

# 15-213/15-513 Recitation: Attack Lab

Your TAs

Monday, Feb 12th, 2024

# Agenda

- Attack Lab Overview
- Stacks Review
- Activity 1
- Procedure Calling Review
- Activity 2

# Quick Reminders

- Attack Lab is due this Thursday
  - You can use one grace day
- Written 3 is due Wednesday
- Written 2 peer reviews are due Wednesday

# Learning objectives

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

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

Refer to Lecture from Tuesday:

*Machine-Level Programming V: Advanced Topics*

# Structure Alignment Review

# Primitive Types

- char = 1 byte
- short = 2 bytes
- int = 4 bytes
- long = 8 bytes
- double = 8 bytes
- pointer = 8 bytes

\*(on Linux under x86\_64, machine dependent)

# Alignment Principles

- A structure has the alignment of its largest field/element.
- The initial address and size of the struct must be a multiple of its alignment.

# What's the Size?

```
struct x {  
    int a;  
    char *b;  
    char c[4];  
    struct {  
        short d;  
        double e;  
    } y;  
};
```

- What is struct y's alignment?
- What is the size of y in bytes? i.e what would sizeof(y) return?
- What is struct x's alignment?
- What is the size of struct x in bytes?
- How could we arrange elements in the struct to save space?



# Lab Overview

# 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:

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

## Shrinking the stack:

# 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`

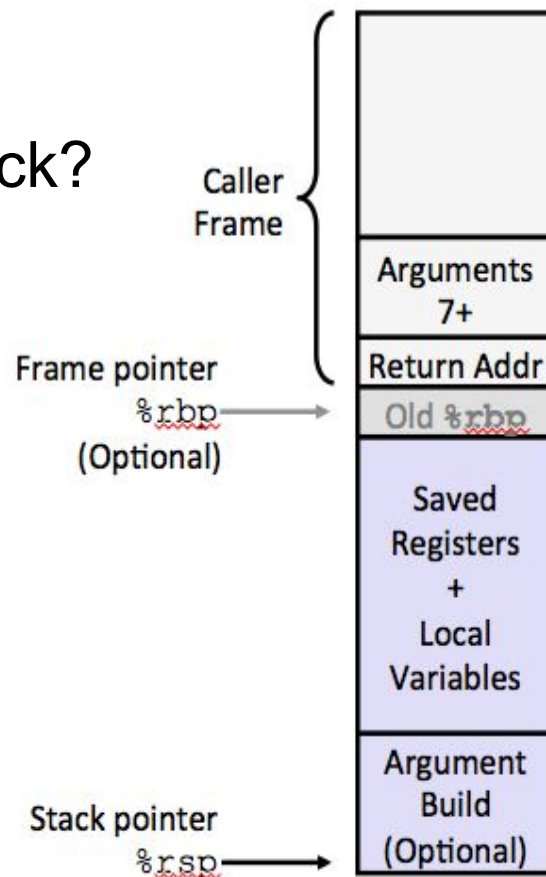
# x86-64 Stack Frames

What kinds of data are stored on the stack?

# x86-64 Stack Frames

What kinds of data are stored on the stack?

