An Introduction to Reverse Engineering

Stuart Nevans Locke

Whoami - Stuart Nevans Locke

- Low level enthusiast
  - Binary Exploitation
  - Processor Architecture
  - Reversing
- Was on coop doing Vulnerability Research last summer
- Hobbies
  - Chess

Overview

- What is Reversing Engineering?
- What is the point of Reverse Engineering?
- An Introduction to Low Level and x86 Assembly

If you have questions at any point, feel free to interrupt me.

What is Reverse Engineering?

Reverse engineering is the process of discovering the technological principles of a device, object, or system through analysis of its structure, function, and operation

What is Reverse Engineering?

- Figuring out how something accomplishes what its purpose
- We’re going to focus on Software Reverse Engineering
    - Given a program, figure out what it does and how it does what it does

What is the purpose of Reversing?

- Depends on who you talk to
    - It can be fun!
    - You can find bugs
    - You just want to understand a product
    - Maybe you want to change the behaviour of something (anything from fixing a bug to adding functionality to totally changing the product)
    - If you want to replicate something and there is no information on how to do that, you might need to reverse engineer it

Intro to Low Level - Disclaimers

- The goal of this is to teach a bit about x86/x86-64 programming, so some might be specific to that
- Some of this might be oversimplified so we don’t get bogged down in irrelevant details

Intro to Low Level - Storage

What types of storage can you name?

Intro to Low Level - Storage

- Disk
  - Very, very slow
  - Huge
- Memory
  - Slow (quick compared to disk!)
  - Large
- Cache
  - Fast
  - Small
- Registers
  - Tiny
  - Very Few of
  - Insanely Fast

Intro to Low Level - Variables

When programming, you might do:
int x = 5;

That variable, **x**, would be stored in a register or in memory.

Typically variables that are only used briefly aren’t stored in memory.

Typically variables are loaded into registers before manipulating them much.

Intro to Low Level - Memory

- Semi-permanent storage medium
- Referred to by address
    - #0, #1, #ff, #1234 (all hex)
    - Allows you to store a variable amount of data
    - Can give greater flexibility
    - Addresses are just integers and can thus be added to or subtracted from
- Used to store variables that live for a long time

Intro to Low Level - Registers

- So few registers that they all have names
- Live on the CPU
- Used to manipulate data
- Fixed size (64 bit on x86-64, 32 bit on x86)

Intro to Low Level - Register Types

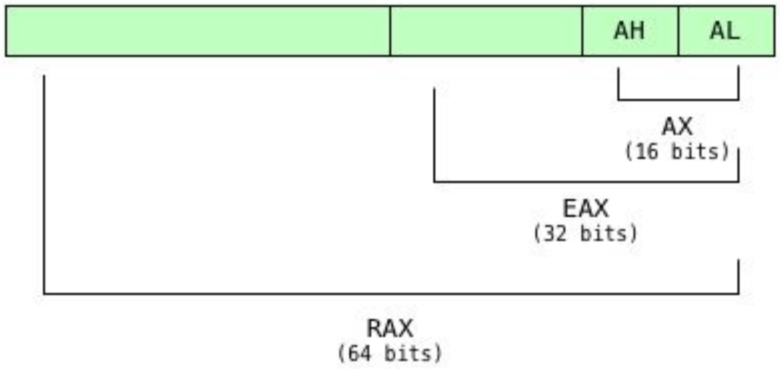
- General Purpose
  - Used for general data manipulation, storing integers and pointers
  - eax, ebx, ecx
- Instruction Pointer
  - Contains an address pointing to the next instruction to be executed
- EFLAGS
  - Contain information about temporary state
- Floating Point
  - Store floating point numbers

Intro to Low Level - Register Types

- Stack Register
  - Going to talk about this more later
  - Tracks the top of the stack
- Many more!

Intro to Low Level - General Purpose Registers (GPRs)

- Most instructions operate on GPRs
- Used for doing math, moving data
- Can access variable sizes of



Intro to Low Level - Instruction Pointer

- You might see it called program counter (pc)
- On x86, instruction pointers is rip/eip/ip
- Tells the CPU what instruction to executing next
- After executing an instruction, the CPU adds the size of it to the instruction pointer so that it points to the next instruction
- Assembly programs just run linearly top to bottom unless there are any jumps or calls

Intro to Low Level - Dereferencing

- Registers can contain addresses (remember address are just integers)
- To get the data at a given address, you dereference the address
- RAX = [RBX]   rax = *rbx
- RAX = [RBX+4]   rax = *(rbx+4)

X86 - Instructions

- Instructions perform operations on data
  - Math, moving data, ...
- More or less tiny functions
- They are why programs actually do anything

X86 - Instructions

- I’m going to use intel syntax
- Intel syntax uses destination-source syntax
- OPERATION DST, SRC
    - Basically means perform OPERATION to SRC and store the result in DST

X86 - mov

Mov is one of the most common instructions. Mov moves (copies) the second (source) operand to the first (destination) operand.

MOV A, B - copies B to A

Used to: Copy data, Store data to memory, get data from memory, load constant value

X86 - mov

mov eax, ebx

mov eax, 5

mov edx, [ecx] -- lea edx, [ecx*4+4] -- mov edx, ecx*4+4

mov [ebx], ecx

X86 - add

Adds the second (source) operand to the first (destination) operand.

ADD A,B   - adds b to a

X86 - add

```
add eax, 5

add eax, ebx
```

X86 - mul

Does an unsigned multiplication of the destination operand with the source operand. Mul has an implied operand, meaning the destination operand is always assumed to be rax/eax/ax. Sometimes stores data in rdx/edx/dx if the multiplication is too large.

MUL EBX -- Multiplies EAX by EBX

X86 - mul

mul 5

mul rax

Useful tools/reference

https://carlosrafaelgn.com.br/asm86/ - x86 emulator

https://www.felixcloutier.com/x86/ - x86 reference

Gdb - debugger. I'll be talking about this more later.

Nasm - assembler. Allows you to assemble code to run it.

Questions?