15-213 Attack Lab Bootcamp

Your TAs

Tuesday, June 7th, 2022

Agenda

- Attack Lab Overview
- Stacks Review
- Activity 1
- Procedure Calling Review
- Activity 2
- Activity 3 (If time)

Learning objectives

By the end of this bootcamp, we want you to know:

- Stack discipline and calling conventions
- How to perform a simple buffer overflow attack

Reminders and Lab Overview

Reminders

- Attack Lab is due this Fr**iday, June 10**
- C Review Bootcamp this **Friday,June. 10**
  - Will be very useful for cachelab coming up!

Attack Lab overview

- Attack programs by crafting buffer overflow attacks that hijack the control flow
- Provide inputs to the `rtarget` and `ctarget` programs that cause them to call certain functions
- Unlike in bomblab, the targets don't explode!

Stacks Review

Manipulating the stack

What instructions do we typically use to change the stack pointer, %rsp?

Growing the stack: **Shrinking the stack:**

Manipulating the stack

What instructions do we typically use to change the stack pointer, %rsp?

Growing the stack: **Shrinking the stack:**

- `sub $0x28, %rsp`
- `push %rbx`
- `callq my_function`

Manipulating the stack

What instructions do we typically use to change the stack pointer, %rsp?

Growing the stack:

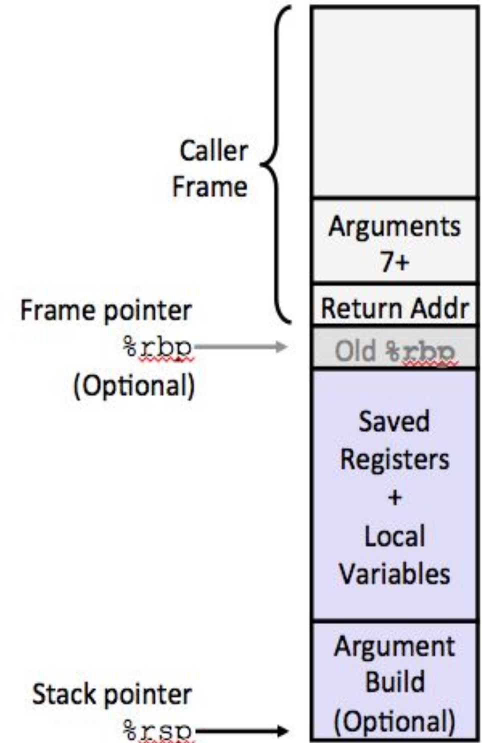
- `sub $0x28, %rsp`
- `push %rbx`
- `callq my_function`

Shrinking the stack:

- `add $0x28, %rsp`
- `pop %rbx`
- `retq`

What does a stack frame look like?

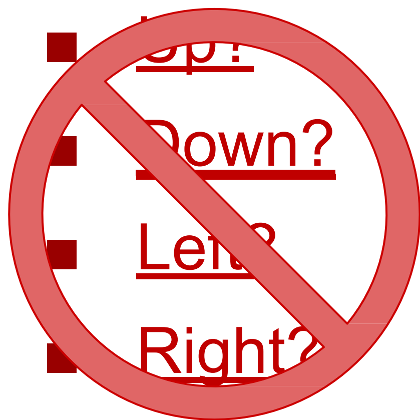
1. Caller pushes arguments 7+ if they exist
2. Caller executes callq <addr>, pushing next instruction address (return address) onto stack and jumping to addr
3. Callee optionally pushes %rbp, address of start of previous stack frame (sometimes optimized out)
4. Callee may push saved registers, local variables



Which way does the stack grow?

- Up?
- Down?
- Left?
- Right?

Which way does the stack grow?



It depends on how you draw it!

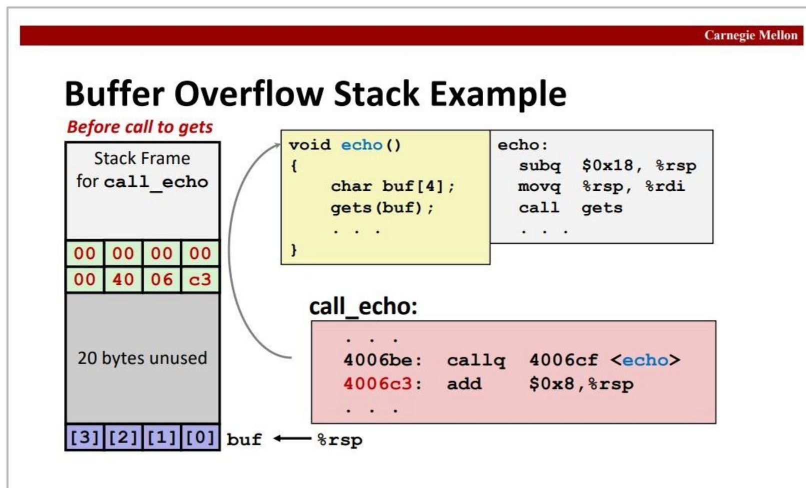
The stack always grows towards **lower addresses** in x86-64.