- Saved registers
- Local variables
- Arguments (7+)
- Saved return address

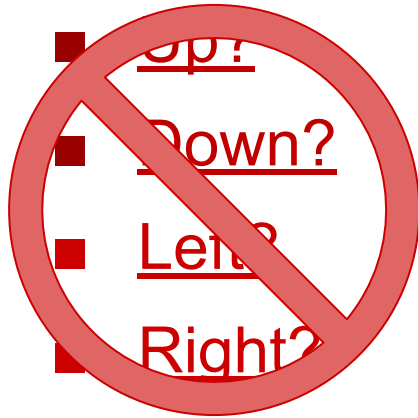




# Which way does the stack grow?

- 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

Carnegie Mellon

### Buffer Overflow Stack Example

*Before call to gets*

Stack Frame for call\_echo

00	00	00	00
00	40	06	c3

20 bytes unused

[3]	[2]	[1]	[0]
-----	-----	-----	-----

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    ...
}
```

```
echo:
    subq $0x18, %rsp
    movq %rsp, %rdi
    call gets
    ...
```

```
call_echo:
    ...
    4006be: callq 4006cf <echo>
    4006c3: add $0x8, %rsp
    ...
```

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

## Everything else

Carnegie Mellon

### Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

zip\_dig cmu;

1	5	2	1	3
16	20	24	28	32

zip\_dig mit;

0	2	1	3	9
36	40	44	48	52

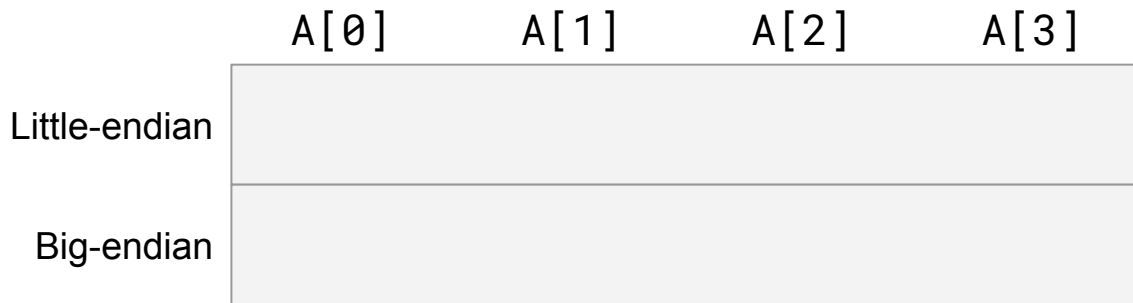
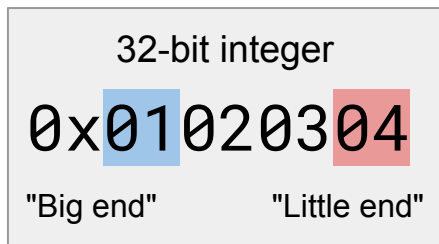
zip\_dig ucb;

9	4	7	2	0
56	60	64	68	72

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

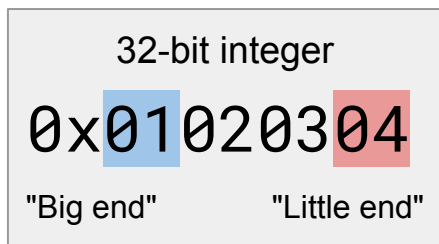
# Endianness

- Describes how integers are represented as bytes.
- Little-endian means that the **least-significant** 8 bits of an integer are stored at the lowest address.



# Endianness

- Describes how integers are represented as bytes.
- Little-endian means that the **least-significant** 8 bits of an integer are stored at the lowest address.



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

## But wait - types are a lie

- Bytes in memory are just bytes: typing is just a way for the compiler to specify how they should be treated.
- E.g. use `x/ 8i (addr)` vs `x/ 8gx (addr)`. They're different!

JULIA EVANS  
@b0rk

## 8 bytes, many meanings

The same bytes can mean many different things. Here are 8 bytes and a bunch of things they could potentially mean:

0x.....0	0x....7							
63	6f	6d	70	75	74	65	72	
c	o	m	p	u	t	e	r	8 ASCII characters
99	111	109	112	117	116	101	114	8 8-bit integers
28515		28781		29813		29285		4 unsigned 16-bit integers
1886220131				1919251573				2 unsigned 32-bit integers
8243122740717776739								1 unsigned 64-bit integer
0x72657475706d6f63								a 64-bit pointer
99.111.109.112				117.116.101.114				2 IPv4 addresses
arpl	WORD PTR	jo 0x7a		je 0x6c		.byte 0x72		x86 machine code
[edi+0x6d],bp								
c	o	m	p	u	t	29285		6 ASCII characters + 1 16-bit integer
2.93930e+29				4.54482e+30				2 32-bit floating point numbers
1.144493e+243								1 64-bit floating point numbers
rgba(99, 111, 109, 0.44)				rgba(117, 116, 101, 0.45)				2 RGBA colours

don't worry if you don't understand all this right now! We'll explain everything



# 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. **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/rec5.tar  
$ tar xvf rec5.tar  
$ cd rec5
```

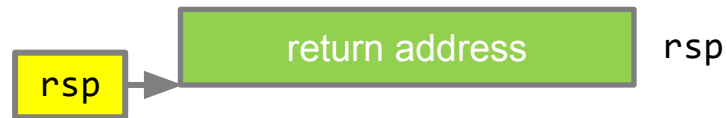
- Take a look at the code in `src/activity.c`.

