



General

FA2026 • 2026-09-20

x86-64 Assembly

Slides by Julius White, Emma Shu, Sam Ruggerio, and Ryan Yin

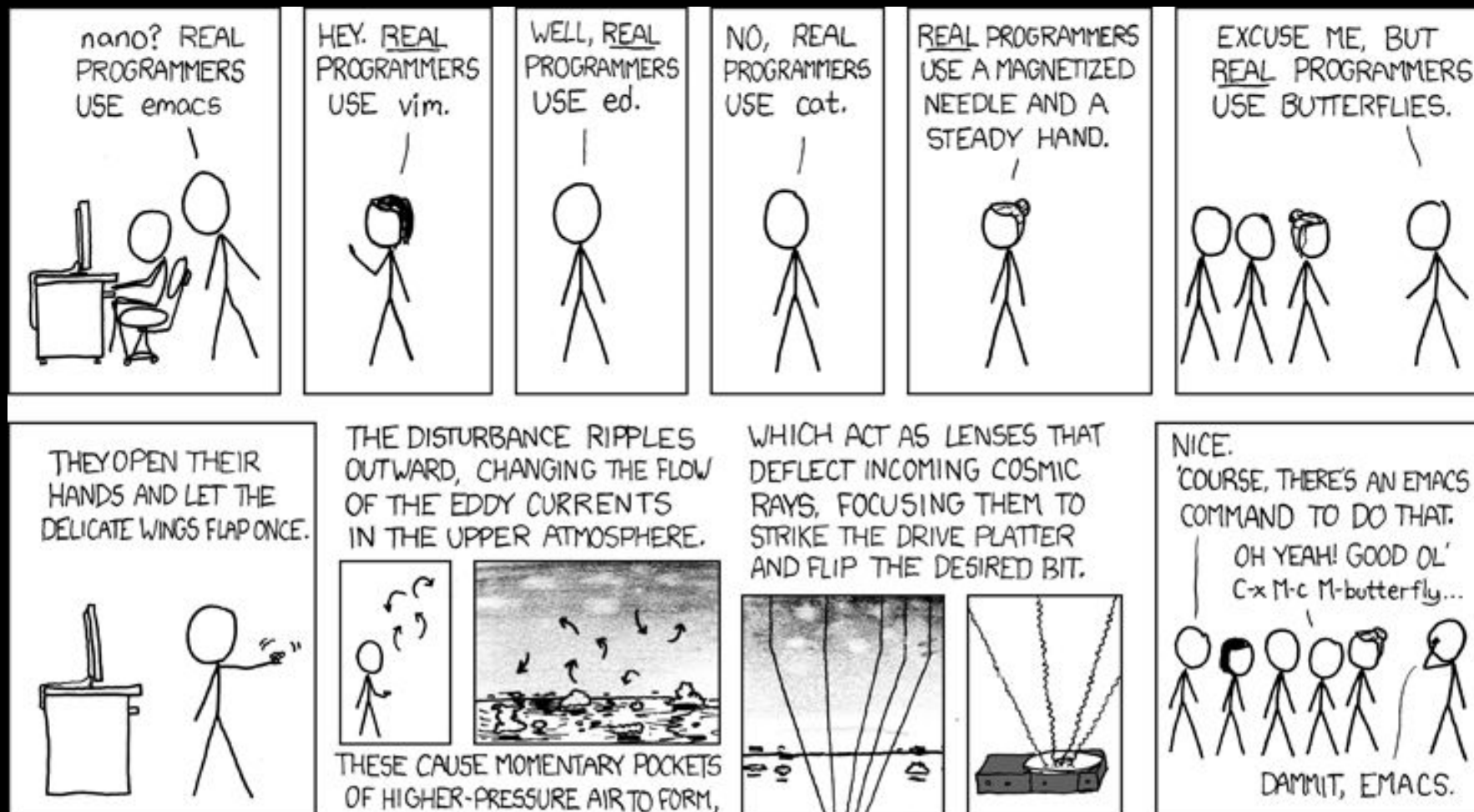
Ryan Yin

- Computer Science Sophomore
- I played mixed doubles badminton in high school
- Admin in SIGPwny



ctf.sigpwny.com

sigpwny{x0r_e4x_e4x}



Announcements

- AmateursCTF this weekend is unfortunately canceled
NEW PLAN: Game night in Siebel with pizza on Friday!
 - Details will be sent in an announcement on Discord
- First lockpicking social Monday 10/13 at 7 PM!
 - Will talk briefly about the history of magstripe and do a demo



What is Assembly?

