CS7038 - Malware Analysis - Wk06

Deeper Dive: x86 32/64 Assembly

Coleman Kane
kaneca@mail.uc.edu

February 15, 2018

UNIVERSITY OF Cincinnati

A Deeper Look

Previous lecture focused on introducing assembly language and how program can be broken up from source code and translated into assembly language.

Our discussion will focus on the Intel/AMD x86 instruction set. Primarily there are 2 versions of it in use today: 32-bit and 64-bit. This instruction set has the following characteristics:

- Register-memory architecture
- Some purpose-specific registers
- 8 (32-bit) or 16 (64-bit) direct-access *general purpose registers*
- Segment/Section *selectors*
- Direct / indirect memory access modes
- Variable-length instructions
- Internal flags, instruction pointer, and machine registers

Assembly Language Source Variations

All instructions are compact binary structures.

Since Assembly Language is basically a textual representation of binary encoded instructions.

One thing you will encounter is that the Assembly source code is just a textual representation of the binary data. Due to this, there are differing approaches to translating it into human-readable source. The following two methods are predominantly used:

- AT&T, UNIX, or GNU syntax. In general: `OP SRC DEST`, read `OP SRC -> DEST`. Frequently registers are prefixed with % symbol.
  Example: "`movl %rax,%rbx`" puts a copy of what's in `RAX` into `RBX`

- Intel or Microsoft syntax. In general: `OP DEST SRC`, read `OP DEST <- SRC`. Frequently op names don't include data width (l, b, w, q) specifiers.
  Example: "`mov rbx,rax`" same as above

Instruction Lengths

x86 instructions are a sequence of bytes describing a unique atomic CPU operation. It is described as a *variable-width* instruction set, meaning that there is no fixed size for all instructions, but rather the number of bytes that comprises an instruction is based upon the instruction and its parameters.

Examples:

- `jmp %rax`: Jump to address provided in `%rax`
  2 bytes
- `movq $0x0000000011111111,%rbx`
  Copy numeric value `0x011111111` into `%rbx`, **10 bytes**
- `addl $0x12,-0x8(%ebp)`:
  Add the 32-bit value `0x00000012` to the 32-bit value stored in the memory location specified by performing the pointer calculation `%ebp - 0x08`, **4 bytes**.
- `nop`: Do nothing, **1 byte**

Generalized Instruction Formats

Instructions can be anywhere from 1 byte to ~15 bytes depending upon their purpose. As a general rule, x86 attempts to reserve the shorter instructions for the more frequent ones, and relegates longer instructions to represent less common operations.

A common layout is below:

Operation	ModR/M	SIB	[Displacement]	[Immediate] [...]

In the above:

- Operation is typically 1 or 2 bytes, but can be 4 bytes to reflect a "modified" version of an operation using a 1-2 byte reserved prefix value
- ModR/M (*when required*) is a single byte that informs the CPU as to how to interpret operands (memory, register, immediate) as well as addressing modes
- SIB (*when required*) for a scaled addressing operation, what mem register to use and what multiplier to use
- Displacement, Immediate, ... operand values that are provided - lengths may vary

CPU Registers

Local memory directly addressable by (most) instructions in the CPU. In the 64bit x86 architecture, there are 16 registers that can be addressed using the names `R01-R15`. The architecture also assigns some of these registers to "special purposes" in many instructions, an oddity that isn't shared by all CPU architectures.

`R0`	`RAX`	Accumulator (freq. result storage)
`R1`	`RCX`	Counter (freq. used as `i` in iterations/loops
`R2`	`RDX`	Data (freq. additional argument in operation)
`R3`	`RBX`	Base (freq. used as a base address/counter)
`R4`	`RSP`	Stack pointer (used to keep track of top of CPU stack)
`R5`	`RBP`	Base pointer (used to keep track of base of CPU stack)
`R6`	`RSI`	Source index (keeps track of indexes of source arrays)
`R7`	`RDI`	Destination index (keeps track of indexes of destination arrays)

There are `R8-R15` that are 8 more general registers with no other specialized purpose. In most operations that accept a register, any `R0-R15` may be used.

Memory Addressing

Instructions can have varying addressing modes that represent common programming patterns where it comes to addressing memory.

The below *addressing modes* relate directly to various methods you may use in array & matrix operations in higher level languages, as well as those containing complex data structures as elements.

- Register addressing: `movq %rax,%rbx`
- Immediate addressing: `movq $0x0a,%rbx`
- Direct memory: `movq 0x1000,%rax`
- Register indirect mem.: `movq (%rbx),%rax`
- Reg-indirect + offset mem.: `movq 0x8(%rbx),%rax`
- Reg-indirect*C + offset mem.: `movq 0x8(,%rbx,4),%rax`
- Base + Reg-indirect*C + offset mem.:
  `movq 0x8(%rdx,%rbx,4),%rax`