Attack Lab Workshop

by Jordan Lin and Siddharth Nandy

adapted from Julia Baylon and Jamie Liu’s workshop

sli.do code: #attack

Contents

- Workshop demo (follow along!)
- Stack review
- Return-oriented programming (ROP)
  - Finding gadgets
- Some other helpful links and tips :D

Workshop Demo

www.tinyurl.com/cs33-attacc

The demo essentially walks through a different version of phase 1, i.e., it is very helpful :)

"We protecc, we hacc, but most importantly, we attacc the stacc."

— Jamie Liu, 2019

The Attack Lab Workflow

1. Unzip your target folder and `make` your target.
2. Play around with your target—have some fun, why not?
3. Inspect the assembly code of with `objdump -d ./target > target.txt`.
4. Following the PDF instructions to see which function(s) you will have to get to with which specific conditions (e.g., values passed into said functions) for each phase and try your best to solve them.
5. Input your attacks for each phase into your target.

Inputting Attacks

1. Store your attacks, in hex, in some text file (e.g., `weapon_1.txt`). It is fine to include additional spaces and/or new-lines for clearer formatting (for you).

```
00 00 00 00 00 00 00 00  /* You can put comments too ... */
d4 fe 6d 00 00 00 00 00  /* ... just in this specific format! */
```

2. Run `./hex2raw < weapon_1.txt | ./target` (whatever your target name is).

With GDB

Instead of step 2, do the following.

1. Run `./hex2raw < weapon_1.txt > weapon_1` (convert from hex to raw file).
2. In GDB, do `run < weapon_1`. (Remember to set breakpoints!)

not this kind :(

The Stack—An Introduction

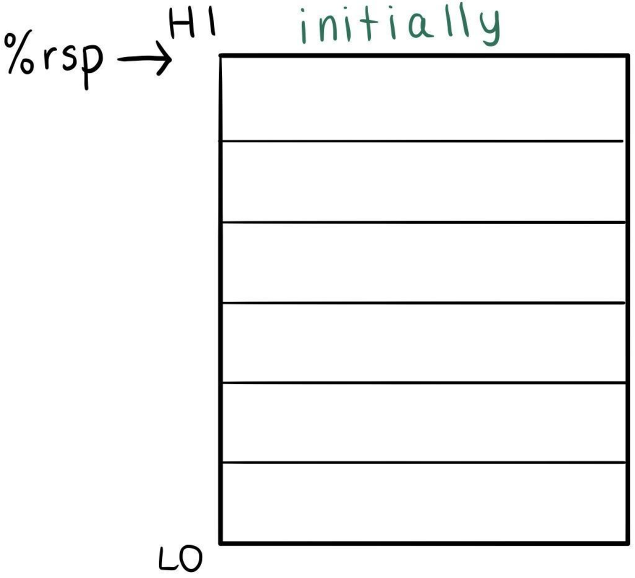
What Instructions Affect the Stack?

1. Adding to or subtracting from the stack pointer, `%rsp`
2. Pushing onto or popping from the stack
3. Calling and returning from functions

This is very important for us to know where our return address is relative to when when the input is written to the stack!

Adding To or Subtracting from the Stack Pointer

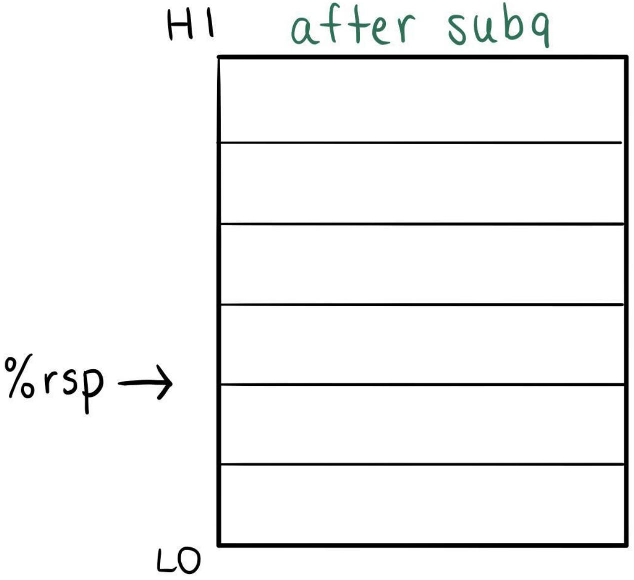
```
subq    $0x20, %rsp
addq    $0x10, %rsp
```



