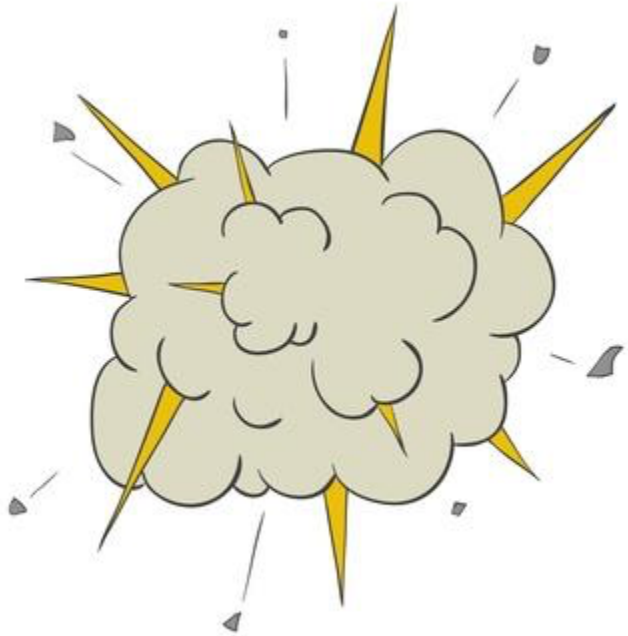
Reminder

- Bomb lab is due **tomorrow!**
- Attack lab is released **tomorrow!!**



Agenda

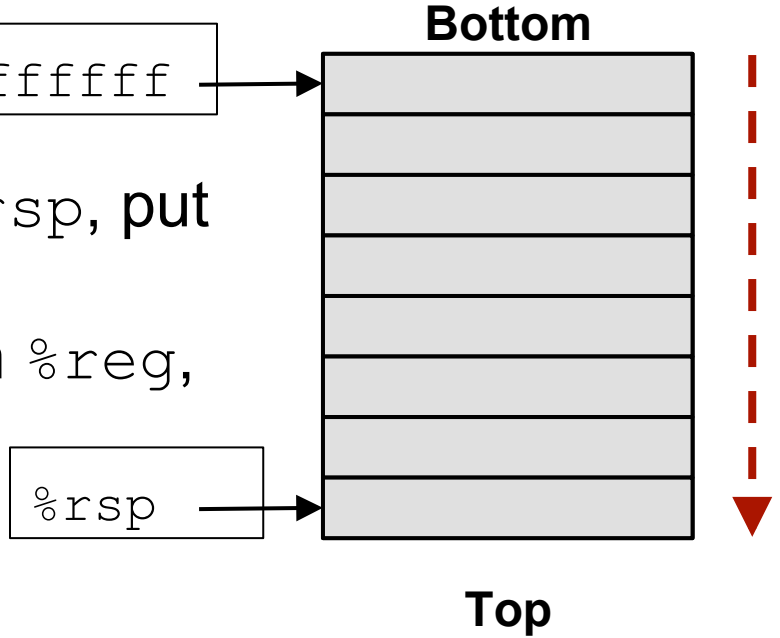
- Stack review
- Attack lab overview
  - Phases 1-3: Buffer overflow attacks
  - Phases 4-5: ROP attacks

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: The Stack

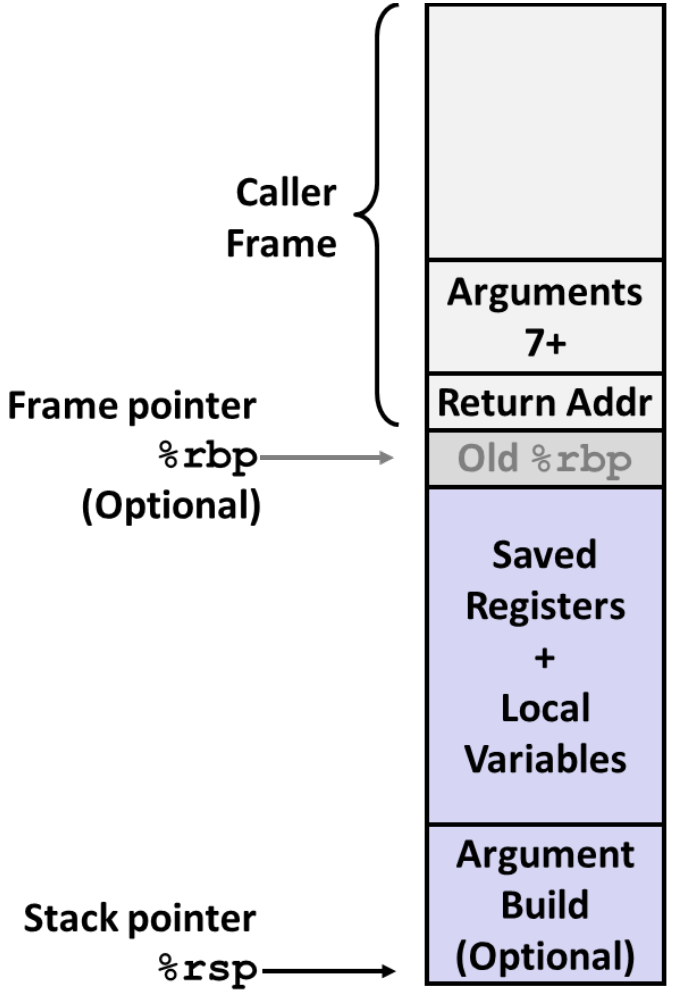
- Grows **downward** towards **lower** memory addresses
- `%rsp` points to **top** of stack



