15-213 Recitation
Attack Lab

Your TAs

Friday, September 20th

Reminders

- **`bomblab`** was due yesterday (September 19th)
- **`attacklab`** has been released, and is due on ***Thursday (September 26th)***

Agenda

- Review: Structs and Alignment
- Review: Calling Procedures, Stack Frames
- Stacks
- Endianness
- Intro to Attack Lab
- Activity!

Review: `structs`

Alignment Requirements

- Badly aligned data can harm performance:
  - e.g. may need multiple memory accesses instead of just one.

- *Primitive* types have pre-determined alignments (machine dependent):
  - **`char`** = 1 byte
  - **`short`** = 2 bytes
  - **`int`** = 4 bytes
  - **`long`** = 8 bytes
  - **`double`** = 8 bytes
  - **`pointer`** = 8 bytes

Alignment Requirements: Compound Types

- Compound types:
  - Arrays
  - Structs
  - Unions
- Alignment rules for these types:
  1. Alignment requirement of the type = Largest alignment requirement of its fields/elements.
  2. Initial address and size must both be multiples of the alignment requirement.

Alignment Requirements: Example

```
double d;
```

- What is the alignment requirement for **d**?
  - *Primitive:* has pre-defined alignment requirement.
  - **Alignment: 8**
- What is its size?
  - **Size: 8 bytes**

Alignment Requirements: Example

```
struct y {
   double d;
}
```

- What is the alignment requirement for **y**?
  - Rule (1): struct alignment = max alignment of fields.
  - **Alignment: 8**
- What is its size?
  - **Size: 8 bytes**

Alignment Requirements: Example

```
struct y {
   short c;
   double d;
}
```

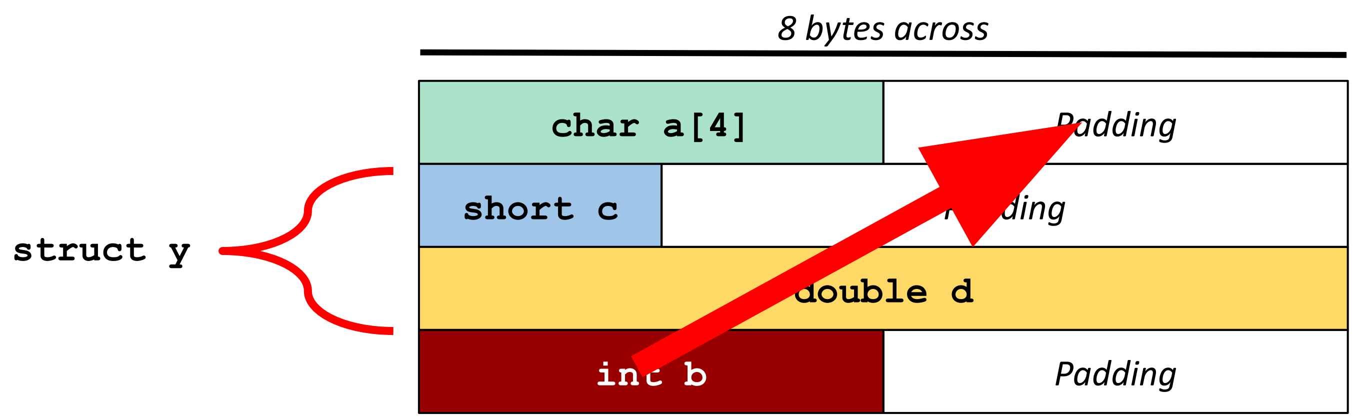
- What is the alignment requirement for **y**?
  - Rule (1): struct alignment = max alignment of fields.
  - **Alignment: 8**
- What is its size?
  - Rule (2): have to add padding after **c** so that **d** is 8-byte aligned
  - **Size: 16 bytes**

Alignment Requirements: Example

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

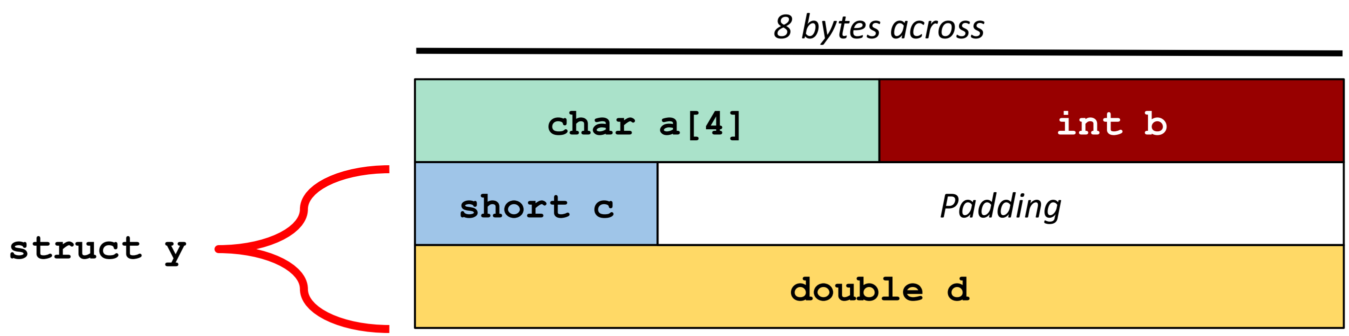
- What is the alignment requirement for **x**?
  - Rule (1): struct alignment = max alignment of fields.
  - **Alignment: 8**
- What is its size?
  - Rule (2): have to add padding after **a** so that **y** is 8-byte aligned
  - Rule (2): have to add padding after **b** so that size of **x** is multiple of 8.
  - **Size: 32 bytes**

`structs`: Reordering Fields



- **`struct x`** takes up *32 bytes*.
- Can we reorder the fields to do better?

`structs`: Reordering Fields



- **`struct x`** now takes up 24 bytes!
- Compiler *cannot* do this optimization. It's up to the programmer (you!)
- *Note:* Can't move field into or out of **`y`** without also changing how you access those fields in your code.

Review: Calling Procedures, Stack Frames

Review: Calling Procedures

Procedure Call: **`call label`**

- Push *return address* onto the stack (so that we can pass control back to the caller!)
- Jump to **`label`**

Procedure Return: **`ret`**

- Pop address from stack
  - This is the address of the next instruction of the *caller*
- Jump to that address

Example

```
int outer_function() {
    int result = inner_function(1, 2, 3, 4, 5, 6, 7, 8, 9);

    return result + 1;
}
```

Lots of arguments!

