X86 Disassembly Crash Course

Levels of Abstraction
Reverse-Engineering
X86 Architecture
Levels of Abstraction

• High-level Language Programs are compiled, interpreted, or a mixture of both.

• **Compilation**: Translation of a program in a *higher-level* language (C, C++) to an equivalent program in a *lower-level* language (x86 object code). Translation to assembly language is often one step in this process.

• **Interpretation**: Direct execution of a program represented in a *higher-level* language (python) by a compiled program written in some other *higher-level* language (C).

• Most interpreters actually involve translation from source code into some intermediate form, often bytecodes (similar to machine language language).
What is this *machine* language?

- The CPU hardware of a computer is characterized by its *instruction set*.
- Instructions are encoded as binary numbers that are interpreted by the hardware to identify what operations to perform.
- These bytes sequences are said to be the *machine language* of the CPU.
- The CPU is a *hardware interpreter* of its own machine language.
- The units of a machine language program are *instructions*, each of which consists of an operation code (op-code) followed by its operands. (People sometimes refer to the entire instruction as an op-code.)
- Machine-language programs are referred to as *object code*. 
Hardware Implementation Layers

- Although the CPU hardware of a machine directly implements its instruction set, even this hardware is often programmed.
- The hardware of a CPU may be controlled by a program that causes it to emulate the behavior of a CPU that directly executes the instruction set. This emulation layer is referred to as microcode or firmware.
Reverse-Engineering

• Malware artifacts are *most often* presented as object code.
• Reverse-engineering is necessary to understand this object code, because its purpose is not directly ascertainable from its external representation.
• Object code differs from one machine to another.
• Our interest is *specifically* x86 object code, that is object code from the 80x86 family of microprocessors.
• We will deal primarily with the 32 bit Pentium instruction set programmed in *protected* mode.
• One subgoal of reverse-engineering in this setting is to convert object code in assembly language by *disassembling* it.
The x-86 Architecture

- Diagram from University of Calgary Dept. of Computer Science
Typical Program Memory Layout

Low memory addresses

Data

Code

Heap

Stack

High memory addresses
Little-Endian vs. Big-Endian

- In modern computers, the byte is the largest unit that has an unit representation. Larger values are composed of sequences of bytes.
  - Word: 2 bytes
  - Double word: 4 bytes
  - Quad word: 8 bytes
- These multi-byte units must be stored in some sequence, either least-significant byte first or most-significant byte first.
- These are referred to (respectively) as little-endian and big-endian (nomenclature taken from Jonathan Swift's *Gulliver's Travels*).
- So, though we write a large number in big-endian notation, its constituent bytes are stored little-endian in an Intel computer.
Some Useful Links

- Zack Smith's Intel Instruction Set Reference
- Sparks and Flame's x86 OpCode Chart
- Intel Software Developer's Manuals
Instruction Binary Representation

• Consider this sequence of Byte values:

   B8 EB 05

• Opcode B8: \texttt{mov ax, Immediate Word}

• Operand: 05EBh = 1515 (in decimal)
Operand Types

● Register
  Efficiently operated upon by the Arithmetic-Logic Unit (ALU)

● Memory
  Require moving data between the CPU and memory

● Immediate
  Operand value is contained within the instruction

● Implied
  Operand value is known because of the OpCode
X86 Registers
(32-bit Architectures)

General Purpose

EAX  AX
    AH  AL
EBX  BX
    BH  BL
ECX  CX
    CH  CL
EDX  DX
    DH  DL

Segment

CS  FS
DS  GS
ES  SS

Pointers

EBP  BP
ESP  SP
ESI  SI
EDI  DI

Index

Control

EIP
FLAGS
Simple Instructions

• The *mov* instructions move data from one location to another without modifying it. They are most often used to move data to and from memory, but may exchange register values as well.

• All of these are all of the form:

  \[ \text{mov destination, source} \]

• The destination is always either *memory* or a *register*, the source can be an *immediate* value.

• The source can be either *direct* (move the specified value) or *indirect* (move the value at the address specified by the operand).

