

15-213/15-503 Recitation Attack Lab

Josh Kim

Tuesday, June 3rd

Reminders

- **bomblab** was due yesterday (June 2nd)
- **attacklab** has been released, and is due on ***Tuesday (June 10th)***
- Written 3 has also been released - note that there is less time to complete this one!

Agenda

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

Review: structs

Alignment Requirements

- Badly aligned data can harm performance:
 - May need multiple memory accesses instead of just one.
- *Primitive* types have pre-determined alignments:
 - **char** = 1 byte
 - **short** = 2 bytes
 - **int** = 4 bytes
 - **long** = 8 bytes
 - **double** = 8 bytes
 - **pointer** = 8 bytes

Alignment : Compound Types

- Compound types:
 - Arrays
 - Structs
 - Unions
- Alignment rules for these types:
 1. Takes largest alignment requirement of its fields.
 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;  
}
```

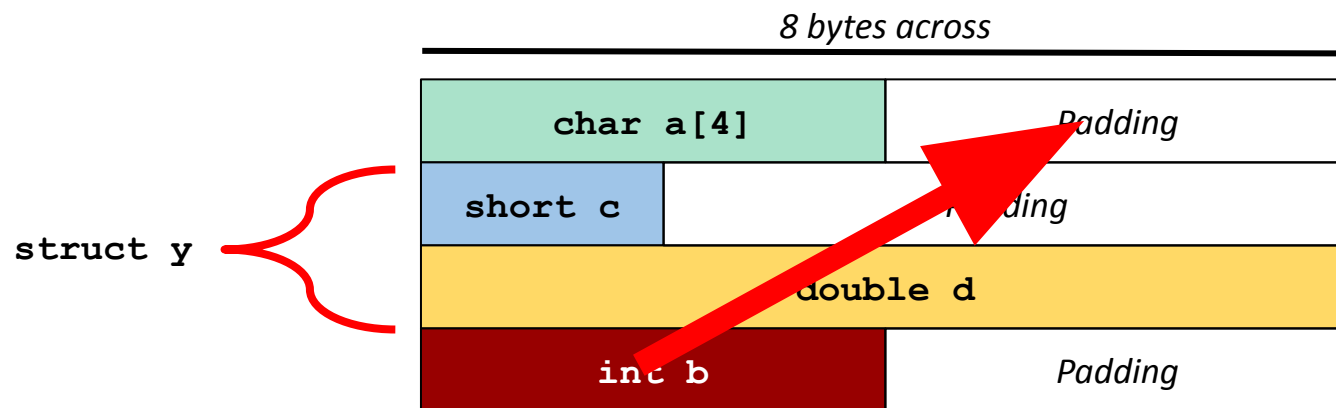
- What is the alignment requirement for **y**?
 - **Alignment: 8**
- What is its size?
 - **Rule (2):** have to add padding after **c** so that **d** is 8-byte aligned
 - **Size: 16 bytes**

Alignment Requirements: Example

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

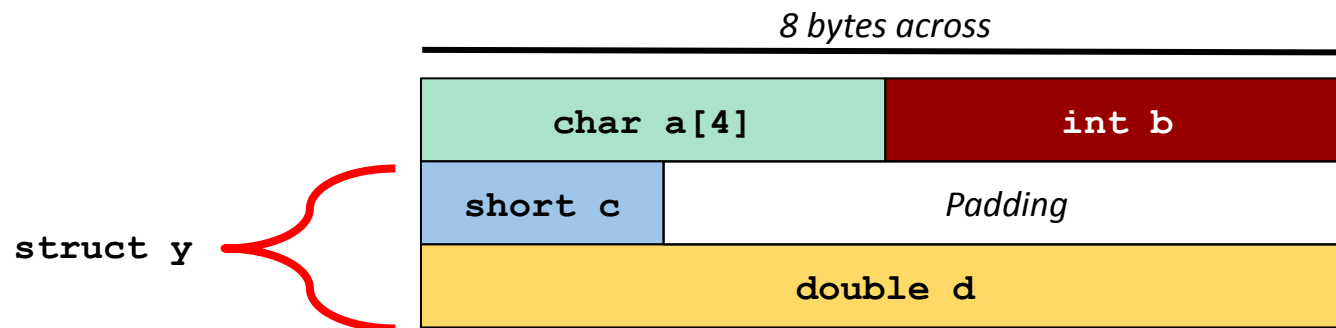
- What is the alignment requirement for **x**?
 - **Alignment: 8**
- What is its size?
 - Remember, the entire struct must also follow alignment rules
 - **Size: 32 bytes**

structs: Reordering Fields



- `struct x` takes up 32 bytes to store 18 bytes of data.
- 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.

Stacks

Manipulating the Stack

- Certain instructions *grow* the stack, and certain instructions *shrink* the stack:

Growing the stack

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

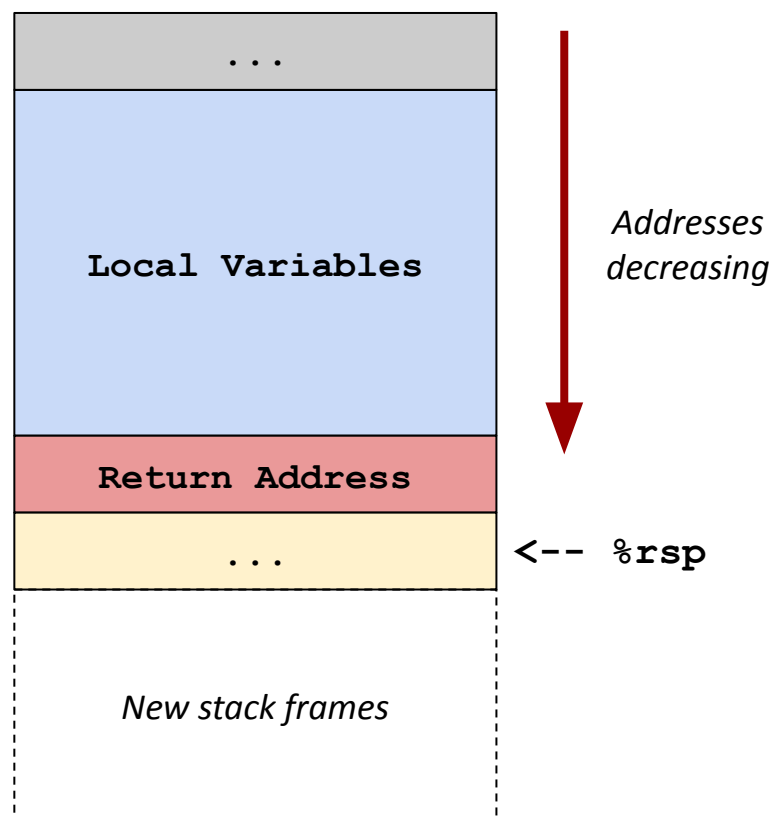
Shrinking the stack

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

- But what does this look like in memory?

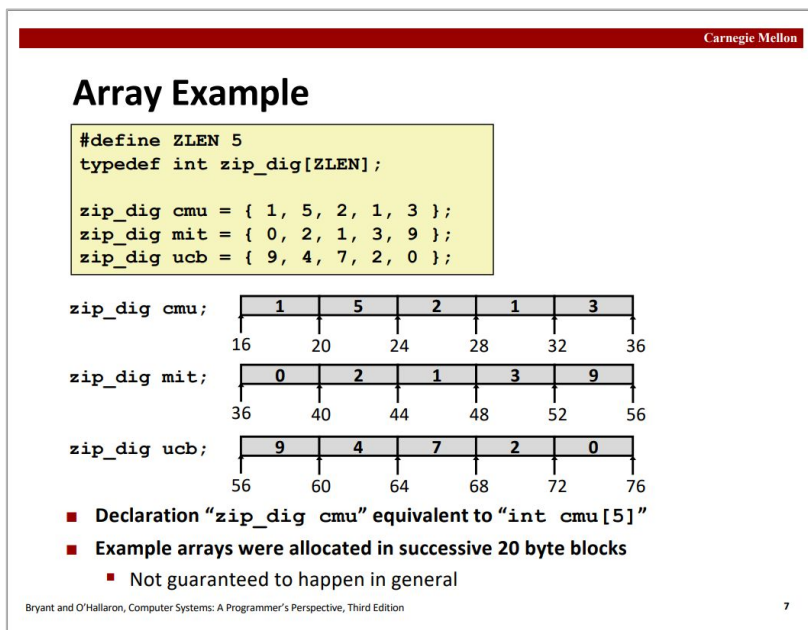
Which way does the stack grow?

- 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
 - But you can draw them in any way that makes sense to you!



Drawing Memory

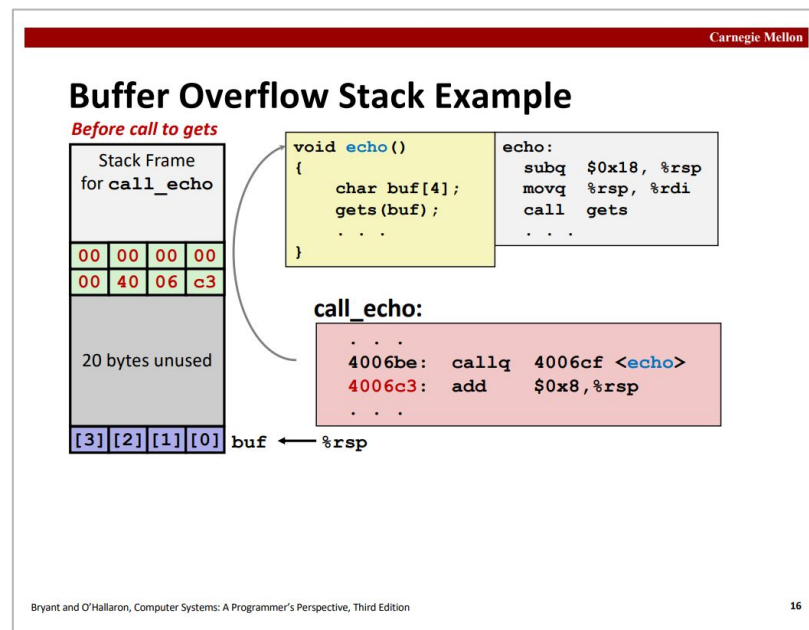
Conventional Memory Diagram



Addresses Increasing:

- Towards the right
- Then downwards

Stack Diagram



Addresses Increasing:

- Towards the left
- Then upwards

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

- Take a look at the following code snippet:

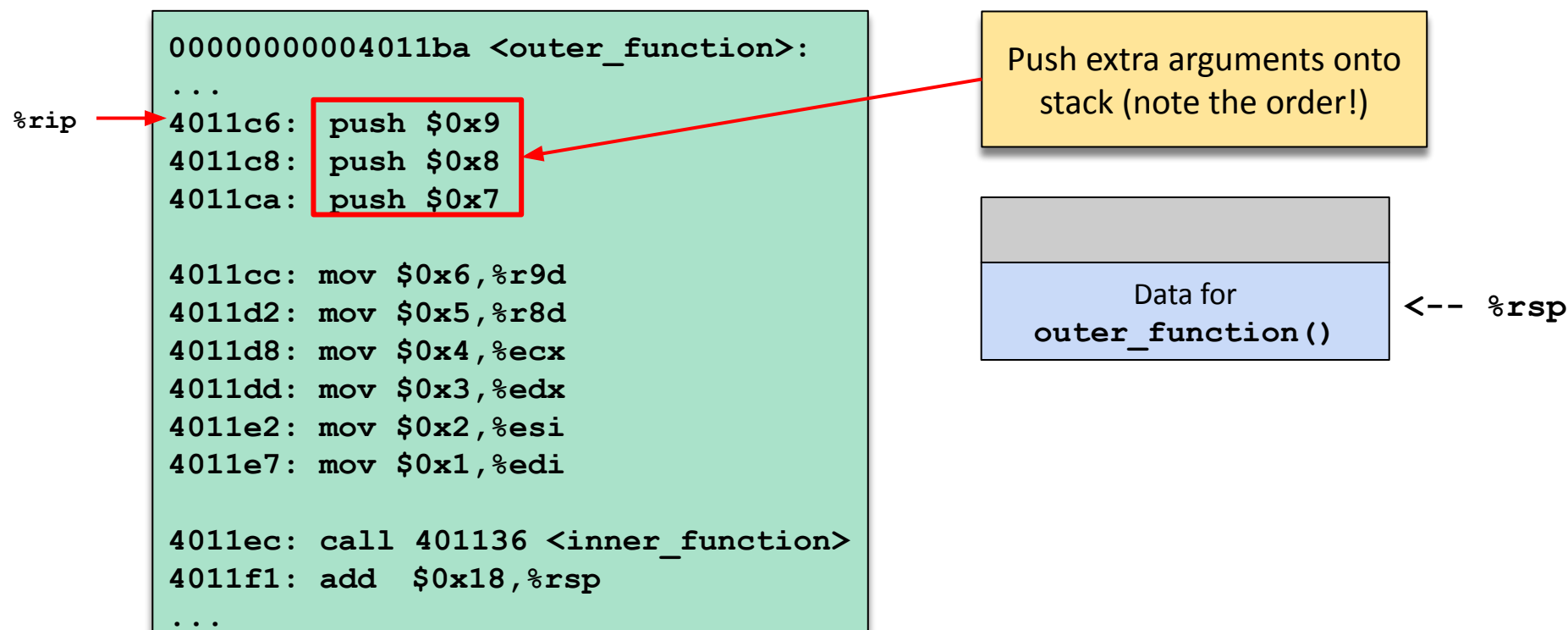
```
int outer_function() {  
    int result = inner_function(1, 2, 3, 4, 5, 6, 7, 8, 9);  
  
