# Function Pointers, Vulnerabilities

## CS 2130: Computer Systems and Organization 1

**Xinyao Yi** Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

## Announcements

- Homework 10 due Monday
- Final exam: 7-10pm April 30, Gilmer 301 (different room!)
  - Cumulative, see practice tests
  - Exam conflict form in email
- Remember to fill out course evaluations

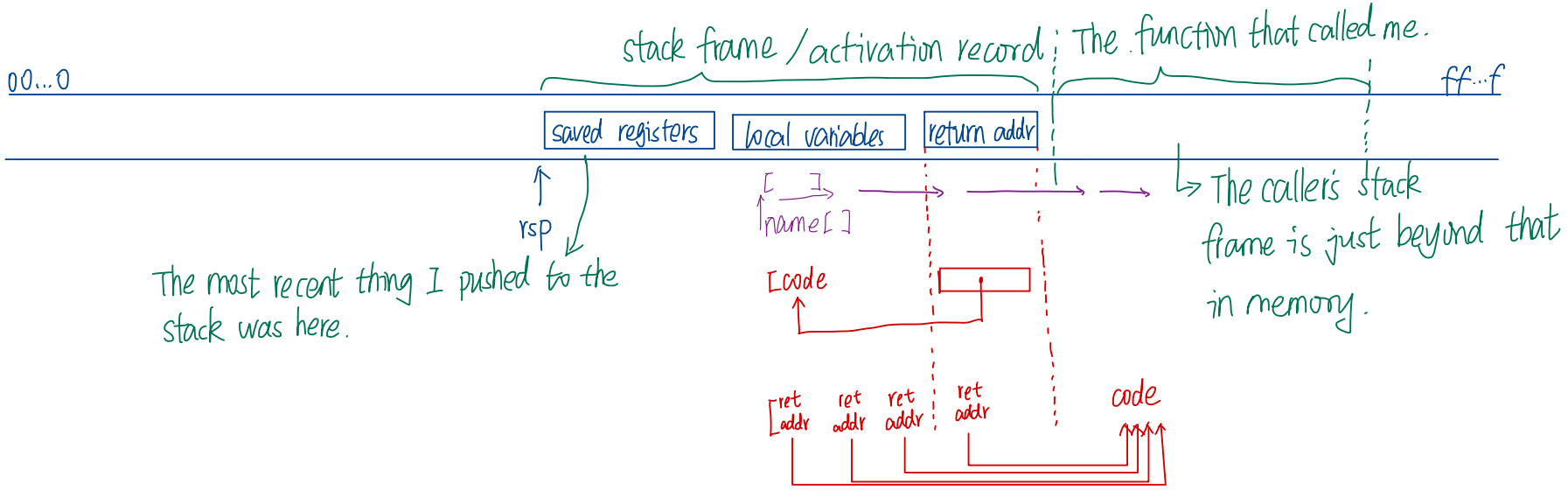
Vulnerabilities...
...and when to report them

# Memory

Common Memory Problems (from reading)

- Memory leak
- Uninitialized memory
- Accidental cast-to-pointer
- Wrong use of ‘sizeof‘
- Unary operator precedence mistakes
- Use after free
- Stack buffer overflow
- Heap buffer overflow
- Global buffer overflow
- Use after return
- Uninitialized pointer
- Use after scope

# Vulnerabilities



# Vulnerabilities

**Vulnerability**: a program for which something like this could happen (security holes)

- Ex: stack buffer overflow possibility
- Not necessarily malicious (like when we talked about backdoors)

**Exploit**: a way to use a vulnerability or backdoor that has been created

- Ex: the magic long word to type into our program

**1. What happens if a program asks for your name but doesn't check the length?**

Imagine the program allocates:

```
char name[20];
```

But then it reads **100 bytes** into it.

Where do the extra 80 bytes go?

They overwrite whatever comes next:

- other local variables
- saved registers
- the return address
- the caller's frame

This is the fundamental danger of buffer overflows."

**2. Overwriting the return address**

"The return address is just a number — it is the address the CPU jumps to when the function returns.

If an attacker can overwrite the return address, the attacker can choose **where the program jumps next**."

**3. Classic attack: Inject code + jump to it**

"This is the traditional stack–based exploit:

1. Write malicious machine instructions into the buffer (as input).
2. Keep writing until you reach the return address.
3. Overwrite the return address with the starting address of that buffer.