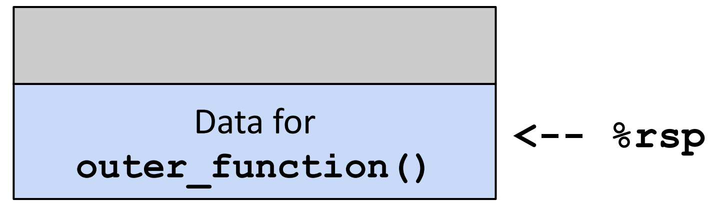
Example: outer_function() calls inner_function()

```
%rip -->
00000000004011ba <outer_function>:
...
4011c6: push $0x9
4011c8: push $0x8
4011ca: push $0x7

4011cc: mov $0x6,%r9d
4011d2: mov $0x5,%r8d
4011d8: mov $0x4,%ecx
4011dd: mov $0x3,%edx
4011e2: mov $0x2,%esi
4011e7: mov $0x1,%edi

4011ec: call 401136 <inner_function>
4011f1: add  $0x20,%rsp
...
```

Push extra arguments onto stack (note the order!)



Example: `outer_function()` calls `inner_function()`

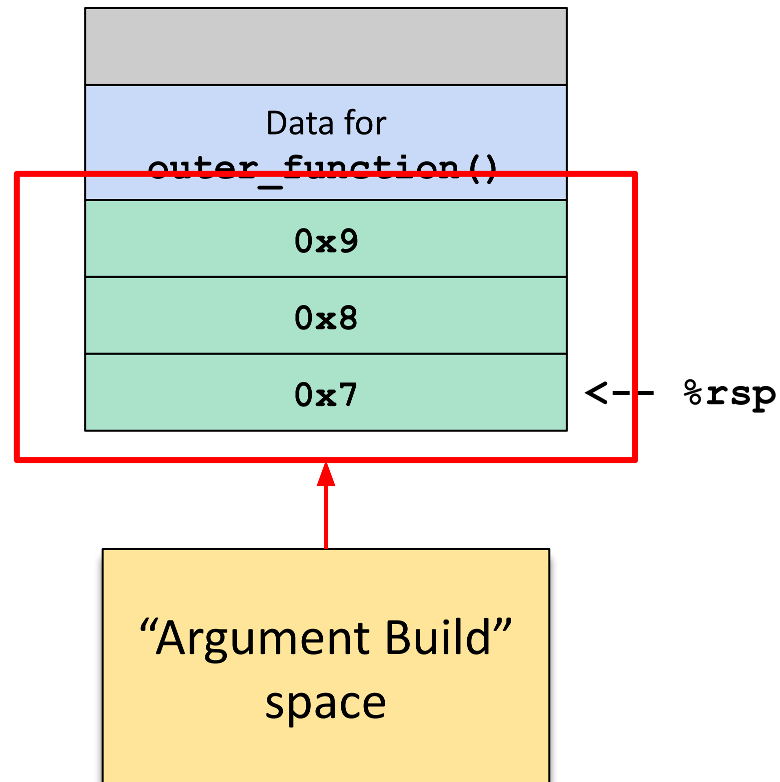
```
00000000004011ba <outer_function>:
...
4011c6: push  $0x9
4011c8: push  $0x8
4011ca: push  $0x7

4011cc: mov $0x6,%r9d
4011d2: mov $0x5,%r8d
4011d8: mov $0x4,%ecx
4011dd: mov $0x3,%edx
4011e2: mov $0x2,%esi
4011e7: mov $0x1,%edi

4011ec: call 401136 <inner_function>
4011f1: add  $0x20,%rsp
...
```

%rip

Push extra arguments onto stack (note the order!)



Example: outer_function() calls inner_function()

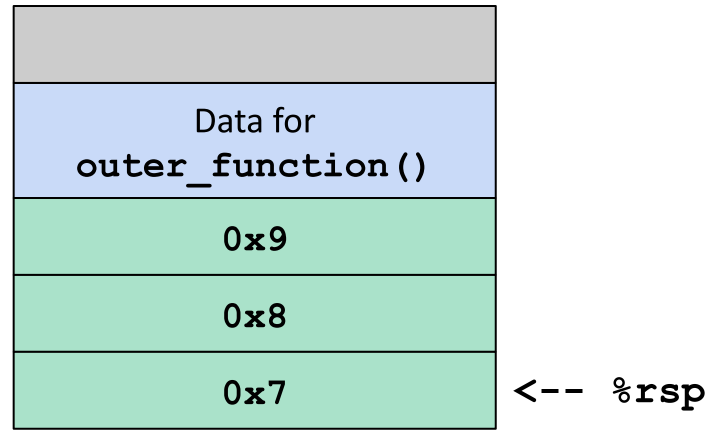
```
00000000004011ba <outer_function>:
...
4011c6:  push $0x9
4011c8:  push $0x8
4011ca:  push $0x7

4011cc: mov $0x6,%r9d
4011d2: mov $0x5,%r8d
4011d8: mov $0x4,%ecx
4011dd: mov $0x3,%edx
4011e2: mov $0x2,%esi
4011e7: mov $0x1,%edi

4011ec: call 401136 <inner_function>
4011f1: add  $0x20,%rsp
...
```

%rip → 4011ec

Load up first 6 arguments



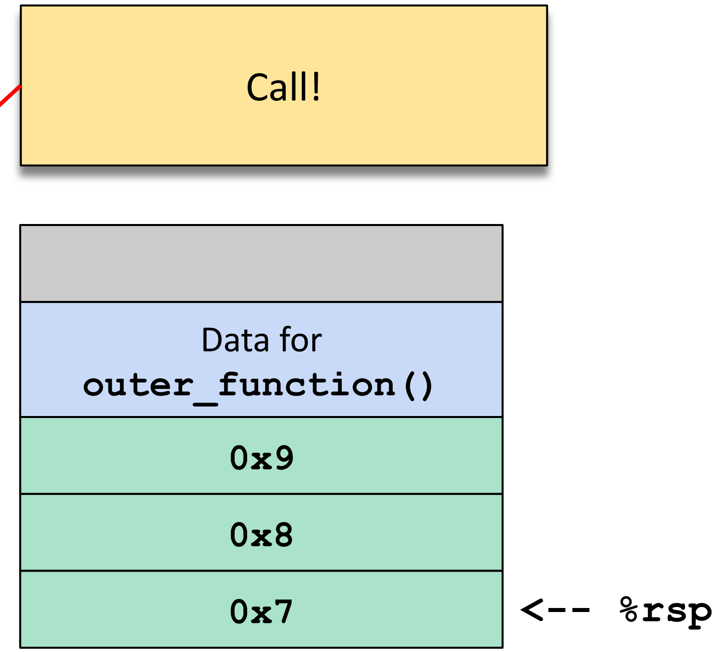
Example: outer_function() calls inner_function()

```
00000000004011ba <outer_function>:
...
4011c6:  push $0x9
4011c8:  push $0x8
4011ca:  push $0x7

4011cc: mov $0x6,%r9d
4011d2: mov $0x5,%r8d
4011d8: mov $0x4,%ecx
4011dd: mov $0x3,%edx
4011e2: mov $0x2,%esi
4011e7: mov $0x1,%edi

4011ec: call 401136 <inner_function>
4011f1: add  $0x20,%rsp
...
```

%rip → 4011ec



Example: `outer_function()` calls `inner_function()`

```
00000000004011ba <outer_function>:
...
4011c6:  push $0x9
4011c8:  push $0x8
4011ca:  push $0x7

4011cc: mov $0x6,%r9d
4011d2: mov $0x5,%r8d
4011d8: mov $0x4,%ecx
4011dd: mov $0x3,%edx
4011e2: mov $0x2,%esi
4011e7: mov $0x1,%edi

4011ec: call 401136 <inner_function>
4011f1: add  $0x20,%rsp
...
```

Call!