    return result + 1;  
}
```

Lots of arguments!

- What would this look like in assembly? How would having many arguments affect the stack?

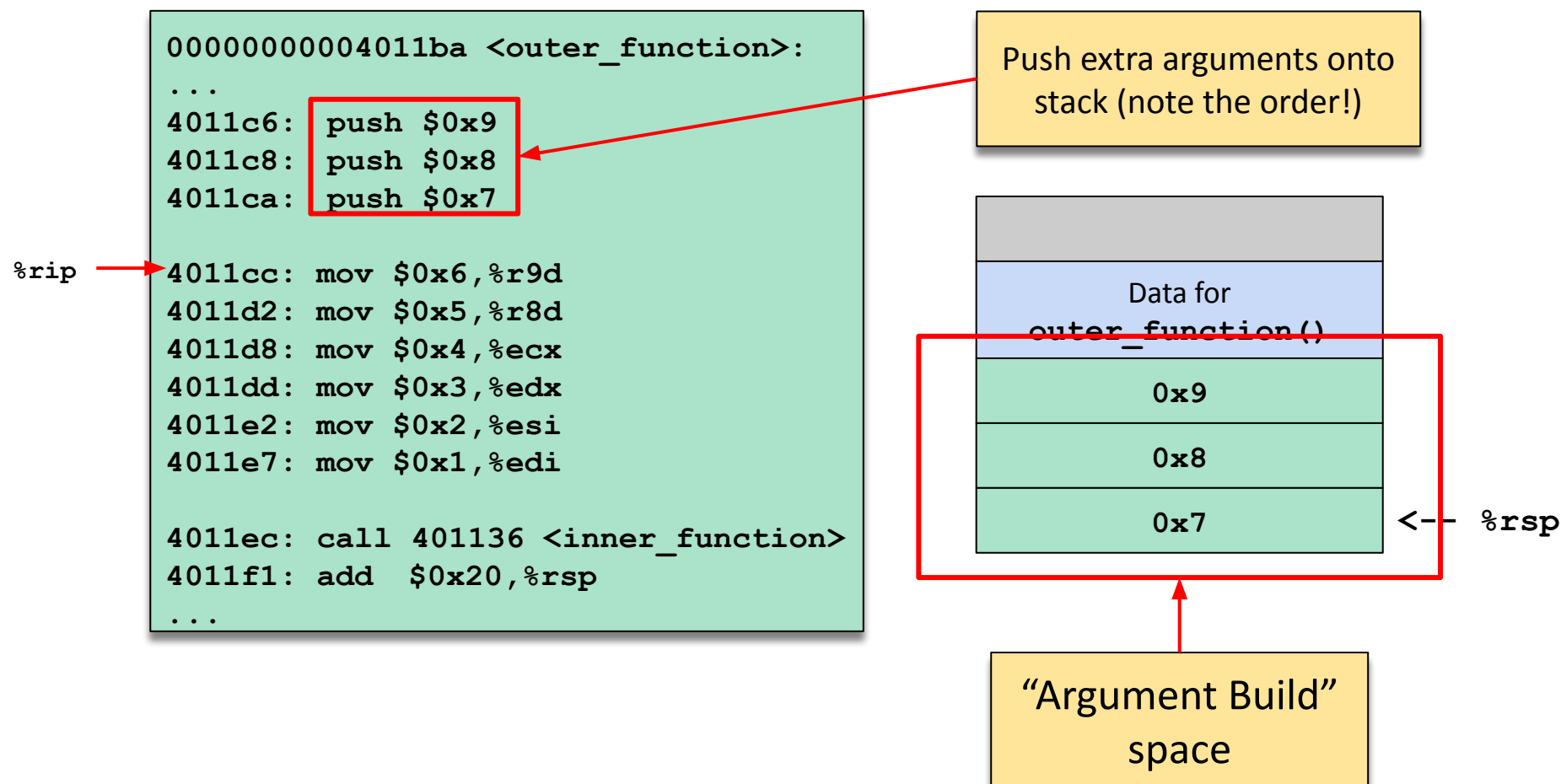
Example: `outer_function()`

- Here is the assembly for `outer_function`:



Example: `outer_function()`

- Here is the assembly for `outer_function`:



Example: `outer_function()`

- Here is the assembly for `outer_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
```

```
%rip → 4011ec: call 401136 <inner_function>
```

```
4011f1: add $0x20,%rsp
```

```
...
```

