SIGPwny

FA2023 Week 05 • 2023-10-01

# x86-64 Assembly

Sam Ruggerio

Announcements

- Right NOW until 7 PM: BuckeyeCTF + Maple CTF
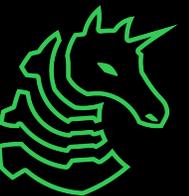
sigpwny{im_in_the_mines_of_linux}
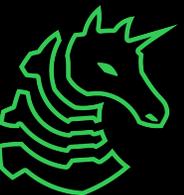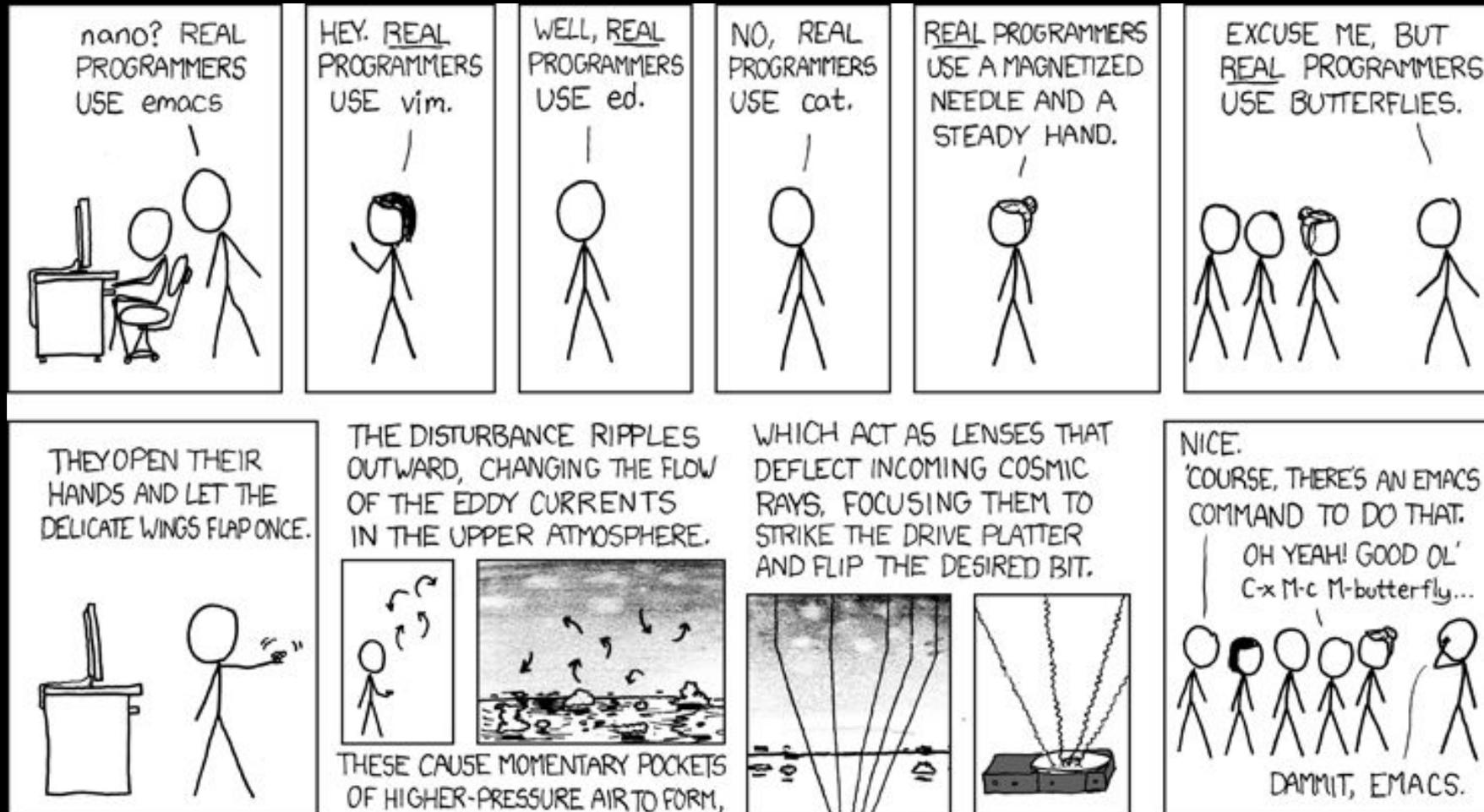
What is Assembly?