Data for `outer_function()`
`0x9`
`0x8`
`0x7`
`0x4011f1` <-- `%rsp`

```
(gdb) x /4gx $rsp
0x7fffffffe3c8: 0x00000000004011f1     0x0000000000000007
0x7fffffffe3d8: 0x0000000000000008     0x0000000000000009
```

Push address of *next* instruction to be executed

```
00000000000401136 <inner_function>:
401136:  endbr64
...
```

`%rip` → `401136`

Pass control to `inner_function()`

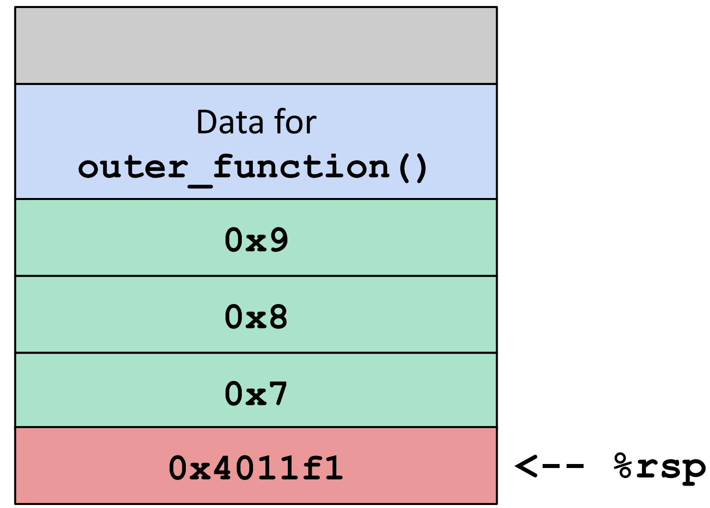
Example: `outer_function()` calls `inner_function()`

```
0000000000401136 <inner_function>:
401136:   endbr64
40113a:   sub $0x38,%rsp     <-- %rip

...

4011b5:   add $0x38,%rsp
4011b9:   ret
```

Function allocates any space it needs



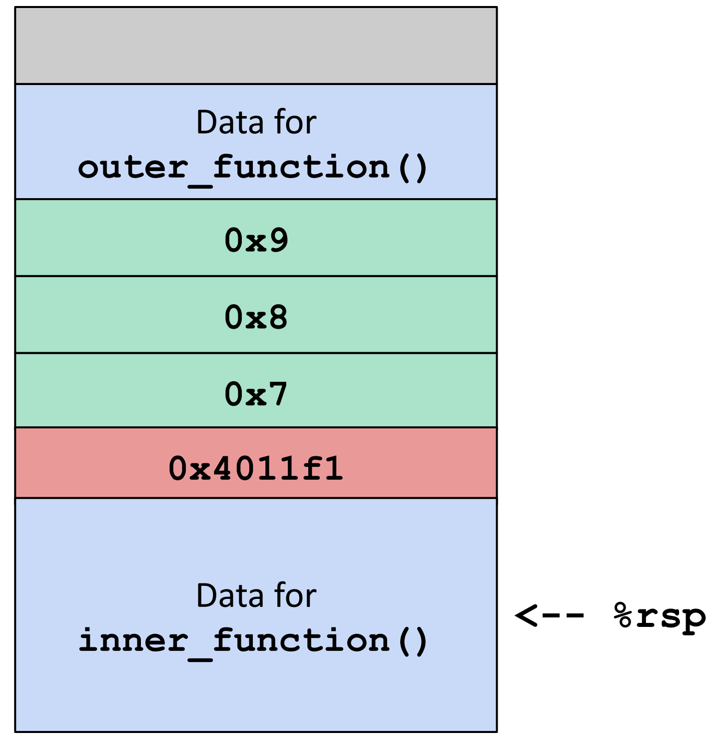
Example: `outer_function()` calls `inner_function()`

```
0000000000401136 <inner_function>:
401136:   endbr64
40113a:   sub $0x38,%rsp

%rip -> ...

4011b5:   add $0x38,%rsp
4011b9:   ret
```

Function allocates any space it needs



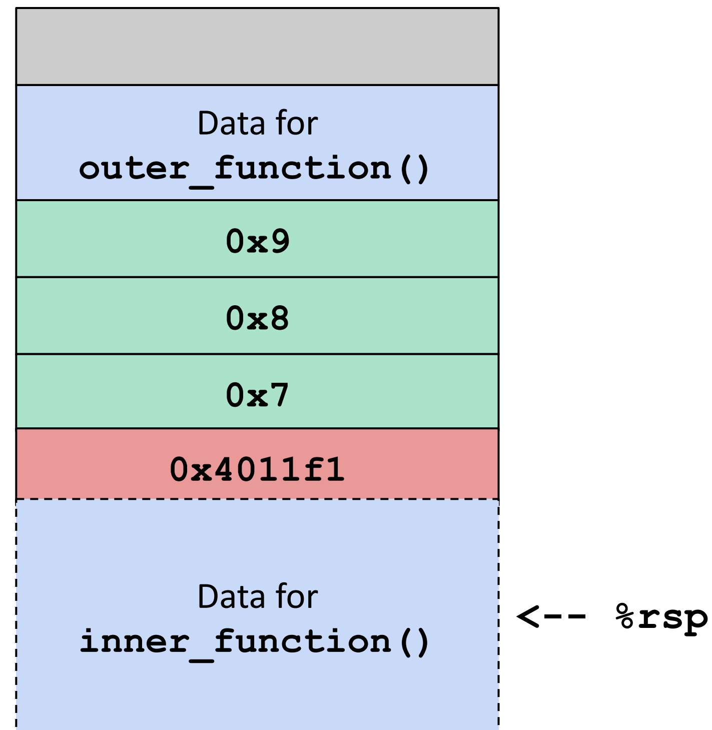
Example: outer_function() calls inner_function()

```
0000000000401136 <inner_function>:
401136:   endbr64
40113a:   sub $0x38,%rsp

...

4011b5:   add $0x38,%rsp      <- %rip
4011b9:   ret
```

De-allocate space



Example: outer_function() calls inner_function()

