Chapter 2: Beyond Basic Static Analysis-x86 Disassembly

# DATA SCIENCE IN SECURITY

# Introduction

- To thoroughly understand a malicious program
  - we often need to go beyond basic static analysis
  - involves reverse engineering a program's assembly code
  - Indeed
    - disassembly and reverse engineering lie at the heart of deep static analysis of malware samples

Goal here is to introduce you engineering to apply it to malware data science

# Disassembly Methods

- Disassembly
  - the process of translating malware’s binary code into valid assembly language.
  - is no easy feat
    - malware authors employ tricks to thwart reverse engineers.
  - Perfect disassembly in the face of deliberate obfuscation is an unsolved problem
    - E.G. self-modifying code

# Disassembly Methods

- Disassembly
  - we must use imperfect methods
    - linear disassembly:
      - involves
        - identifying the contiguous sequence of bytes in the PE file
        - then decoding these bytes.
    - The key limitations :
      - it ignores subtleties about how instructions are decoded by the CPU
      - it doesn’t account for the various obfuscations

# Disassembly Methods

- Disassembly
  - There are other methods
    - we won't cover here
    - used by disassemblers such as IDA Pro.
    - simulate or reason about program execution
    - discover which instructions might reach as a result of a series of conditional branches
    - can be more accurate than linear disassembly
    - it's far more CPU intensive than linear disassembly
    - less suitable for data science purposes
      - disassembling thousands or even millions of programs.

# Basics of x86 assembly language

- lowest-level human-readable programming language
- maps closely to the binary instruction format of CPU.
- A line is almost always equivalent to a single CPU instruction
- reading disassembled malware x86 code is easier than you might think

# Basics of x86 assembly language

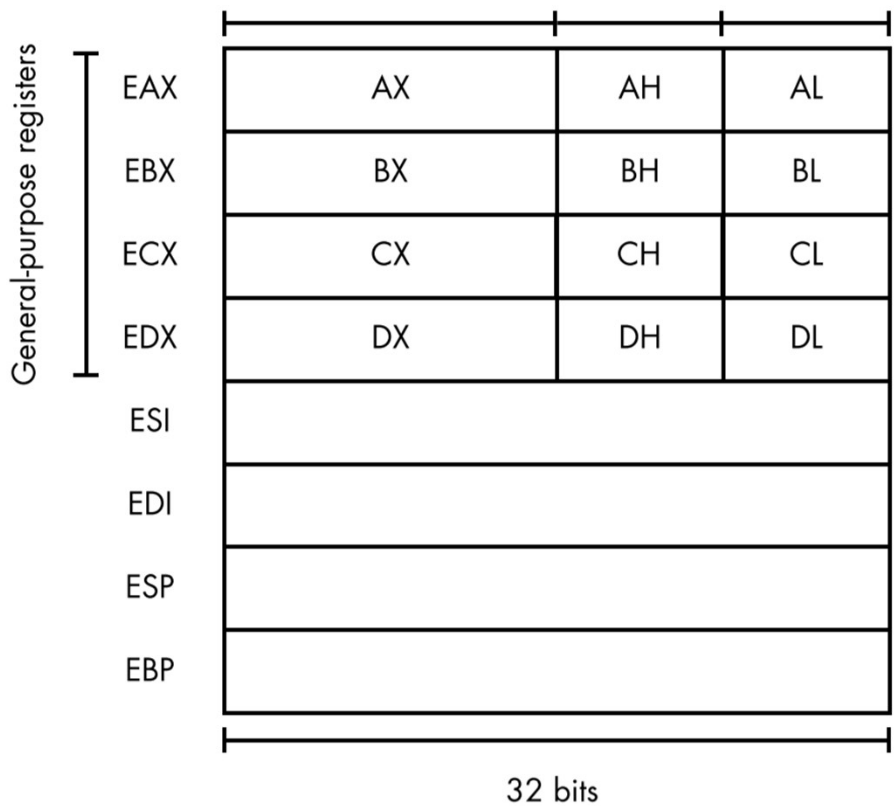
- malwares spend most of time calling into the operating system
  - by way of the DLLs
  - do most of the real work
    - modifying the system registry
    - moving and copying files
    - communicating via network protocols, and so on.
  - following malware assembly code often involves
    - understanding the ways in which function calls are made from assembly
    - understanding what various DLL calls do

