# CSE 351: The Hardware/Software Interface

## Section 5

## Structs as parameters, buffer overflows, and lab 3

# Structs as parameters

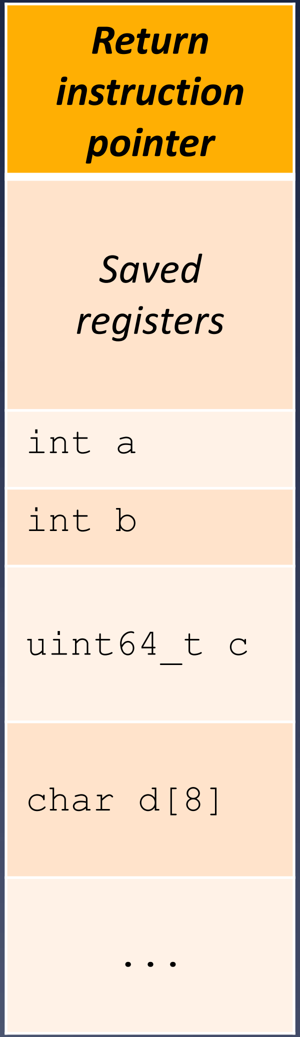
- In the previous section, we looked at how integers and pointers were passed as arguments to functions
- If we were to pass a struct by value to a function (as in not passing a pointer), how would the compiler use assembly instructions to achieve this?
- Let's take a look at an example (see pass_struct.c from the section material)

# Buffer overflows

- C performs no bounds-checking on array accesses; this makes it fast but also unsafe
  - What would we need to add to C to support checked array accesses?
- For example: `int arr[10]; arr[15] = 3;`
  - No compiler warning, just memory corruption
- What symptoms are there when programs write past the end of arrays?
  - Hint: we saw an example of this in lab 0

# Stack layout

- As we've seen previously, when values are declared on the stack, the compiler shifts %rsp (in x86-64 assembly) to allocate space for them
- When a function returns, the return instruction pointer indicates where to begin executing again



***Return instruction pointer***

*Saved registers*

`int a`

`int b`

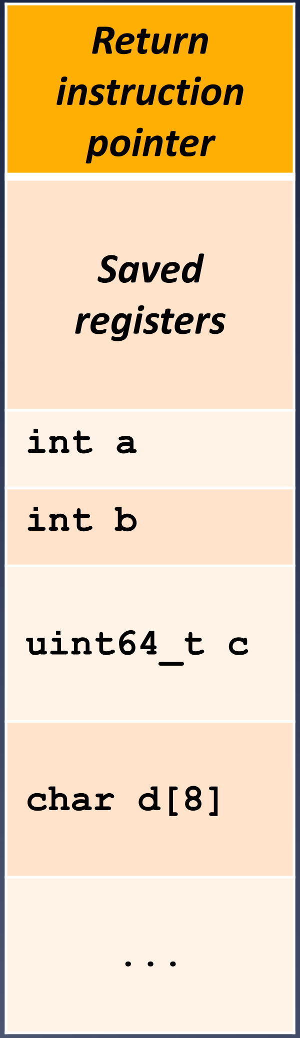
`uint64_t c`

`char d[8]`

`...`

# Stack layout

- Note that the top of the diagram represents higher addresses, and the bottom is lower addresses
- To which memory does `d[10]` refer in this example?



***Return instruction pointer***

***Saved registers***

**`int a`**

**`int b`**

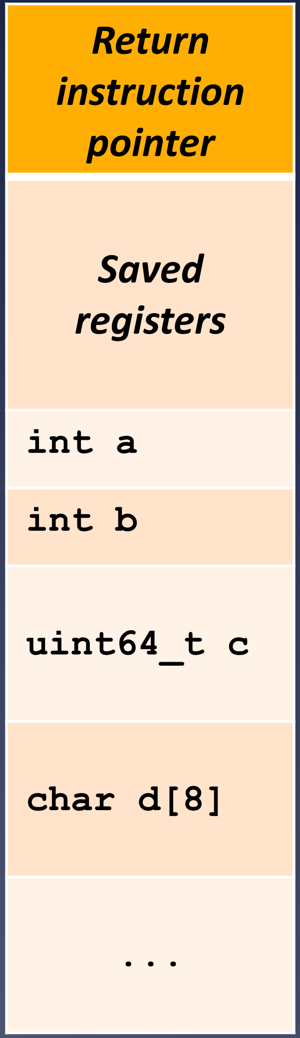
**`uint64_t c`**

**`char d[8]`**

**`...`**

# Buffer overflow attacks

- In buffer overflow *attacks*, malicious users pass values to attempt to overwrite important parts of the stack or heap
- For example, an attacker could overwrite the return instruction pointer with the address of a malicious block of code



***Return instruction pointer***

***Saved registers***

`int a`

`int b`

`uint64_t c`

`char d[8]`

`...`

# Buffer overflow attacks

- C has some inherently unsafe functions that facilitate buffer overflows, including `gets` and `strcpy`
- `gets(char* s)` reads from standard input until reaching a newline character ('\n') or EOF (end of file)
  - How long should `s` be to contain the entire input string?
- `strcpy(char* dest, const char* src)` copies the contents of the `src` string into the `dest` string
  - What happens if `dest` is smaller than `src`?

# Protecting against overflows

- As a programmer, you can protect against buffer overflow bugs/attacks by checking buffer lengths and using safer string-related functions
  - `fgets(char* s, int size, FILE* stream)` takes a size parameter and will only read that many bytes from the given input stream
  - `strncpy(char* dest, const char* src, size_t n)` will copy at most `n` bytes from `src` to `dest`

# Protecting against overflows

- Stack canaries
  - At runtime, programs place a (pseudo-)random integer on the stack immediately before the return instruction pointer. If the integer value doesn’t match when the function returns, the program generates a segmentation fault
- Data execution prevention
  - Some parts of memory (notably the stack) are marked as non-executable. The CPU will refuse to execute instructions from such locations and the program will terminate

# Lab 3: Buffer overflows

- The purpose of lab 3 is to become familiar with how buffer overflow attacks work
- The various stages of the lab require different types of attacks to achieve certain goals
- If you have become comfortable with GDB and understanding assembly instructions, you should have no problem

# Lab 3: Buffer overflows

- The exploitable function in lab 3 is called `Gets` (capital ‘G’) and is called from the `getbuf` function
- `getbuf` allocates a small array and reads user input into it via `Gets`. If the user input is too long, then certain values on the stack within the `getbuf` function will be overwritten…

# Lab 3: Buffer overflows

- The first thing to do is to become familiar with the provided tools for the lab
- To generate malicious strings for testing buffer overflows, use the provided `sendstring` tool. It takes a list of space-separated hex values and translates them to the corresponding Ascii characters
- Each lab is slightly different as determined by the username given to it; when you run the `bufbomb` binary, you have to pass in "`-u [cse-username]`"
- Let's take a look at how this works