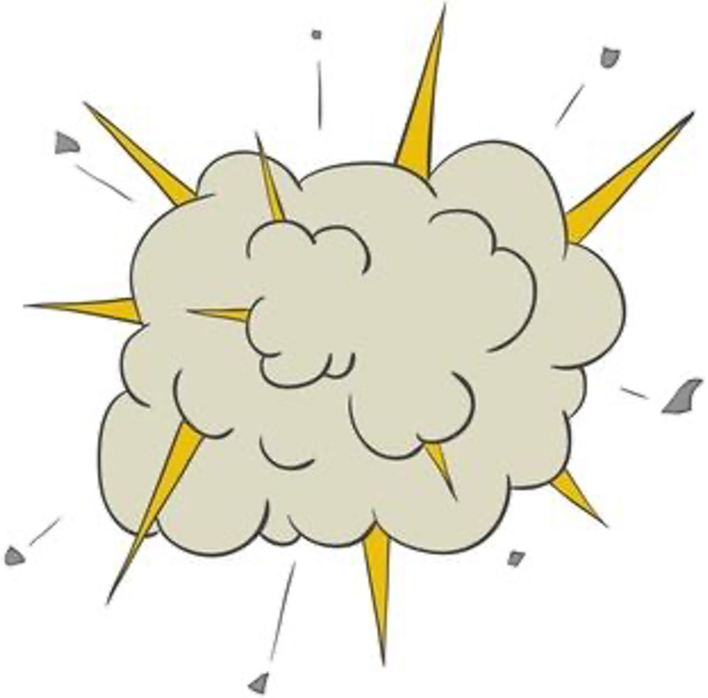
Attack Lab

- Attack lab is released **today!!**



Agenda

- Logistics
- Stack review
- Attack lab overview
  - Phases 1-3: Buffer overflow attacks
  - Phases 4-5: ROP attacks

Logistics

- Use remote Linux Server to complete this assignment

```
ssh YourNetID@10.230.11.37 -p 4410
```

- Or use CentOS VM

  https://drive.google.com/file/d/1QLhvcIoK5nrkv40PnfHBb1ZKcPlNibBG

- Obtain Files from:

  http://DCLAP-V1111-CSD.ABUDHABI.NYU.EDU:15513/

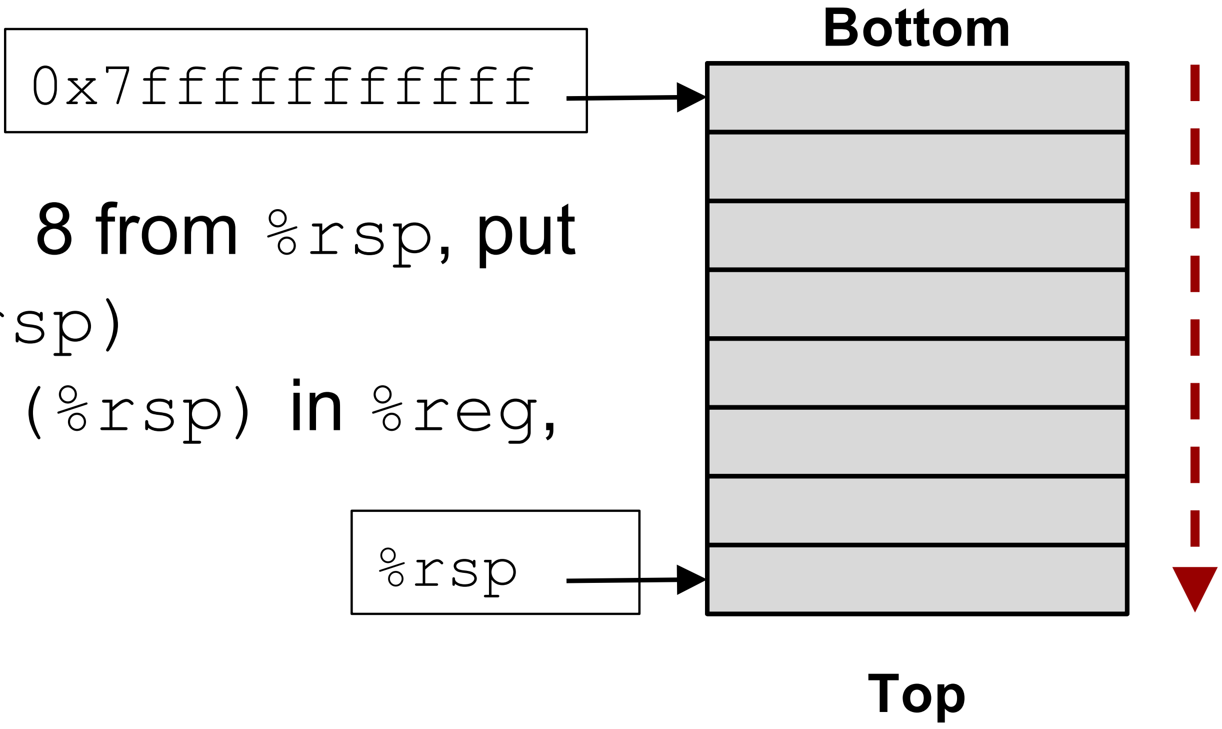
- Score Board:

  http://DCLAP-V1111-CSD.ABUDHABI.NYU.EDU:15513/scoreboard

- You need an active VPN connection to NYU/NYUAD network

x86-64: The Stack

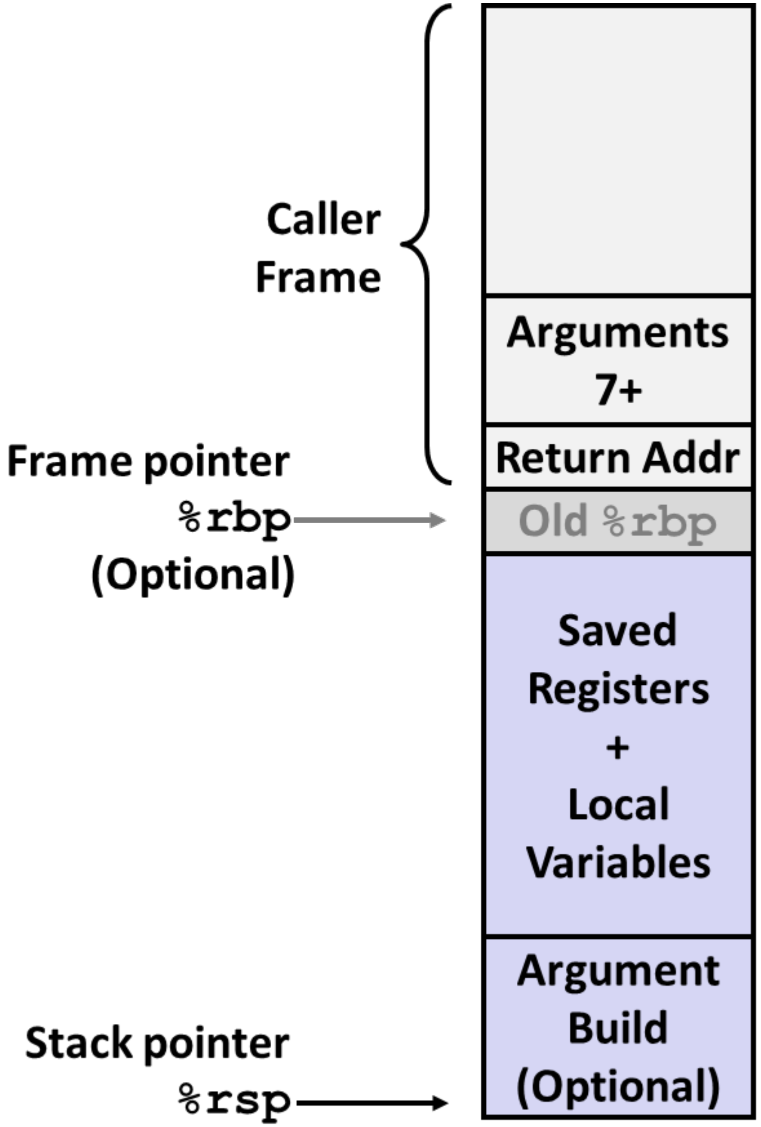
- Grows **downward** towards **lower** memory addresses
- `%rsp` points to **top** of stack