# Part 1: Diving into assembly

- Look at the disassembly of `so1ve()`.
- 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?**

# Part 1: Drawing the stack diagram

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



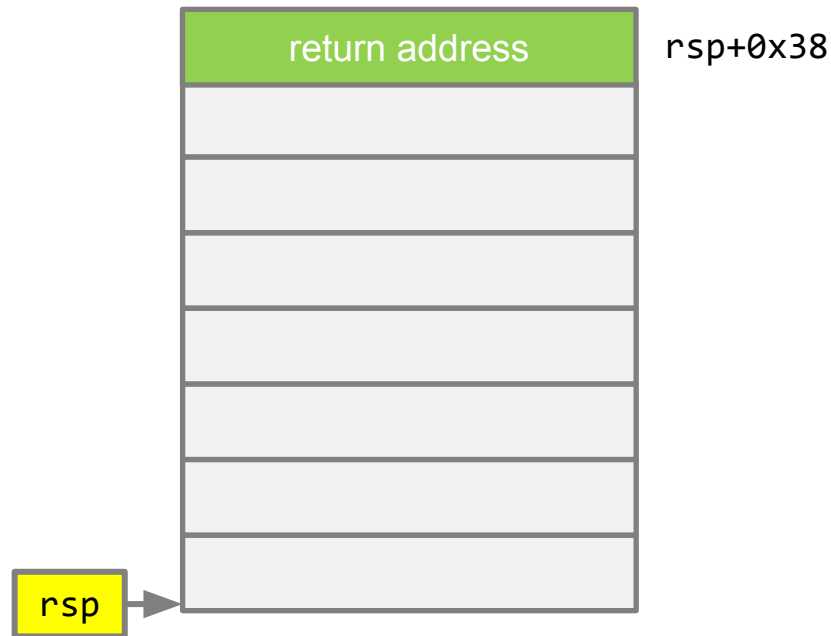
Addresses  
increase towards  
the top of the slide



# Part 1: Drawing the stack diagram

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

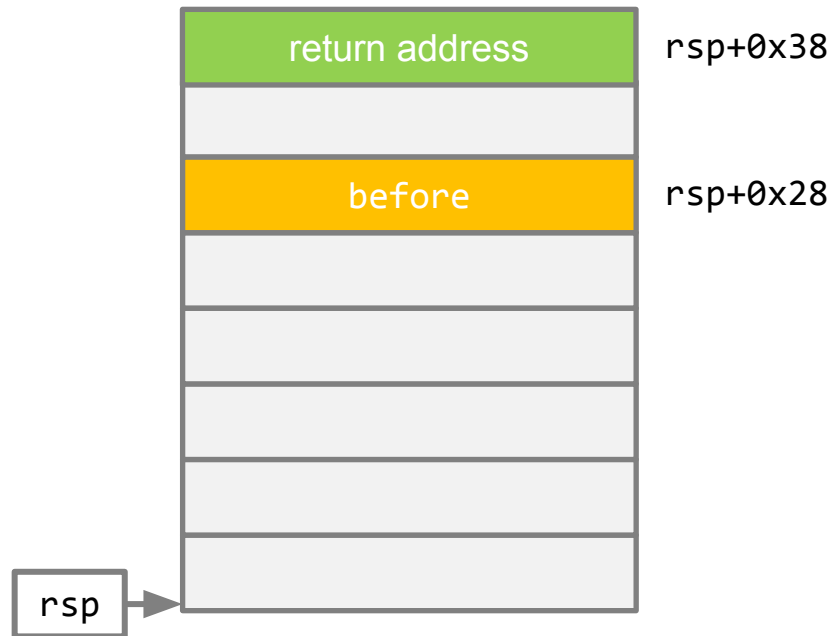
Addresses  
increase towards  
the top of the slide



# Part 1: Drawing the stack diagram

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

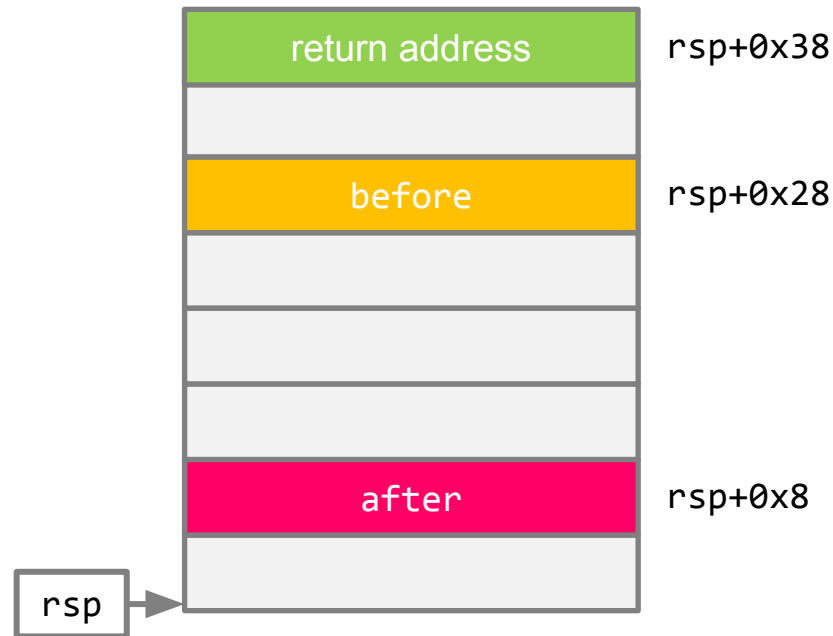
Addresses  
increase towards  
the top of the slide