(Informally, this usually means "down".)

Be aware of this possible ambiguity when reading diagrams.

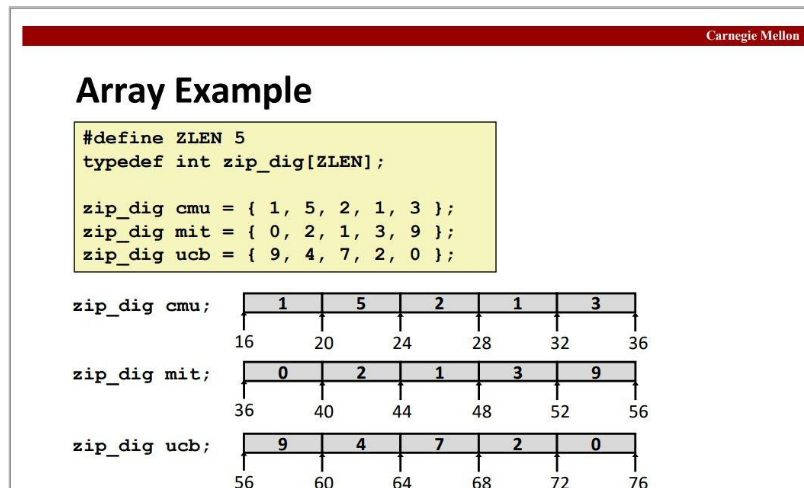
Drawing memory

Stack diagrams



Addresses are displayed increasing to the **left**, and then **upwards**.

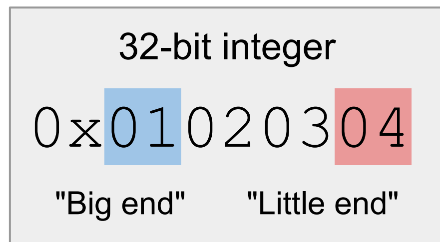
Everything else

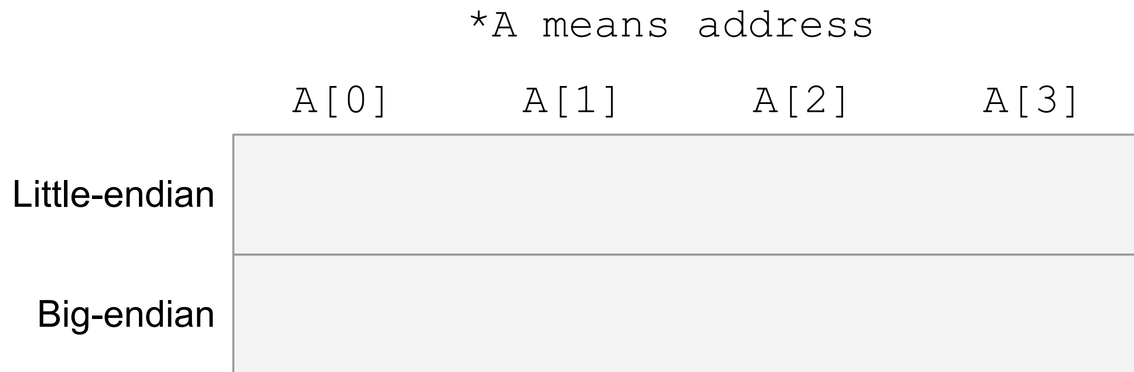


Addresses are displayed increasing to the **right**, and then **downwards**.

Endianness

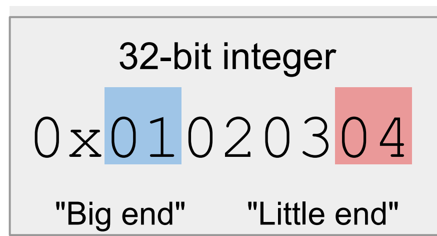
- Describes how integers are represented as bytes.
- Little-endian means that the least significant bytes of and integer are stored at the lowest address.





Endianness

- Describes how integers are represented as bytes.
- Little-endian means that the least significant bytes of an integer are stored at the lowest address.
- Shark machines use little-endian, so that is what you will work with in attack lab.
    - Reverse how addresses look in exploit string



*A means address

	A[0]	A[1]	A[2]	A[3]
Little-endian	0x04	0x03	0x02	0x01
Big-endian	0x01	0x02	0x03	0x04

Activity 1

Part 1: Introduction to solve()

Let's look at `solve()` in the `src/activity.c` file.

What is it doing?

Is it possible for the program to call `win()`?

```
void solve(void) { long
    before = 0xb4; char
    buf[16];
    long after = 0xaf;

    Gets(buf);

    if (before == 0x3331323531)
        win(0x15213);

    if (after == 0x3331323831)
        win(0x18213);
}
```

Part 1: The gets() function

```
char *gets(char *s);
```

- `gets()` reads from standard input and writes characters into `s` until it reaches a newline.
- Since it has no information about the **size** of the buffer `s`, its design is fundamentally flawed. Input that is too long will overflow past the space allocated for the buffer and cause nasty things to happen. **Never use `gets()` yourself!**
- `Gets()` is a CS:APP wrapper function that checks for errors, and exits if it encounters any.