- A human-readable abstraction over CPU machine codes

01001000000001011101111011000000011011100010011

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 ← 0x646967657465567
*R11 0x202
*R12 0x401110 (_start) ← endbr64
*R13 0x7fffffffddc0 ← 0x1
R14 0x0
R15 0x0
*RBP 0x7fffffffddcd0 ← 0x0
*RSP 0x7fffffffddcb0 ← 0x0
*RIP 0x401220 (main+42) ← mov
```

Stack

```
0x7fffffffddcb0 ← 0x0
0x7fffffffddcb8 → 0x401110 (_start)
0x7fffffffddcc0 → 0x7fffffffddc0
0x7fffffffddcc8 ← 0x0
0x7fffffffddcd0 ← 0x0
0x7fffffffddcd8 → 0x7ffff7de3083
```



Instruction Memory

- Contiguous memory of executable data
 - “This is where your compiled program lives in RAM”
- Normally, only read & execute permissions (security feature)
- At very low address space (below the heap!)
- Managed by the special purpose Instruction Pointer register (rip) also called program counter
- Simply, Instruction memory is just RAM that holds your program's code. The CPU fetches from it, executes it, and moves on



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 "Base Pointer" register
 - rsp which is the "Stack Pointer" 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

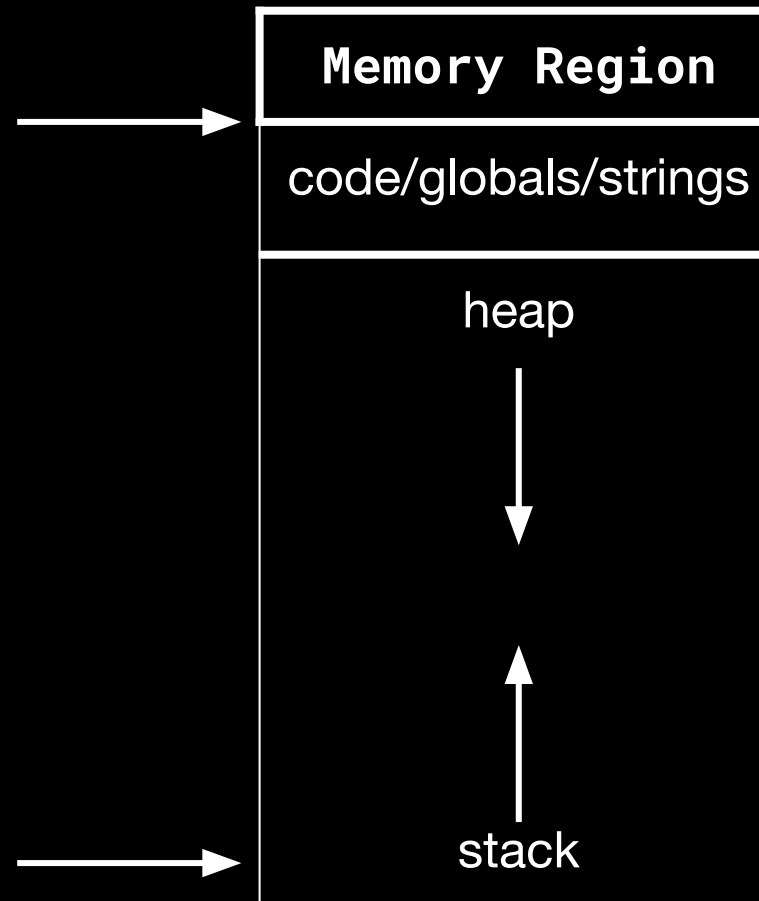
01 23 45 67 89 ab cd ef
rax eax ax al

← 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. (LIFO)
 - If the heap and stack meet you get a stack overflow or out of memory error
- Bottom of Memory
(0x00000000)
- Top of Memory
(0xFFFFFFFF)

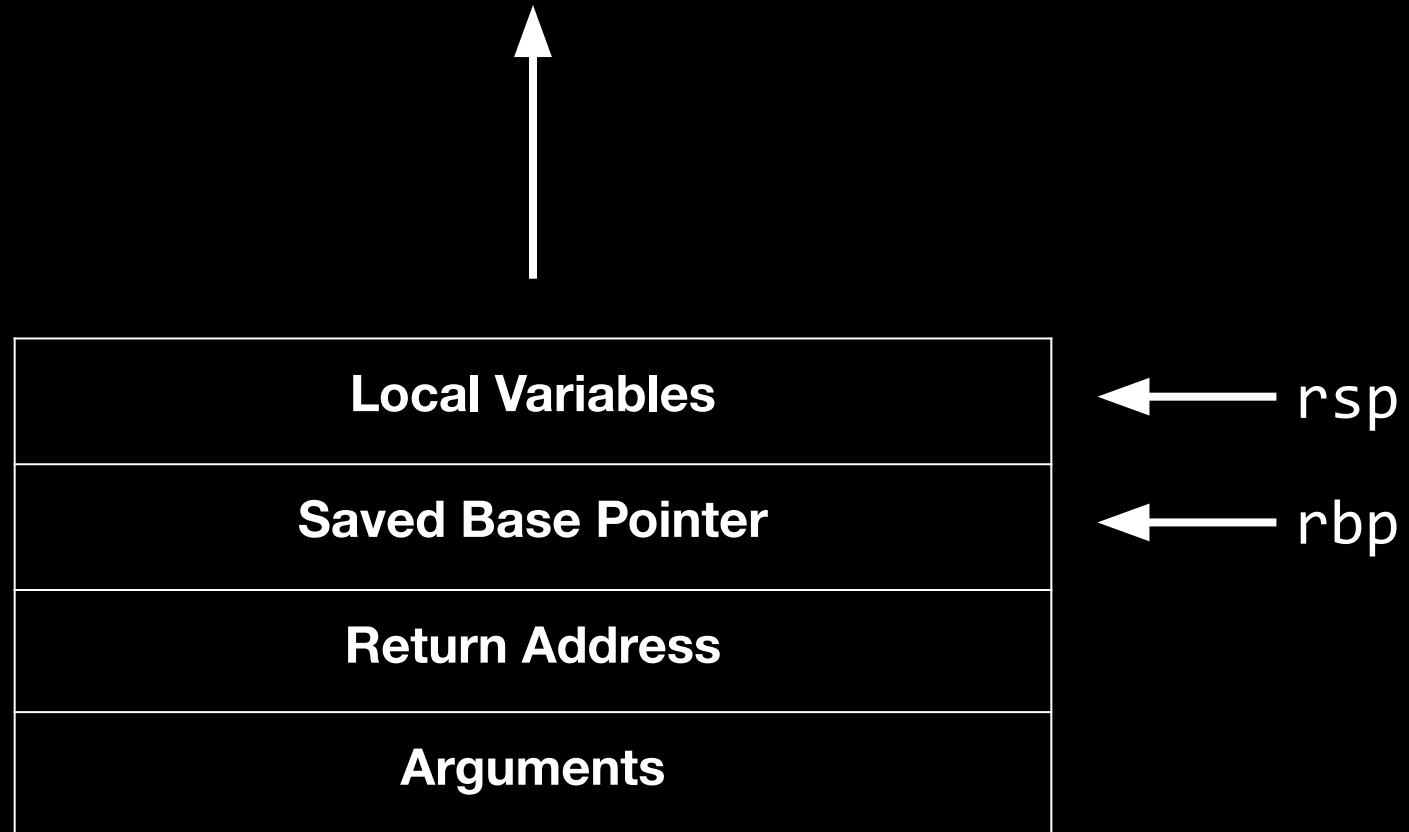


Stack & Registers

- 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



Saved Base Pointer vs Return Address

(what's the difference?)

saved base pointer - the scope of the previous function

return address - the line to go back to in the previous function