# 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

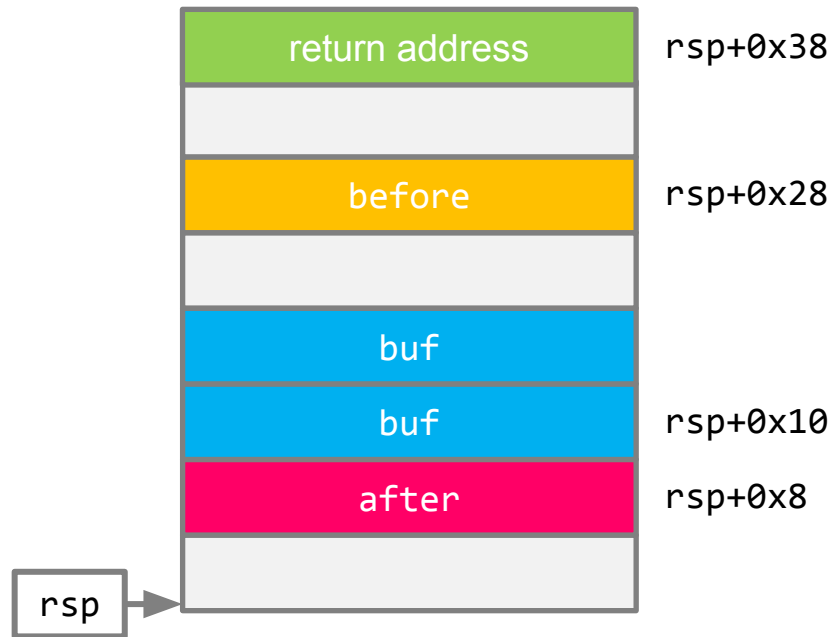




# Part 1: Drawing the stack diagram

```
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

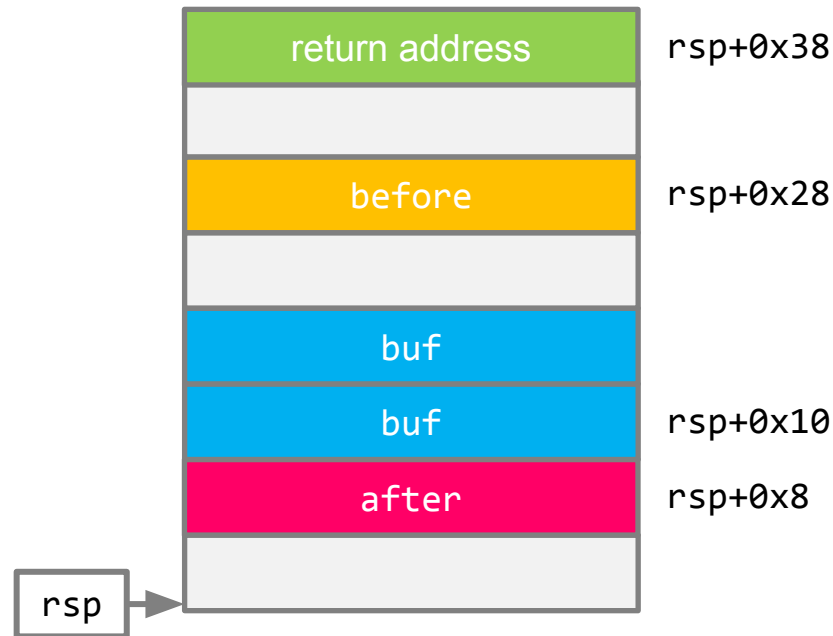


# Part 1: Comparing with GDB output

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
```