Part 1: Activity setup

- Split up into groups of 2-3 people
- One person needs a laptop
- Log in to a Shark machine, and type:

```
$ wget https://www.cs.cmu.edu/~213/activities/attack-lab-activity.tar
$ tar xvf attack-lab-activity.tar
$ cd attack-lab-activity
```

- Take a look at the code in `src/activity.c`.

Part 1: Diving into assembly

- Look at the disassembly of `solve()`.
- Try drawing a stack diagram.
  - How large is the stack frame?
  - Where is the saved return address?
  - Where are `before`, `buf`, and `after`?
- **Which variable will be overwritten if we perform a buffer overflow, `before` or `after`?**

Hex	Char
30	0
31	1
32	2
33	3
34	4
35	5
36	6
37	7
38	8
39	9
61	a
62	b
63	c
64	d
65	e
66	f
67	g
68	h

Part 1: Drawing the stack diagram

```
=> 0x4006b5 <+0>:     sub    $0x38,%rsp
```

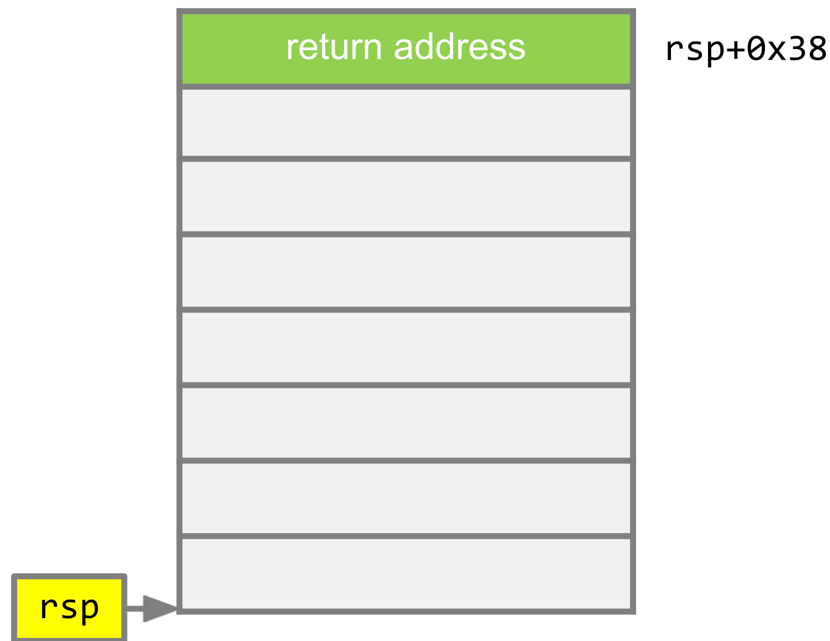
rsp+0x38

Addresses increase towards the top of the slide

Part 1: Drawing the stack diagram

```
   0x4006b5 <+0>:      sub    $0x38,%rsp
=> 0x4006b9 <+4>:      movq   $0xb4,0x28(%rsp)
```

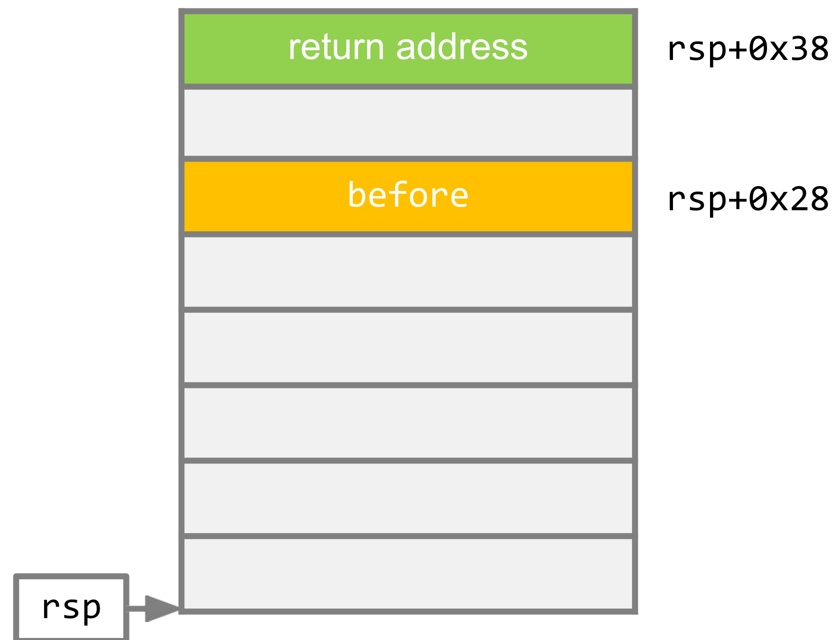
Addresses increase towards the top of the slide