Load up first 6 arguments
into argument registers

Data for <code>outer_function()</code>
0x9
0x8
0x7

<-- %rsp

Example: `outer_function()`

- Remember, `call` loads the return address onto the stack

```

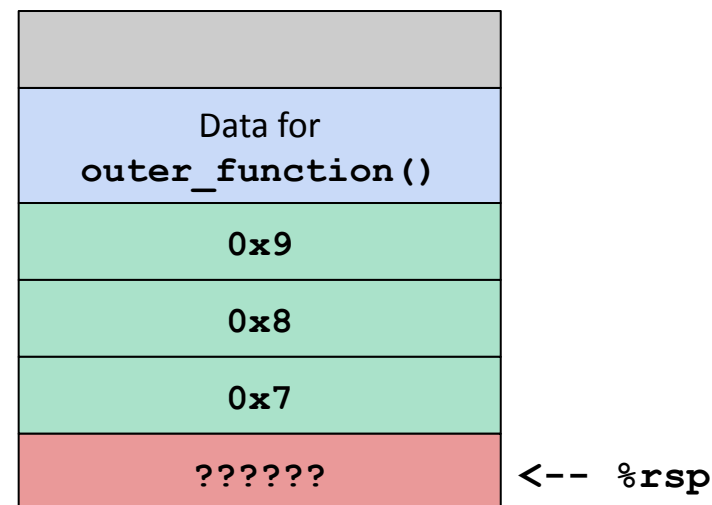
00000000004011ba <outer_function>:
...
4011c6:  push $0x9
4011c8:  push $0x8
4011ca:  push $0x7

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
...

```

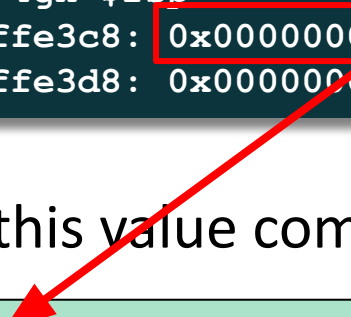
Now we're ready to call!



Example: `outer_function()`

- What is the return address we should store? Let's inspect gdb to find out!

```
(gdb) x /4gx $rsp
0x7fffffffef3c8: 0x00000000004011f1  0x0000000000000007
0x7fffffffef3d8: 0x0000000000000008  0x0000000000000009
```



- Where does this value come from?

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

- It's the address of the instruction we want to jump to after completing the call to `inner_function`

Example: `outer_function()`

- State of our program before starting `inner_function`

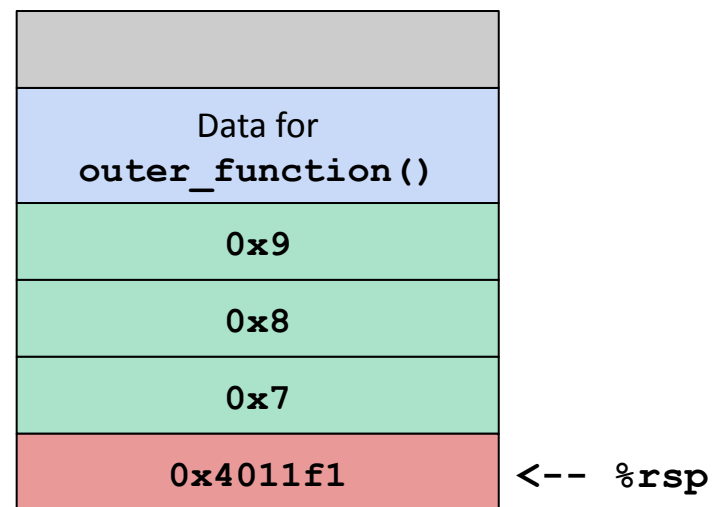
```

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
...

```



Pass control to `inner_function()`
=> Set `%rip` to `0x401136`

Example: `inner_function()`

- Here is the assembly for `inner_function`

```
0000000000401136 <inner_function>:
```

`%rip` →

```
401136:  endbr64
```

```
40113a:  push %rbp
```

```
40113b:  mov %rsp,%rbp
```

```
40113e:  sub $0x38,%rsp
```

```
...
```

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

```
4011bd:  pop %rbp
```

```
4011be:  ret
```

Example: `inner_function()`

- Here is the assembly for `inner_function`

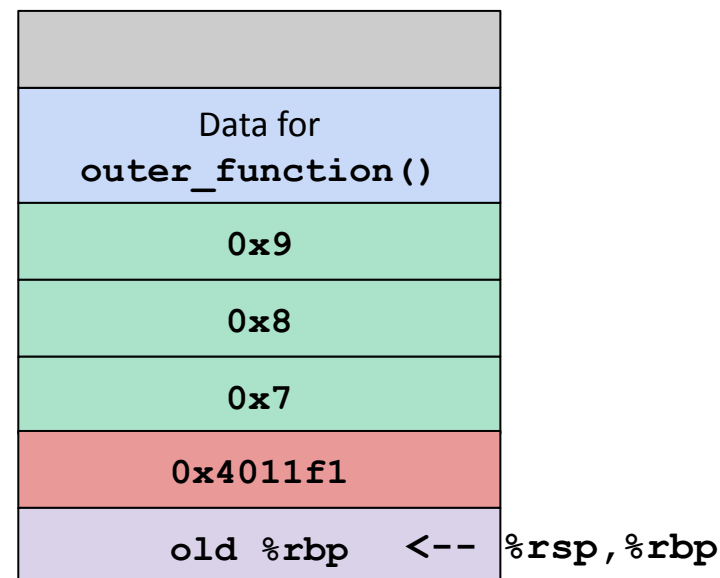
```

0000000000401136 <inner_function>:
401136:  endbr64
40113a:  push %rbp
40113b:  mov %rsp,%rbp
40113e:  sub $0x38,%rsp
...
4011b9:  add $0x38,%rsp
4011bd:  pop %rbp
4011be:  ret