- A human-readable abstraction over CPU machine codes

0100100000000101110111101100000000110111100010011

48 05 DE C0 37 13

add rax, 0x1337c0de
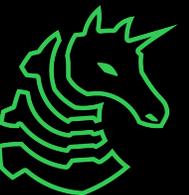
What is Assembly?

```
int method(int a){
    int b = 6;
    char c = 'c';
    return a+b;
}
```

```
method:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     DWORD PTR [rbp-4], 6
        mov     BYTE PTR [rbp-5], 99
        mov     edx, DWORD PTR [rbp-20]
        mov     eax, DWORD PTR [rbp-4]
        add     eax, edx
        pop     rbp
        ret
```
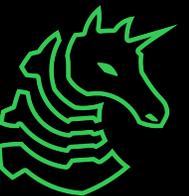
Basic CPU Structures

Instruction Memory

```
[0x00401000]
    ;-- section..text:
    ;-- segment.LOAD1:
entry0 ();
push     rsp
pop      rsi
xor      dl, 0x60
syscall
ret
```

Registers

```
*RAX  0x3e8
*RBX  0x401300 (__libc_csu_init) <-
*RCX  0x7ffff7ea311b (getegid+11) <-
 RDX  0x0
*RDI  0x7ffff7fad7e0 (_IO_stdfile_1
 RSI  0x0
 R8   0x0
*R9   0x7ffff7fe0d60 (_dl_fini) <-
*R10  0x400502 <- 0x64696765746567
*R11  0x202
*R12  0x401110 (_start) <- endbr64
*R13  0x7fffffffddc0 <- 0x1
 R14  0x0
 R15  0x0
*RBP  0x7fffffffdcd0 <- 0x0
*RSP  0x7fffffffdcb0 <- 0x0
*RIP  0x401220 (main+42) <- mov
```

Stack

```
0x7fffffffdcb0 <- 0x0
0x7fffffffdcb8 -> 0x401110 (_star
0x7fffffffdcc0 -> 0x7fffffffddc0
0x7fffffffdcc8 <- 0x0
0x7fffffffdcd0 <- 0x0
0x7fffffffdcd8 -> 0x7ffff7de3083
```
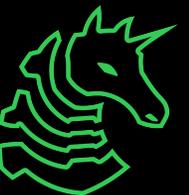
Instruction Memory

- Contiguous memory of executable data

- Normally, only read & execute permissions.

- At very low address space (below the heap!)

- Managed by the special purpose
  Instruction Pointer register: `rip`
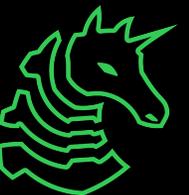
Registers

- 16 general purpose "variables" that the CPU can operate on. On a 64 bit architecture, each are 64 bits wide.

- Most can be used for whatever you want within a function, except for:
  - `rbp` which is the "<u>B</u>ase <u>P</u>ointer" register
  - `rsp` which is the "<u>S</u>tack <u>P</u>ointer" register

- We can access lower bits using various namings for each register
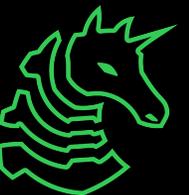
Registers

| 8 Byte	4 Byte	2 Byte	1 Byte
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rsp	esp	sp	spl
rbp	ebp	bp	bpl
rX	rXd	rXw	rXb

```
01 23 45 67 89 ab cd ef
```
rax

eax

ax al

← These registers are named r8 through r15
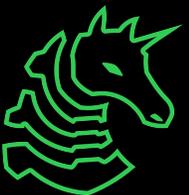
Stack

- The region of memory dedicated to functions and local variables

- Push to the stack to add data, pop to remove newest element.

Bottom of Memory
(`0x00000000`)

Top of Memory
(`0xFFFFFFFF`)

