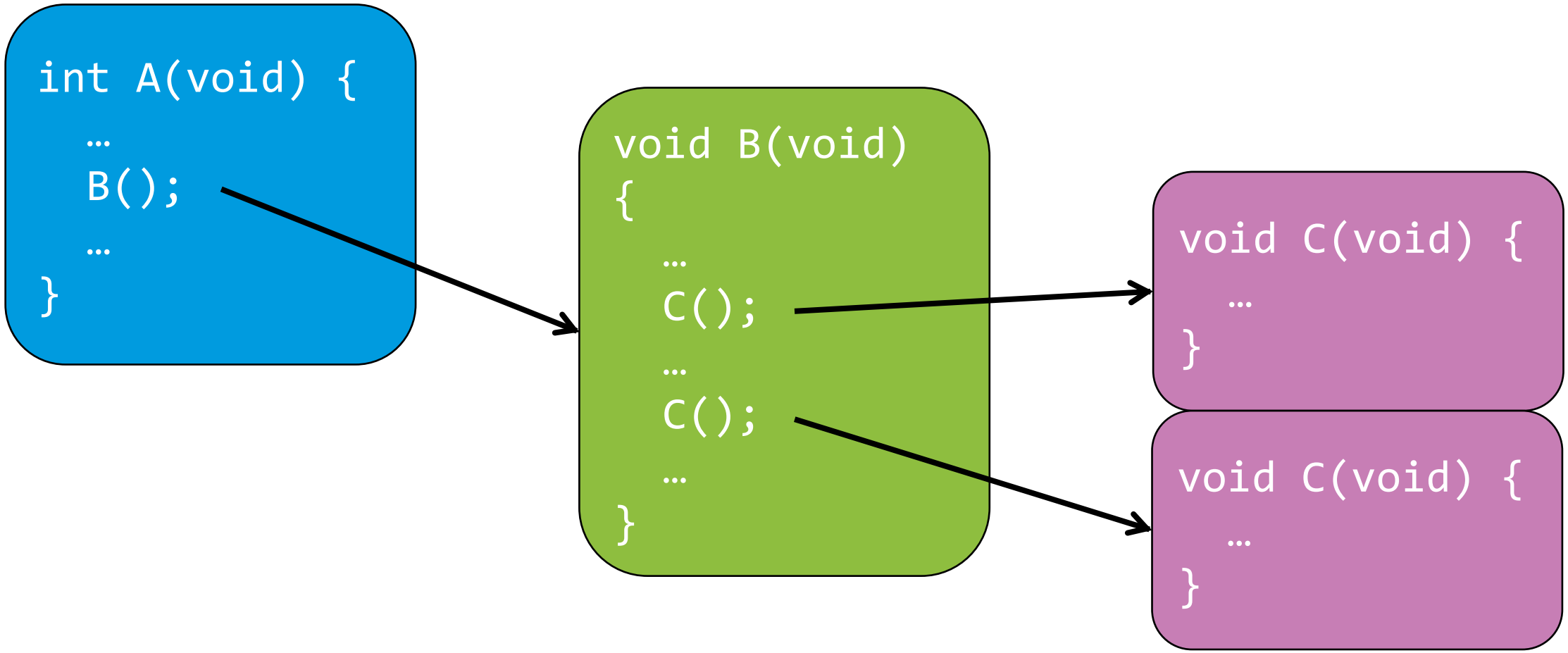
Buffer Overflow

- Cleverly write extra data into a buffer to alter the runtime stack.



