Cosc 69.16: Reverse Engineering

Prishita Dharampal

For this lab assignment, you will be compiling short C programs and then reverse engineering them to see what they look like inside. This assignment will be performed individually, first on the Unix command line and then in the GHIDRA reverse engineering framework.

Problem 1. The following is a short C program. Place it in a file named `first.c` and compile it with `cc -o first first.c` to produce an executable. You can run this with `./first`.

```
#include <stdio.h>

int main(int argc, char **argv) {
    if (argc > 1) {
        printf("Hello world. %s\n", argv[1]);
    } else {
        printf("Aloha!\n");
    }
    return 0;
}
```

1. What file format and architecture is your file? You can find this by running the `file` command on your executable.
2. Disassemble your executable with the `objdump -d` command, and write down the `main` function's disassembly.
3. Add some comments to the disassembly, marking the return value (`0`) of `main`, the address of two strings (`"Hello world argv[1]"` and `"Aloha!"`), and the registers that store `int argc` and `char **argv`. Note we typically will call adding such comments *annotation*.
4. `objdump -d` often doesn't include strings in its listing, but `objdump -D` will. Include the lines of the string in your document.
5. Why are the bytes of the string interpreted as machine language?

6. Use an ASCII table to decode the bytes of the strings. C strings end with a null byte (`0`), so you should include each and every byte from the beginning to the null.

Problem 2. Let's try a program that's a little more complicated, which we'll call `second.c`.

```
#include <stdio.h>

int square(int i) {
    return i * i;
}

int main(int argc, char **argv) {
    for (int i = 0; i < 10; i++) {
        printf("The square of %d is %d.\n", i, square(i));
    }
}
```

1. Compile the program with both `cc -O3 second-fast second.c` and `cc -Os second-small second.c`. The `-O` flags tell the compiler to optimize the build, with `-O3` making the fastest code and `-Os` making the smallest code.
2. Disassemble both of these files with `objdump -d`. What's different about their `main` functions? Which one is easier to understand, and why?

Problem 3. Our first two programs included symbols, so we could easily tell where `main` was and what the variable names were. For this third program, we will strip the symbols to leave an executable more like those that are distributed commercially. Compile it with `cc -o third third.c` and do `objdump -d` before and after stripping the binary with `strip third`.

What are the differences between these two `objdump` outputs, and why?

```c
#include <stdio.h>
#include <stdlib.h>

__attribute__((noinline))
void guess(int g) {
    if (g == 42)
        printf("That's right!\n");
    else
        printf("Nope, that's the wrong number.\n");
}

int main(int argc, char **argv) {
    if (argc == 2) {
        guess(atoi(argv[1]));
    } else {
        printf("Give me a number?\n");
    }
}
```

Instead of `objdump`, we'll use GHIDRA for this assignment, a free reverse engineering tool developed by the National Security Agency.

After opening GHIDRA, create a New Project. After the file is imported, open it and click Yes to have GHIDRA auto-analyze the executable.

1. In the Symbols pane, practice navigating between functions.
2. Because the binary is stripped, GHIDRA does not know local function or variable names, though it may identify standard library calls such as `atoi()`.
3. Navigate to the function that calls `atoi()`. Rename it `main`.
4. Find and name the `guess()` function.
5. Rename the local variables in `guess()` to match the original source code.
6. Comment each line of the assembly for `guess()` and paste the commented disassembly into your report.

Problem 4. For extra credit, repeat the third section using tools such as Binary Ninja, IDA Pro, Radare2, or Cutter.