• At most one operand can be a memory address.
Sample mov Instructions

- `mov eax, ebx` ; move from register to register
- `mov eax, 0x42` ; immediate to register
- `mov eax, [0x4037C4]` ; indirect immediate to reg.
  ; loads value found at
  ; address 4037C4h
- `mov eax, [ebx]` ; indirect register to reg. Loads value
  ; found at address in ebx
- `mov eax, [ebx+esi*4]` ; calculates memory address
  ; for indirect operand
- `mov [ebp+4], eax` ; loads to indirectly specified
  ; memory address
lea vs. mov

- **lea destination, value**
  - Load Effective Address
  - Calculates an indirect address using arithmetic like `mov`, but stores the address, not the memory contents of that address in the destination.

- **Suppose**
  - address 0030A304h contains 00000005h
  - register `ebp` contains 0030A300h

```
mov eax, [ebp + 4] ; stores 5 in eax
lea eax, [ebp + 4] ; stores 0030A304h in eax
```
Arithmetic (add)

- **add destination, value**
  - Adds the value specified to the value stored in destination and replaces destination with the result.
  - Like C `destination += value`.

- Supports the following addressing modes:
  - Register, Register
  - Register, Memory
  - Register, Immediate
  - Memory, Register
  - Memory, Immediate
Arithmetic sub, inc, dec

• sub destination, value
  - Subtracts value from destination and stores in destination.
  - Like C destination -= value.
  - Same addressing modes as add.

• inc destination
  - Increments destination by 1.
  - Like C destination++.
  - Destination can be register or memory.

• dec destination
  - Decrements destination by 1.
  - Like C destination--.
  - Destination can be register or memory.
Flags Set by add and sub

- **OF**: Overflow flag (Result would not store entire result.)
- **SF**: Sign flag (Last operation yielded value with MSB set.)
- **ZF**: Zero flag (Set if the result of an arithmetic operation is 0.)
- **AF**: Adjust flag (Carry or borrow in least 4 significant bits)
- **PF**: Parity flag (1 if number of 1 bits is even, 0 if odd)
- **CF**: Carry flag (Arith. carry or borrow occurred in MSB position)

\[ \text{inc and dec do not set CF.} \]
Multiplication and Division

- **mul** *value*
  - *eax* must contain operand to be multiplied by *value*.
  - The 64-bit result is stored with most-significant bits in *edx* and least-significant in *eax*.

- **div** *value*
  - *edx* and *eax* must contain the 64-bit operand to be divided by *value*.
  - Result is stored in *eax*. Remainder is stored in *edx*. 
Logic Operations

- Binomial logic operations \( \text{or, and, xor} \)
  - Same addressing modes as add and sub.
  - Operation is performed between source and destination and result is left in destination.
  - Like C \( \text{destination} \neq \text{source} \) where ? is |, &, or ^.
  - Sets SF, ZF, PF. Clears OF and CF.

- Idiomatic usage:
  \( \text{xor eax, eax} \)
  Stores 0 in eax using a 2-byte instruction.
Shift Operations

- `shr destination, count`
- `shl destination, count`
  - Destination must be register or memory.
  - Count is immediate.
  - Bits shifted beyond boundary are shifted through \( CF \) (only flag set by operation).
  - Zero bits are used for fill.
- Often used for multiplication/division by powers of 2.
Rotate Operations

- **ror** destination, count
- **rol** destination, count
  - Destination is register or memory.
  - Rotates bits in specified direction (no bits are lost)
  - CF is set to bit last rotated around end.
- **rcl** destination, count
- **rcr** destination, count
  - Rotates value into CF augmenting destination value by 1 bit.
NOP

- **nop**
  - Does nothing
  - No flags set
- **Implented with** `xchg eax, eax`
- **xchg destination, source**
  - Source and destination are exchanged
  - Operands
    - Register, Register/Memory
    - Register/Memory, Register
  - No flags set
Next Class

- Stack
- Function calls
- Conditional execution
- Branching
- Rep instructions