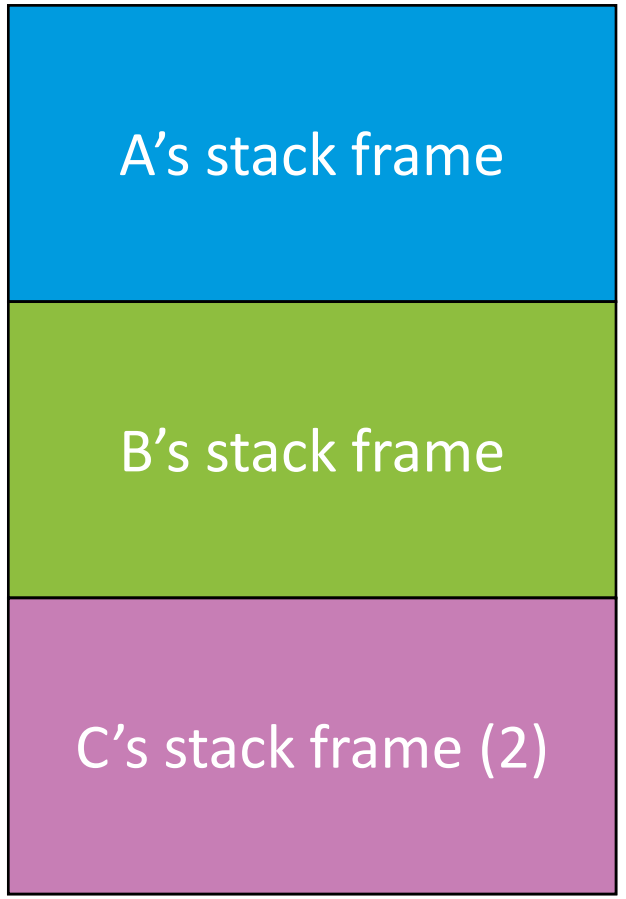

```
void spoon(void) {
  char buf[8];
  gets(buf);
  printf("My %s is too big\n",
         buf);
}
```

Buffer Overflow: The Runtime Stack



Buffer Overflow: The Runtime Stack

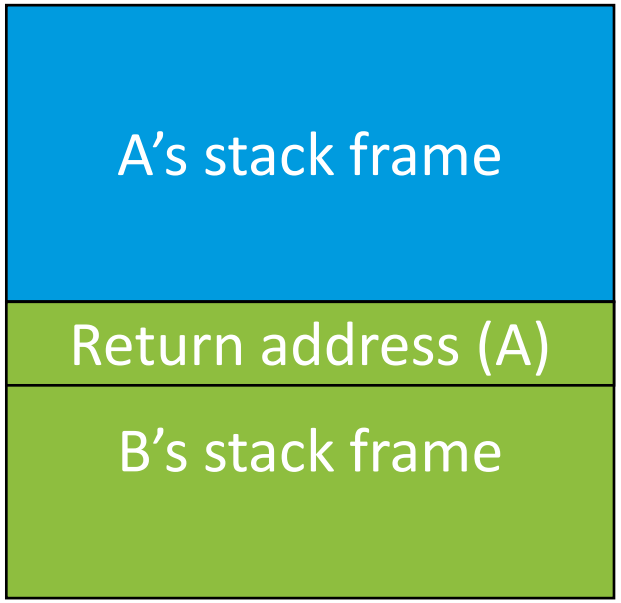
- Each function call has its own stack frame.
- A stack frame tracks:
  - Local variables
  - Parameters to pass to other functions
  - Other temporary space
- Function call: push a new frame onto the stack
- Function return: pop a frame from the stack



