CS 69/169: Lab Assignment, Week 01 Day 02

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.

Deliverable: Write up your results in a document. We won't force you to a single style, but professionalism in reporting is a good habit to learn as a reverse engineer. Generally this means a summary of what you found, followed by a section with technical details. For formatting, it is typical to use a monospaced font for any disassembly and consistent indentation. Syntax highlighting might be handy. You may wish to consider LaTeX or Markdown.

First Program

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){
  printf("Hello world.\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` and the address of the string ("Hello World"). We'll 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 string. C strings end with a null byte (`0`), so you should include each and every byte from the beginning to the null.

Second Program

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 with `cc -Os second-small second.c`. `-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?

Third Program

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 strip the binary with `strip third`.

```
#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. You can download it for free from `https://ghidra-sre.org/`.

After opening GHIDRA, create a New Project. GHIDRA is somewhat unique among reverse engineering tools in that it has projects of more than one file, which is very handy when reverse engineering files for an entire course. You might want to maintain one project for the whole semester, so that you can look back at past lab assignments while working on new ones.

After the file is imported, open it and click `Yes` to have GHIDRA auto-analyze the executable. Analysis serves many purposes, but for now you can think of it as running through the disassembly to figure out where functions begin and end.

After GHIDRA has analyzed the file, you can scroll through a disassembly of the executable in the Listing pane, which you'll see in the middle. GHIDRA also includes a decompiler, and the right pane will show you an attempt at recreating the original C code.[1]

[1]We'll discuss this and it's challenges more later in the course, but you should comment on where you see automated decomompilation matching or deviating from the disassembly.

1. In the Symbols pane on the left, you should see a few Functions. Practice navigating between them.

2. Because you stripped the binary, GHIDRA doesn't know the names of these local functions or the local variables inside. It can notice some standard library functions, such as `atoi()`.

3. Navigate to the function that calls `aoti()`. It will have an auto-assigned name like `entry` or `FUN_10000eac4`. Click on the function name to select it, and then press the `L` key to rename the function. Call this one `main`.

4. Now find and name the `guess()` function. You might look for the literal `42`, or you might look for the strings that are printed by that function.

5. Inside of the `guess()` function, change the names of the local variables in the decompiler view with the `L` key to match the original source code.

6. You can use the semicolon key (`;`) to assign comments to lines of code in both the Decompiler and Listing views. By using the Decompiler view as a cheat sheet, first understand and then comment each line of the assembly code for the `guess()` function. Paste this commented disassembly into your report.

Other Tools (Bonus)

GHIDRA is an excellent tool for reverse engineering, but it's not the only one available. For extra credit, repeat the third section in trial editions of commercial tools such as Binary Ninja and IDA Pro, or in free tools, such as Radare2 and Cutter.