CSC 325 Lab 5

Binary Puzzle

Table of Contents

Partners for this lab are in Lab5.

1. Lab Goals

- Gain experience reading and tracing through the execution of ARM64 assembly instructions.
- Enhance your understanding of how assembly translates to C instructions, data structure access, and function calls.
- Practice with tools for examining binary files.
- Put your GDB skills to work to solve an assembly code puzzle.

2. Lab Overview

In this assignment, you and your partner are going to receive a binary program. Your goal is to reverse engineer what it does.

3. Lab Starting Point Code

3.1. Getting Your Lab 5 Lab Repo

Both you and your partner should download the puzzle binary to your ARM64 machine.

3.2. Running your program

Your program is compiled for an ARM machine, so it will only run on an ARM CPU.

You are almost always going to want to run your program in gdb. You'll likely want to start it in gdb, set breakpoints, and then, in order to step through its execution:

```
$ gdb ./puzzle
(gdb) layout asm
(gdb) break main
(gdb) run <group_number>   # where <group_number> is your Lab5 group, e.g., 5
```

You can also just run your puzzle program from the command line.

```
$ ./puzzle <group_number>
```

4. Lab Details

The binary is a program that consists of a sequence of assembly language algorithms, one for each Lab5 group.

4.1. Solving Your Puzzle

- You can use many tools to help you solve your puzzle. Look at the hints section below for some tips and ideas. **The best way is to use `gdb` to step through the execution of the disassembled binary**.

5. Lab Requirements

- You must solve your puzzle by examining it at the assembly code level using tools like `gdb`, `strings`, `objdump`, and other similar tools for examining binary files.
- For the **checkpoint** you need to get past floors 1 and 2, and you need to edit the `puzzleID` file and push it to your repo.
- For the **complete solution** you need to get past floors 1-4, complete and submit the write-up of how you solved each floor, and submit the solution to each floor.

Write-up Requirements (for the completed solution only)

- Edit `how_we_solved.txt` in an editor to include a short explanation of how you solved your puzzle and a short explanation of what each section of the control flow is doing.
- **Describe at a high-level** what the original C code is doing for each section. For example
  - Is it doing some type of numeric computation, string processing, function calls, etc?
  - Describe the specific computation it is doing (i.e. what type of string processing and how is that being used?).

  - Is there a loop? What is the number of iterations?
  - Is there a function? What is its purpose? What are its formal parameters and return value?
  - Is there a function call? What are the arguments and the return value?

- **Don't describe in terms of registers and assembly code for this part**, but describe what the puzzle on each floor is doing at a higher-level in terms of C semantics. You do not need to reverse engineer the assembly code and translate every part of it to equivalent C code. Instead, give a rough idea of equivalent C or pseudo code for the main part of each puzzle section.

  For example, something like "uses an if-else to choose to do X or Y based on the input value Z" is an appropriate right level of explanation. Something like "moves the value at `sp + 8` into register `x0` " is **way too low-level**.

- The lab write-up lab should not be onerous; you should be able to explain each section in a short paragraph or two (maybe with a few lines of C or pseudo code to help explain). I recommend doing the write-ups for each section as you solve them.

- Excessively verbose, low-level descriptions will be penalized, as will vague descriptions; you want to clearly demonstrate that you figured out what the code is doing by examining the assembly code for each section in your puzzle executable.

- If you are unable to solve a section, you can still receive partial credit for it in the write-up part by telling me what you have determined about it.

6. Tips

There are many ways of solving your puzzle. There are various tools for examining the program binary without running the puzzle program. These may provide some helpful information for solving some sections. The most useful tool will be `gdb`, which will allow you to run the puzzle program, set breakpoints, step through parts of its instruction's execution, and examine its execution state. This will help you to discover information about what the program does, and you can use this information to solve your puzzle.

Remember that the puzzle program must be run on an ARM64 CPU.

6.1. How to solve the puzzle lab!

There are many tools that are designed to help you figure out both how programs work, and what is wrong when they don't work. Here is a list of some of the tools you may find useful in analyzing your puzzle, and hints on how to use them.

- `gdb ./puzzle` The GNU debugger will be your most useful tool. You can trace through a program line by line, examine memory and registers, look at the assembly code, set breakpoints, and set memory watch points.

- draw the stack and register contents as you are tracing through code in gdb, and take notes as you go (this will also help you with the write-up part of the lab assignment).
- `strings puzzle`: display the printable strings in your puzzle.
- `objdump`: objdump may provide some information that is helpful, but `gdb` will be your most useful tool
  - `objdump -t puzzle` prints out the puzzle's symbol table. The symbol table includes the names of all functions and global variables in the puzzle, the names of all the functions the puzzle calls, and their addresses. You may learn something by looking at the function names.
  - `objdump -d puzzle`: disassemble all of the code in the puzzle. You can also just look at individual functions. Although `objdump -d` gives you a lot of information, it doesn't tell you the whole story. Calls to system-level functions are displayed in a cryptic form. For example, a call to `sscanf` might appear as:

    ```
    401010:    97fffefc    bl  400c00 <__isoc99_sscanf@plt>
    ```

    To determine what that call to `sscanf` is doing, you need to disassemble within a running puzzle program using `gdb`.

Looking for documentation about a particular tool? The `man` command will help you find documentation about unix utilities, and in `gdb` the `help` command will explain gdb commands:

```
$ man objdump

(gdb) help ni
```

6.2. Notes on odd instructions or code sequences

You may find some code in your puzzle lab that uses instructions that we have not talked about in class. Here are some notes about some of these:

- **`adrp`**: The `adrp` instruction is loading an address from some other (usually far away) region of memory. You'll most commonly see this when the program is referencing a static string in your program. Examples might include a format string to `printf` or `scanf` like `"%d"`, but it could be other strings that live in the data region of the puzzle program.

This list is (probably) incomplete — we're doing this lab with ARM64 assembly for the first time ever this semester. If you encounter weird instructions, please try to decipher them, and notify your instructor. I'll try to help you and may add them to this list of "odd instructions".

7. Submitting

Zip a folder containing your "`how_we_solved.txt`" file and a screenshot of your ARM64 terminal running gdb.

8. Handy Resources

- C Debugging Tools (textbook Chapter 3)
- ARM64 Reference Sheet
- gdb for IA32 assembly debugging # most is also applicable to ARM64
- Some Useful Unix Commands
- vi (and vim) quick reference
- man and Manual pages documentation for libraries and commands