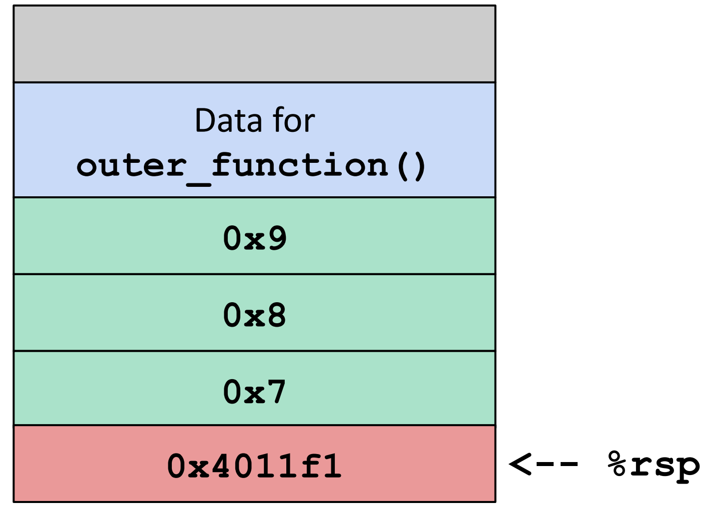
```
0000000000401136 <inner_function>:
401136:   endbr64
40113a:   sub $0x38,%rsp

...

4011b5:   add $0x38,%rsp
4011b9:   ret
```

%rip → 4011b9

De-allocate space → add $0x38,%rsp



Example: outer_function() calls inner_function()

```
0000000000401136 <inner_function>:
401136:   endbr64
40113a:   sub $0x38,%rsp

...

4011b5:   add $0x38,%rsp
4011b9:   ret
```

%rip → 4011b9: ret

Pop return address from stack, and jump to it

Data for outer_function()
0x9
0x8
0x7
0x4011f1 <-- %rsp

Load popped address into %rip

```
00000000004011ba <outer_function>:
...
4011ec: call 401136 <inner_function>
4011f1: add  $0x20,%rsp
...
```

Example: outer_function() calls inner_function()

```
0000000000401136 <inner_function>:
401136:   endbr64
40113a:   sub $0x38,%rsp

...

4011b5:   add $0x38,%rsp
4011b9:   ret
```

Pop return address from stack, and jump to it

Data for outer_function()
0x9
0x8
0x7 <-- %rsp

```
00000000004011ba <outer_function>:
...
4011ec: call 401136 <inner_function>
%rip → 4011f1: add  $0x20,%rsp
...
```

Stacks

Manipulating the Stack

- We saw that certain instructions ***grow*** the stack, and that certain instructions ***shrink*** the stack:

Growing the stack

- `sub 0x38, %rsp`
- `push %rbp`
- `call`

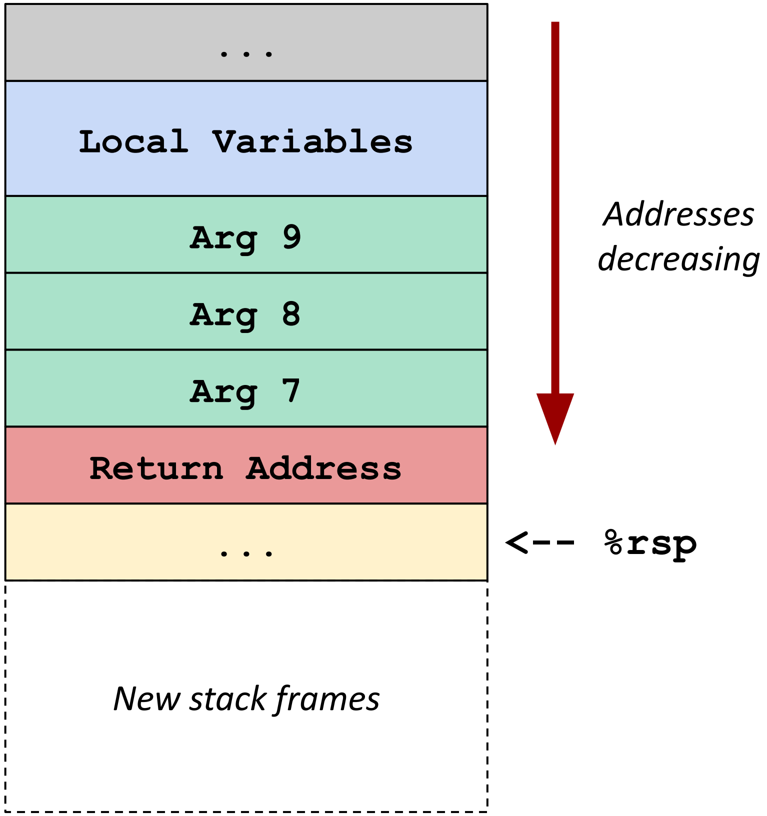
Shrinking the stack

- `add 0x38, %rsp`
- `pop %rbp`
- `ret`

- But what does this look like in memory?

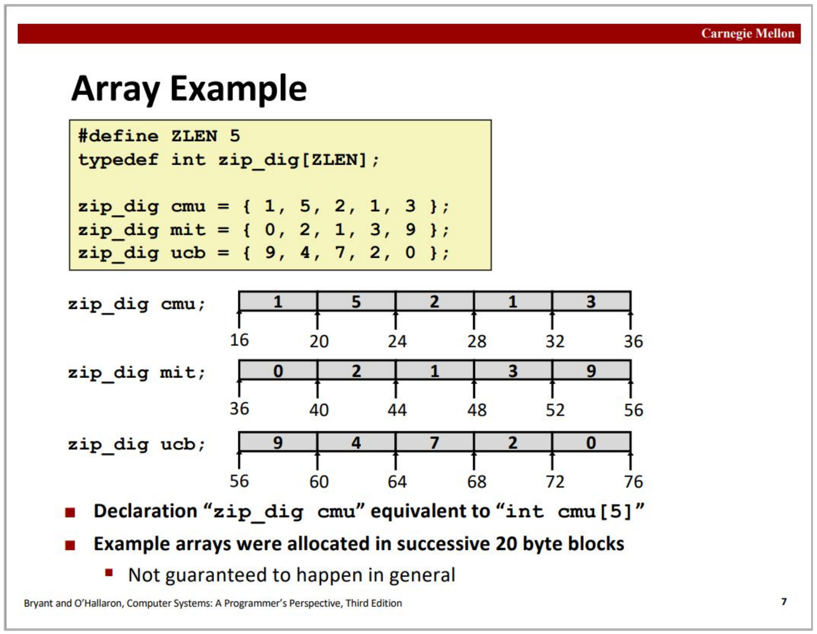
Which way does the stack grow?

- We say that the stack grows ***"down"*** because it grows towards ***lower addresses***:
  - e.g. **`sub 0x38, %rsp`**
- We will draw them this way in **`attacklab`** examples, too.
  - But you can draw them in any way that makes sense to you!