- `push %reg`: subtract 8 from `%rsp`, put val in `%reg` at `(%rsp)`
- `pop %reg`: put val at `(%rsp)` in `%reg`, add 8 to `%rsp`

x86-64: Stack Frames

- Every function call has its own **stack frame**.
- Think of a frame as a workspace for each call.
  - Local variables
  - Callee & Caller-saved registers
  - Optional arguments for a function call



x86-64: Register Conventions

- Arguments passed in registers:
  `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
- Return value: `%rax`
- Callee-saved: `%rbx, %r12, %r13, %r14, %rbp, %rsp`
- Caller-saved: `%rdi, %rsi, %rdx, %rcx, %r8, %r9, %rax, %r10, %r11`
- Stack pointer: `%rsp`
- Instruction pointer: `%rip`

x86-64: Function Call Setup

Caller:

- Allocates stack frame large enough for saved registers, optional arguments
- Save any caller-saved registers in frame
- Save any optional arguments (in **reverse order**) in frame
- `call foo`: push `%rip` to stack, jump to label `foo`

Callee:

- Push any callee-saved registers, decrease `%rsp` to make room for new frame

x86-64: Function Call Return

Callee:

- Increase `%rsp,` pop any callee-saved registers (in **reverse order**), execute `ret: pop %rip`

Attack Lab Overview: Phases 1-3

Overview

- Exploit x86-64 by overwriting the stack
- Overflow a buffer, overwrite return address
- Execute injected code

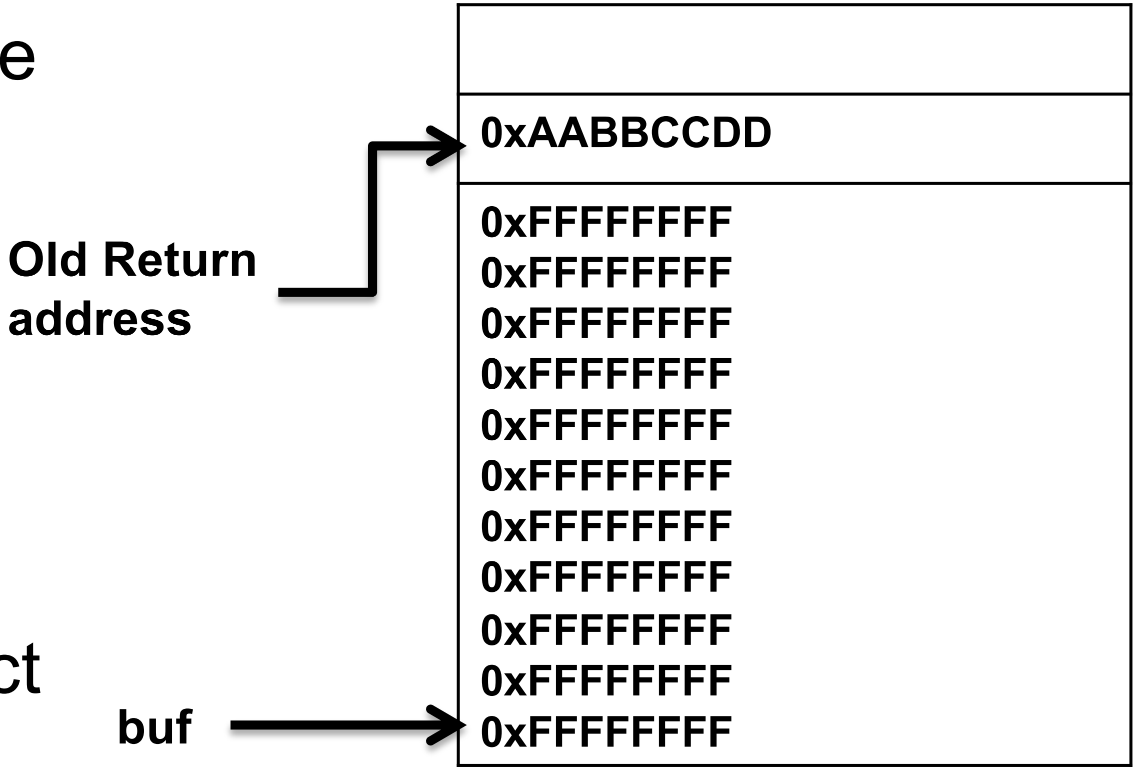
Key Advice

- Brush up on your x86-64 conventions!
- **Use objdump –d** to determine relevant offsets
- **Use GDB** to determine stack addresses

Buffer Overflows

- Exploit *strcpy vulnerability* to overwrite important info on stack
- When this function returns, where will it begin executing?
  - Recall

    `ret:pop %rip`
- What if we want to inject new code to execute?