When the function returns, it jumps directly into the attacker's code.

This is how early exploitation worked:
**'put code on the stack, then return into it.'**"

**4. Problem: How do attackers know exactly how far to overwrite?**

"You might not know the exact number of bytes between the buffer and the return address.

Attackers solved this by writing *tons* of return addresses, over and over, so eventually one of them lands exactly on the real return address.

Because of 8-byte stack alignment, eventually one of those overwrite attempts matches the correct alignment."

**5. Compiler defense: Stack Canary**

“To defend against overwriting the return address, compilers insert a random value — a ‘canary’ — right before the return address.

Memory layout becomes:

```
[ local variables ]
[ random canary ]
[ return address ]
```

Before the function returns, the compiler-generated prologue checks that the canary is unchanged.

- If it changed → abort with ‘stack smashing detected’.
- If it’s intact → normal return.

To exploit the overflow, the attacker must guess the canary — extremely difficult.”

# Vulnerabilities

**Modern systems prevent executing code on the stack**

"This classic attack doesn't usually work today.

Why?

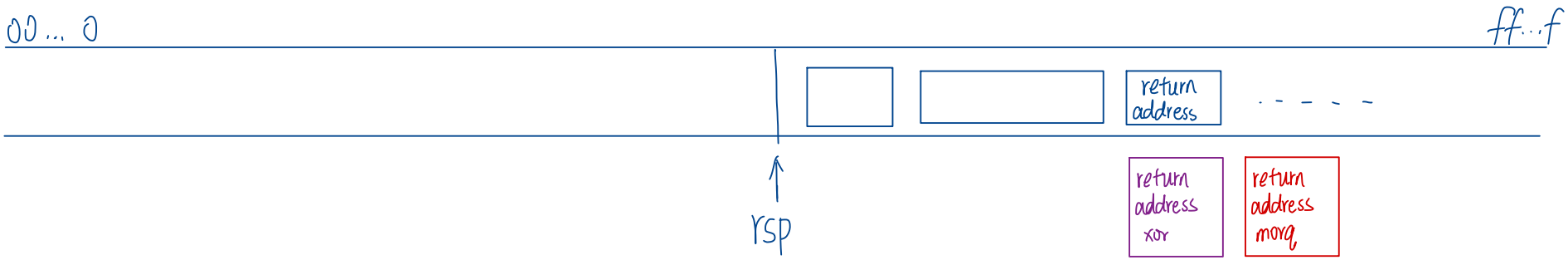
Because modern operating systems mark the stack as:

- readable
- writable
- **not executable**

This is the NX bit (No-eXecute).

If you try to execute instructions stored on the stack:
→ immediate segmentation fault."

# Vulnerabilities




```
xorl %eax, %eax
movq %rdi, %rsi
      ⋮
(The code "I" want to do)
```

```
xorl %eax, %eax
ret
(functions pre-compiled)
movq %rdi, %rsi
ret
```

# Vulnerabilities

**The modern workaround: ROP (Return-Oriented Programming)**

"So if we cannot execute our own code, we must use code that **already exists**.

This leads to Return-Oriented Programming (ROP).

The idea is:

- The program and libc already contain thousands of tiny instruction sequences ending in `ret`.
- Each small sequence is called a *gadget*.
- If you can control the return address, you can chain these gadgets together:
  - first return to a gadget that clears a register
  - then return to a gadget that loads a value
  - then return to a gadget that calls a function

By chaining dozens of gadgets, you can build an arbitrary computation **without injecting any code**.

You only manipulate return addresses."

**OS defense: ASLR (Address Space Layout Randomization)**

"To defeat ROP, modern operating systems randomize memory layout:

- the stack address
- the heap address
- the location of shared libraries
- the base address of the program

This means attackers cannot predict where gadgets are located.

Without knowing the address of libc functions or gadgets, building a ROP chain becomes much harder."

## Warning

Anytime you can modify memory the programmer did not expect you to be able to modify, there's something you can do to give yourself power or rights the programmer didn't mean to give you

# Memory

Common Memory Problems (from reading)

- Memory leak
- Uninitialized memory
- Accidental cast-to-pointer
- Wrong use of ‘sizeof‘
- Unary operator precedence mistakes
- Use after free
- Stack buffer overflow
- Heap buffer overflow
- Global buffer overflow
- Use after return
- Uninitialized pointer
- Use after scope

# Vulnerabilities

What should you do when you find a vulnerability?