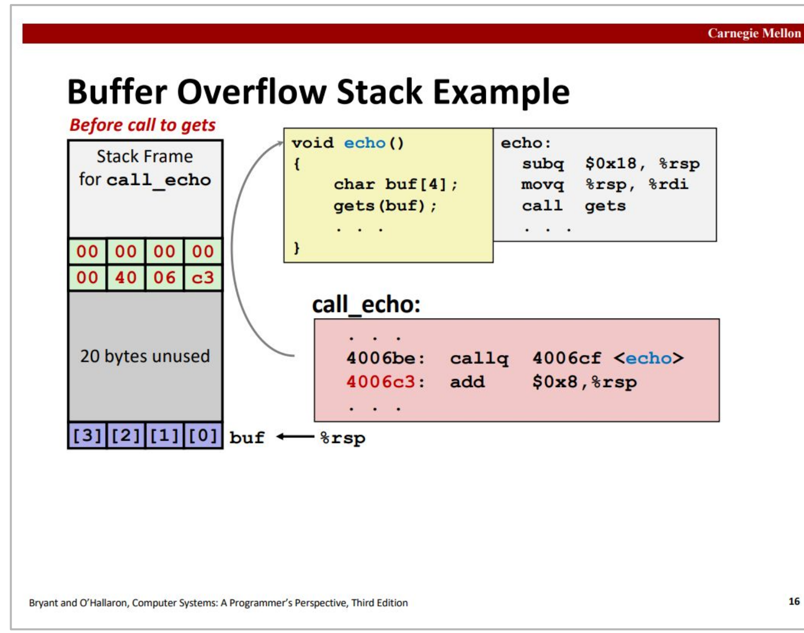


Drawing Memory

Conventional Memory Diagram



Stack Diagram



Addresses Increasing:

- Towards the <u>right</u>
- Then <u>downwards</u>

Addresses Increasing:

- Towards the <u>left</u>
- Then <u>upwards</u>

Endianness

Endianness

- Under the hood, we represent everything as a series of contiguous *bytes.*
- ***Endianness*** refers to how we order the ***bytes*** for **“simple”** types (integers and floats).

Endianness

- ***Little-Endian***:
  - *Least* significant byte is stored at the *lowest* address.
  - Shark Machines are Little-Endian.
  - Can assume everything in this class is little-endian unless otherwise stated.
- ***Big-Endian***:
  - *Most* significant byte is stored at the *lowest* address.

32-bit integer

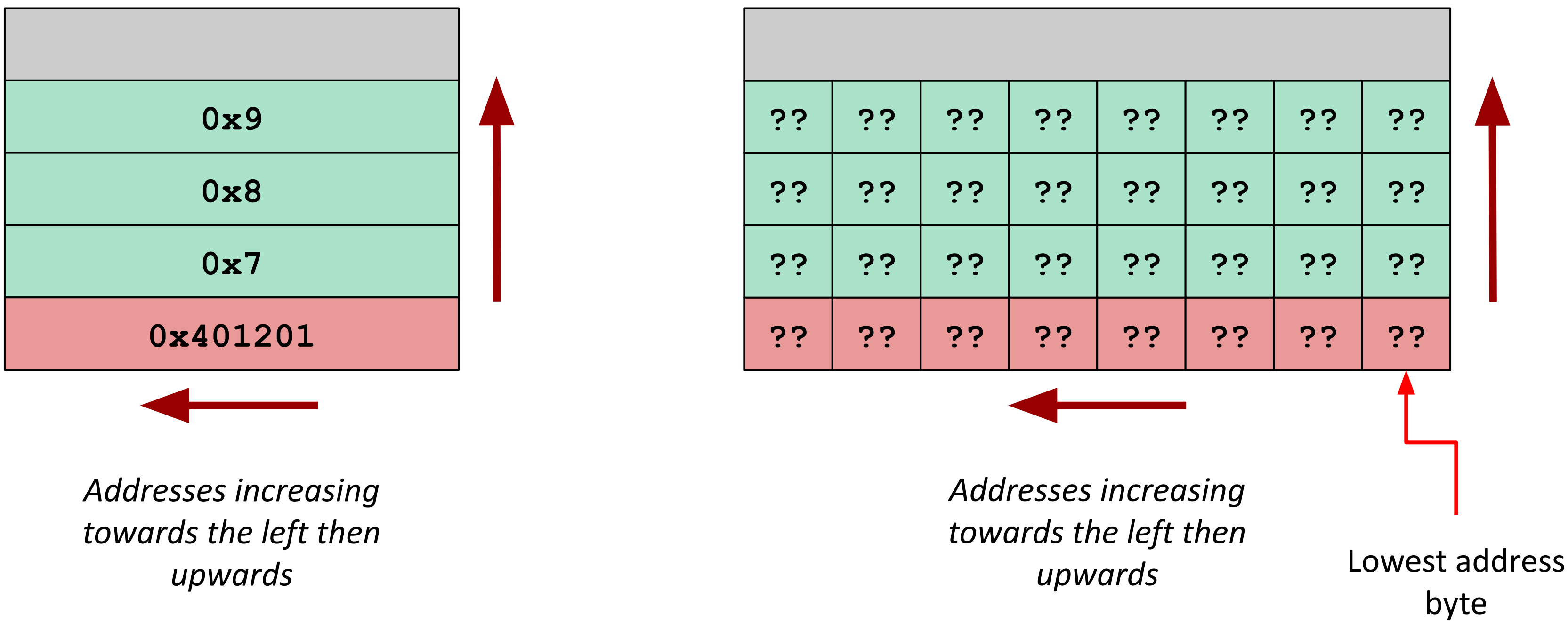
0x01020304

"Big end" "Little end"

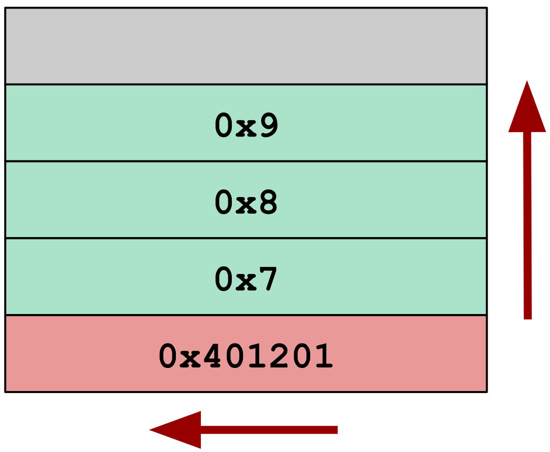
	Mem[0]	Mem[1]	Mem[2]	Mem[3]
Little-Endian	0x04	0x03	0x02	0x01
Big-endian	0x01	0x02	0x03	0x04