```
1 helper(int arg0) {  
2     int temp = 67; // lol  
3     int temp2 = 100;  
4     return;  
5 }  
6  
7 main() {  
8     helper(3);  
9 }  
10  
11  
12  
13 }
```

temp = 67 = “we are holding 67 in a register”

Saved base pointer = “you were running the main() function”

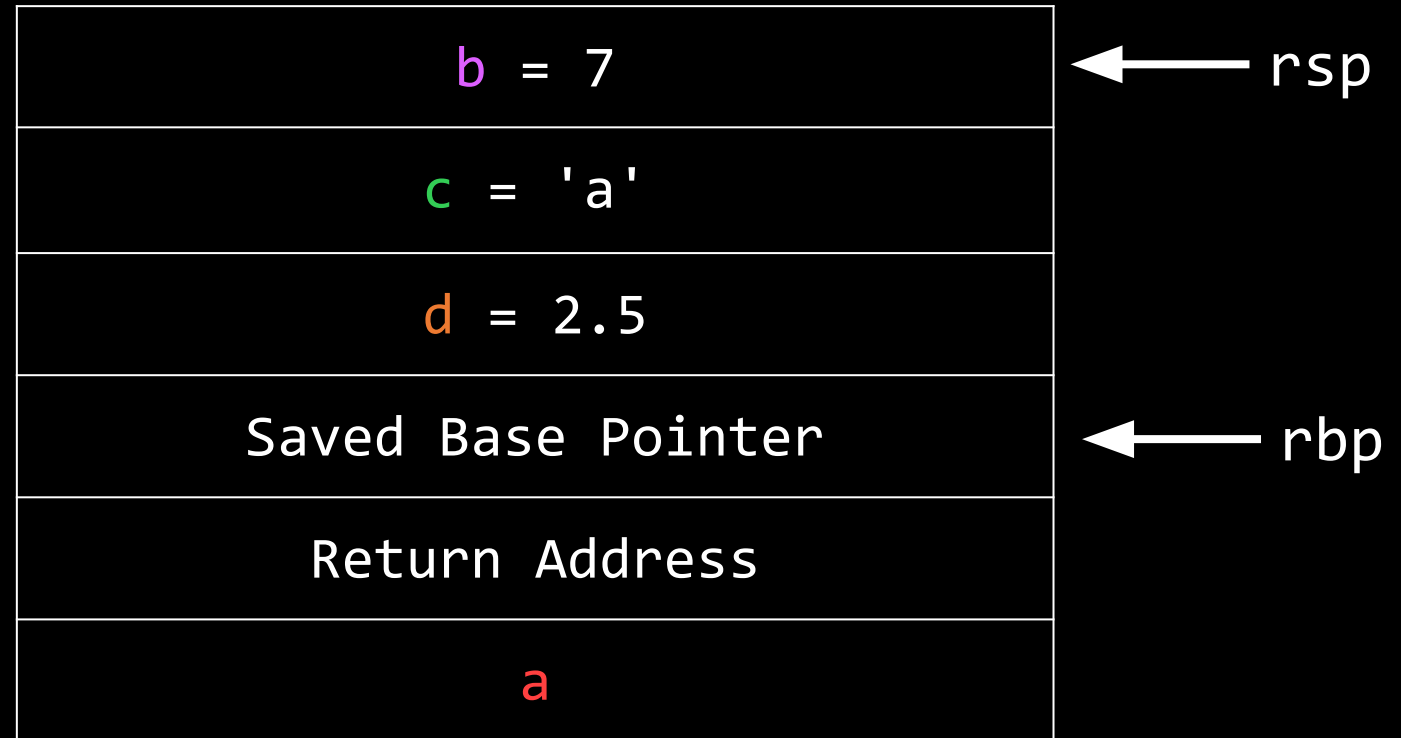
Return Address = “you were at line 11 in main()”

arguments = **arg0 = 3**



Stack & Functions

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



Function calls