| Memory Region |
| --- |
| code/globals/strings |
| heap |
| stack |
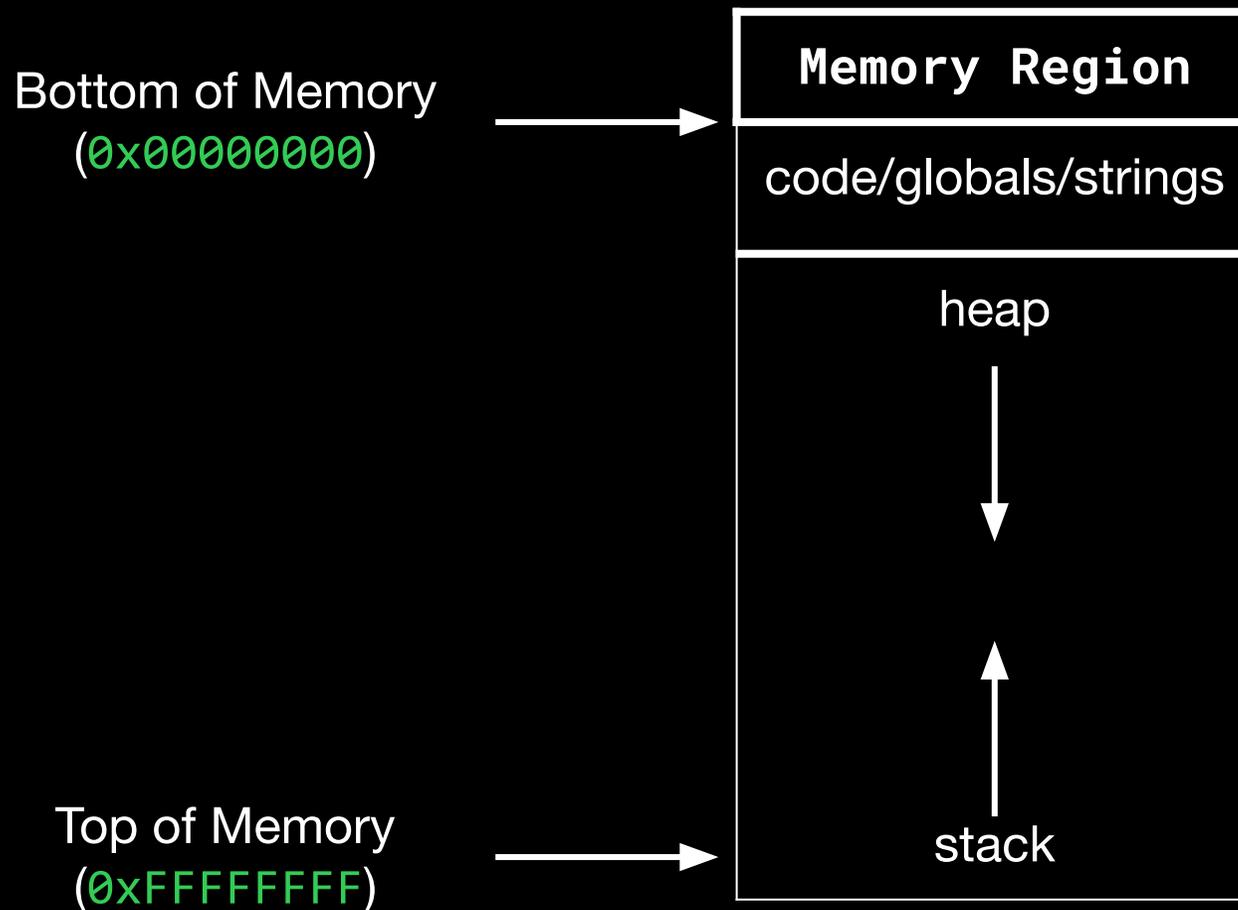
Stack & Registers

- There are two registers dedicated to managing the stack

- `rsp` - holds the address of the *top* of the stack
    - If you want to allocate memory on the stack, you subtract from `rsp`
    - Likewise to deallocate, add to `rsp`.

- `rbp` - holds the address of the start of the stack frame
