1 Lab 9

Date: Oct 17, 2019

This document first describes the aims of this lab. It then describes the exercises which need to be performed.

1.1 Aims

The aim of this lab is to familiarize you with examining assembly language translations of C programs. After completing this lab, you should be familiar with the following topics:

- Using the `objdump` utility to disassemble programs.
- Using gdb to disassemble programs and examine memory and registers.

1.2 Exercises

1.2.1 Starting up

Use the startup directions from the earlier labs to create a `work/lab9` directory. Make sure that your `lab9` directory contains a copy of the files directory. Run the `script` command to start recording your terminal session:

```
$ script -a lab9.log
```

As you come across a new command, briefly scan it's `man` page to get an idea of its capabilities.

1.2.2 Exercise 1: Examining Object Files

The files directory contains a hello.c program. Read the source code to understand what the program does. Build it by typing `make`. This will build the `hello` executable with debugging information included. Run the `hello` executable to ensure that the program works correctly.

Examine the sizes of the individual segments in the executable using

```
$ size hello
```

which will print sizes for the basic `text` segment (where code lives), `data` segment (where initialized static data lives) and `bss` segments (where uninitialized static data lives (initialized to 0 at load-time)) along with the total size (in both decimal and hexadecimal).

Then try

```
$ size -A hello
```

which will print information not only for the above basic segments, but also for segments which contain debugging information.

Then do a `objdump` of the executable code:

```
$ objdump -d hello > hello.objdump
```

and look at `hello.objdump` using a text editor. You should note the following:

- In addition to the assembly language symbolic representation of the instructions, the dump also contains something close to the machine code (in hex) which will be loaded into memory at runtime. Note also that the leftmost column contains the relative address of each instruction in hex.
- The variable length of the instructions which range from 1 byte instructions like `push`, `leaveq` and `retq` as well as longer 5 byte instructions like the `callq` instruction.
- Note that immediate operands are represented in little-endian order. For example, in the code for the `main` function, look for the `0xdeadbeef` argument to the `exit()` function. Note that within the instruction that constant is specified in a normal left-to-right manner but when stored in memory, its bytes appear scattered because of the little-endian order.

You can get even more information using

```
$ objdump -d -s -x hello > hello.objdump
```

The `-d` option disassembles the code, `-s` shows full-contents and `-x` displays all headers in the object file.

Look at the result again using a text editor. You are not expected to understand most of this information, but seeing it allows you to appreciate the amount of extra information generated for debugging.

You can use `grep` to filter the output of `objdump` to get 40 lines following the `<main>:` label:

```
$ objdump -d hello | grep  -A40 main.:
```

Look at the size of the executable using `ls`:

```
$ ls -l hello
```

Now strip out the debugging information in the `hello` executable:

```
$ strip hello
```

and repeat the `ls`. You should notice an appreciable reduction in the size.

Finally, clean things up for subsequent exercises using `make clean`.

1.2.3 Exercise 2: Using objdump to Peek into Object Files

In Exercise 3 for *Lab 4* , you were required to identify a mask by looking at the behavior of a program. It is trivial to identify that mask by simply disassembling the object file. That file `mystery.o` has been copied over to the `files` directory. Use a command from the previous exercise to examine the file and discover what the mask was.

One lesson you should take away from this exercise is that depending on the intricacies of binary formats to protect a secret provides no real protection. Any such secret hidden away in a object file can easily be compromised.

1.2.4 Exercise 3: Using gdb to Examine Generated Instructions

Once again run `make` to build a fresh `hello` executable containing debugging information.

Now run the executable within `gdb` by typing `gdb hello`. At the `gdb` prompt, put a breakpoint on `main` using `b main`. Type `r` to run the program, your program should stop at `main()`. Type `disas /m` to disassemble the current function and dump out the memory. You should see a assembly listing with an arrow just before the next instruction to be executed (which should be the test for `argc == 1`). Note that putting the breakpoint at the start of `main()` did not insert the breakpoint at the absolute beginning of `main()`, instead it inserted it after the function prolog.

Now examine the registers. Type `i reg` to get a dump of all the registers. You can refer to the value of an individual register by preceeding the name with a `$`. Obviously, these names are dependent on the specific machine you are debugging. However, gdb has some generic register names:

`$pc` always refers to the program counter (`$rip` for x86-64).

`$sp` always refers to the stack pointer (`$rsp` for x86-64).

`$fp` always refers to the frame pointer (`$rbp` for x86-64).

`$ps` always refers to the program-status word (`$rflags` for x86-64).

If you examine the code in for `main()`, you should realize that the first argument `argc` has been put into register `-0x14(%ebp)` and the second argument `argv` is in `-0x20(%ebp)`.