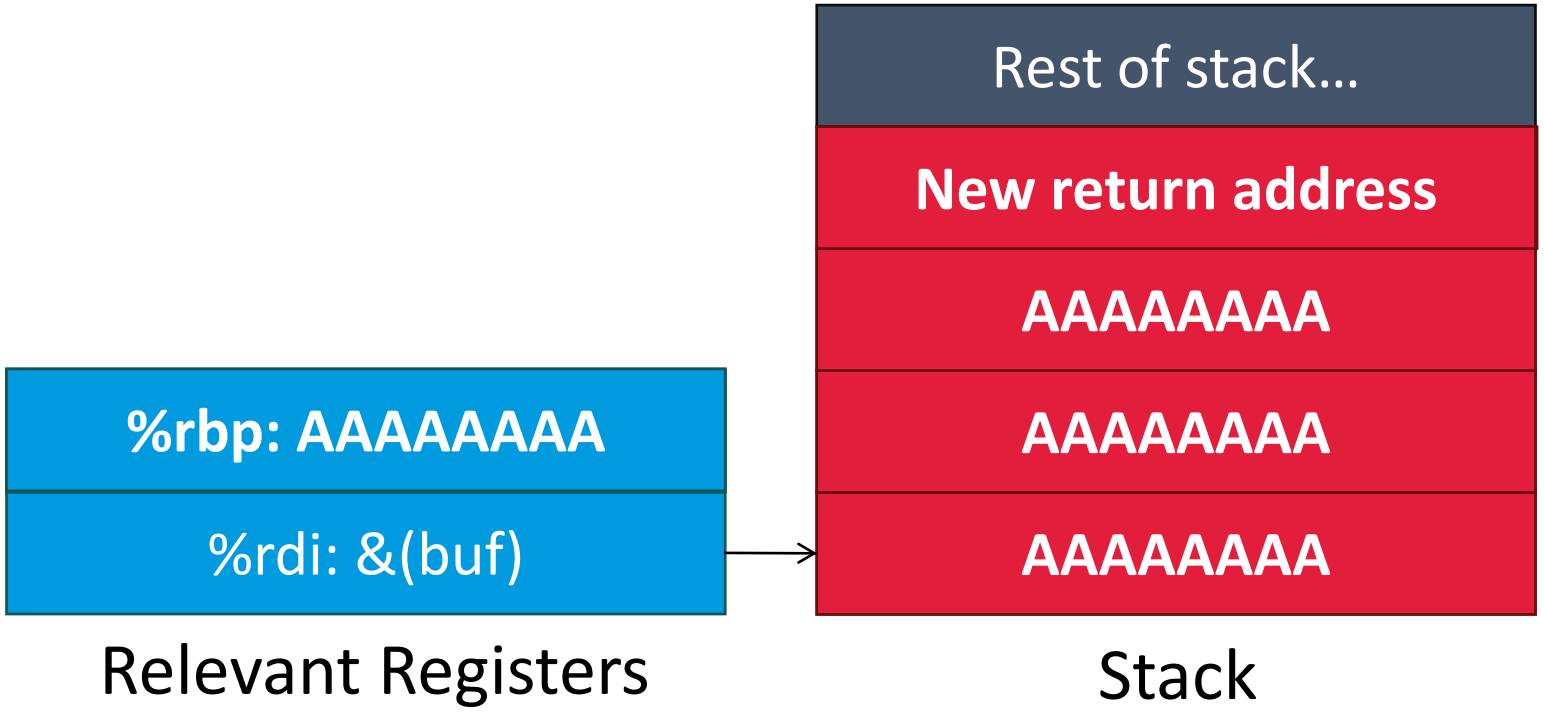
Buffer Overflow: The Runtime Stack

```
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $3, %esi
movl    $2, %edi
call    B
movl    %eax, -4(%rbp)
nop
leave
ret
```

```
pushq   %rbp
movq    %rsp, %rbp
subq    $48, %rsp
movl    %edi, -36(%rbp)
movl    %esi, -40(%rbp)
...
movl    %eax, -8(%rbp)
movl    -8(%rbp), %eax
leave
ret
```