```

`%rip` →

Function store the original base pointer and repositions it for this stack frame



Example: `inner_function()`

- Here is the assembly for `inner_function`

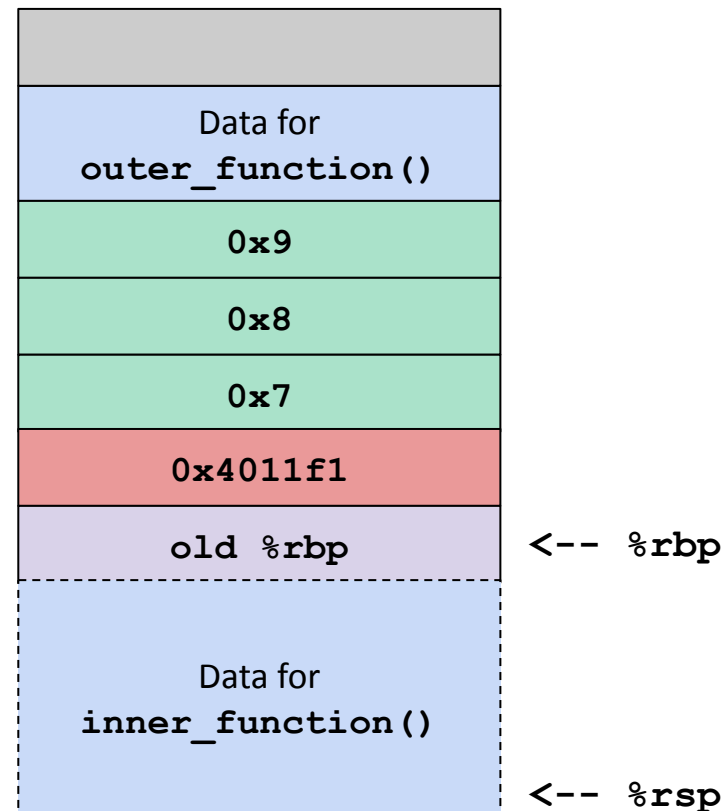
```

0000000000401136 <inner_function>:
401136:  endbr64
40113a:  push %rbp
40113b:  mov %rsp,%rbp
40113e:  sub $0x38,%rsp
...
4011b9:  add $0x38,%rsp
4011bd:  pop %rbp
4011be:  ret

```

`%rip` →

Function allocates any space it needs



Example: `inner_function()`

- Here is the assembly for `inner_function`

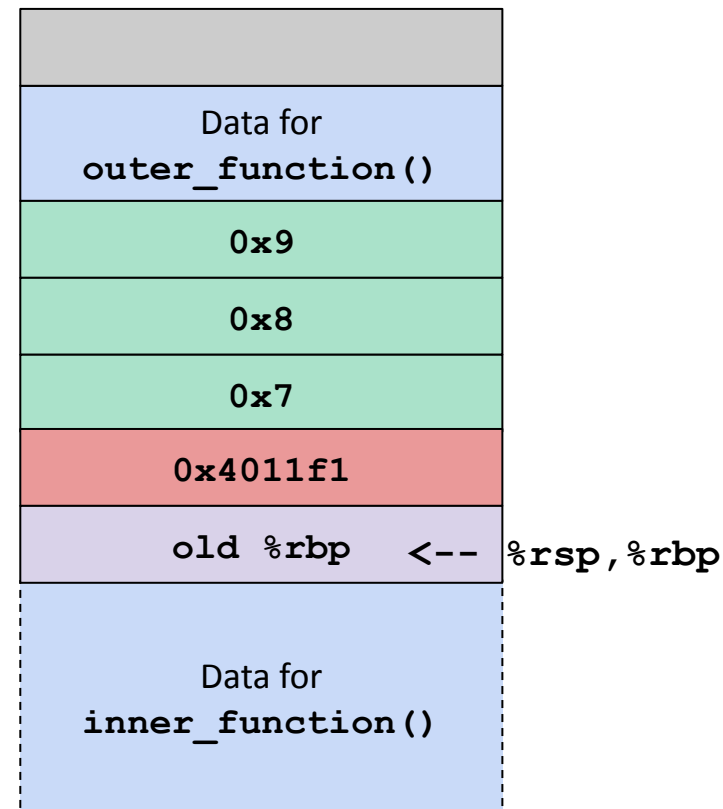
```

0000000000401136 <inner_function>:
401136:  endbr64
40113a:  push %rbp
40113b:  mov %rsp,%rbp
40113e:  sub $0x38,%rsp
...
4011b9:  add $0x38,%rsp
4011bd:  pop %rbp
4011be:  ret

```

`%rip` →

De-allocate any stack space
used by the function



Example: `inner_function()`

- Here is the assembly for `inner_function`

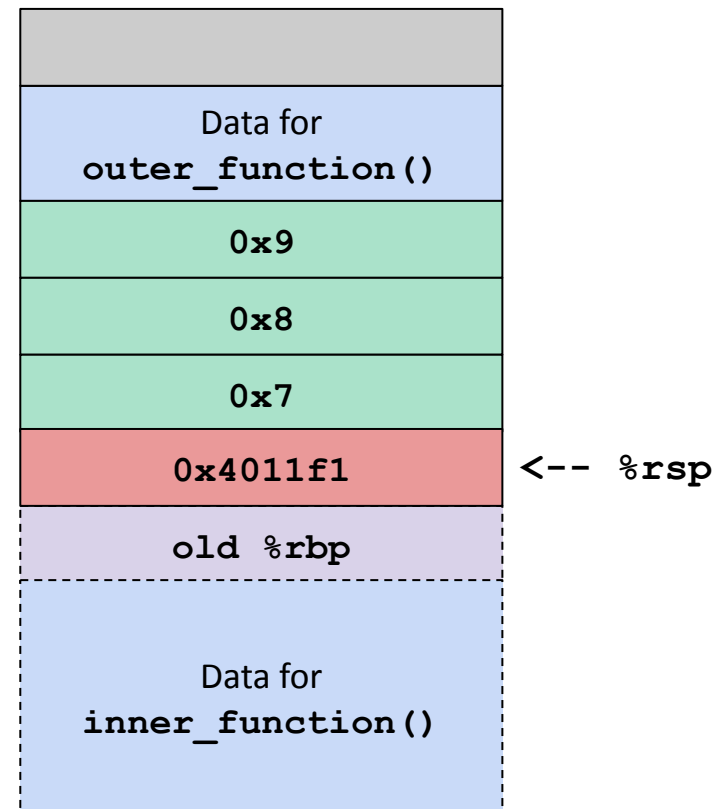
```

0000000000401136 <inner_function>:
401136:  endbr64
40113a:  push %rbp
40113b:  mov %rsp,%rbp
40113e:  sub $0x38,%rsp
...
4011b9:  add $0x38,%rsp
4011bd:  pop %rbp
4011be:  ret

```

`%rip` →

Restore `%rbp` to prepare to return to original stack frame



Example: `inner_function()`

- Here is the assembly for `inner_function`

```

0000000000401136 <inner_function>:
401136:  endbr64
40113a:  push %rbp
40113b:  mov %rsp,%rbp
40113e:  sub $0x38,%rsp
...
4011b9:  add $0x38,%rsp
4011bd:  pop %rbp
4011be:  ret

```

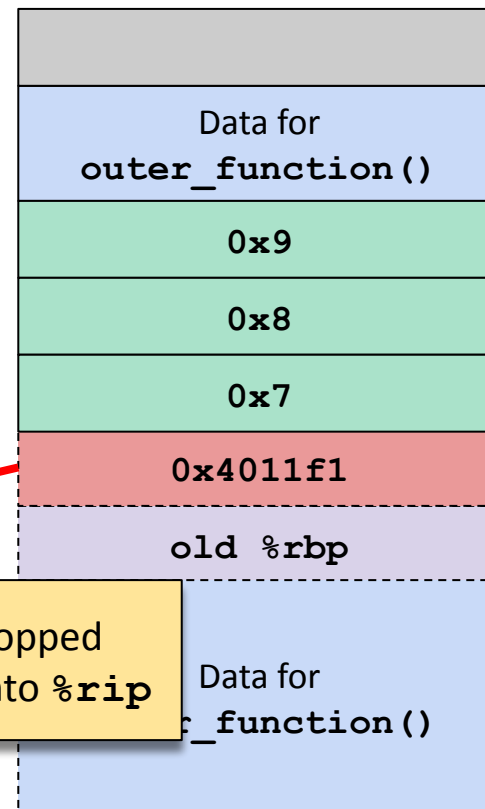
Return! We pop the return address and jump

```

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

```

Load popped address into `%rip`



`<-- %rsp`

`%rip` →

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).