Print out the first argument by doing `p argc`. It should print out a 1. Now let's try to print it out directly from where its value is stored in memory. `-0x14(¬ $rbp)` refers to the memory addressed by `rbp - 20`, hence the value of `argc` is in the memory location addressed by `rbp - 20`. So let's try `p *($rbp - 20)`. Unfortunately, that results in *generic pointer dereference* error.

The problem is that `gdb` has no idea what `$rbp - 20` is pointing to as registers are totally untyped. Hence we need to help it out by providing a suitable type via a cast: `p *(int *)($rbp - 20)` should correctly print out a 1.

Now let's attempt to print out the second argument. `p argv` will print out the value of `argv` in hex cast to a `(const char **)` (the type is there because an array parameter of type T for a function is replaced by a pointer parameter to type T; hence `const char *argv[]` becomes `const char **argv`, explaining the cast).

Let's try to do the same thing using the value stored in memory for `argc` rather than having `gdb` do it for us. The address of `argc` is `rbp - 0x20`. Since `argv` has type `const char **` and `rbp - 0x20` contains its address, `rbp - 0x20` should have type `(const char ***)`. Hence let's try to print it using `p *(const char ***)($rbp - 32)`. This should print out the same result as `p argv`.

Now let's try printing out the first string in the `argv[]` array. So if we simply use `p argv[0]` we get the path via which the program was invoked. Let's do the same thing but accessing the `argv` stored in memory directly rather than having `gdb` do it for us. Based on our success in printing out the value of `argv` from memory, the command `p (*(const char ***)($rbp - 0x20))[0]` should do the job.

You should still be stopped at the test of `argc == 1`. Set up `gdb` to always print out the current instruction using `display /i *$pc`. You should see that you are about to execute a compare instruction, so look at the flags using `p $eflags`. Now execute the compare instruction by using the `nexti` (abbreviated `ni`) command. The `display` you setup earlier should result in the next instruction being printed. Now if you print out the flags using `p $eflags` you should see the `ZF` zero flag set.

You should be at a conditional jump instruction `jne` which will jump if the `Z` flag is not set. Since it is set, the jump will not be done and typing `ni` should put you in the code to print out the usage message using:

```
fprintf(stderr, "usage: %s NAME...\n", argv[0]);
```

The next few instructions set things up to call `fprintf()` by adding the arguments to the stack **right-to-left**. If you look at the code, you will see a large hexadecimal constant being loaded into the 2nd argument via register `%rsi`; this is the address of the format string. If that address is `0x`*nnnnn*, then `p (char *)0x`*nnnnn* should print out the format string.

Single step the code using the `ni` command until the next instruction about to be executed is the `call` instruction (you can repeat the previous command by simply typing an empty line). At that point, the first argument `stderr` should be in `rdi`, the 2nd argument the format string in `rsi` and the program name should be in `rdx`.

Using techniques similar to what you did earlier, print out the format argument using `rsi` and the program name using `rdx`.

Continue single-stepping (simply type an empty line to repeat the previous command) until you see the usage message and the program terminates. Quit `gdb` using `q`.

1.2.5 Exercise 4: More Use of gdb to Examine Generated Instructions

Now run `gdb` on the program once again using `gdb hello`. This time setup a breakpoint on `hello()`. Now run the program, but unlike last time provide a argument, say `r joe`. Once `hello()` is entered, you should regain control via the breakpoint.

Now use techniques similar to those used in the past exercise to print out the value of `hello's` argument `who` directly from the passed in argument (rather than simply `p who`).

1.2.6 Exercise 5: Modifying a Register

Making sure you have quit the previous gdb session, load the `hello` program again into `gdb` and `start joe` which should run the program with the argument `joe` but with a temporary breakpoint on `main`. Disassemble `main()` (use `disass /m`) and find the code containing the `for`-loop within `main()`. Note that the disassembly attempts to show the assembly language in order by source lines; hence assembly instructions are not necessarily contiguous in memory when the code for a source line gets split up into non-contiguous portion. Put a breakpoint on the compare instruction which compares the `for`-loop index with its bounds; specifically, if this instruction is at address `nnnn`, do `b *nnnn`.

Continue running the program using `c`, you should hit the above breakpoint. Verify that `i` is indeed 1 by typing `p i`. Then continue running the program using `c`; it should print out a `hello joe` line before stopping again at the same breakpoint. If you print out `i` again, you should get 2 and the compare should be ready to fail leading to termination of the `for`-loop.

However, you can force the `for`-loop to execute one more time. Find out which register stores `i`. Let's assume that it is `eax`. Then decrement the value of `eax` using `set $eax = $eax-1`.

Now if you continue using `c`, the loop should execute one additional time and `hello()` should be called once again. However since we have only provided a single argument and have not taken any steps to provide that value, the value actually printed will be `null`.

1.3 References

GDB Manual

GNU BinUtils Manuals