- `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: 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

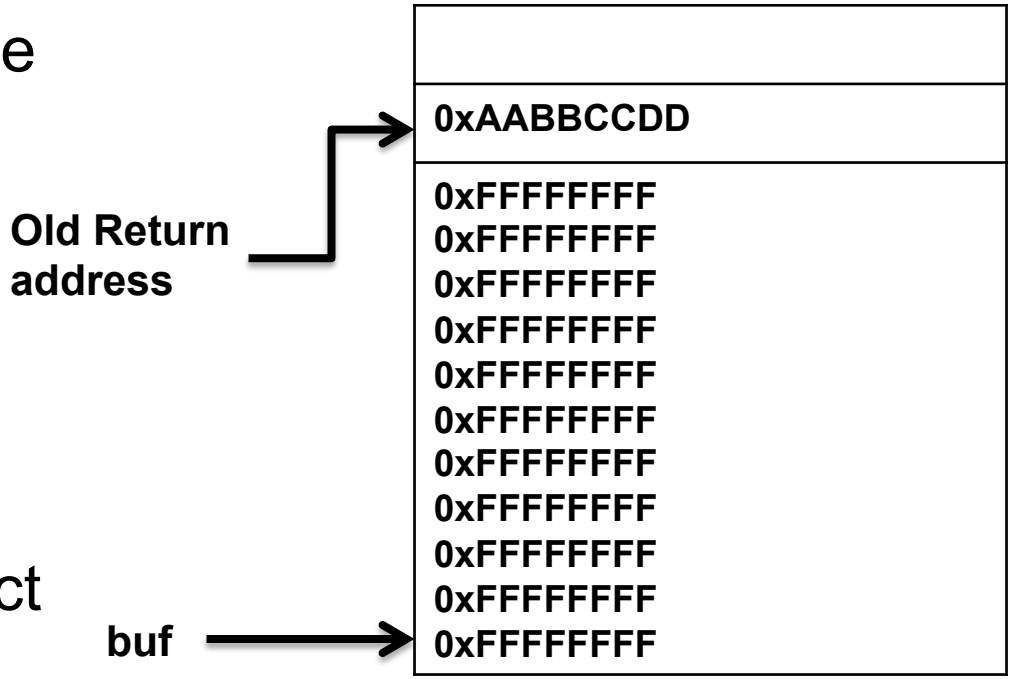
Key Advice

- 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?



Demonstration: Generating Byte Codes

- Use **gcc** and **objdump** to generate byte codes for assembly instruction sequences

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

- Draw a stack diagram and 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

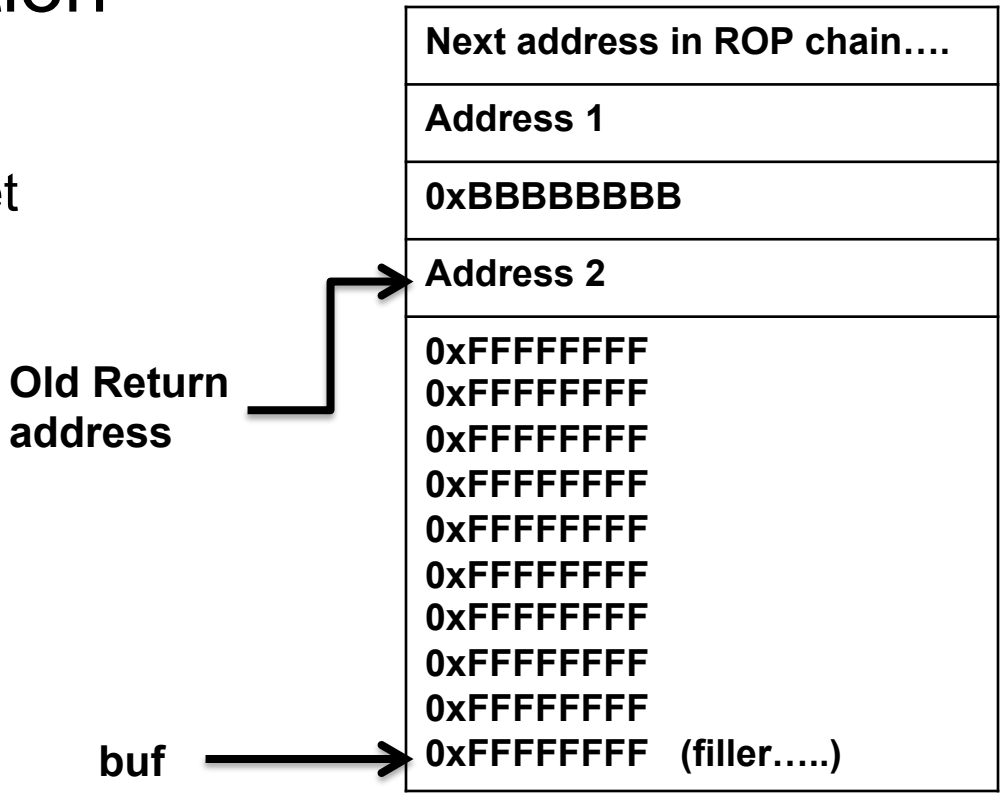
ROP Example: Solution

Gadgets:

Address 1: mov %rbx, %rax; ret

Address 2: pop %rbx; ret

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



ROP Demonstration: Looking for Gadgets

- How to identify useful gadgets in your code

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?