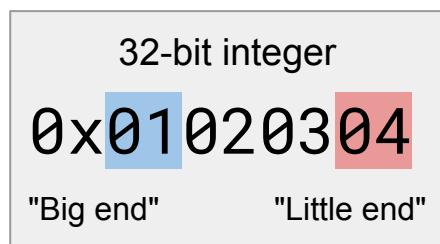
Endianness

■ *Little-Endian:*

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

■ *Big-Endian:*

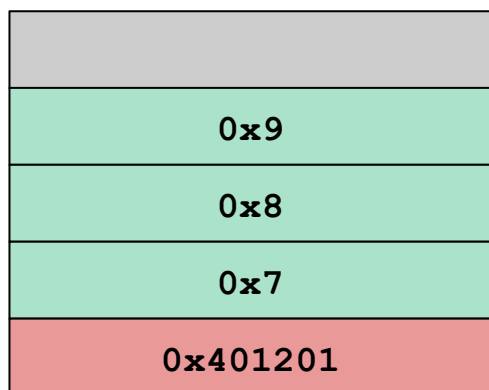
- *Most significant byte* is stored at the *lowest* address.



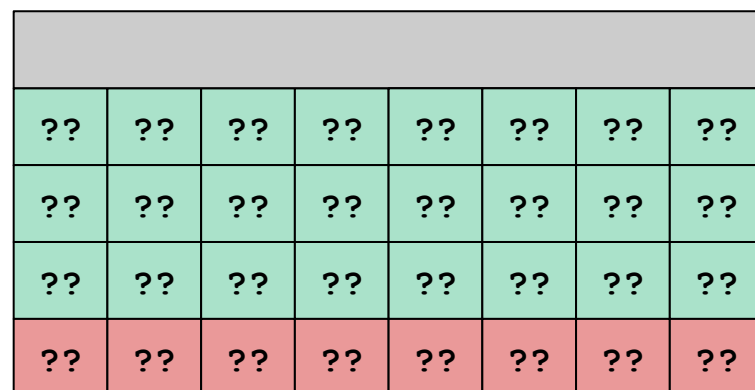
	Mem[0]	Mem[1]	Mem[2]	Mem[3]
<i>Little-Endian</i>	0x04	0x03	0x02	0x01
<i>Big-endian</i>	0x01	0x02	0x03	0x04

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?



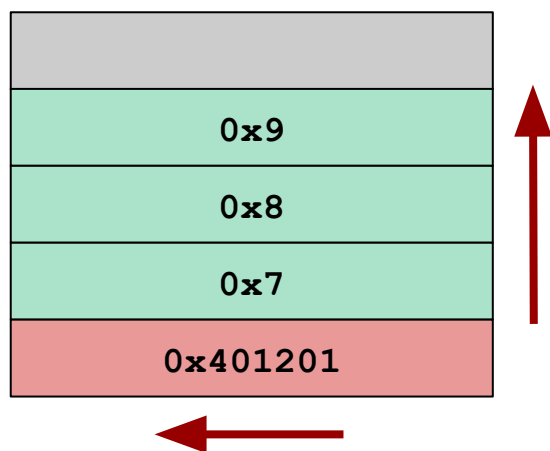
*Addresses increasing
towards the left then
upwards*



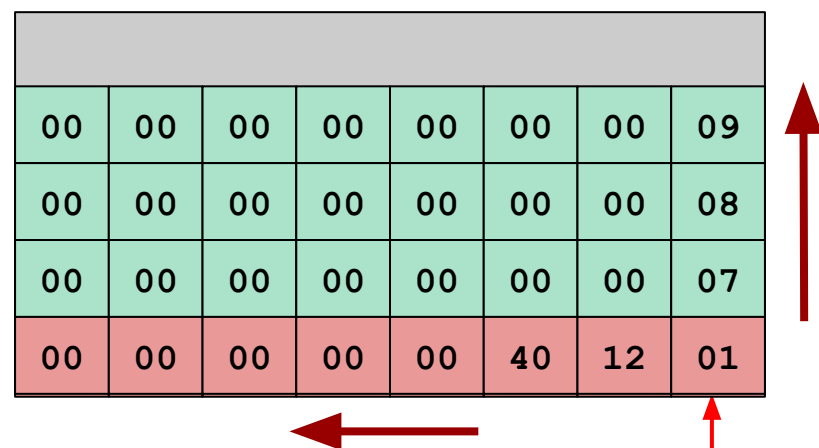
*Addresses increasing
towards the left then
upwards*

Lowest address
byte

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
```

```
0x7fffffffef3e8: 0x01 0x12 0x40 0x00 0x00 0x00 0x00 0x00
0x7fffffffef3f0: 0x07 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffef3f8: 0x08 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffef400: 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!

Activity: Endianness with Memory Layout

Here is an array with 6 elements:

```
short arr[] = {0x1234, 0x8326, 0x9742, 0x4200, 0x1521, 0x3531};
```

What do these statements print? Justify your answers.

```
printf("A. %lx\n", *((long*)&arr[2]));  
printf("B. %x\n", (unsigned char) *(((char*)arr)+ 3));
```

Activity: Representation in Memory

Here is an array with 6 elements:

```
short arr[] = {0x1234, 0x8326, 0x9742, 0x4200, 0x1521, 0x3531};
```

What does this look like in memory?

*Addresses increasing
towards the right
then downwards*



??	??	??	??	??	??	??	??
??	??	??	??	??	??	??	??



Activity: Representation in Memory

Here is an array with 6 elements:

```
short arr[] = {0x1234, 0x8326, 0x9742, 0x4200, 0x1521, 0x3531};
```

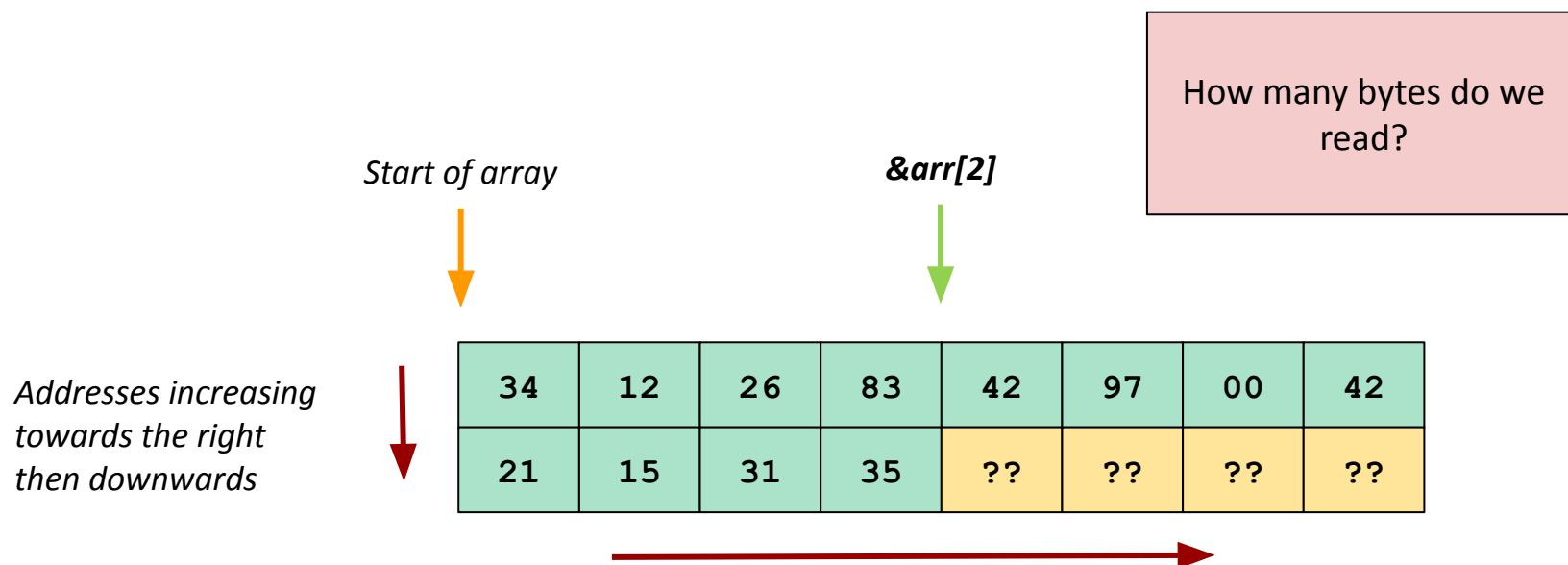
What does this look like in memory?