Part 1: Drawing the stack diagram

```
   0x4006b5 <+0>:     sub    $0x38,%rsp
   0x4006b9 <+4>:     movq   $0xb4,0x28(%rsp)
=> 0x4006c2 <+13>:    movq   $0xaf,0x8(%rsp)
```

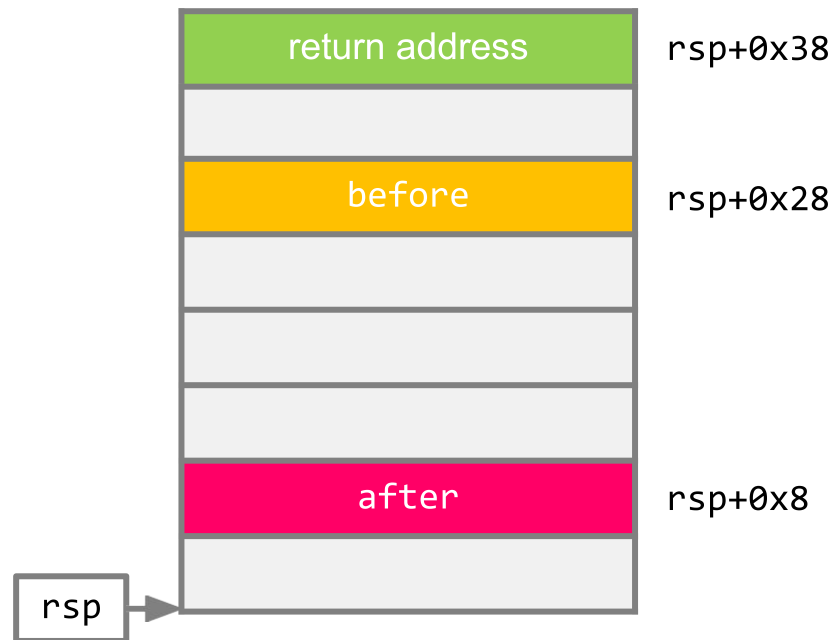
Addresses increase towards the top of the slide



Part 1: Drawing the stack diagram

```
   0x4006b5 <+0>:     sub    $0x38,%rsp
   0x4006b9 <+4>:     movq   $0xb4,0x28(%rsp)
   0x4006c2 <+13>:    movq   $0xaf,0x8(%rsp)
   0x4006cb <+22>:    lea    0x10(%rsp),%rdi
=> 0x4006d0 <+27>:    callq  0x40073f <Gets>
```

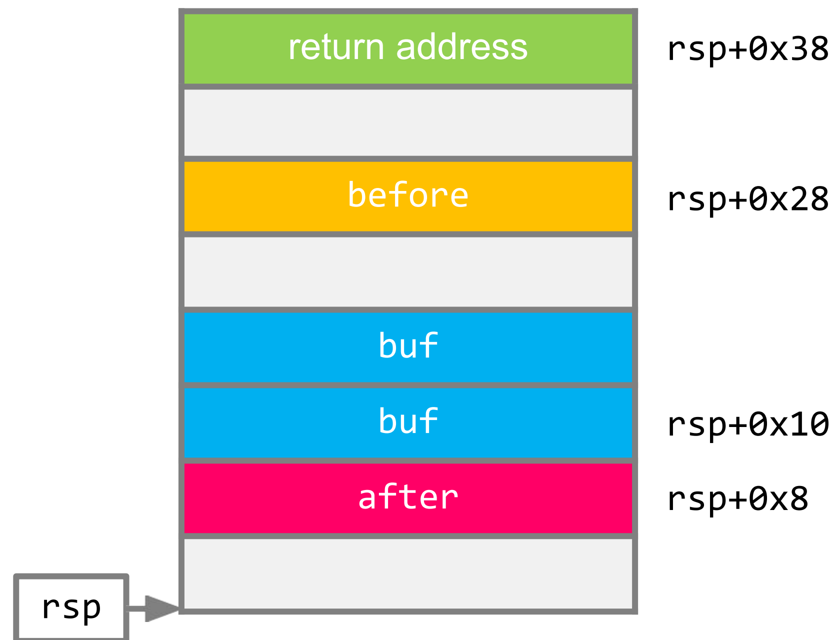
Addresses increase towards the top of the slide



Part 1: Drawing the stack diagram

```
   0x4006b5 <+0>:     sub    $0x38,%rsp
   0x4006b9 <+4>:     movq   $0xb4,0x28(%rsp)
   0x4006c2 <+13>:    movq   $0xaf,0x8(%rsp)
   0x4006cb <+22>:    lea    0x10(%rsp),%rdi
   0x4006d0 <+27>:    callq  0x40073f <Gets>
=> 0x4006d5 <+32>:    mov    0x28(%rsp),%rdx
```