    - The value at the address holds the base ptr of the calling function

Stack & Functions

Local Variables
Saved Base Pointer
Return Address
Arguments

rsp

rbp
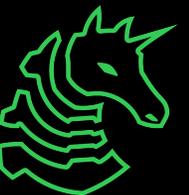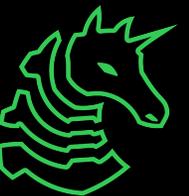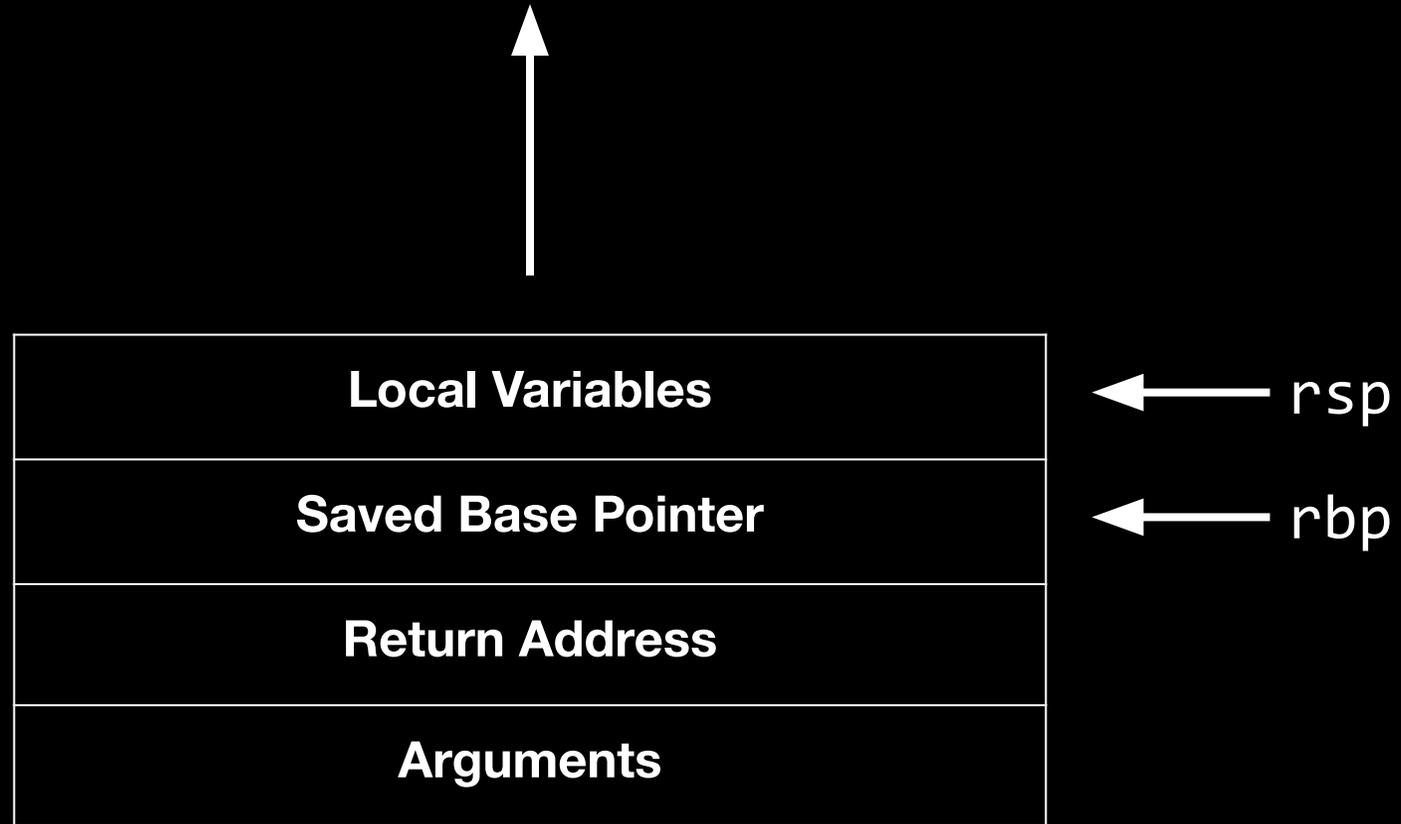
Stack & Functions

```
method_1(int a){
    int b = 7;
    char c = 'a';
    float d = 2.5;
    return a+b
}
```