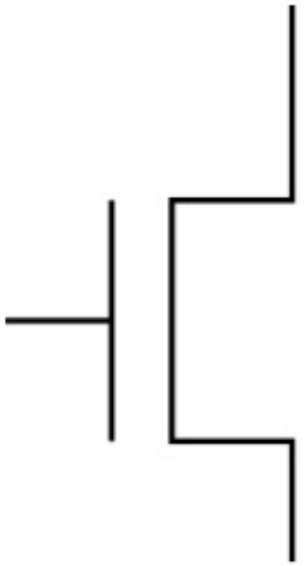
# Good Practices

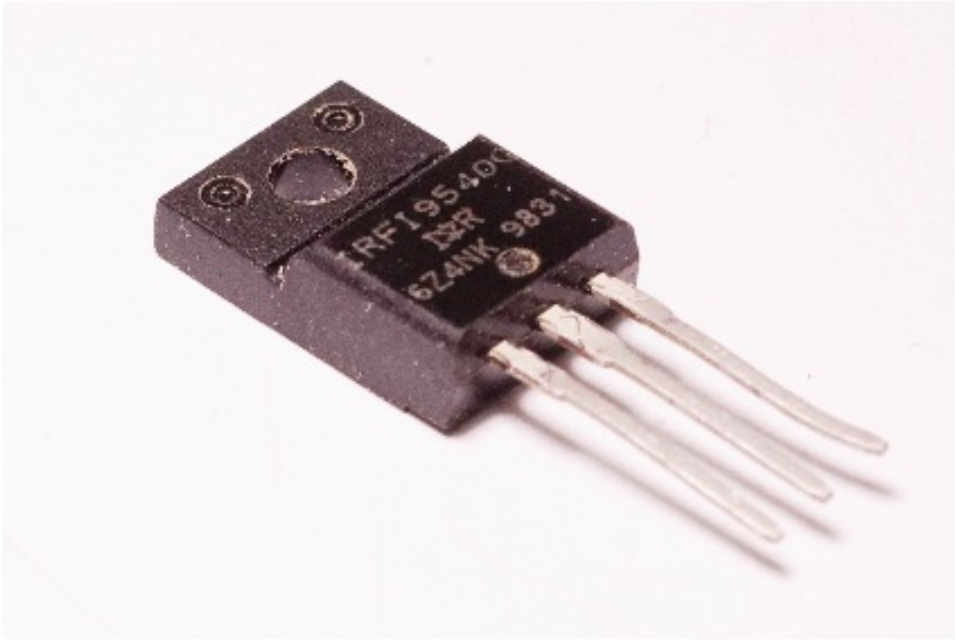
Good practices when finding a vulnerability:

1. Tell the owner
2. Wait (a reasonable amount of time for a fix)
3. Publish

# Where have we been?

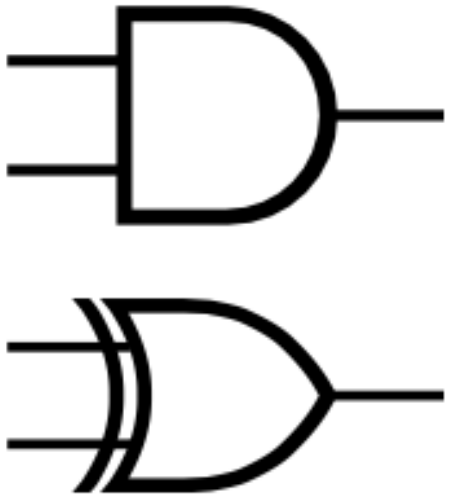
# Where are we going?