Endianness: Example

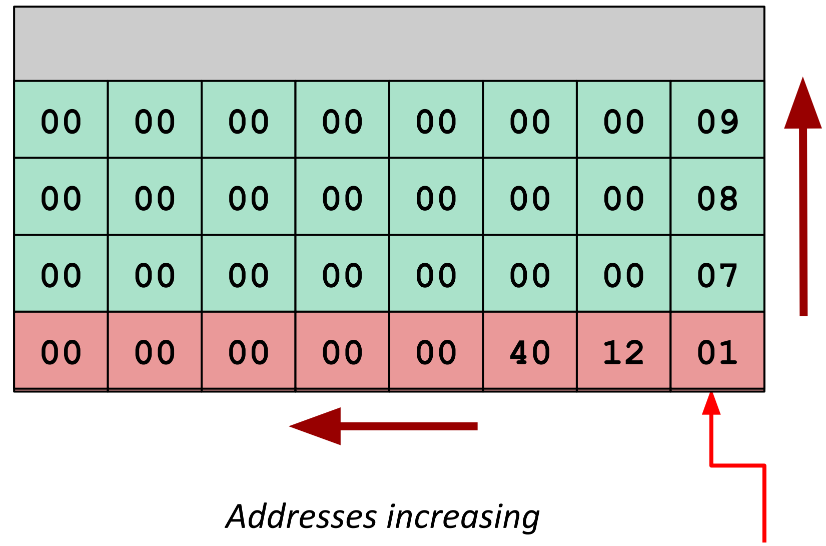
- Suppose we draw our diagram with addresses increasing towards the left, then upwards.
- How are the bytes ordered on a little endian machine?



Endianness: Example



Addresses increasing towards the left then upwards



Addresses increasing towards the left then upwards

Lowest address byte

Endianness Example: Comparing with gdb

00	00	00	00	00	00	00	09
00	00	00	00	00	00	00	08
00	00	00	00	00	00	00	07
00	00	00	00	00	40	12	01

Addresses increasing towards the left then upwards

```
(gdb) x /32bx $rsp
0x7fffffffe3e8: 0x01 0x12 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffffe3f0: 0x07 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffe3f8: 0x08 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffe400: 0x09 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

Addresses increasing towards the right then downwards

- `gdb` draws its diagram with addresses increasing towards the right then downwards.
- Both diagrams are correct, and are still little-endian!

Attack Lab

Attack Lab: Overview

- Exploit vulnerabilities in target programs using the techniques you learned in lecture.
- Hijack their control flow and make them do something else!
- Targets do *not* explode like in **`bomblab`**.
- We'll get some practice right now!

Activity

Activity 1

- Download this week's handout from the *Schedule* page.
- Also download the code.
- For now:
  - Just open up the source code under **`src/activity.c`**.
  - We'll start by walking through the code together!

```
$ wget https://www.cs.cmu.edu/~213/activities/rec5.tar
$ tar xvf rec5.tar
$ cd rec5
```

Activity 1: solve()

```
void solve(void) {                                          src/activity.c
    long before = 0xb4;
    char buf[16];
    long after = 0xaf;

    Gets(buf);