Addresses increase towards the top of the slide

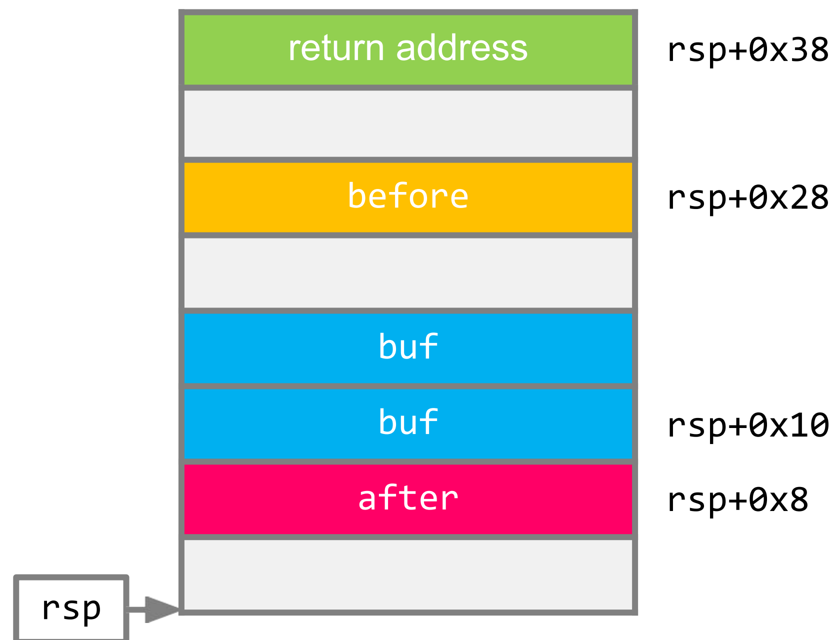


Part 1: Comparing with GDB output

Let's compare the stack diagram we drew with the actual values on the stack after `Gets()` returns.

```
      0x4006d0 <+27>:  callq 0x40073f <Gets>
   => 0x4006d5 <+32>:  mov   0x28(%rsp),%rdx
```

```
(gdb) break *0x4006d5
(gdb) run
Starting program: act1
abcdefgh12345678
(gdb) x/8gx $rsp
(gdb) x/64bx $rsp
```

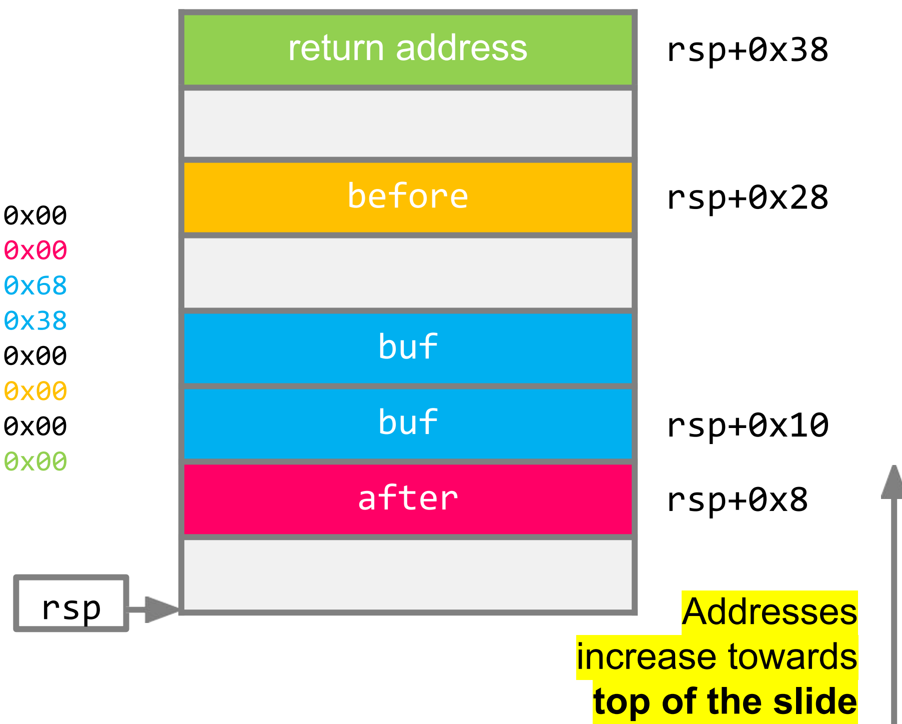


Part 1: Comparing with GDB output

```
(gdb) x/8gx $rsp
0x602020:    0x0000000000000000     0x00000000000000af
0x602030:    0x6867666564636261     0x3837363534333231
0x602040:    0x0000000000000000     0x00000000000000b4
0x602050:    0x0000000000000000     0x0000000000400783

(gdb) x/64bx $rsp
0x602020:   0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x602028:   0xaf   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x602030:   0x61   0x62   0x63   0x64   0x65   0x66   0x67   0x68
0x602038:   0x31   0x32   0x33   0x34   0x35   0x36   0x37   0x38
0x602040:   0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x602048:   0xb4   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x602050:   0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x602058:   0x83   0x07   0x40   0x00   0x00   0x00   0x00   0x00
```