each line is 8 bytes

Adding To or Subtracting from the Stack Pointer

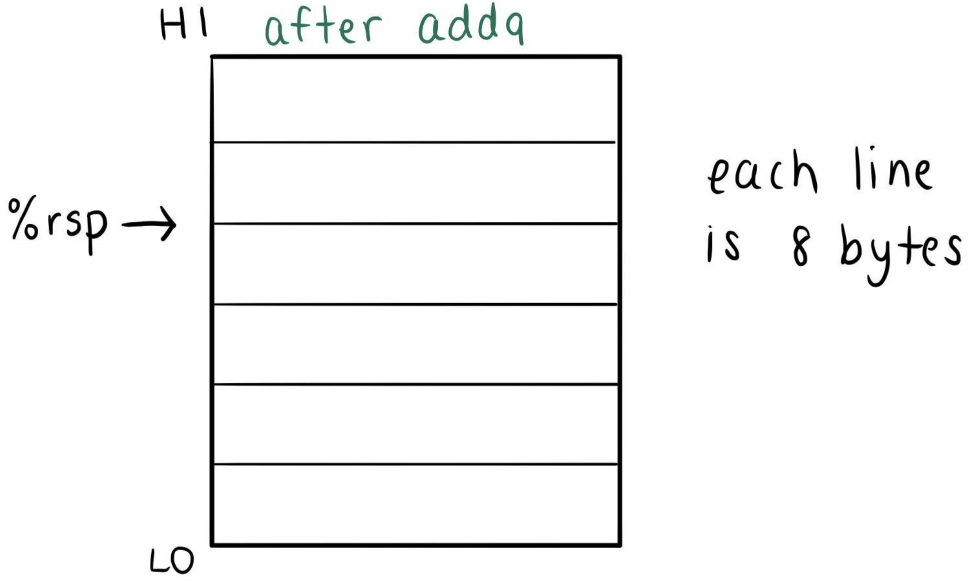
```
subq    $0x20, %rsp
addq    $0x10, %rsp
```



each line is 8 bytes

Adding To or Subtracting from the Stack Pointer

```
subq    $0x20, %rsp
addq    $0x10, %rsp
```



Pushing onto or Popping from the Stack

When we execute a **`push`** instruction, what really happens?

1. Allocate space on the stack for the data being pushed by subtracting from the stack pointer.
2. Move the data onto the stack.

So, we can think of the instruction

```
pushq %rax
```

as the following two instructions.

```
subq $0x8, %rsp
movq %rax, (%rsp)
```

When we execute a **`pop`** instruction, what really happens?

1. Move the data from the stack to the desired location,
2. Deallocate space on the stack by adding to the stack pointer.