*Addresses increasing
towards the right
then downwards*

34	12	26	83	42	97	00	42
21	15	31	35	??	??	??	??

Activity: Part A

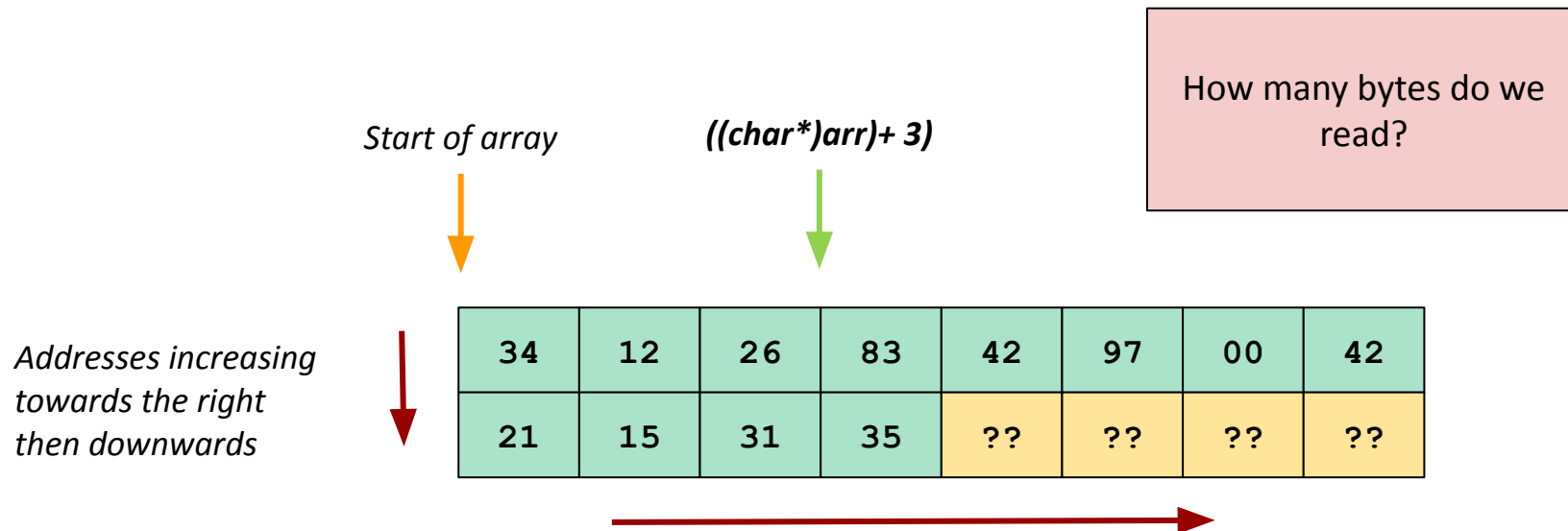
```
printf("A. %lx\n", *((long*)&arr[2]));
```



A. 0x3531152142009742

Activity: Part B

```
printf("B. %x\n", (unsigned char) *(((char*)arr)+ 3));
```



A. 0x83

SULIA 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:

*Addresses increase
towards the right*

8 bytes →

63 6f 6d 70 75 74 65 72

c o m p u t e r

99 111 109 112 117 116 101 114

28515 28781 29813 29285

1886220131 1919251573

8243122740717776739

0x72657475706d6f63

99.111.109.112 117.116.101.114

arpl WORD PTR jo 0x7a je 0x6c .byte
[edi+0x6d],bp 0x72

c o m p u t 29285

2.93930e+29 4.54482e+30

1.144493e+243

rgba(99, 111, 109, 0.44) rgba(117, 116, 101, 0.45)

8 ASCII characters

8 8-bit integers

4 unsigned 16-bit integers

2 unsigned 32-bit integers

1 unsigned 64-bit integer

a 64-bit pointer

2 IPv4 addresses

x86 machine code

6 ASCII characters + 1 16-bit integer

2 32-bit floating point numbers

1 64-bit floating point numbers

2 RGBA colours

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



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.
- 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/s25-rec4.tar
$ tar -xvf s25-rec4.tar
$ cd s25-rec4
```

Activity 1: solve ()

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

    Gets(buf);

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

src/activity.c

- 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?

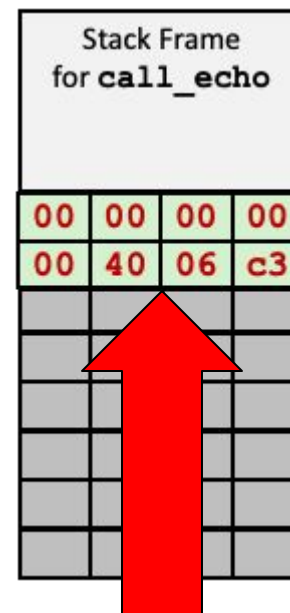
Recall: Unsafe Functions

- 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



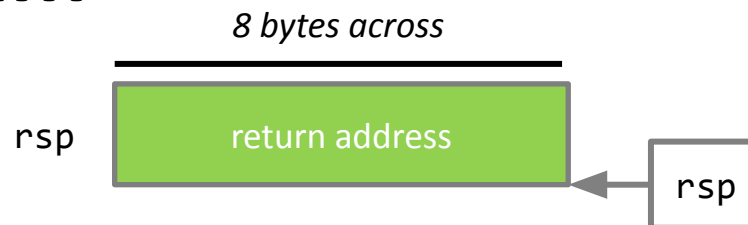
Can overwrite anything
before the buffer!

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



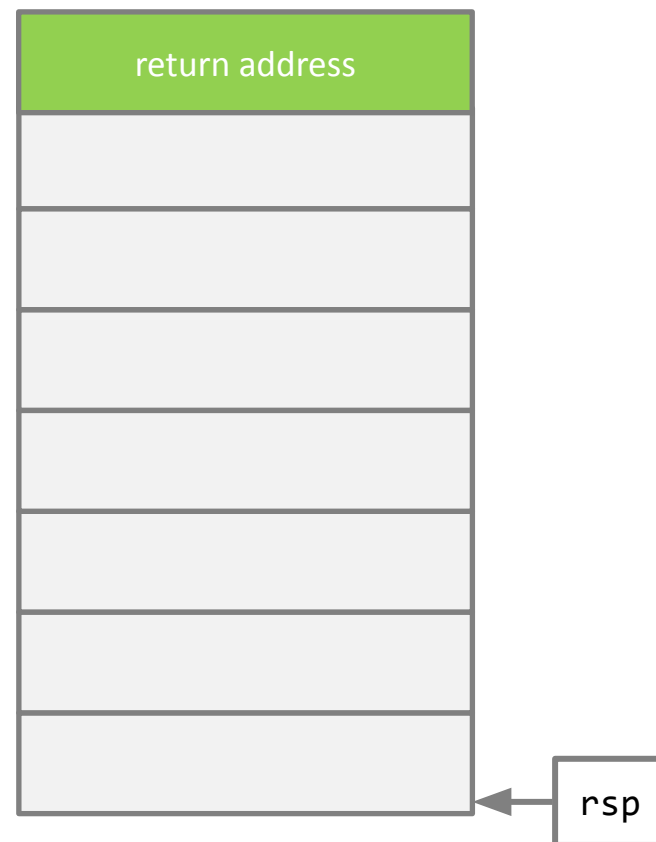
Activity 1: Stack diagram

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

rsp+0x38

return address

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)
```