Addresses increase towards **bottom of the slide**



Part 1: Comparing with GDB output

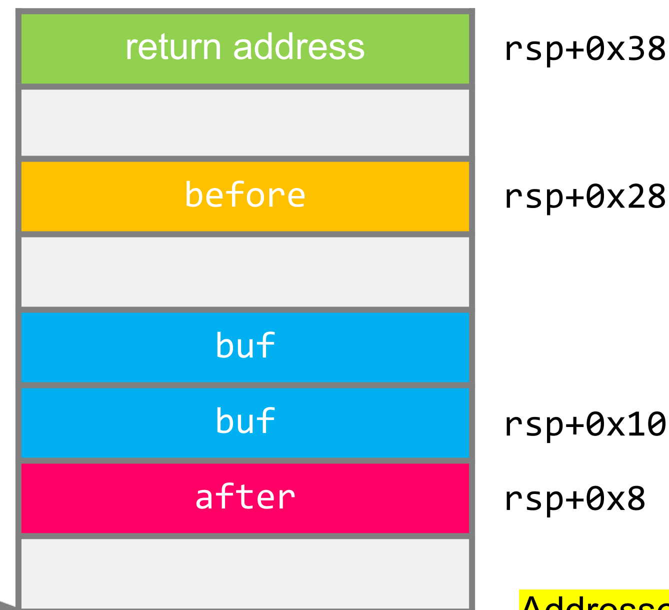
```
(gdb) x/8gx $rsp
0x602020:   0x0000000000000000   0x00000000000000af
0x602030:   0x6867666564636261   0x3837363534333231
0x602040:   0x0000000000000000   0x00000000000000b4
0x602050:   0x0000000000000000   0x0000000000400783

(gdb) x/64bx $rsp
0x602020:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x602028:
0x602030:
0x602038:  0x31  0x32  0x33  0x34  0x35  0x36  0x37  0x38
0x602040:
0x602048:  0xb4  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x602050:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x602058:  0x83  0x07  0x40  0x00  0x00  0x00  0x00  0x00
```

1. Input one byte at a time
2. Input one byte at a time
3. Input one byte at a time

Addresses increase towards **bottom of the slide**

Remember Endianness!



rsp

Addresses increase towards **top of the slide**

Part 1: Comparing with GDB output

```
(gdb) x/8gx $rsp
0x602020:    0x0000000000000000      0x00000000000000af
0x602030:    0x6867666564636261      0x3837363534333231
0x602040:    0x0000000000000000      0x00000000000000b4
0x602050:    0x0000000000000000      0x0000000000400783

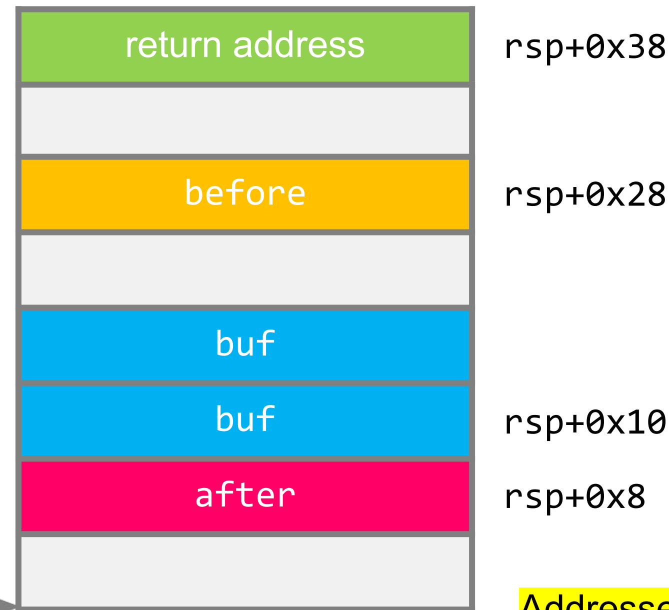
(gdb) x/64bx $rsp
0x602020:   0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x602028:   0xaf   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x602030:   0x61   0x62   0x63   0x64   0x65   0x66   0x67   0x68
0x602038:   [illegible]
0x602040:   [illegible]
0x602048:   0xb4   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x602050:   0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x602058:   0x83   0x07   0x40   0x00   0x00   0x00   0x00   0x00
```

Read the whole word this way

Addresses increase towards **bottom of the slide**

Remember Endianness!

rsp



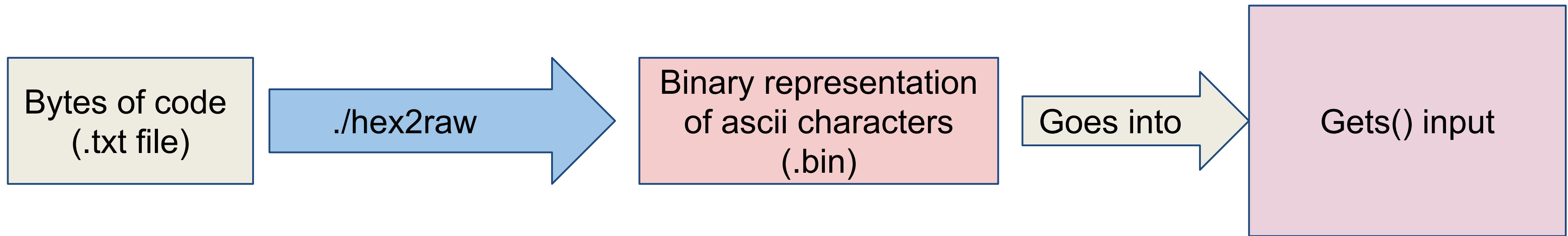
Addresses increase towards **top of the slide**

Part 1: Exploitation

- Try to find an input string that wins 1 cookie!
  - What do we need to overwrite `before` with if we want to have `before == 0x3331323531`?
- Constructing an exploit
  - `gets()` stops reading once it sees a newline. In the buffer, it replaces the newline with a null terminator.
  - gets() does **not** stop reading at a null terminator.