b = 7	← rsp
c = 'a'	
d = 2.5	
Saved Base Pointer	← rbp
Return Address	
a	
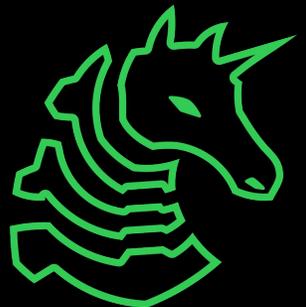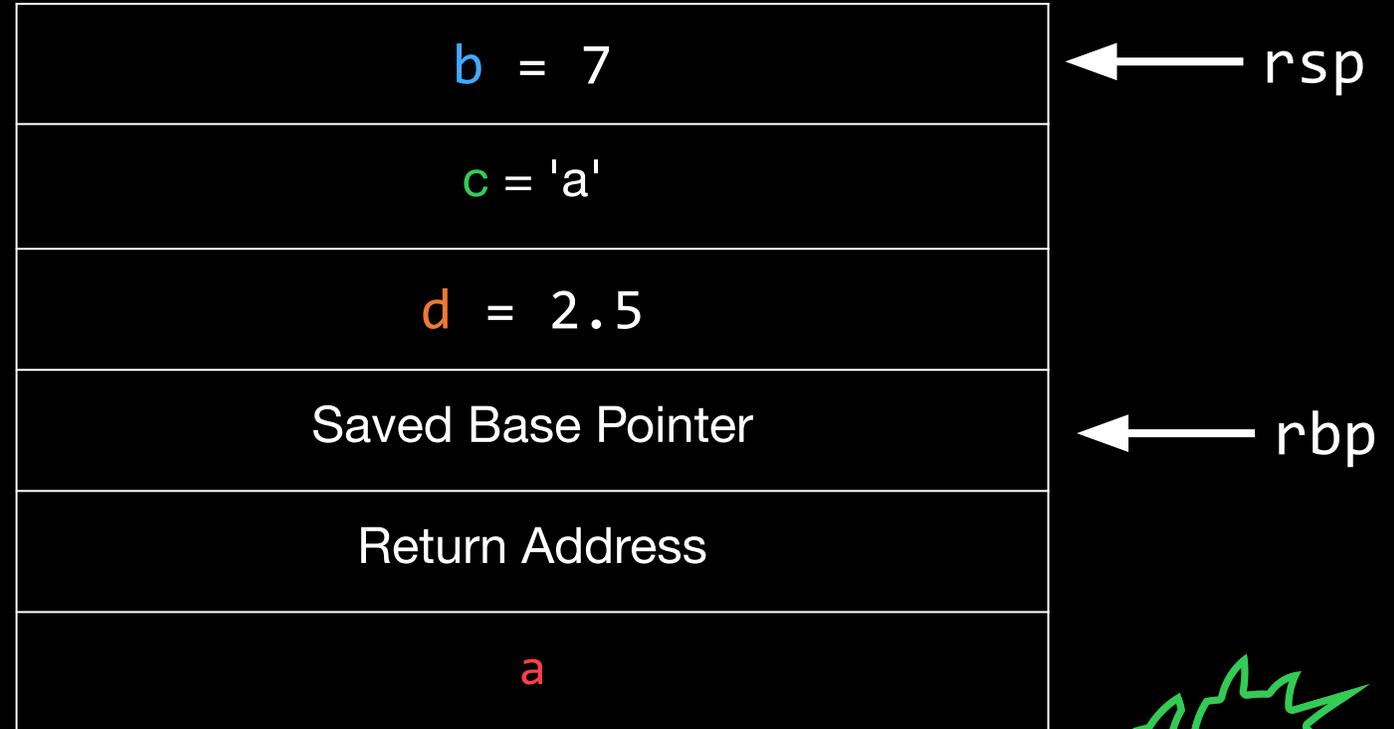
A Note on Syntax
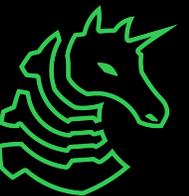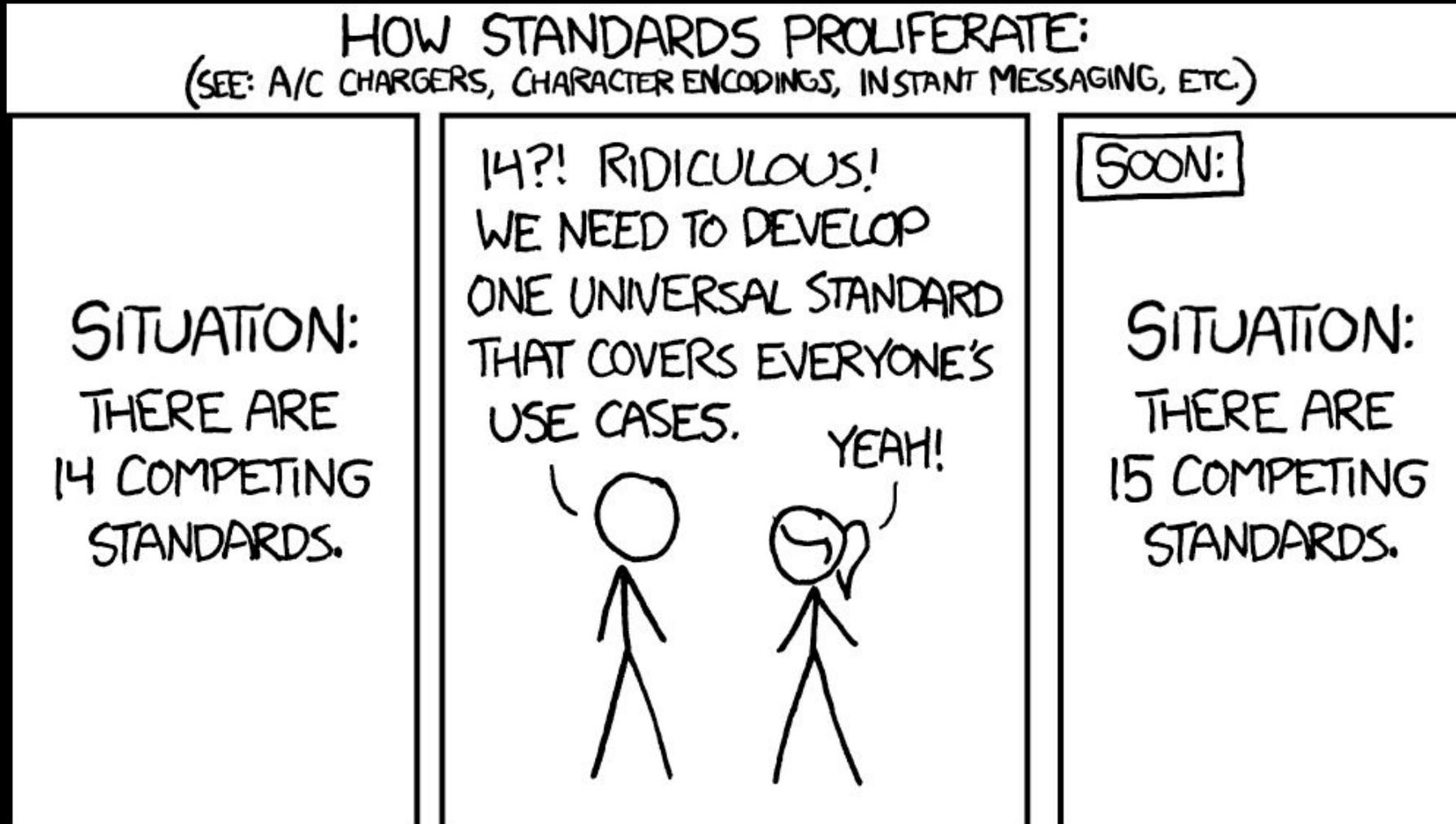
Intel vs AT&T