```
int method_1(int a);
```

How do we call this?

How do we know what it returns?



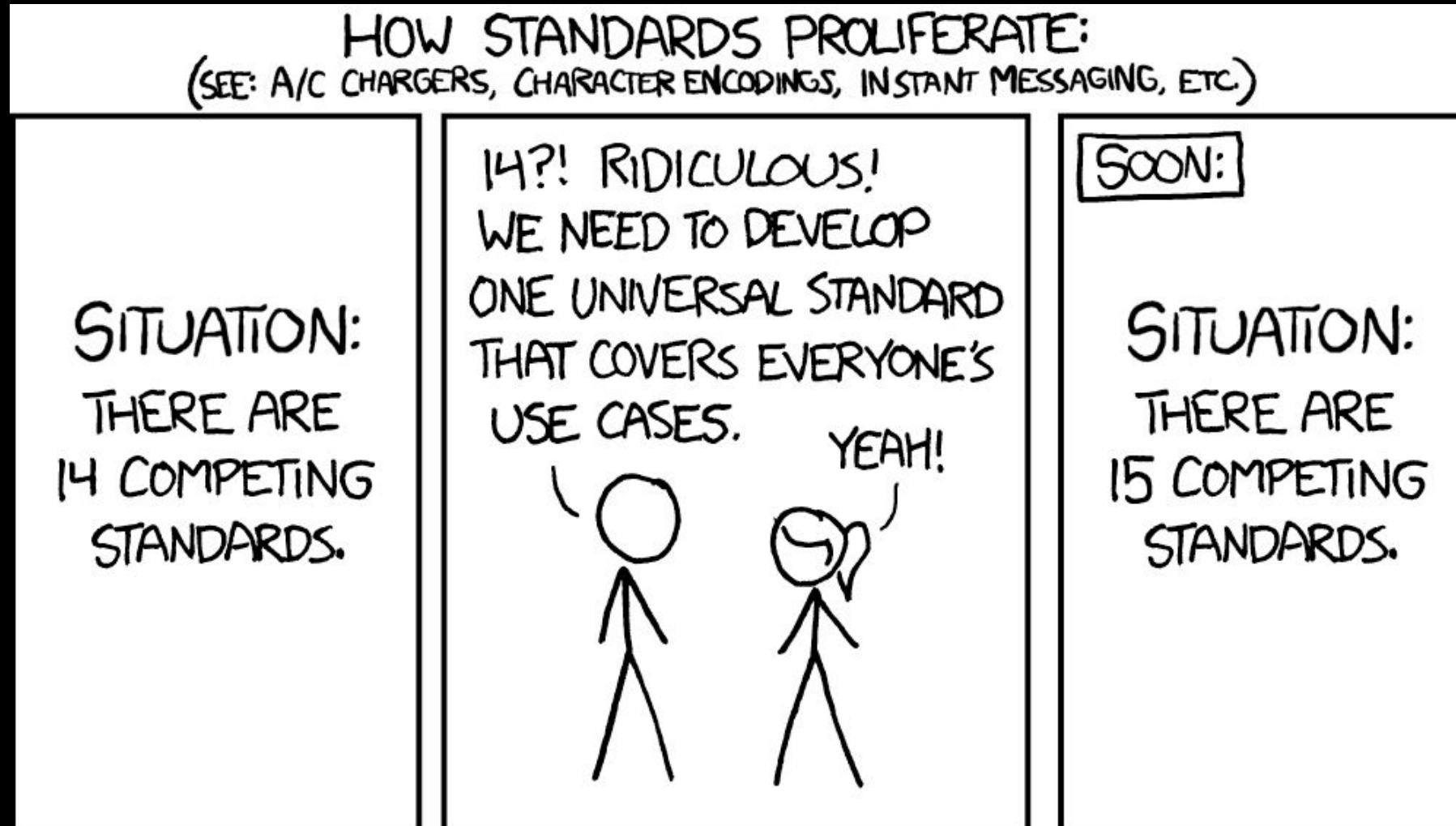
Calling Conventions