Putting together your input

- You want to directly change the bytes on the stack.
- However, gets() only accepts ascii characters as input
- To get around this? Use hex2raw, which is provided for you!



For this activity, ./hex2raw is automatically run when using make, so after you make all you have to do is do is

```
(gdb) run < inputs/input1.bin
```

Part 1: Recap

- Buffer overflows can **overwrite** parts of the stack frame, including other local variables
- Stack frames may include **padding**, so looking at the assembly is crucial to drawing a correct diagram
- GDB prints output starting at the **lowest** address, whereas our stack diagrams start at the **highest**

Procedure Calling Review

Call and return instructions

Which registers do callq and retq change?

%rax
%rdi
%rsi
%rdx
%rcx
%r8
%r9
%r10
%r11

%rbx
%r12
%r13
%r14
%rbp
%rsp
%rip

Call and return instructions

Which registers do callq and retq change?

%rax
%rdi
%rsi
%rdx
%rcx
%r8
%r9
%r10
%r11

%rbx
%r12
%r13
%r14
%rbp
%rsp
%rip

Stack/Procedure Review

```
0000000000400540 <multstore>:
  •
  •
=>400544: callq   400550 <mult2>
  400549: mov%rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:     mov  %rdi,%rax
  •
  •
  400557    retq
  •
```

0x130
0x128
0x120

•
•
•

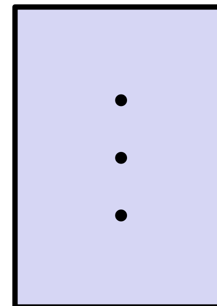
%rsp 0x120

%rip 0x400544

Stack/Procedure Review

```
0000000000400540 <multstore>:
  •
  •
=>400544: callq   400550 <mult2>
  400549: mov%rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:     mov  %rdi,%rax
  •
  •
  400557    retq
  •
```



%rsp 0x120

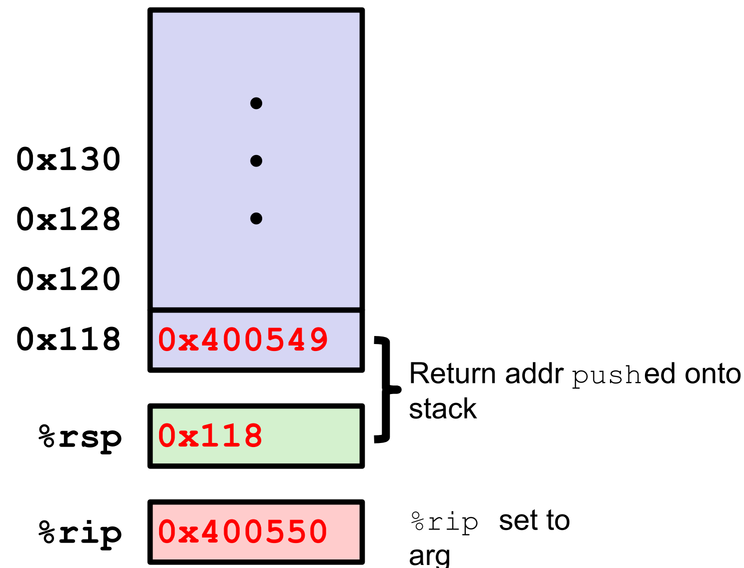
%rip 0x400544

What happens next?

Stack/Procedure Review

```
0000000000400540 <multstore>:
  •
  •
  400544: callq   400550 <mult2>
  400549: mov%rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
=>400550:     mov  %rdi,%rax
  •
  •
  400557    retq
  •
```



Stack/Procedure Review

```
0000000000400540 <multstore>:
  •
  •
  400544: callq   400550 <mult2>
  400549: mov%rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:     mov  %rdi,%rax
  •
  •
=>400557   retq
```

0x130	•
0x128	•
0x120	•
0x118	0x400549

%rsp	0x118
%rip	0x400557

Stack/Procedure Review

```
0000000000400540 <multstore>:
  •
  •
  400544: callq   400550 <mult2>
=>400549: mov %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:     mov  %rdi,%rax
  •
  •
  400557    retq
```

Address	Stack
	•
0x130	•
0x128	•
0x120	
0x118	0x400549

Register	Value
%rsp	0x120
%rip	0x400549

Stack `pop` to `%rip`

Let's Rewind…

```
0000000000400540 <multstore>:
  •
  •
  400544: callq   400550 <mult2>
  400549: mov%rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov  %rdi,%rax
  • ?????
  • ?????
=>400557:  retq
```

0x130	•
0x128	•
0x120	•
0x118	0xbadbad

%rsp	0x118
%rip	0x400557

What if we mess up the return address?