So, we can think of the instruction

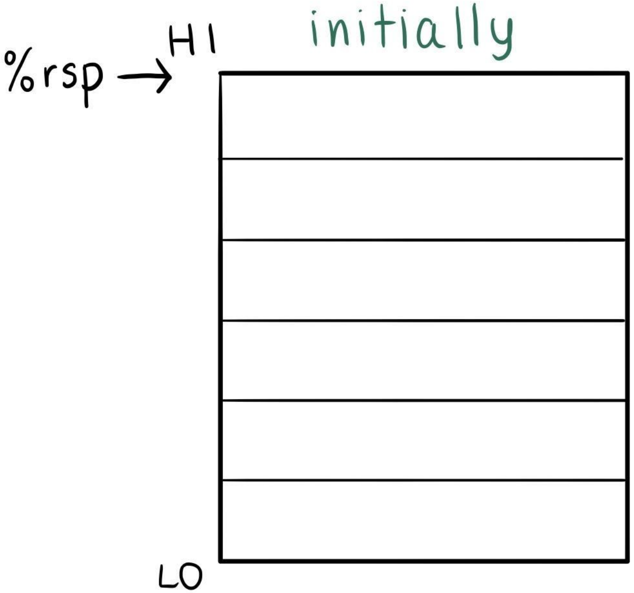
```
popq %rax
```

as the following two instructions

```
movq (%rsp), %rax
addq $0x8, %rsp
```

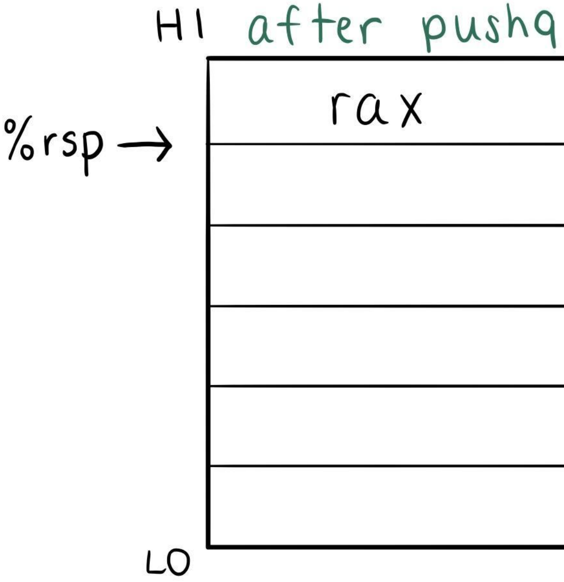
Pushing onto or Popping from the Stack

pushq %rax
popq %rax



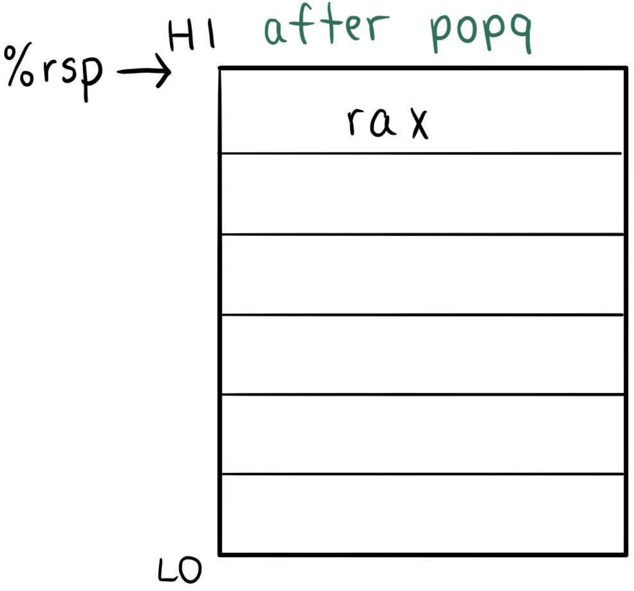
Pushing onto or Popping from the Stack

pushq %rax
popq %rax



Pushing onto or Popping from the Stack

pushq %rax
popq %rax



Calling and Returning From Functions

What happens to the stack when we **call** a function within another function?

- We push the address of the next instruction to be executed onto the stack

Note: We also move the address of the function we are calling into `%rip`, but this does not affect the stack.

What happens to the stack when we **return** from a function?

- We pop the top of the stack into `%rip`
    - Suppose `func2` is being called inside of `func1`.
    - When we called `func2`, we pushed the address of the next instruction to be executed in `func1` on the stack.
    - Now that we are resuming execution of `func1`, we move that instruction address into `%rip` to continue execution.

Calling and Returning From Functions

Consider the following disassembled functions:

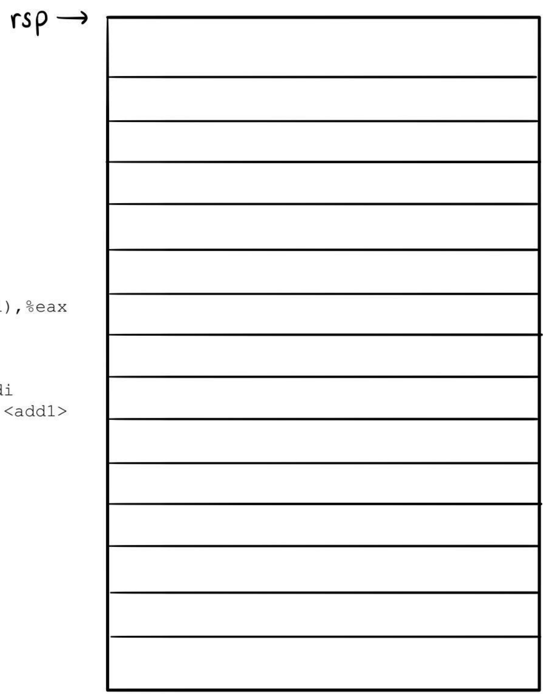
```
400492 <add1>:
  400492:   lea     0x1(%rdi),%eax
  400495:   retq

400496 <main>:
  400496:   mov     $0x1,%rdi
  40049b:   callq   400492 <add1>
  4004a0:   retq
```

What does the stack look like during execution?