```
int method_1(int a);
```

Platform	Return Value	Parameter Registers	Additional Parameters	Stack Alignment	Scratch Registers	Preserved Registers	Call List
System V i386	eax, edx	none	stack (right to left) ¹		eax, ecx, edx	ebx, esi, edi, ebp, esp	ebp
System V X86_64 ²	rax, rdx	rdi, rsi, rdx, rcx, r8, r9	stack (right to left) ¹	16-byte at call ³	rax, rdi, rsi, rdx, rcx, r8, r9, r10, r11	rbx, rsp, rbp, r12, r13, r14, r15	rbp
Microsoft x64, UEFI X64	rax	rcx, rdx, r8, r9	stack (right to left) ¹	16-byte at call ³	rax, rcx, rdx, r8, r9, r10, r11	rbx, rdi, rsi, rsp, rbp, r12, r13, r14, r15	rbp
ARM (32-bit)	r0, r1	r0, r1, r2, r3	stack	8 byte ⁴	r0, r1, r2, r3, r12	r4, r5, r6, r7, r8, r9, r10, r11, r13, r14	



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
```

```
sub dx, 0x1235
```

```
and rsp, rbp
```

```
xor rsi, rsi
```

```
inc ecx
```

```
nop
```

```
mov rbp, rsp
```

```
imul r8, r10, 0x20
```

```
shl rcx
```

```
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

```
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
```



Logic Flow

```
first = 32    # move 32 into rbx
```

```
second = 21   # move 21 into rax
```

```
while (first != second): # compare first and second  
    first -= 1           # decrement first
```

```
return        # why are python and x86 different  
              # (in terms of returning variables)
```



Logic Flow

