Reverse Engineering: An Overview

What is Reverse Engineering?

- The process of analyzing software to identify its components and their relationships
- Understanding the software’s design and implementation
- Commonly used in cybersecurity to analyze malware and identify vulnerabilities

Why Reverse Engineering is Important

- Analyzing malware to understand its behavior and develop countermeasures
- Identifying vulnerabilities in software to improve security
- Recovering lost source code for legacy systems

Tools for Reverse Engineering

- **Ghidra:** A software reverse engineering suite developed by the NSA
- **IDA Pro:** Interactive Disassembler, a popular tool for static analysis
- **Radare2:** An open-source framework for reverse engineering and analyzing binaries
- **Binary Ninja:** A reverse engineering platform with a focus on ease of use

Introduction to Assembly Language

High-Level vs. Low-Level Languages

- **High-Level Languages:** Python, Java, C++
    - Easier for humans to read and write
    - Compiled/interpreted into machine code
- **Low-Level Languages:** Assembly (x86, ARM)
    - Closer to machine code
    - Directly interacts with hardware

Program Build Flow

- **Source Code:** Written in C/C++
- **Compilation:** Converts to Object Code (.o files)
- **Linking:** Combines Object Code into Executable
- **Disassembly:** Converts Executable back to Assembly for analysis

x86 Registers Overview

General Purpose Registers

- **EAX:** Accumulator for operands/results
- **EBX:** Base pointer to data
- **ECX:** Counter for loops/strings
- **EDX:** I/O pointer
- **ESI/EDI:** Source/Destination for strings
- **ESP:** Stack Pointer
- **EBP:** Stack Base Pointer
- **EIP:** Instruction Pointer (next instruction to execute)

Flags and Segment Registers

- **Flags:**
  - **OF:** Overflow Flag
  - **SF:** Sign Flag
  - **ZF:** Zero Flag
- **Segment Registers:**
  - **CS:** Code Segment
  - **DS:** Data Segment
  - **SS:** Stack Segment
  - **ES/FS/GS:** Extra Segments

Basic Instructions

Data Transfer Instructions

- **mov:** Transfer data
  - `mov destination, source`
- **lea:** Load effective address
  - `lea destination, [source]`

Control Transfer Instructions

- **jmp:** Jump to address
  - `jmp address`
- **call:** Call procedure
  - `call address`
- **ret:** Return from procedure
  - `ret`

Arithmetic Instructions

- **add:** Addition
  - `add destination, value`
- **sub:** Subtraction
  - `sub destination, value`
- **mul:** Multiplication
  - `mul source`
- **div:** Division
  - `div source`

Addressing Modes

Immediate Addressing

- Operand is a constant value
  - Example: `mov eax, 5`

Register Addressing

- Operand is a register
  - Example: `mov eax, ebx`

Memory Addressing

- Operand is an address in memory
  - Example: `mov eax, [ebx]`

The Stack

Overview

- **LIFO Data Structure:** Last In, First Out
- **Push/Pop:**
    - `push value`: Save data on the stack
    - `pop destination`: Retrieve data from the stack

Common Stack Instructions

- **pusha:** Push all general-purpose registers
- **pushad:** Push all 32-bit registers
- **popa:** Pop all general-purpose registers
- **popad:** Pop all 32-bit registers