Old Return address

0xAABBCCDD

0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF

buf

Attack Lab Overview: Phases 4-5

Overview

- Utilize return-oriented programming to execute arbitrary code
  - Useful when stack is non-executable or randomized
- Find gadgets, string together to form injected code

Key Advice

- Use mixture of pop & mov instructions + constants to perform specific task

ROP Example

- Use ROP exploit to **pop a value 0xBBBBBBBB into %rbx** and **move it into %rax**

```
void foo(char *input){
    char buf[32];
    ...
    strcpy (buf, input);
    return;
}
```

Gadgets:

address$_1$: mov %rbx, %rax; ret

address$_2$: pop %rbx; ret

Inspired by content created by Professor David Brumley

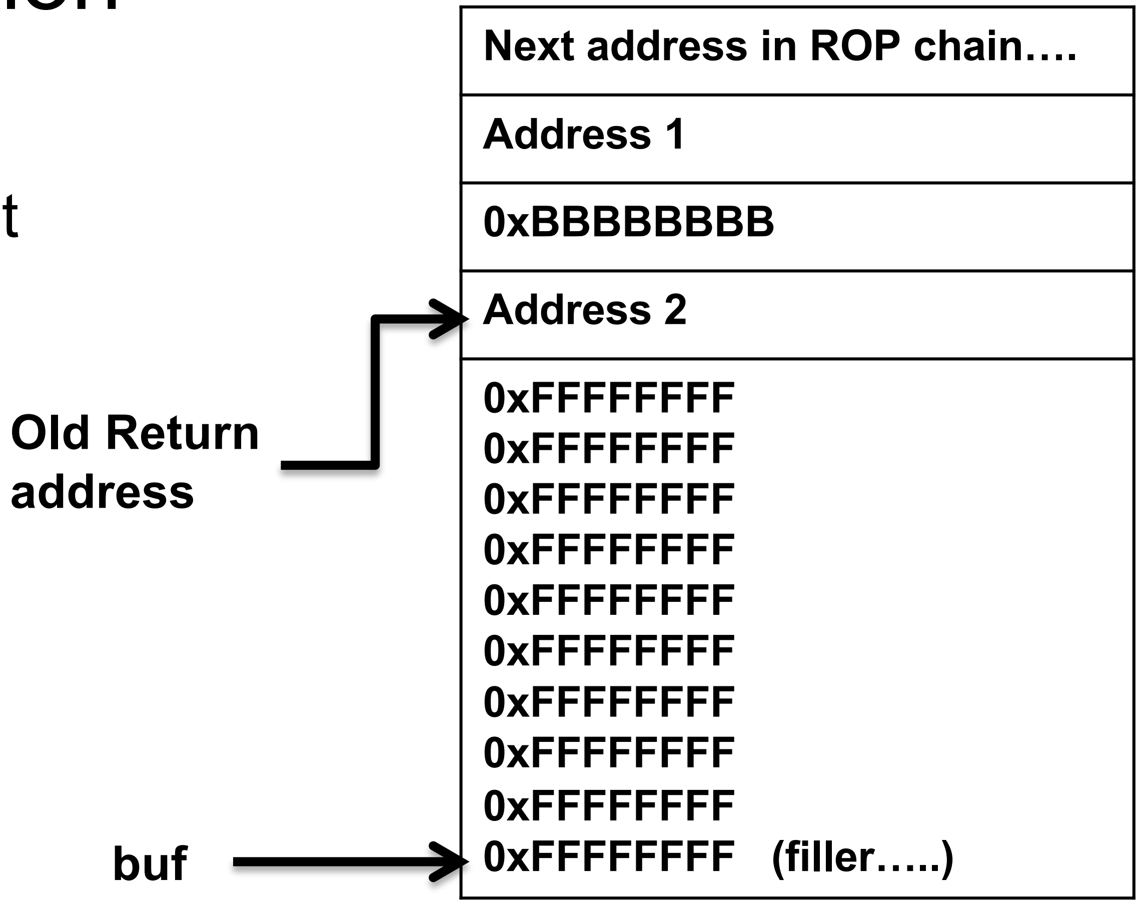
ROP Example: Solution

Gadgets:

Address 1: mov %rbx, %rax; ret

Address 2: pop %rbx; ret

```
void foo(char *input){
   char buf[32];
   ...
   strcpy (buf, input);
   return;
}
```



Next address in ROP chain….

Address 1

0xBBBBBBBB

Address 2

0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF
0xFFFFFFFF (filler…..)