	Intel	AT&T
Registers	`rax, rsp, r15`	`%rax, %rsp, %r15`
Immediates (Constants)	`0x123`	`$0x123`
Command Order / Typing	`add eax, bx`	`addzqd %bx, %eax`
Comments	`; this is a comment.`	`// this is, too.`
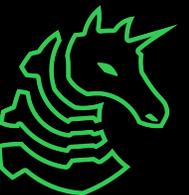
Basic Assembly

mnemonic destination, source(s)

e.g.

```
add rax, rbx              nop

sub dx, 0x1235            mov rbp, rsp

and rsp, rbp              imul r8, r10, 0x20

xor rsi, rsi             shl rcx

inc ecx                  sar rdi, 5
```
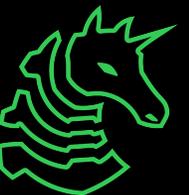
Logic Flow

- We can use `jmp addr` to jump to nearby addresses in our instruction code

- near/short jumps are relative, but when writing we can use labels!

- This is one of the few ways to modify `rip` (hopefully) safely.
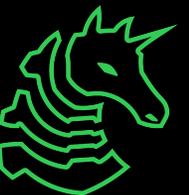
Logic Flow

- Assembly compares values by subtracting values (a-b)
  - If we get 0, a=b
  - If we get a positive number, a>b, otherwise, a<b

- `cmp` subtracts two registers and sets flags (RFLAGS register) for later use

- `jCC` jumps to address if condition is met, based on flags set by `cmp`. There's 64 of them.
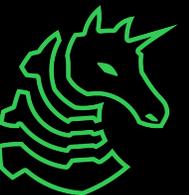
Logic Flow

```asm
mov rbx, 0x20     ; move 32 into rbx
mov rax, 0x15     ; move 21 into rax
foo:
    cmp rax, rbx  ; compare rax and rbx
    jne bar       ; if not equal, jump to bar label
    xor rax, rax  ; zero out rax
    ret           ; return
bar:
    dec rbx       ; decrement rbx
    jmp foo       ; jump to foo label
```
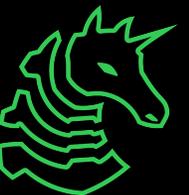
Using the Stack

- Use `push (reg/imm)` to push a 16 bit, 32 bit or 64 bit value onto the stack.
    - `rsp` is *automatically* decremented

- Use `pop reg` to pop a value from the stack into a register
    - `rsp` is *automatically* incremented

Using the Stack

```
mov rax, 0x1337c0de
push rax
xor rax, rax
pop rbx
```

Saved Base Pointer
Return Address

← rsp
rbp
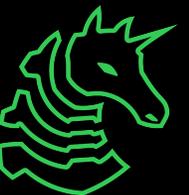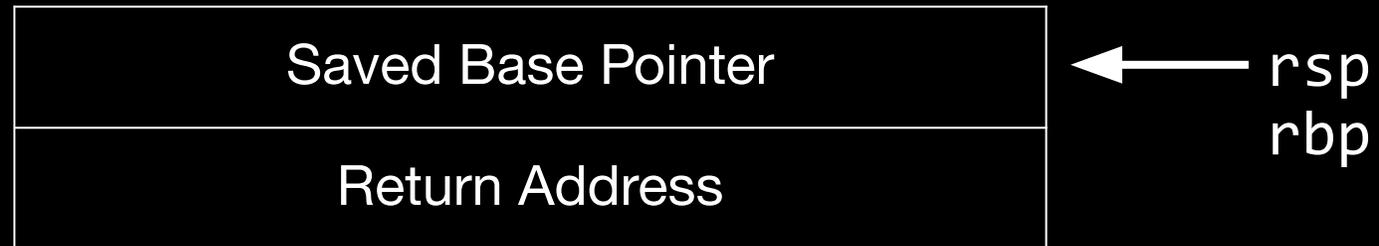
rax: 0x1234567890abcdef
rbx: 0x1234567890abcdef
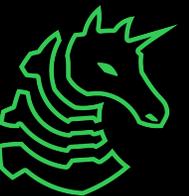
Using the Stack

```
mov rax, 0x1337c0de
push rax
xor rax, rax
pop rbx
```