# Basics of x86 assembly language

- CPU Registers
  - General-purpose registers
    - are like scratch space for assembly programmers.
    - On a 32-bit system
      - each register contains 32, 16, or 8 bits of space

# Basics of x86 assembly language

- CPU Registers
  - General-purpose registers



# Basics of x86 assembly language

- CPU Registers
  - Stack and Control Flow Registers
    - store critical information about the stack which is
      - responsible for storing
        - local variables for functions
        - arguments passed into functions
        - and control information relating to the program control flow
    - ESP register points to the top of the stack for the currently executing function
    - EBP register points to the bottom of the stack for the currently executing function
    - EIP register contains the memory address of the currently executing instruction
    - EFLAGS is a status register that contains CPU flags

# Basics of x86 assembly language

- Arithmetic Instructions

| Instructions | Description |
|---|---|
| `add ebx, 100` | Adds 100 to the value in EBX and then stores the result in EBX |
| `sub ebx, 100` | Subtracts 100 from the value in EBX and then stores the result in EBX |
| `inc ah` | Increments the value in AH by 1 |
| `dec al` | Decrements the value in AL by 1 |

# Basics of x86 assembly language

- Data Movement Instructions

| Instructions | Description |
|---|---|
| mov ebx,eax | Moves the value in register EAX into register EBX |
| mov eax, [0x12345678] | Moves the data at memory address 0x12345678 into the EAX register |
| mov edx, 1 | Moves the value 1 into the register EDX |
| mov [0x12345678], eax | Moves the value in EAX into the memory location 0x12345678 |

# Basics of x86 assembly language

- Stack Instructions
  - push instruction
    ```
    push 1
    ```
    - points the ESP to a new memory address
    - copies the value from the argument to that memory location
  - pop instruction
    ```
    pop eax
    ```
    - pops the top value off the stack and move it into a specified register.

# Basics of x86 assembly language

- Stack Instructions
  - It is important to understand that stack grows downward in memory
    - the highest value on the stack is actually stored at the lowest address in stack memory
    - push instruction decrements the ESP and then copies the value into that memory location
    - pop instruction copies the top value off of the stack and then increments the value of ESP

# Basics of x86 assembly language

- Control Flow Instructions
  - define a program's control flow
  - often expressed through C-style function calls
  - are closely related to stack
  - the most important are call and ret

# Basics of x86 assembly language

- Control Flow Instructions
  - call instruction calls a function
    - Think of it as a function in a higher-level language like C

```
call address
```

    - does two things
      - First, it pushes the address of the  next instruction onto the top of the stack so that
      - Second, it replaces the current value of EIP with the value specified by the address operand.

# Basics of x86 assembly language

- Control Flow Instructions
  - the ret instruction completes a function call

```
ret
```

  - ret pops the top value off the stack
    - places the popped value back into EIP and resumes execution
  - The jmp is another important control flow instruction `jmp 0x12345678`
    - tells the CPU to move to the memory address specified as its parameter

# Basics of x86 assembly language

- Control Flow Instructions
  - x86 assembly doesn't have high-level constructs like if, then, else, else if
  - branching to an address typically requires two instructions:
    - a cmp instruction
    - a conditional branch instruction

# Basics of x86 assembly language

- Control Flow Instructions
  - Most conditional branch instructions
    - start with a j
    - post-fixed with letters that stand for the condition being tested
  - E.g. jge tells the program to jump if greater than or equal to