    if (before == 0x3331323531)
        win(0x15213);
    if (after == 0x3331323831)
        win(0x18213);
}
```

- Assume **before** and **after** are stored on the stack.
- Is there any way for **solve()** to call **win()**?
- Based on what you learned in lecture, are there any vulnerabilities we can exploit here?

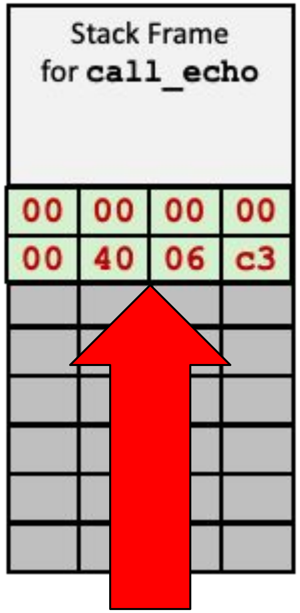
Recall: Unsafe Functions

- C standard library functions like **`gets()`** and **`strcpy()`** write to buffers, but have no length checks!
  - Enables *buffer overflow* attacks.

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

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

Compiler making space
for buffer +
a little bit of padding



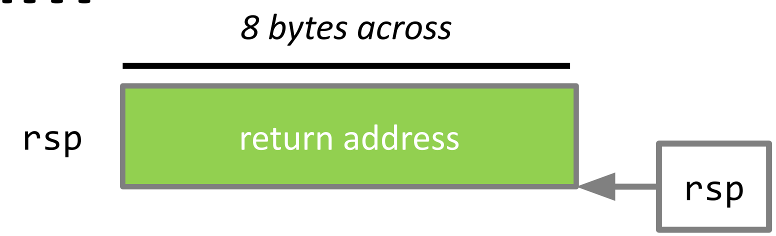
Can overwrite anything
before the buffer!

Activity 1: Back to `solve()`

- Let's see if we can find a similar vulnerability in **`solve()`** by looking at the assembly!
- Source code and assembly code are both reproduced on the back of the handout.
- Draw a stack diagram to see if you can answer the following:
  - What does the stack frame look like?
  - Where is the saved return address?
  - Where do we store **`buf`**, **`before`**, and **`after`** in relation to each other?

Activity 1: Stack diagram

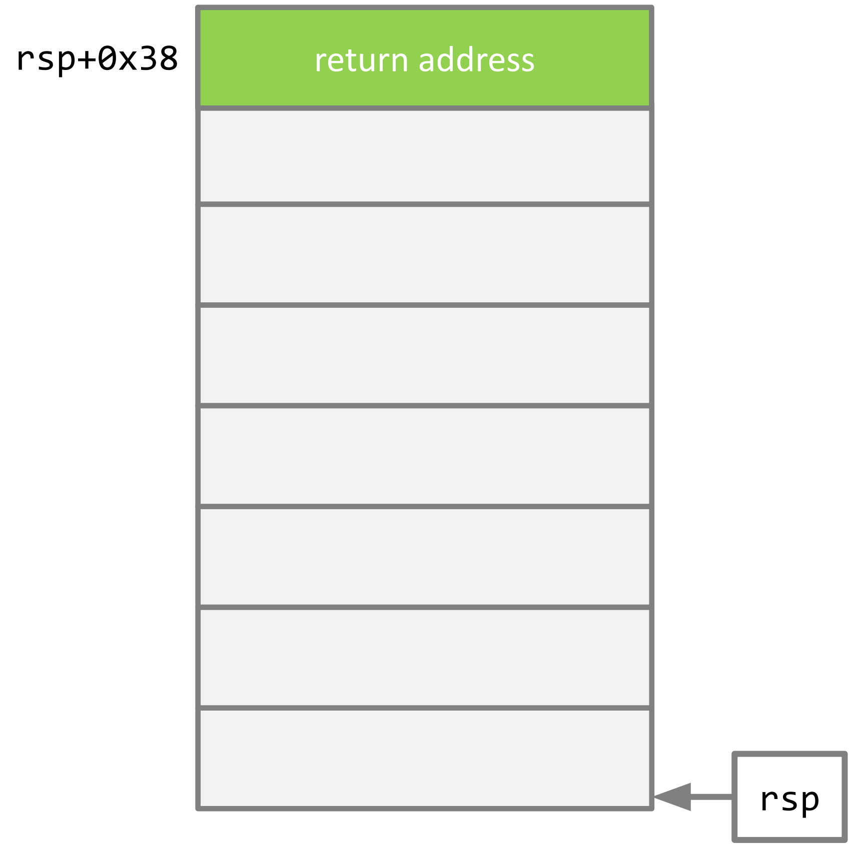
```
=> 0x4006b5 <+0>: sub    $0x38,%rsp
```



Addresses increase
towards the top of
the slide

Activity 1: Stack diagram

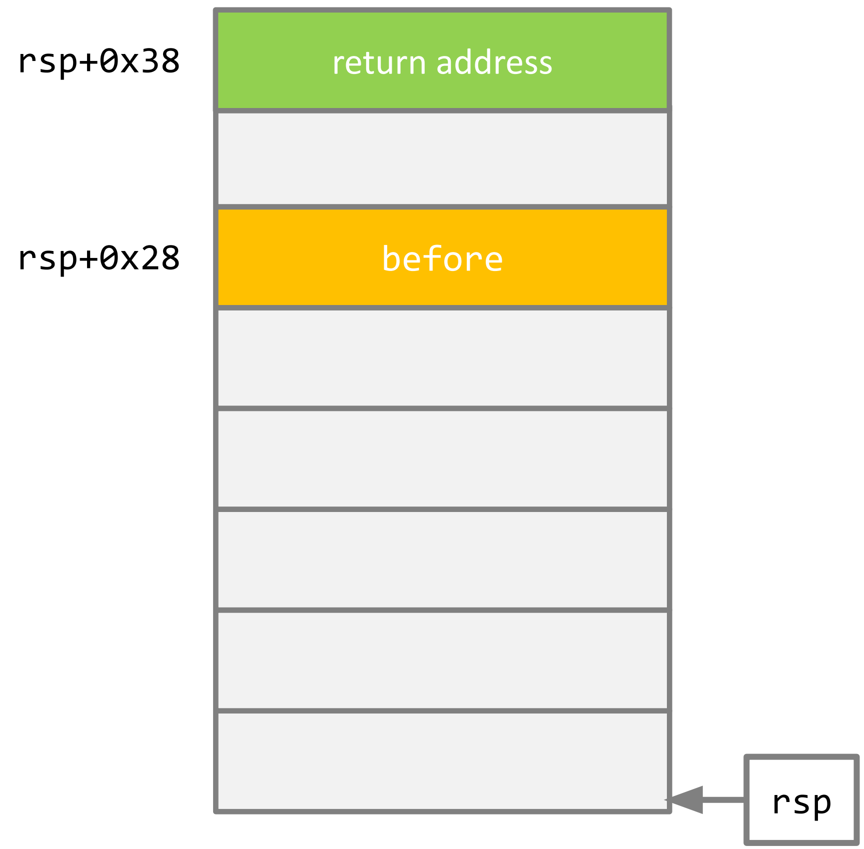
```
   0x4006b5 <+0>:     sub    $0x38,%rsp
=> 0x4006b9 <+4>:     movq   $0xb4,0x28(%rsp)
```



Addresses increase towards the top of the slide

Activity 1: Stack diagram

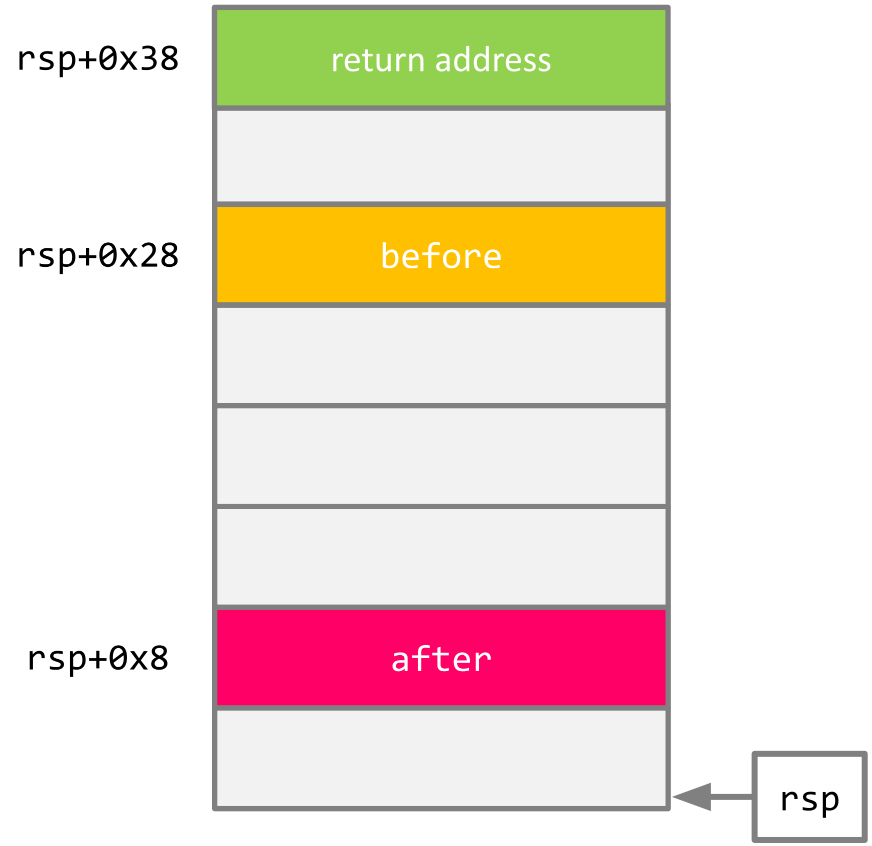
```
   0x4006b5 <+0>:     sub    $0x38,%rsp
   0x4006b9 <+4>:     movq   $0xb4,0x28(%rsp)
=> 0x4006c2 <+13>:    movq   $0xaf,0x8(%rsp)
```



Addresses increase towards the top of the slide

Activity 1: Stack diagram

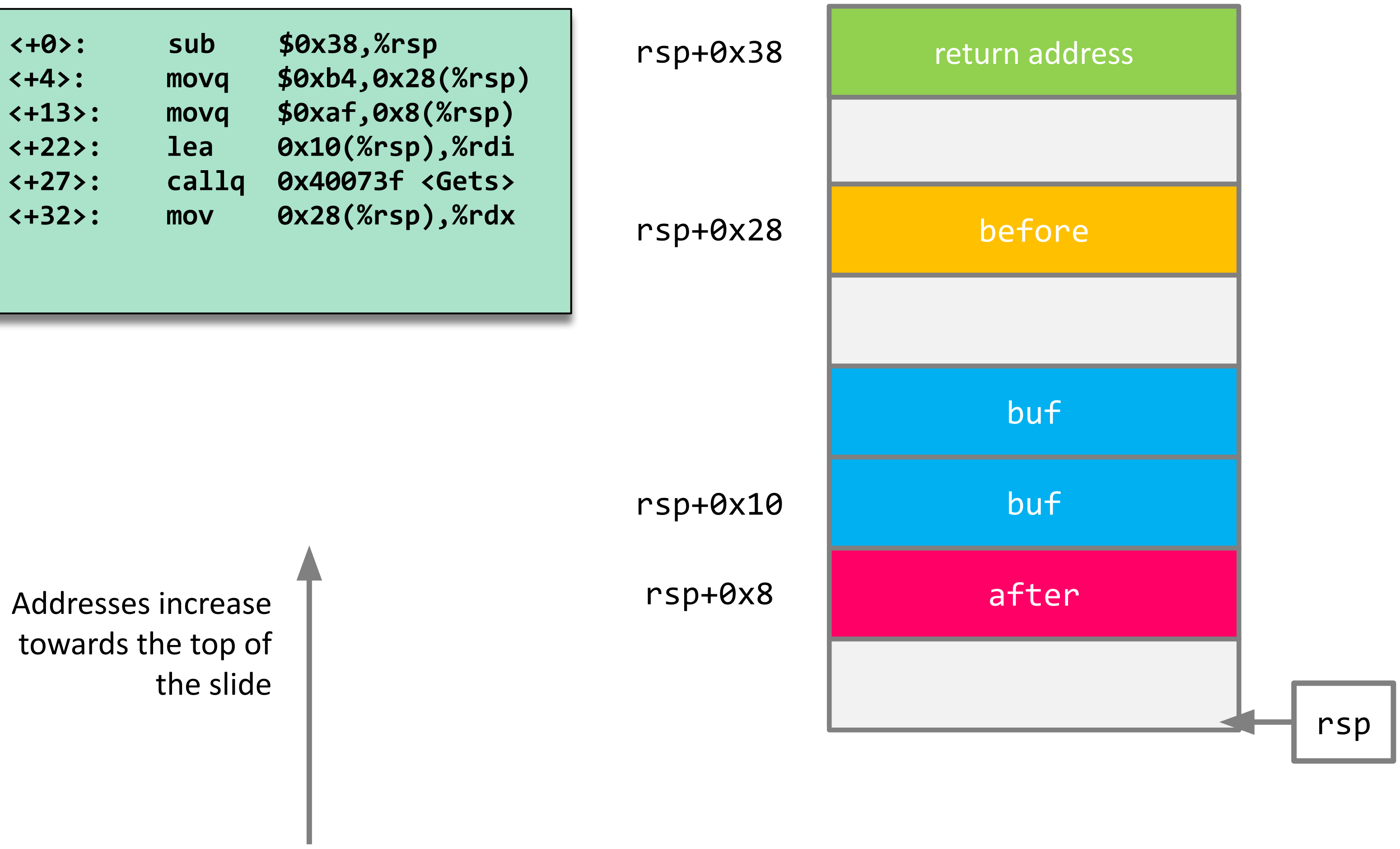
```
   0x4006b5 <+0>:     sub    $0x38,%rsp
   0x4006b9 <+4>:     movq   $0xb4,0x28(%rsp)
   0x4006c2 <+13>:    movq   $0xaf,0x8(%rsp)
   0x4006cb <+22>:    lea    0x10(%rsp),%rdi
=> 0x4006d0 <+27>:    callq  0x40073f <Gets>
```



Addresses increase towards the top of the slide

Activity 1: 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
```