```
push rdi
```

```
mov rdi, 0x20 ; move 32 into rdi
```

```
mov rax, 0x15 ; move 21 into rax
```

```
assembly logic ...
```

```
pop rdi ; we expect our return value to be in rax
```



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

push:

- `mov rsp, reg`
- `rsp -= 4`

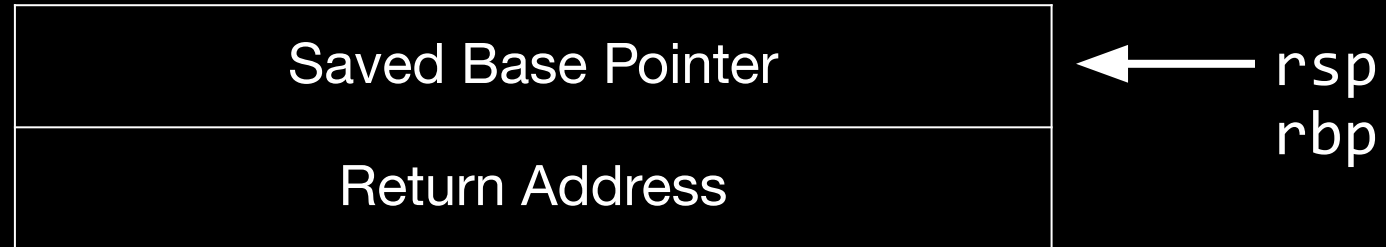
pop:

- `mov reg, rsp`
- `rsp += 4`



Using the Stack

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



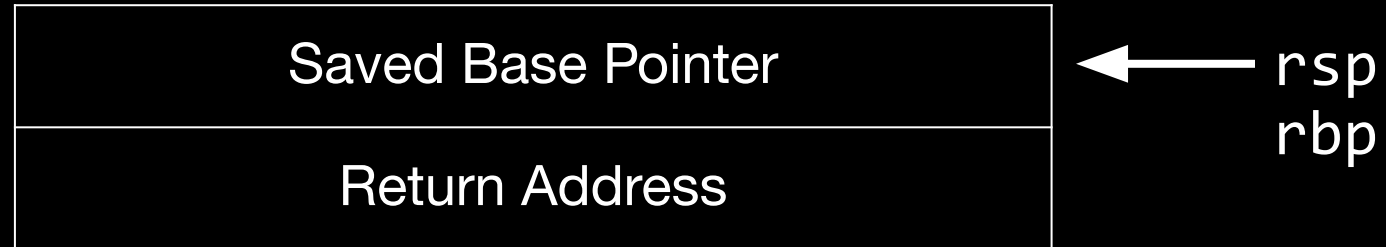
```
rax: 0x1234567890abcdef  
rbx: 0x1234567890abcdef
```



Using the Stack

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

```
rax: 0x000000001337c0de  
rbx: 0x1234567890abcdef
```



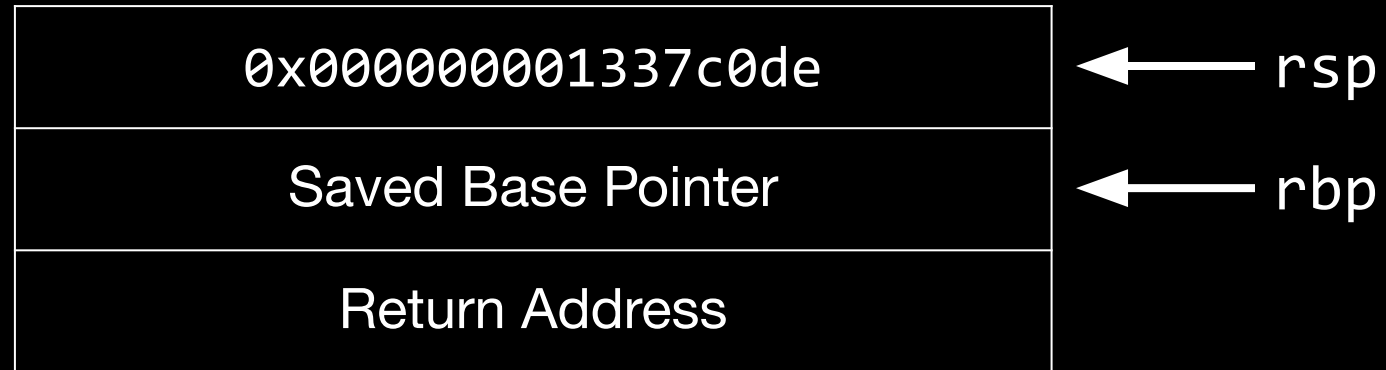
Using the Stack