Calling and Returning From Functions

suppose this is the state of the stack after executing the first instruction in main (400496)

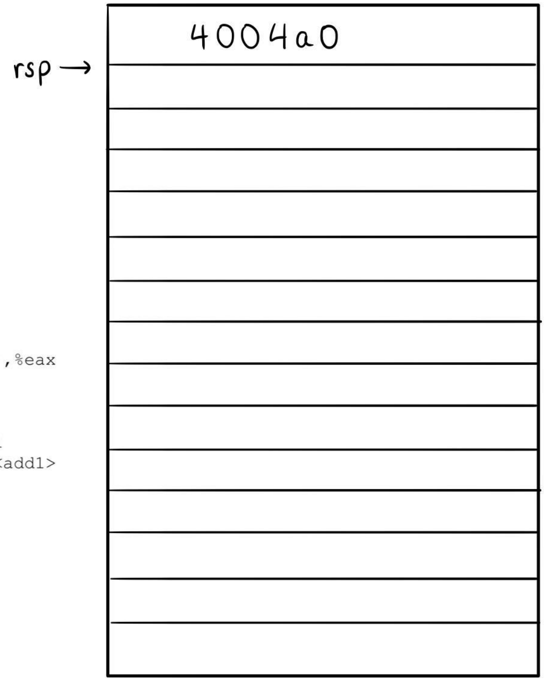
```
400492 <add1>:
  400492:    lea     0x1(%rdi),%eax
  400495:    retq

400496 <main>:
→ 400496:    mov     $0x1,%rdi
  40049b:    callq   400492 <add1>
  4004a0:    retq
```



Calling and Returning From Functions

when we call the add1 function, we push the address of the next instruction in main on the stack

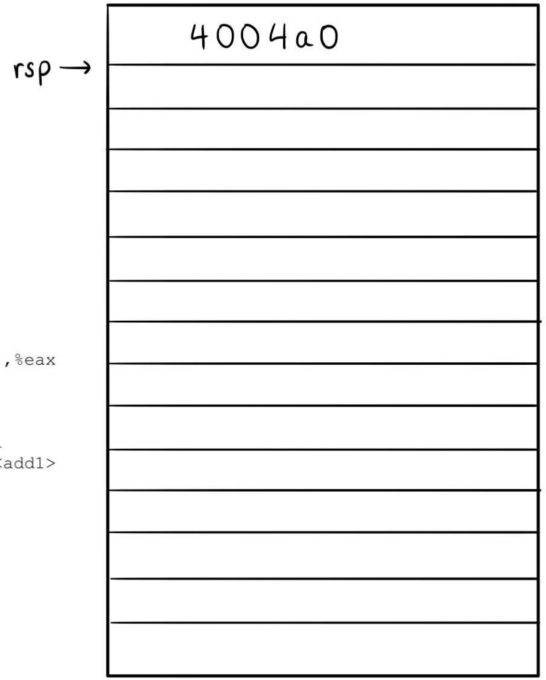
```
400492 <add1>:
  400492:    lea     0x1(%rdi),%eax
  400495:    retq

400496 <main>:
  400496:    mov     $0x1,%rdi
  40049b:    callq    400492 <add1>
  4004a0:    retq
```



Calling and Returning From Functions

now we are executing add1. the first instruction (lea) does not change the stack

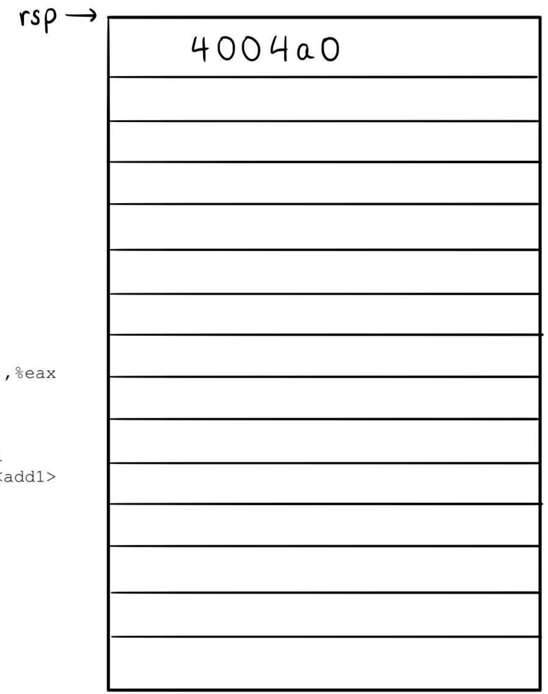
```
400492 <add1>:
→ 400492:    lea     0x1(%rdi),%eax
  400495:    retq

400496 <main>:
  400496:    mov     $0x1,%rdi
  40049b:    callq    400492 <add1>
  4004a0:    retq
```