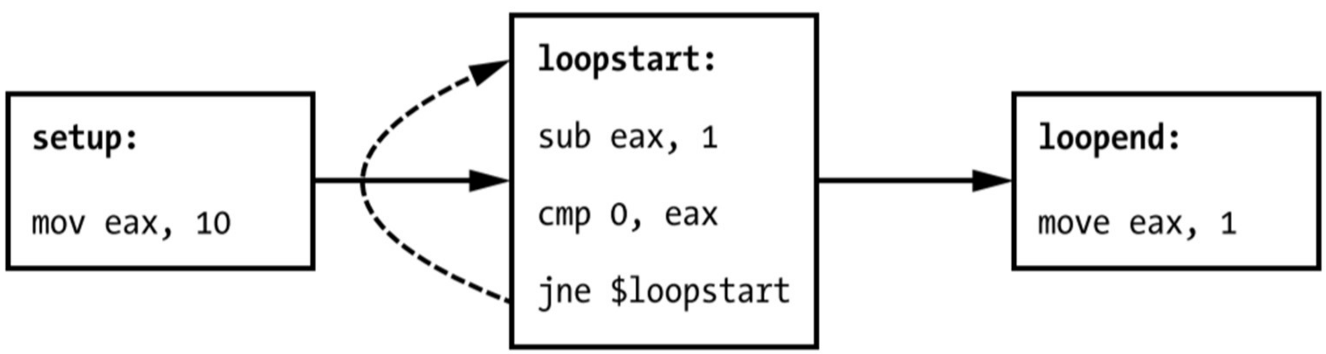
# Basics of x86 assembly language

- Basic Blocks and Control Flow Graphs
  - A basic block is a sequence of instructions that we know will always execute contiguously
    - always ends with either a branching instruction or an instruction that is the target of a branch
    - always begins with either the first instruction of the program or a branch target

# Basics of x86 assembly language

- Basic Blocks and Control Flow Graphs

```
  setup: # symbol standing in for address of instruction on the next line
❶ mov eax, 10
  loopstart: # symbol standing in for address of the instruction on the next
  line
❷ sub eax, 1
❸ cmp 0, eax
  jne $loopstart
  loopend: # symbol standing in for address of the instruction on the next line
  mov eax, 1
  # more code would go here
```



# Disassembling ircbot.exe

```
pip install pefile
pip install capstone
```

- capstone is an open source disassembly library that can disassemble 32-bit x86 binary code

Demo time

# Factors that limit Static analysis

- static analysis has limitations
  - render it less useful in some circumstances
  - malware authors can employ certain offensive tactics
    - are far easier to implement than to defend against

# Factors that limit Static analysis

- Packing
  - the process by which malware authors compress, encrypt, or mangle the bulk of their malicious program
    - it appears inscrutable to malware analysts
    - When the malware is run, it unpacks itself and then begins execution.
  - The obvious way around packing is
    - To actually run the malware in a safe environment
  - is also used by benign software installers for legitimate reasons

# Factors that limit Static analysis

- Resource Obfuscation
  - obfuscates the way program resources are stored on disk, and then deobfuscate them at runtime
    - A simple obfuscation would be
      - to add a value of 1 to all bytes in images and strings stored in the PE resources section
      - subtract 1 from all of this data at runtime
  - one way around resource obfuscation is
    - to run the malware in a safe environment.
  - Another mitigation is to
    - figure out the ways in which malware has obfuscated its resources
    - manually deobfuscate them

# Factors that limit Static analysis

- Anti-disassembly Techniques
  - are designed to exploit the inherent limitations of disassembly techniques to
    - hide code from malware analysts
    - or make malware analysts think that a block of code contains different instructions than it actually does
  - there's no perfect way to defend against them
  - In practice, the two main defenses against are
    - to run malware samples in a dynamic environment
    - to manually figure out where anti-disassembly strategies manifest within a malware sample and how to bypass them

# Factors that limit Static analysis

- Dynamically Downloaded Data
  - involves externally sourcing data and code
    - a malware may load code dynamically from an external server at malware startup time
      - static analysis will be useless against such code
    - A malware may source decryption keys from external servers at startup time
      - then use these keys to decrypt data or code
  - Such techniques are quite powerful
    - the only way around them is to acquire the code, data, or private keys on the external servers by some means