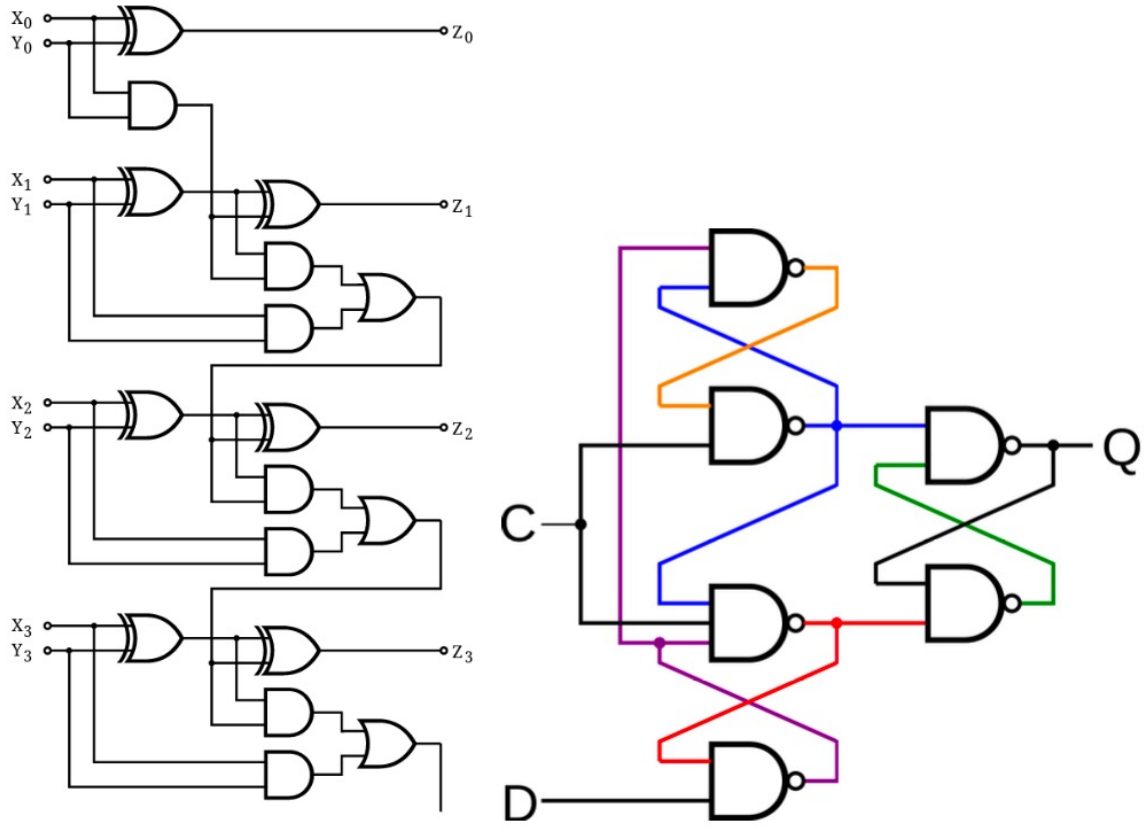



0 and 1

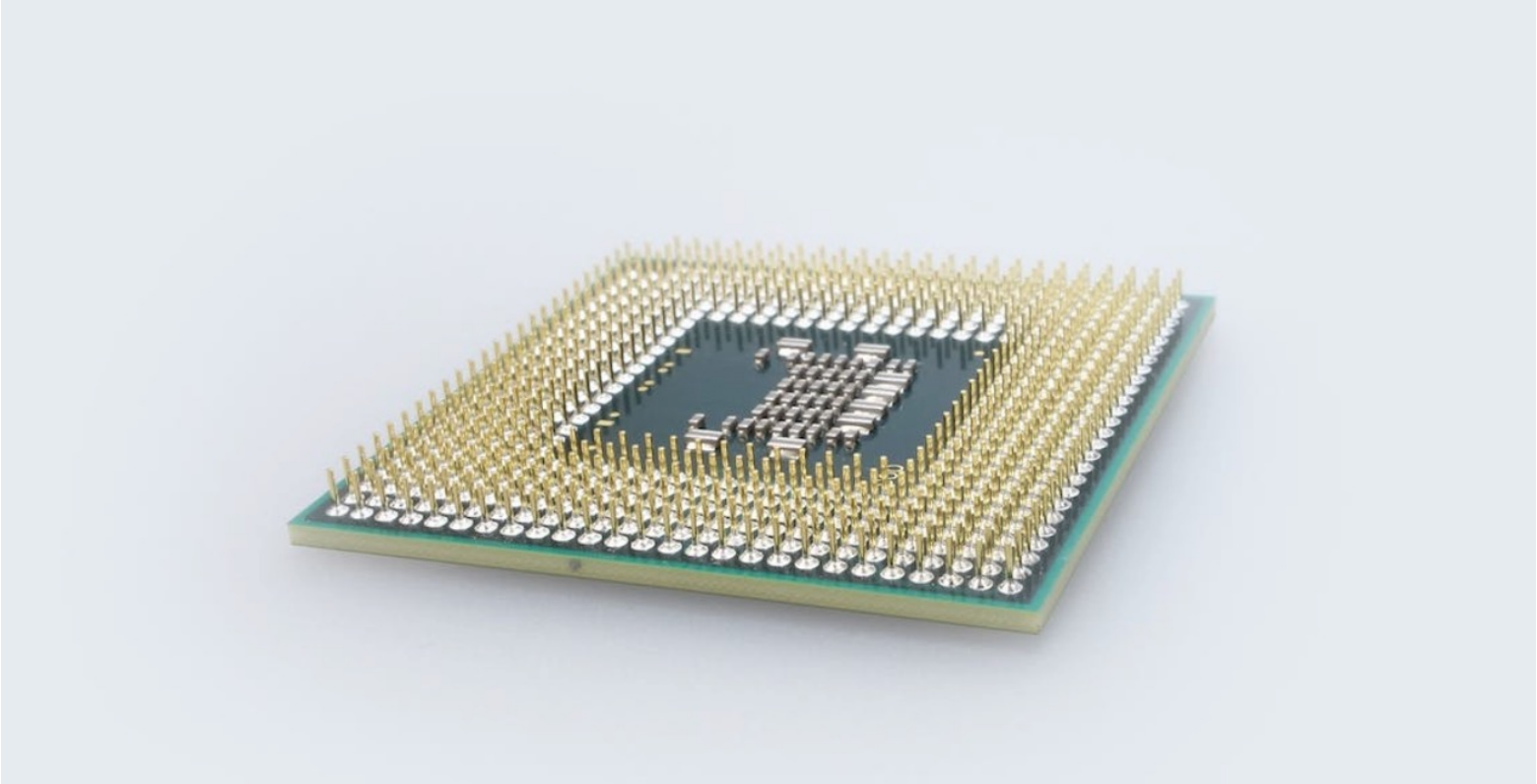
# Where are we going?



# Where are we going?



# Where are we going?



# Where are we going?

```
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   31 c0                   xor    %eax,%eax
   6:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
   d:   c7 45 f8 03 00 00 00    movl   $0x3,-0x8(%rbp)
  14:   48 c7 45 f0 04 00 00    movq   $0x4,-0x10(%rbp)
  1b:   00
  1c:   48 8d 4d f8             lea    -0x8(%rbp),%rcx
  20:   48 89 4d e8             mov    %rcx,-0x18(%rbp)
  24:   48 8d 4d f0             lea    -0x10(%rbp),%rcx
  28:   48 89 4d e0             mov    %rcx,-0x20(%rbp)
  2c:   48 8b 4d e8             mov    -0x18(%rbp),%rcx
  30:   48 63 09                movslq (%rcx),%rcx
  33:   48 89 4d d8             mov    %rcx,-0x28(%rbp)
  37:   48 8b 4d e0             mov    -0x20(%rbp),%rcx
  3b:   48 8b 09                mov    (%rcx),%rcx
  3e:   89 4d d4                mov    %ecx,-0x2c(%rbp)
  41:   5d                      pop    %rbp
  42:   c3                      retq
```

# Where are we going?

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

## Where are we going?

Along the way:

- Interact with the terminal and SSH
- Learn basic command-line tools and editors
- Access command-line documentation
- Practice C and using the C standard library
- Learn how to debug with lldb and the address sanitizer
- Discuss related security and social topics
- Think about the next steps of Generative AI

## Finale

Along the way:

- Interact with the terminal and SSH
- We have covered a LOT
- Electricity on wires
- Transistors to gates (AND, OR, ...)
- Combined gates to make circuits
- Connected circuits and registers to build a 1-byte computer
- Wrote an ISA for that computer (1-byte instructions, Toy ISA)
- Expanded to x86-64 Assembly (saw the binary)
- Concluded with C (how it compiles and connects with Assembly)

## Finale

Thanks for a great semester!