Calling and Returning From Functions

now, we return
from add1 to
main, so we pop
the address we pushed
earlier into %rip.
this means we resume
execution at 4004a0

```
400492 <add1>:
  400492:    lea     0x1(%rdi),%eax
→ 400495:    retq

400496 <main>:
  400496:    mov     $0x1,%rdi
  40049b:    callq   400492 <add1>
  4004a0:    retq
```



Return-Oriented Programming

Otherwise known as **ROP**

Now, the stack memory address is randomized and non-executable—*what do we do* D:

Why ROP? Code Injection Was Just Fine :(

- Executable space protection: operating systems frequently combat buffer overflow bugs by making the memory that stores data as non-executable, so we can’t execute code that we’ve injected into the data area.
    - With ROP, we are executing code that the program already has (e.g., the gadget farm), just perhaps not the intended code as we point our return addresses to the middle of instructions :)
- Address space layout randomization (ASLR): operating systems randomize the location where executables are loaded into memory, so we cannot reliably determine where the injected code will be.
    - However, we can be fairly certain that the gadget farm will remain where it is :P

The Idea Behind ROP

The x86 instruction set is “dense”, meaning that any random sequence of bytes might be interpretable as a valid instruction.

- By searching through the existing program, we can find operation bytecodes to alter control flow (e.g., `ret == 0xc3`)
- The preceding bytes may represent useful instructions (e.g., a move between registers).

Together, these bytes are called a **gadget**, and we can chain together multiple gadgets such that the `ret` instruction of each one jumps to the next by injecting multiple return addresses on the stack that point to said gadgets.

The ROP Workflow

1. Determine your goal (e.g., move `0x69` to `%rdi`).
2. Find gadgets that, when chained together, accomplish that goal (e.g., `pop`, register moves, etc.).
3. Chain together said gadgets with injected return addresses.
4. Profit.

Finding A Gadget

Suppose we decide that we want to execute: `popq %rax`, which has the encoding `0x58`. To find a gadget that does this, we can look for the bytes `0x58` and `0xc3` in our gadget farm—recall that `0xc3` corresponds to `ret`.

Suppose we found the following instructions in the gadget farm.

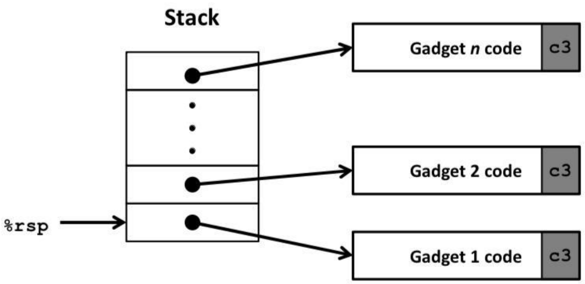
```
40049e:   b8  00  d8  91  58        mov   $0x5891d800, %eax

4004a3:   c3                        ret
```

We can take the last byte in the `mov` instruction along with the following `ret` instruction. Our gadget starts with byte 5 of the `mov` instruction, so to get the address we start with `0x4009e` and go 5 – 1 = 4 bytes past that to get that our gadget starts at address `0x4004a2`.

Takeaway: You might not find the exact instruction that you need in the gadget farm, the idea is to take parts of instructions—or chain multiple of them together—to execute what you want

Chaining Gadgets Together

ROP Execution



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

Imagine we are executing gadget 1.

- `%rsp` points to the address of gadget 2.

When we return in gadget 1, what happens?

- We will pop `(%rsp)` into `%rip` and continue executing
- Since `(%rsp)` stored the address for gadget 2, we now execute gadget 2.

… and so on until we have executed all the gadgets that we have chained together.

Helpful Links and Tips

- `objdump`—basically required :P
- Understand what `gets()` does, i.e., how it puts your input onto the stack.
- The ASCII table, especially when you have to construct a string from hex.
- String byte ordering—endianness is important!
- Understand how the stack changes after certain instructions are executed
    - It may be helpful to visualize what your stack looks like while executing `ctarget`.
- It will be **very** helpful to read the lab specs as you go through each phase.