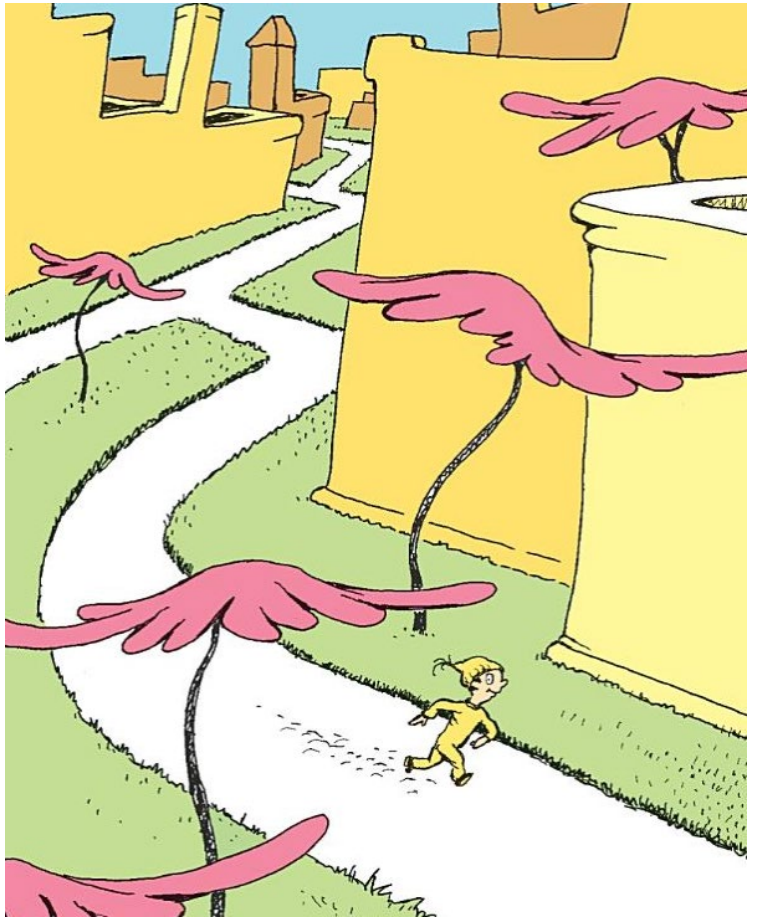


Buffer Overflow: A New Return Address



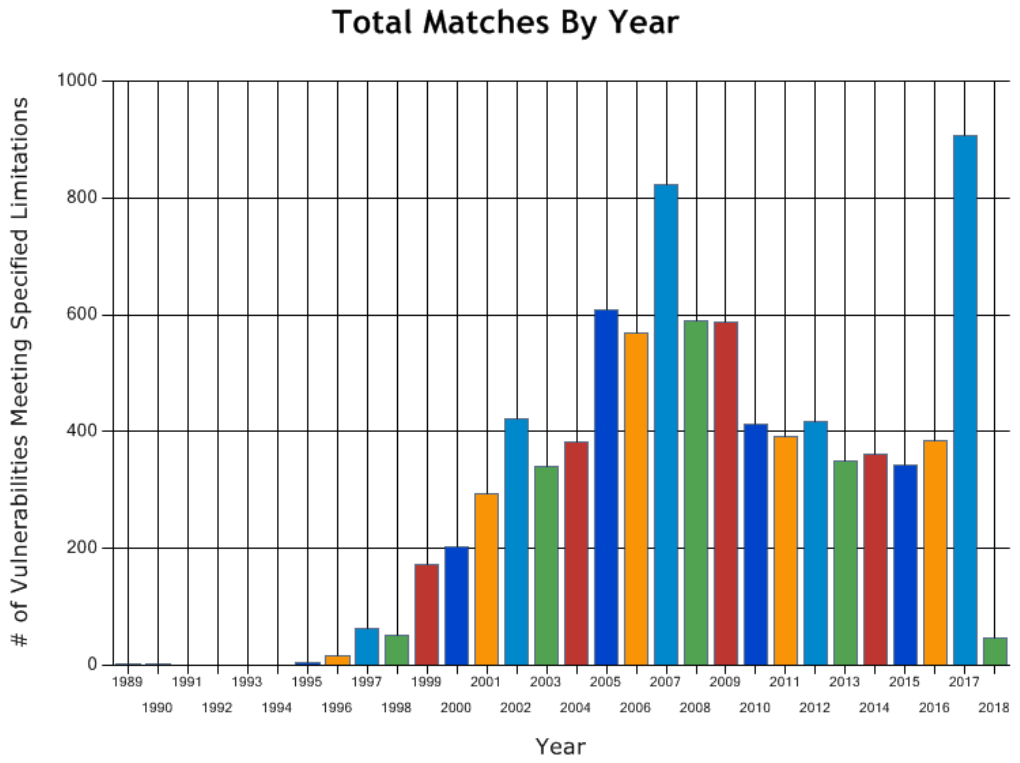
Buffer Overflow: Where To?

- So you've written a new return address - where can you go?
  - A different function in the program
  - Functions in the C standard library
  - Custom assembly code that you wrote
- Most attackers just start a shell so they can do whatever else they want.
  - Surprisingly easy – just call `/bin/sh`.
  - Requires the program to run with some form of elevated privileges.



Buffer Overflow: Summary

- Buffer overflows are powerful exploits with many variants.
- It's a common bug, even today.
- Modern C compilers and operating systems do provide some protection, at least.
- If this is interesting, consider doing a project in it.



Buffer Overflow: Bonus Video

- https://www.youtube.com/watch?v=WWbZFj-cLvk