rsp+0x38

rsp+0x28

return address

before

Addresses increase
towards the top of
the slide

rsp

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



rsp+0x38

return address

rsp+0x28

before

rsp+0x8

after

rsp

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

```

Addresses increase
towards the top of
the slide



rsp+0x38

return address

rsp+0x28

before

rsp+0x10

buf

buf

rsp+0x8

after

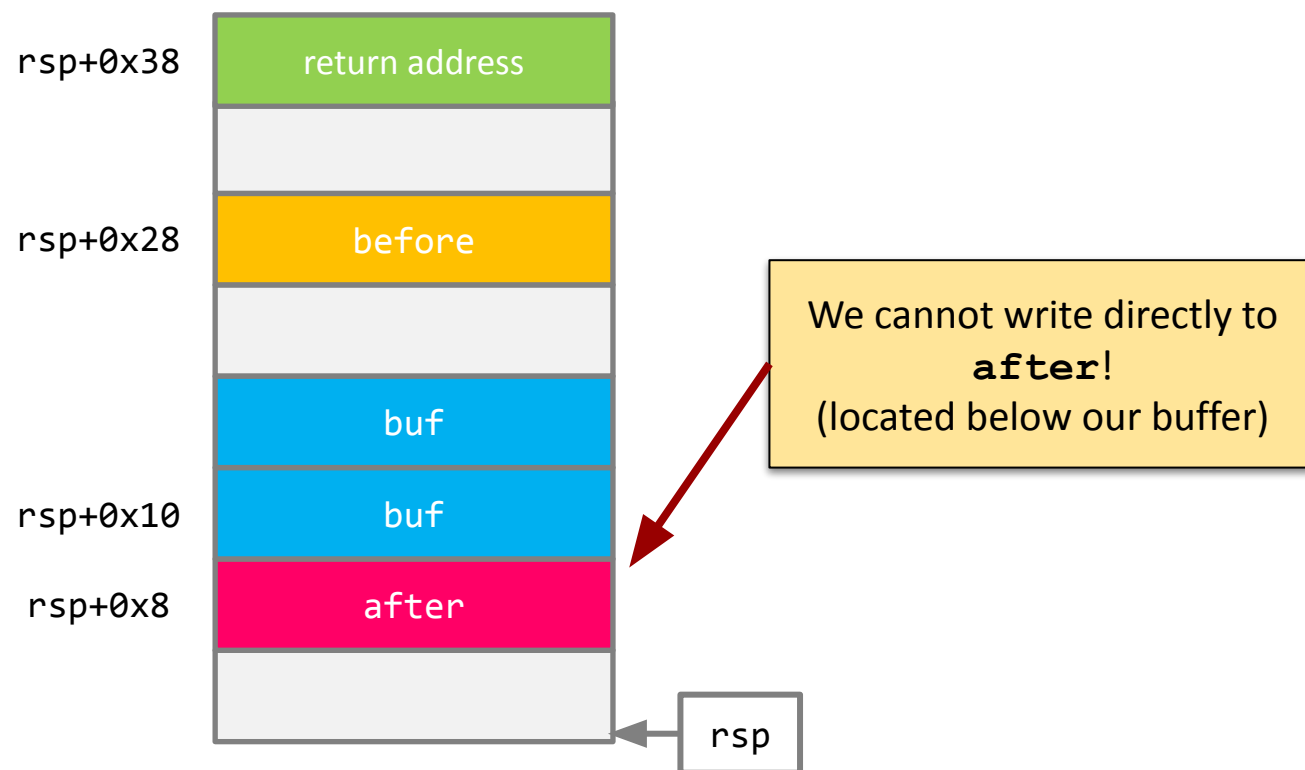
rsp

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

- Objective: call `win(0x18213)`
- How is activity 2 different from activity 1?

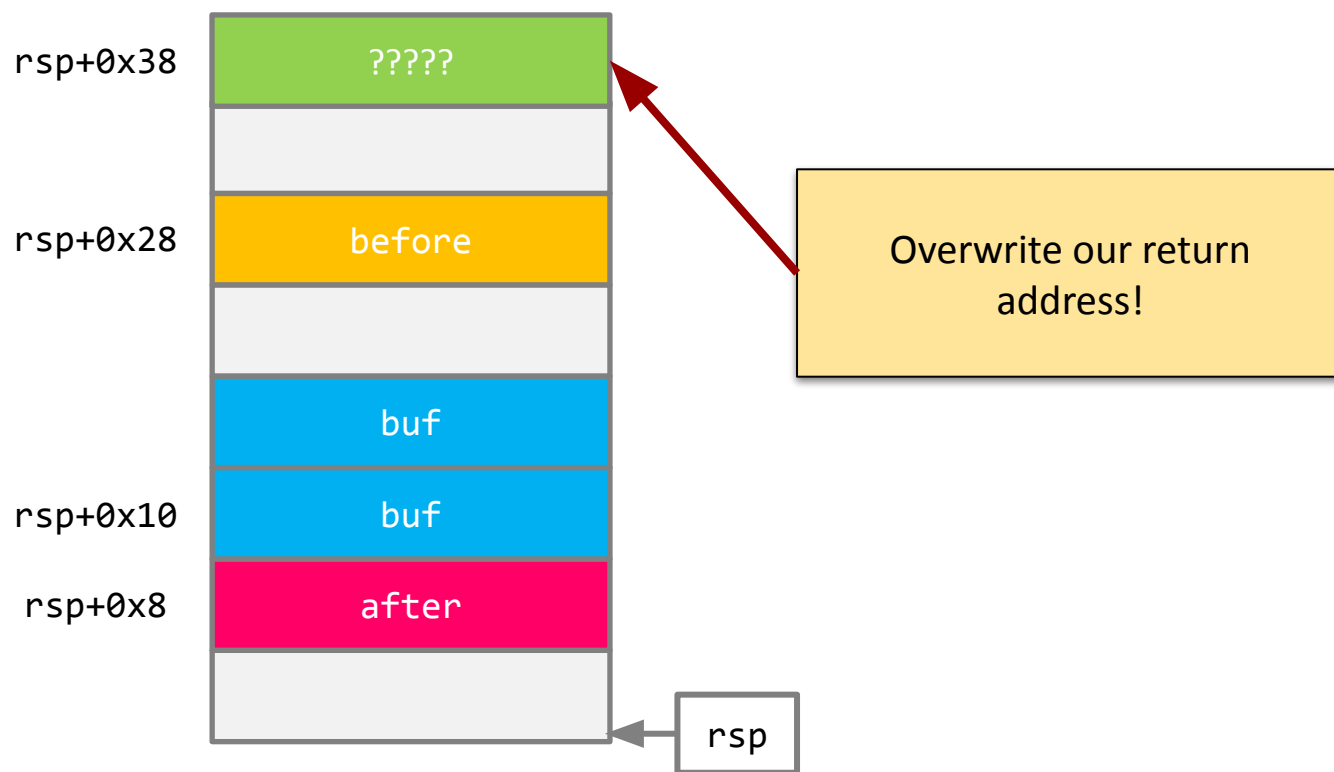


Activity 2: Exploitation

- If we cannot overwrite `after` in order to call `win(0x18213)`, what is another type of attack we can perform?
- *One possible solution:* Instead of setting local variables that result in calling `win(0x18213)`, we can jump to an instruction that directly calls `win(0x18213)`!