```
mov rax, 0x1337c0de
```

```
→ push rax
```

```
xor rax, rax
```

```
pop rbx
```



```
rax: 0x000000001337c0de
```

```
rbx: 0x1234567890abcdef
```



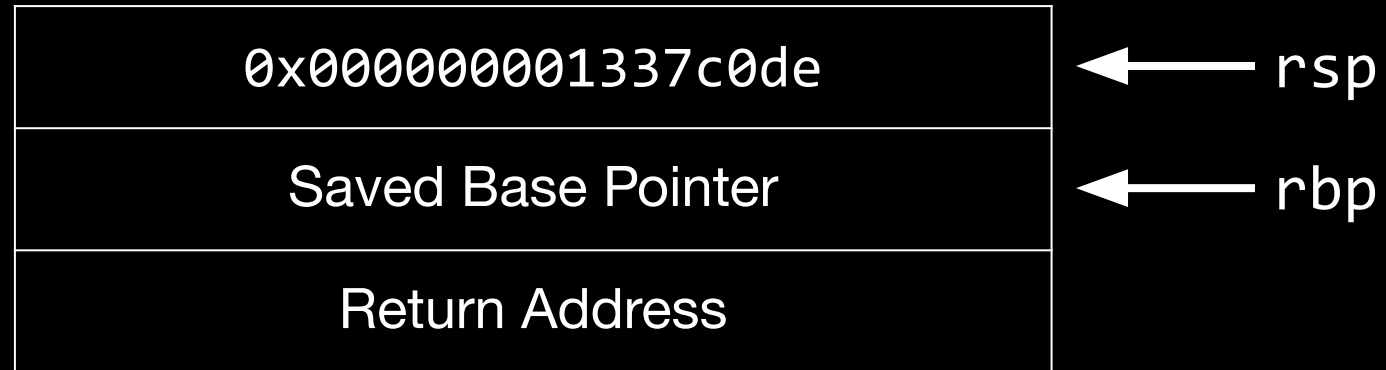
Using the Stack

```
mov rax, 0x1337c0de
```

```
push rax
```

```
→ xor rax, rax
```

```
pop rbx
```



```
rax: 0x0000000000000000
```

```
rbx: 0x1234567890abcdef
```



Using the Stack

```
mov rax, 0x1337c0de
```

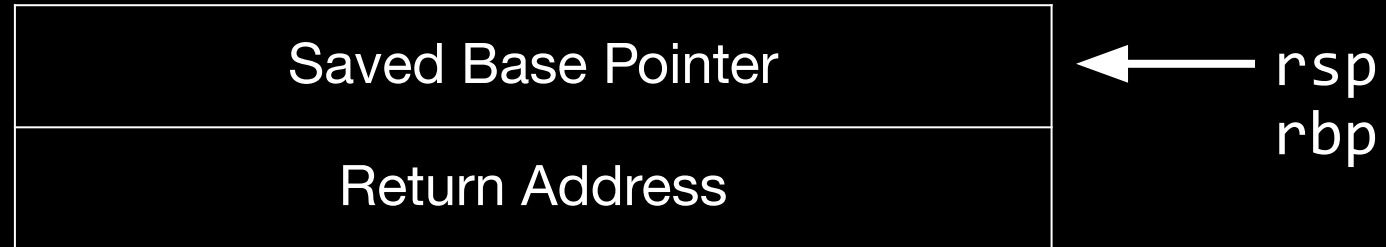
```
push rax
```