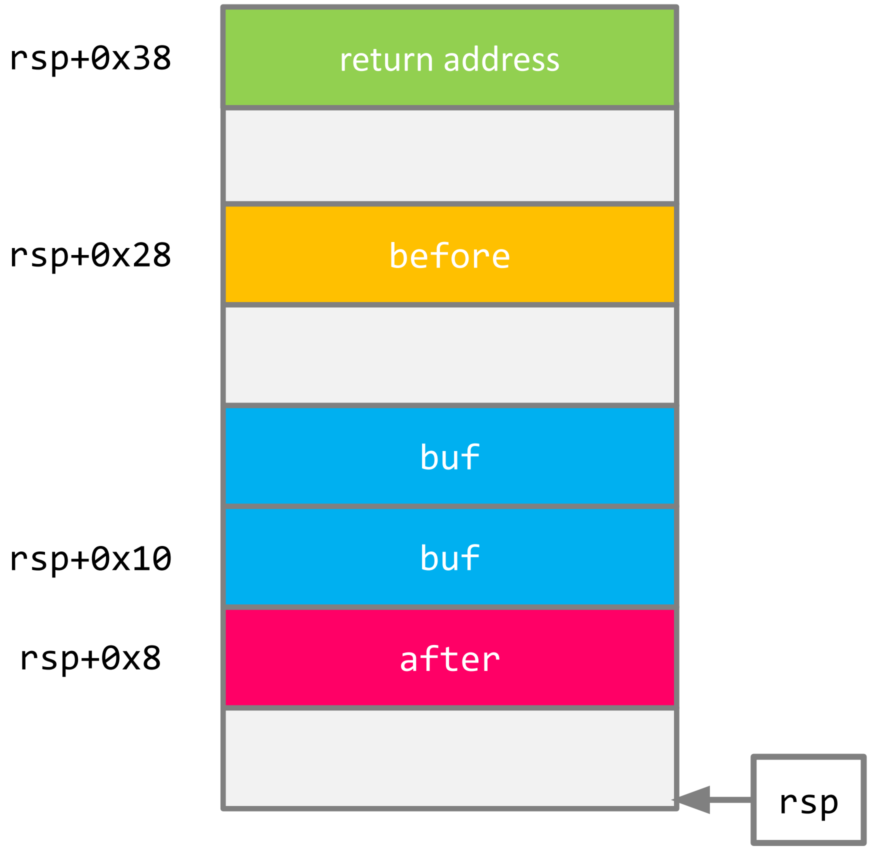


Activity 1: Exploitation

- Goal: call **`win(0x15213)`**
- Take a few minutes to craft an exploit string!
- Crafting an exploit:
  - **`gets()`** stops reading once it sees a newline.
  - Will ***not*** stop reading when it sees a null terminator.

Activity 2

- Goal: call **`win(0x18213)`**
- Is it possible to overwrite **`after`**?
- What *can* we overwrite?
- Where could we jump to call **`win(0x18213)`**?



The End