CS7038 - Malware Analysis - Wk05.2

Assembly Crash Course

Coleman Kane
kaneca@mail.uc.edu

February 8, 2018

What Is Machine Langauge

In order to execute software on in most of your environments, the source code for the software must be converted into machine langauge at some level.

The most common approach, by far, is to perform static compilation of source code from an author, which involves translating it into a target-native machine code.

A lot of malware you will encounter written in C, C++, or Delphi, will come in this form - frequently x86 32-bit or 64-bit code.

This form is the native "language" of your CPU, and is the most efficient form for your software

Architecture Background

Among CPU architectures, there are two predominant models:

- Load-Store Architecture
- Register-Memory Architecture

In both cases, the computer is divided into components that are system memory (typically "RAM") and CPU registers. The CPU registers are given names, and are analogous to variables in your programs.

Frequently, the processor needs to "localize" any data that it needs to operate on. In the case of load-store, this "localization" is actually an architectural requirement.

Instruction Families

The atmoic operations executed at the machine-code level are called "instructions", and these implement the core functionality of the CPU hardware. There are many of them (roughly between 900-1000 implemented on current CPUs), and thus they are divided into instruction families:

- Arithmetic
- Control transfer
- Memory transfer
- System

For the time being, we will focus on the arithmetic, control, and memory instructions. These represent the core feature set of a CPU for implementing algorithms.

Typical Compilation Sequence

Frequently, compilation consists of the following sequence of steps:

1. Write application
2. Compile source code into assembly
3. Assemble assembly source into binary machine code
4. Link chunks (objects) containing binary into EXE / DLL
5. Install on target system

Important note: without some sort of emulation support, the compiled software will frequently only run on machines and OS versions that it was built to be compatible with.

Control Flow Graph Construction

During step 1 from above, prior to translating the C code into assembly, the software is broken up into a graph, dividing the program up into sequences of non-control-transfer operations called "blocks". These blocks are interconnected by the control transfer operations, creating a "control flow diagram".

Those blocks are then independently compiled from the blocks of source code into blocks of assembly instructions that implement the source code.

Following this, the blocks are then arranged in the output file, and the destinations of control-flow instructions are adjusted to match the output organization.

Resources for Assembly Language

I won't go into a lot of detail on the individual instructions here. However there are a number of good references online:

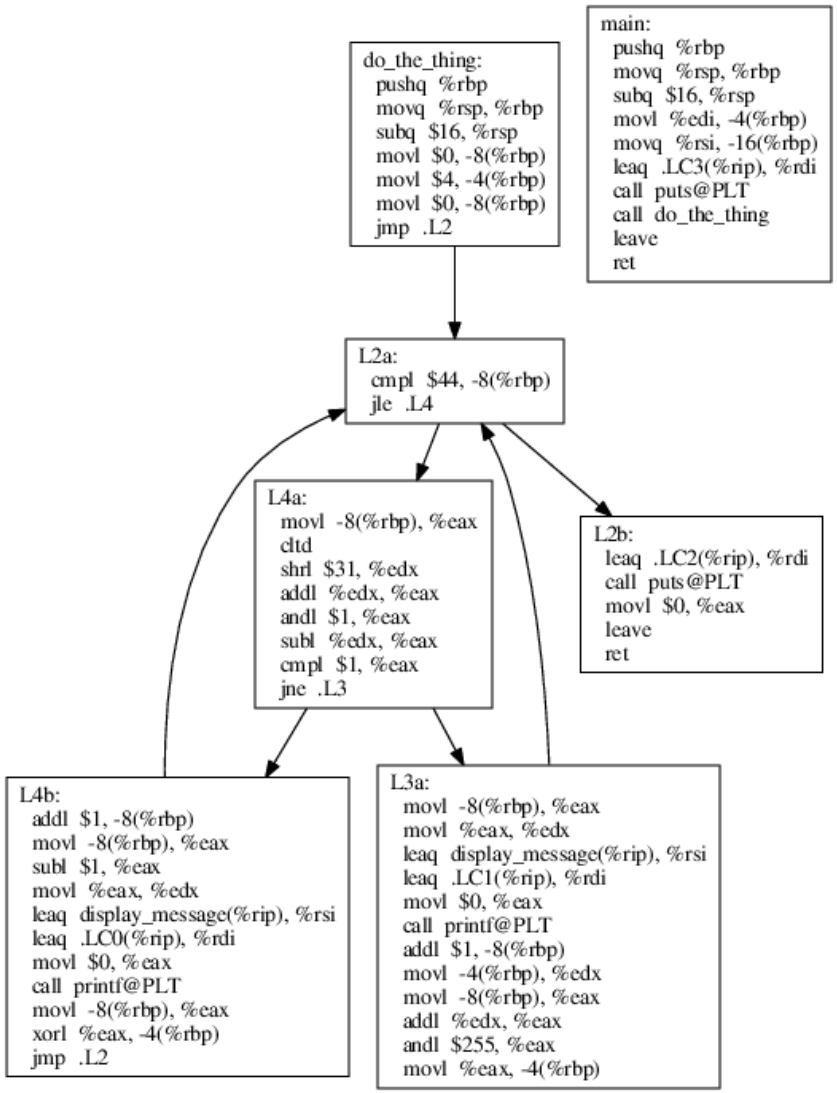
- x86 Instruction reference - simple site - `http://ref.x86asm.net/`
- AMD64 Programmer's reference, Vol 3 - `https://support.amd.com/TechDocs/24594.pdf` (PDF)
- Sandpile - `http://sandpile.org`
- Navigable parsed version of Intel 64 reference - `https://github.com/zneak/x86doc`

In many cases, you can request either print or electronic copies of the instruction set for most CPUs you are working with by contacting the manufacturer of the target architecture.

Example Program

```c
#include <stdio.h>

static char display_message[] = "Current val";

int
do_the_thing(void) {
  int i = 0;

  int b = 4;

  for(i = 0; i < 45;) {
    if((i % 2) == 1) {
      i = i + 1;
      printf("%s (odd): %d\n", display_message, i - 1);
      b = b ^ i;
    } else {
      printf("%s (even): %d\n", display_message, i);
      i = i + 1;
      b = (b + i) & 0xff;
    }
  }

  printf("This is the end\n");
  return 0;
}

int
main(int argc, char **argv) {
  printf("Beginning the program\n");

  return do_the_thing();
}
```

Program Assembled and graphed



UNIVERSITY OF Cincinnati

Disassembly & IDA

When access to the program source code is unavailable, as typically is the case with malware, it is necessary to use various tools to reconstruct the control-flow-graph and help the engineer make sense of the program. In some cases, tools may even help convert some or all of the disassembly into some possibly readable C program code.

Some tools exist to help with this. Each have their own strengths and weaknesses:

- IDA - `https://www.hex-rays.com/` (closed src)
- binary ninja - `https://binary.ninja/` (closed src)
- ROSE - `https://www.rose-compiler.org/` (semi-open src)
- radare2 - `http://rada.re/r/` (open src)
- snowman - `https://github.com/yegord/snowman` (open src)