Activity 2

Part 2: Exploitation

- Hijacking control flow
  - Is it possible to overwrite `after`? If not, what parts of the stack frame *can* we overwrite?
  - Is there anywhere we could jump to call `win(0x18213)`?
- Constructing an exploit

inputs/input2.txt

```
48 65 6c 6c 6f 20 31 35
32 31 33 21   # comment
```

`make`
(runs `hex2raw`)

inputs/input2.bin

```
Hello 15213!
```

Part 2: Exploitation

- Hijacking control flow
  - Is it possible to overwrite `after`? If not, what parts of the stack frame *can* we overwrite? No, the buffer is stored at a higher memory address than `after`.
  - Is there anywhere we could jump to call `win(0x18213)`? We can overwrite the return address to return to just after the conditional where we call win(0x18213)

```
void solve(void)
    { long before
    = 0xb4; char
    buf[16];
    long after = 0xaf;

    Gets(buf);

    if (before ==
        0x3331323531)
        win(0x15213);

    if (after ==
        0x3331323831)
        win(0x18213);
```

Part 2: Exploitation

- Organize back into your groups and try to call win(0x18213) to earn 2 cookies!
- Hint: gdb activity again and disassemble solve. (gdb) disas solve
- Find out what address you need to return to in order to call win(0x18213), and overwrite the return address of solve to return to that.

Solution

- Disassembling solve, we see that at address 0x0000000000400707 we move 0x18213 to %rdi and call win.

- We need to overwrite the return address of solve, which is 40 bytes past where we read into from gets.

```
=> 0x00000000004006b5 <+0>:     sub    $0x38,%rsp
   0x00000000004006b9 <+4>:     movq   $0xb4,0x28(%rsp)
   0x00000000004006c2 <+13>:    movq   $0xaf,0x8(%rsp)
   0x00000000004006cb <+22>:    lea    0x10(%rsp),%rdi
   0x00000000004006d0 <+27>:    callq  0x40073f <Gets>
   0x00000000004006d5 <+32>:    mov    0x28(%rsp),%rdx
   0x00000000004006da <+37>:    movabs $0x3331323531,%rax
   0x00000000004006e4 <+47>:    cmp    %rax,%rdx
   0x00000000004006e7 <+50>:    jne    0x4006f3 <solve+62>
   0x00000000004006e9 <+52>:    mov    $0x15213,%edi
   0x00000000004006ee <+57>:    callq  0x40064d <win>
   0x00000000004006f3 <+62>:    mov    0x8(%rsp),%rdx
   0x00000000004006f8 <+67>:    movabs $0x3331323831,%rax
   0x0000000000400702 <+77>:    cmp    %rax,%rdx
   0x0000000000400705 <+80>:    jne    0x400711 <solve+92>
   0x0000000000400707 <+82>:    mov    $0x18213,%edi
   0x000000000040070c <+87>:    callq  0x40064d <win>
   0x0000000000400711 <+92>:    add    $0x38,%rsp
   0x0000000000400715 <+96>:    retq
```

Part 2: Recap

- `retq` always jumps to the **saved return address**, which it pops off the stack (at `rsp`).
- **Overwriting** the saved return address on the stack allows us to "fool" `retq`, and transfer control to an arbitrary instruction.

Activity 3
(If Time)

Part 3: Return Oriented Programming

- Goal: call win(0x18613)

- Notice the suspiciously named function gadget1
  - (gdb) disas gadget1

- Can we return to this code and use it so that our first argument to win is 0x18613?

Solution

- We notice gadget1 conveniently pops a value from the stack and stores it in %rdi

- We can store 0x18613 on the stack, overwrite the original return address of solve to return to gadget1, have gadget1 pop 0x18613 into %rdi, then have gadget1 return to the call to win at address 0x000000000040070c

```
Dump of assembler code for function gadget1:
   0x0000000000400777 <+0>:     pop    %rdi
=> 0x0000000000400778 <+1>:     retq
```

Attack Lab Tools

- ```
  $ gcc -c test.s
  $ objdump -d test.o
  ```
  Compiles the assembly code in test.s, then shows the disassembled instructions along with the actual bytes.

- ```
  $ ./hex2raw < exploit.txt > exploit.bin
  ```
  Convert hex codes into raw binary strings to pass to targets.

- ```
  (gdb) display /12gx $rsp
  (gdb) display /2i $rip
  ```
  Displays 12 elements on the stack and the next 2 instructions to run
  GDB is also useful to for tracing to see if an exploit is working.
  ```

If you get stuck

- **Please read the writeup carefully.** Not everything will make sense on the first read-through.
- Other resources you can make use of:
  - CS:APP Chapter 3
  - Lecture slides and videos
  - x86-64 and GDB cheat sheets under Resources