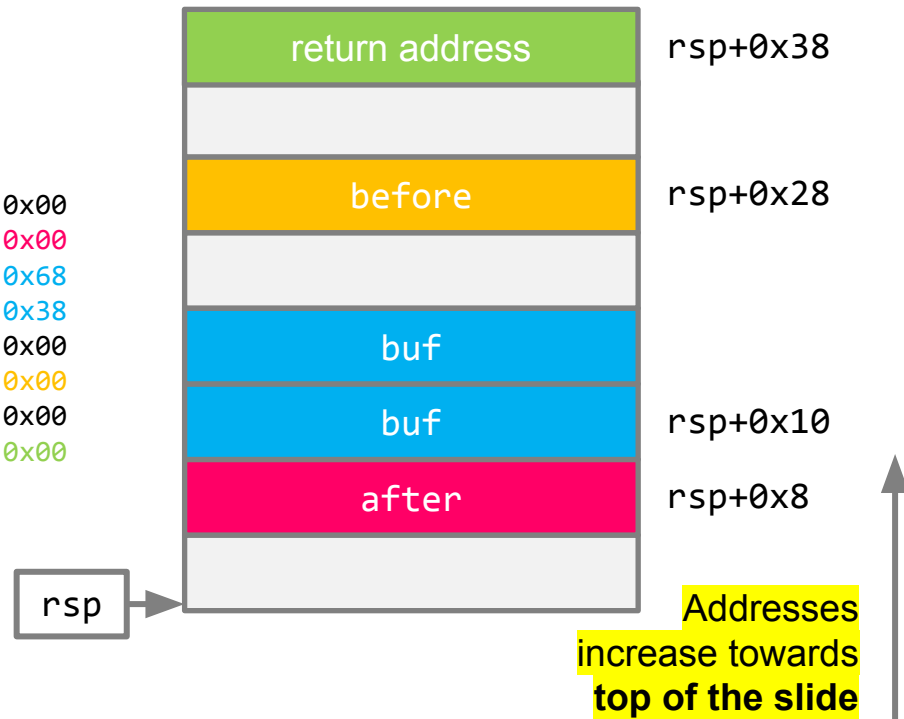
```
(gdb) break *0x4006d5  
(gdb) run  
Starting program: act1  
abcdefgh12345678  
(gdb) x/8gx $rsp  
(gdb) x/64bx $rsp
```



# 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 0x00
0x602028: 0xaf 0x00 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

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.

# 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?

<code>%rax</code>
<code>%rdi</code>
<code>%rsi</code>
<code>%rdx</code>
<code>%rcx</code>
<code>%r8</code>
<code>%r9</code>
<code>%r10</code>
<code>%r11</code>

<code>%rbx</code>
<code>%r12</code>
<code>%r13</code>
<code>%r14</code>
<code>%rbp</code>
<code>%rsp</code>
<code>%rip</code>

# Call and return instructions

Which registers do `callq` and `retq` change?

<code>%rax</code>
<code>%rdi</code>
<code>%rsi</code>
<code>%rdx</code>
<code>%rcx</code>
<code>%r8</code>
<code>%r9</code>
<code>%r10</code>
<code>%r11</code>

<code>%rbx</code>
<code>%r12</code>
<code>%r13</code>
<code>%r14</code>
<code>%rbp</code>
<code>%rsp</code>
<code>%rip</code>



# 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
```

0x130

0x128

0x120

%rsp 0x120

%rip 0x400544

What happens next?

# Stack/Procedure Review

```

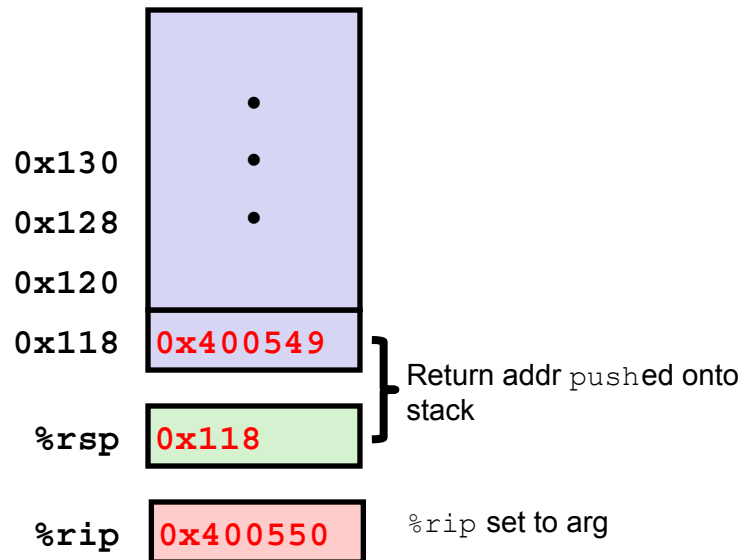
0000000000400540 <multstore>:
.
.
400544: callq 400550 <mult2>
400549: mov  %rax, (%rbx)
.
.

```

```

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
```

0x130

0x128

0x120

0x118

0x400549

%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?

# 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.

- `(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 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](#)



# Appendix

# 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: 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.