ROP Demonstration: Looking for Gadgets

- How to identify useful gadgets in your code

Gadget Example #1

```
long ab_plus_c
  (long a, long b, long c)
{
   return a*b + c;
}
```

```
00000000004004d0 <ab_plus_c>:
  4004d0:   48 0f af fe   imul %rsi,%rdi
  4004d4:   48 8d 04 17   lea (%rdi,%rdx,1),%rax
  4004d8:   c3            retq
```

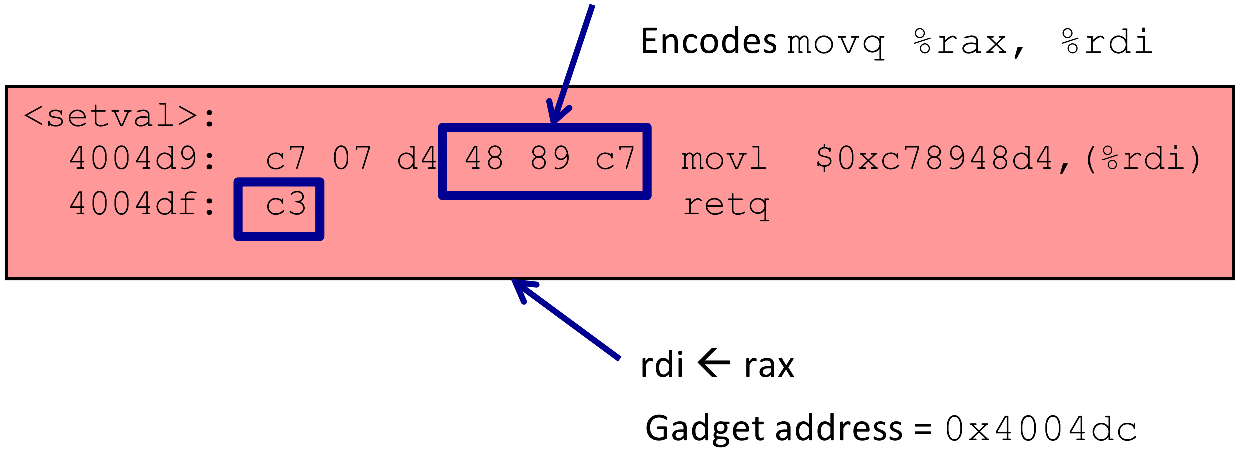
rax ← rdi + rdx

Gadget address = `0x4004d4`

- Use tail end of existing functions

Gadget Example #2

```
void setval(unsigned *p) {
    *p = 3347663060u;
}
```



Encodes movq %rax, %rdi

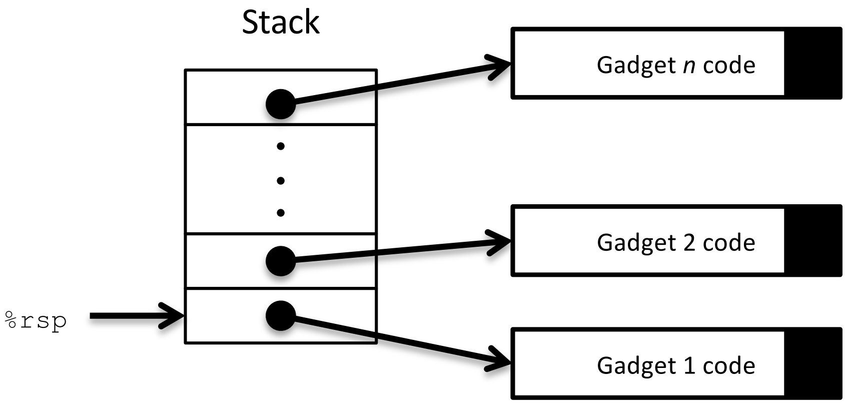
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7  movl  $0xc78948d4,(%rdi)
  4004df:  c3                 retq
```

rdi ← rax

Gadget address = 0x4004dc

- Repurpose byte codes

ROP Execution



- Trigger with `ret` instruction
  - Will start executing Gadget 1
- Final `ret` in each gadget will start next one

Tools

- **objdump –d**
  - View byte code and assembly instructions, determine stack offsets
- **./hex2raw**
  - Pass raw ASCII strings to targets
- **gdb**
  - Step through execution, determine stack addresses
- **gcc –c**
  - Generate object file from assembly language file

More Tips

- Draw stack diagrams
- Be careful of byte ordering (little endian)

Also...

Questions?