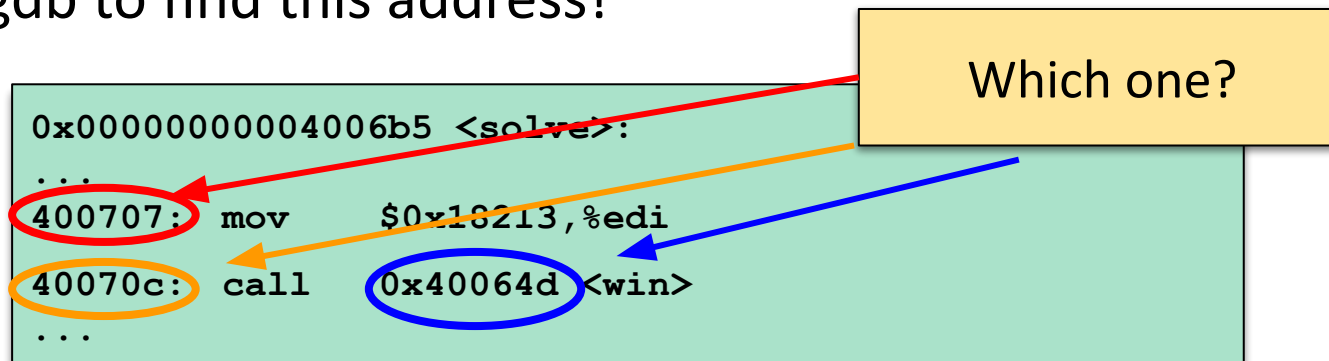
Activity 2

- Change the return address:



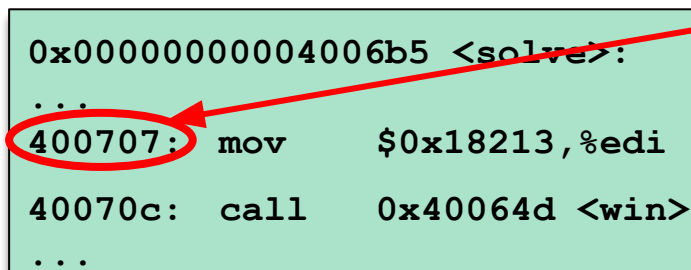
Activity 2: Reflection

- What address should we place in our return address?
- Use gdb to find this address!



Activity 2: Reflection

- What address should we place in our return address?
- Use gdb to find this address!



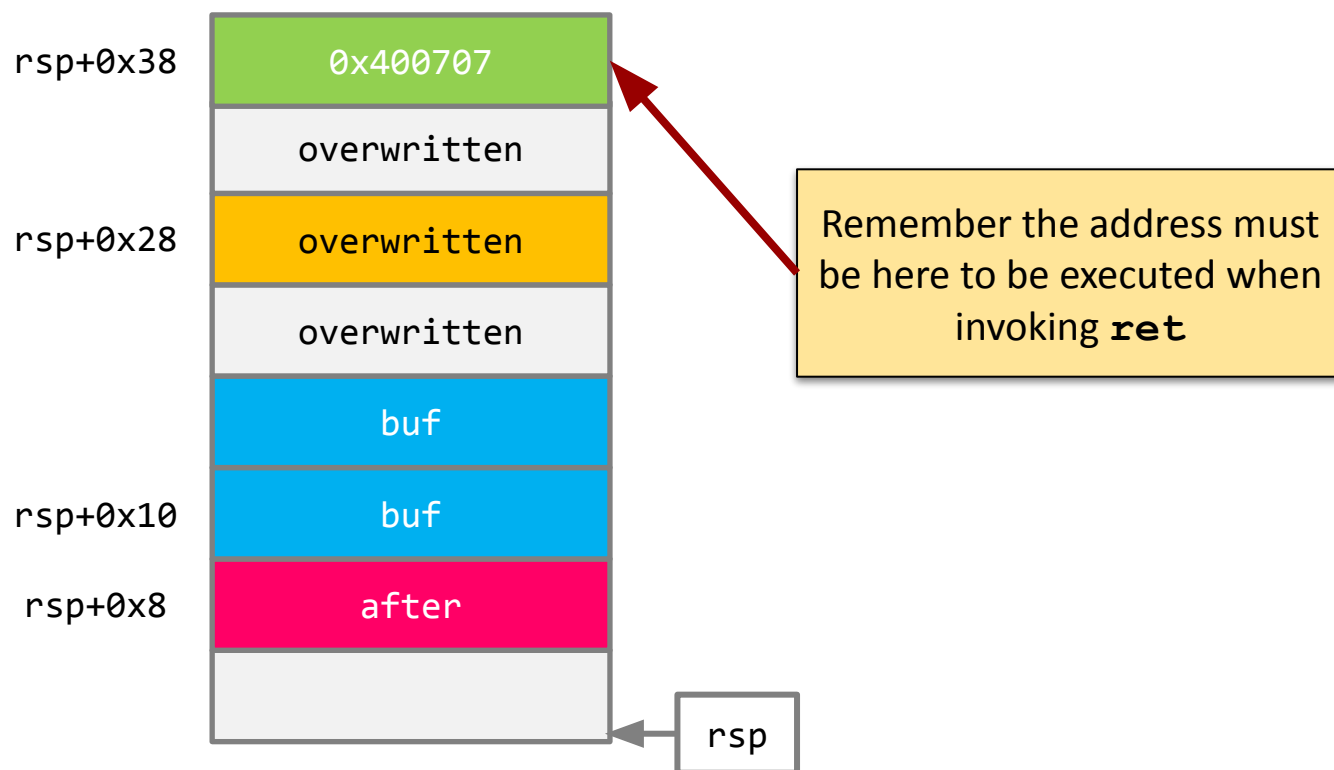
```
0x0000000004006b5 <solve>:  
...  
400707: mov     $0x18213,%edi  
40070c: call   0x40064d <win>  
...
```

Correct!

- Remember that we can't just call **win()**, we need to ensure that the first argument is set to **0x18213**.

Activity 2

- We write this address onto our stack:

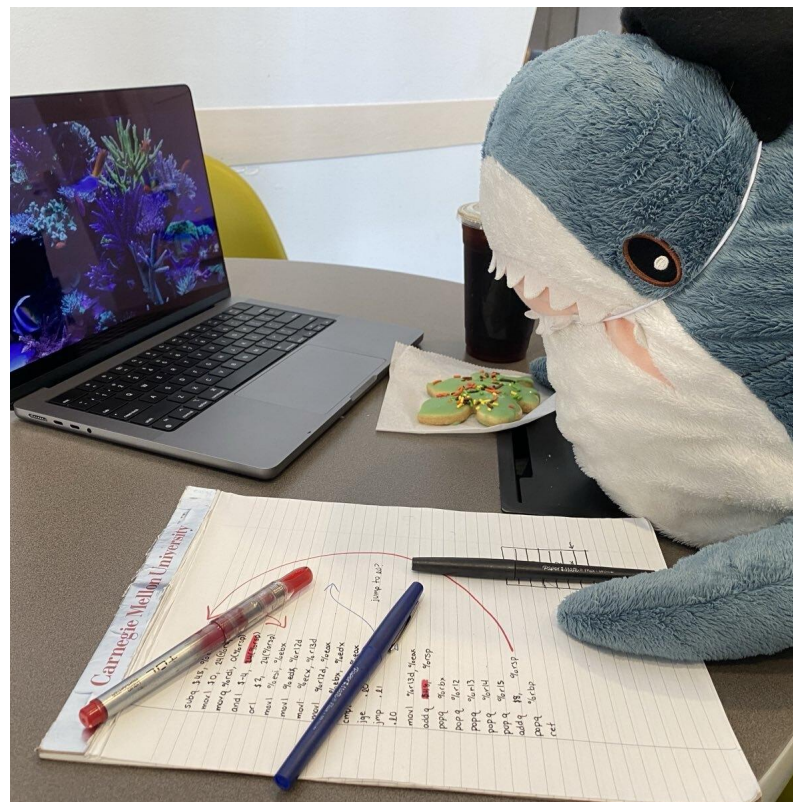


Activity 3 (If we have time)

- Goal: call `win(0x18613)`
- There's no conditional or existing that we can leverage now.

What else can we exploit?

- *Hint:* Are there any gadgets available to us?



The End