```
xor rax, rax
```

```
→ pop rax
```

```
rax: 0x0000000000000000
```

```
rbx: 0x000000001337c0de
```

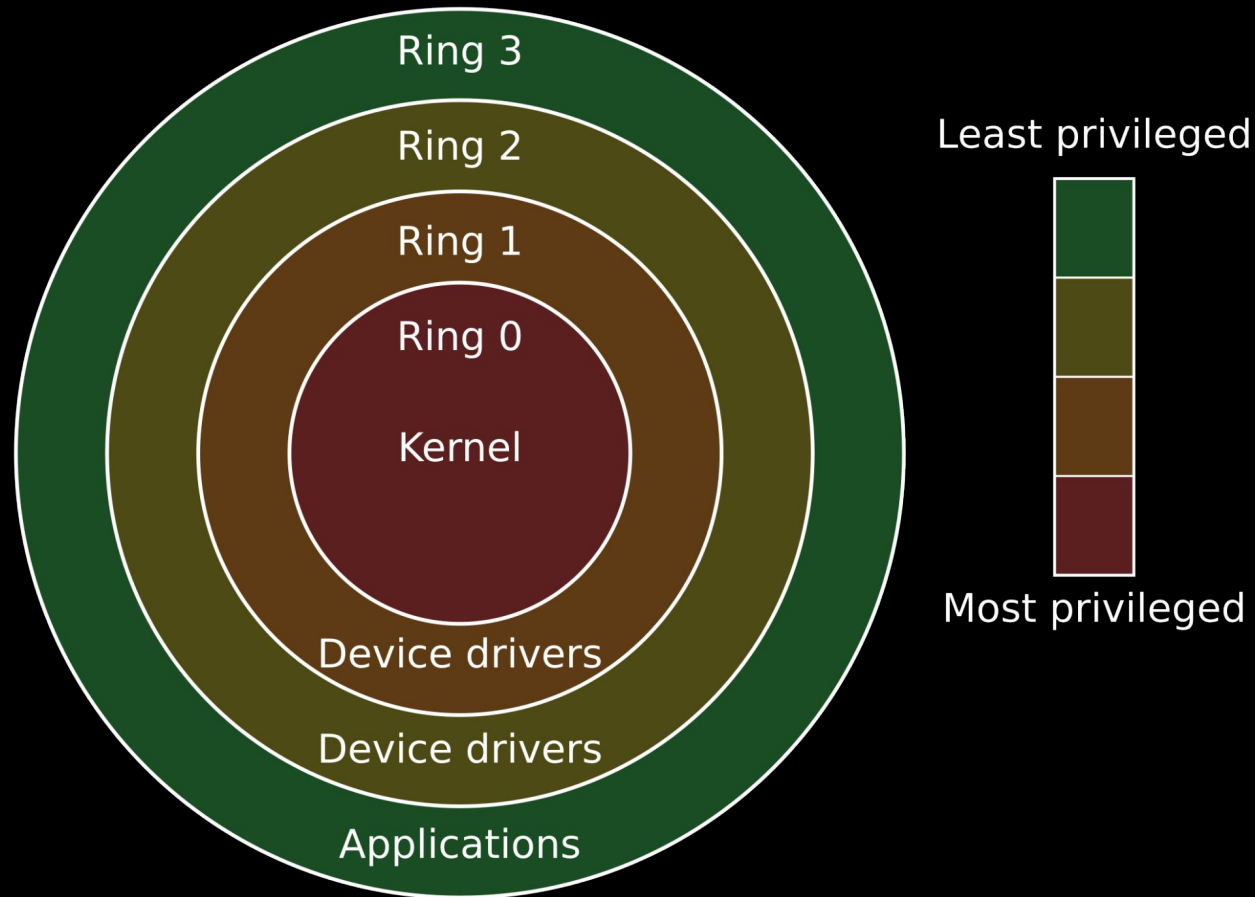


Syscalls

- 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



Why Syscalls?



Hertzprung at English Wikipedia, CC BY-SA 3.0
<<http://creativecommons.org/licenses/by-sa/3.0/>>, via Wikimedia Commons



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
```



A little history lesson (respect your elders)

- **Commercial blockbuster**
 - Sold millions of copies and successfully competed in the late 1990s gaming market
- **Massive scope**
 - pathfinding algorithms
 - financial systems
 - ride physics
 - intricate game logic



**ALL IN X86
ASSEMBLY?!?!?!?!?!?**



A little technical history lesson

(respect your elders)

- **99% x86 assembly written by one developer**
 - Chris Sawyer coded the entire game in assembly, not a higher-level language
- **Ran on 16MB RAM**
 - Managed thousands of guests and complex simulations on minimal hardware
- **Tiny executable of ~15-16 MB**
 - The main program file was remarkably compact, with total installation around 180 MB including all assets



Resources

RTFM: <https://www.felixcloutier.com/x86/>

Online Assembler: defuse.ca/online-x86-assembler

Syscall Table & Argument Convention:

<https://syscalls.pages.dev/>

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()
```



ctf.sigpwny.com

sigpwny{x0r_e4x_e4x}

Meeting content can be found at
sigpwny.com/meetings.