Saved Base Pointer	← rsp
Return Address	rbp

rax: 0x000000001337c0de
rbx: 0x1234567890abcdef

Using the Stack

```
mov rax, 0x1337c0de
push rax
xor rax, rax
pop rbx
```

0x000000001337c0de	← rsp
Saved Base Pointer	← rbp
Return Address

rax: 0x000000001337c0de
rbx: 0x1234567890abcdef

Using the Stack

```
mov rax, 0x1337c0de
push rax
xor rax, rax      ←
pop rbx
```
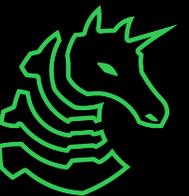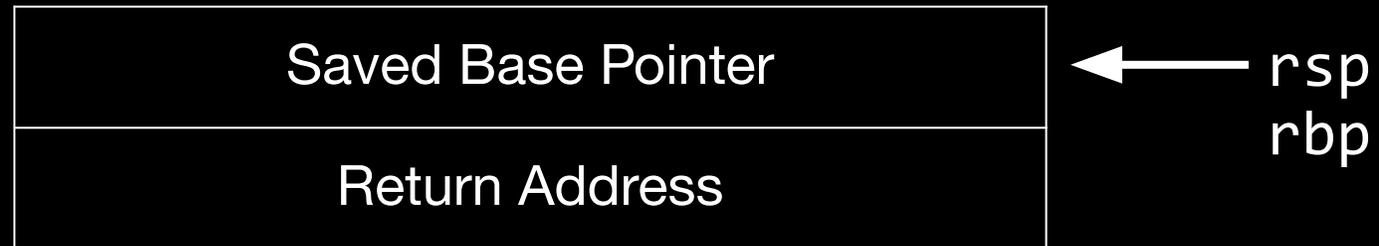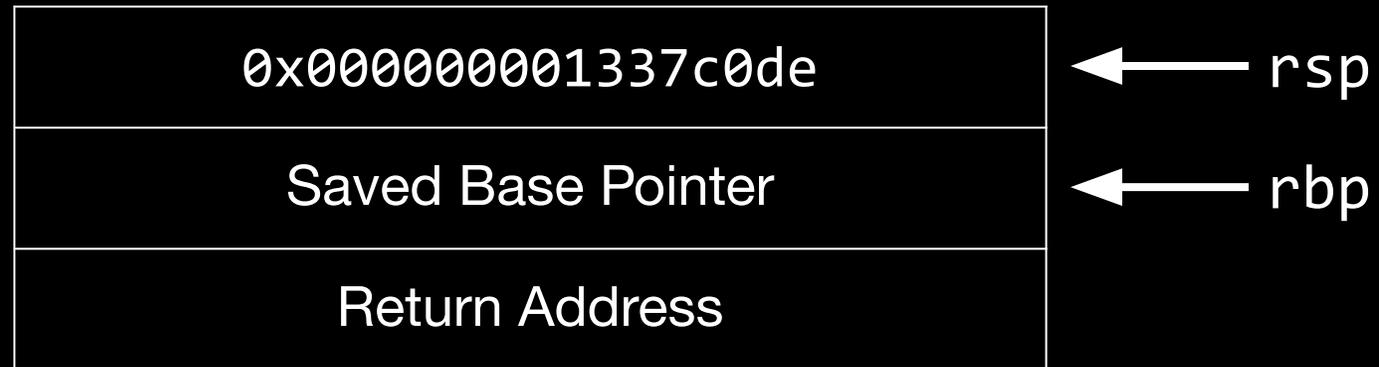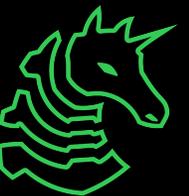
0x000000001337c0de	← rsp
Saved Base Pointer	← rbp
Return Address	
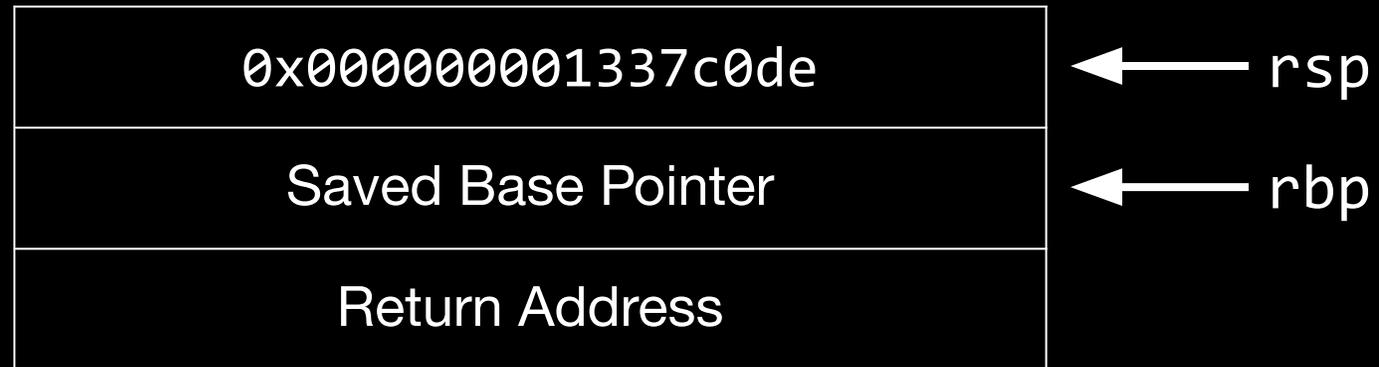
rax: 0x000000000000000
rbx: 0x1234567890abcdef
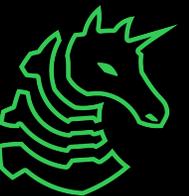
Using the Stack

```
mov rax, 0x1337c0de
push rax
xor rax, rax
pop rbx
```

Saved Base Pointer
Return Address

rsp
rbp

rax: 0x000000000000000
rbx: 0x000000001337c0de
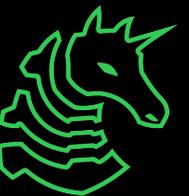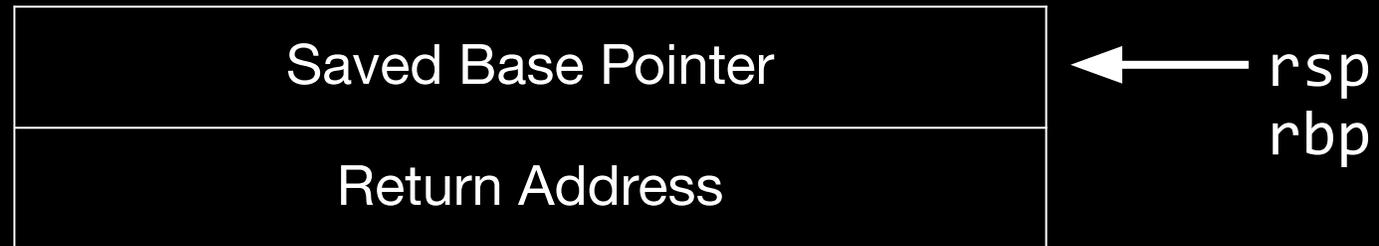
Syscalls

- The linux kernel provides a set of functions to interface with the OS.

- glibc provides wrappers, so *most* programs use glibc calls
  - But you can inline system calls without calling glibc at all!

- Examples of system calls: `read, exit, open, execve`
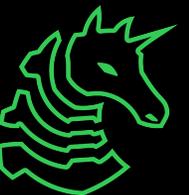
Calling a Syscall

- Load the syscall id into `rax`
  - The most up-to-date resource of ids to syscalls is the abi table: https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl

- Load your arguments into the registers, in order, as follows:

  `rdi, rsi, rdx, r10, r8, r9`

- Use the `syscall` instruction
- return value, if needed, is stored in `rax`
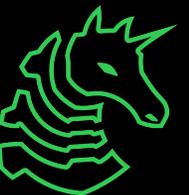
Calling a Syscall

```
exit(10);                mov rax, 0x3c
                         mov rdi, 0x0a
                         syscall
```

```
execve("/bin/sh",        mov rax, 0x3B
NULL, NULL);             mov rdi, rsp ; /bin/sh is on the stack
                         xor rsi, rsi
                         xor rdx, rdx
                         syscall
```
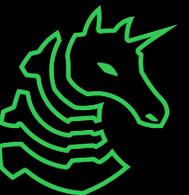
Pointers and Dereferencing

- At a *high level*, use braces to dereference a pointer

  `mov rax, [rbx] ;` `moves the memory pointed by rbx to rax`

- You may use a index register, a scale for that index, and a displacement in a dereference

  `mov rax, [rbx + rcx*4 + 0x1a]`

  - This is useful for iterating through arrays

- Writing to memory can be done the same way

  `inc [rsp] ;` `increments the top value on the stack by 1`

Resources

RTFM: https://www.felixcloutier.com/x86/
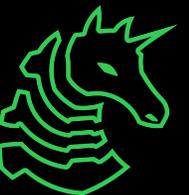
⭐ Online Assembler: defuse.ca/online-x86-assembler

Syscall Table & Argument Convention: Chromium Docs

Flat Assembler/Fasm: https://flatassembler.net/

Compiler Explorer: https://godbolt.org/
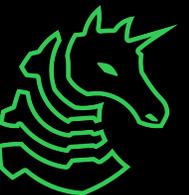
Challenges

- 1 - asm_adder
- 2 - asm_leaver
- 3 - asm_reader
- 4 - asm_shellcode
- 5 - asm_modifier

Use pwntools! An example script:

```
from pwn import *

conn = process("./chal") # or remote("link", port)

conn.sendline(b'your shellcode here')

conn.interactive()
```
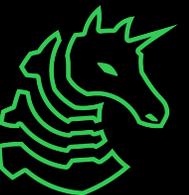
Next Meetings

2023-10-05 - **This Thursday**

- Reverse Engineering II
- Learn how to reverse engineer x86 binaries!

2023-10-08 - **Next Sunday**

- Circumventing DRM with Ojas