Spring 2019

Lab 2: Disassembling and Defusing a Binary Bomb

Assigned: Sunday, April 21, 2019

Due Date: *Wednesday, May 1, 2019 at 11:59 pm*

Video(s): You may find this video (../../videos/tutorials/gdb.mp4) (with captions) (https://www.youtube.com/watch?v=0f8VZBaMj0I) helpful for getting started with the lab.

Overview

Learning Objectives:

- Gain basic familiarity with x86-64 assembly instructions and how they are used to implement comparison, loops, switch statements, recursion, pointers, and arrays.
- Gain experience using the `gdb` debugger to step through assembly code and other tools such as `objdump`.

The nefarious Dr. Evil (https://upload.wikimedia.org/wikipedia/en/1/16/Drevil_million_dollars.jpg) has planted a slew of "binary bombs" on our machines. A binary bomb is a program that consists of a sequence of phases. Each phase expects you to type a particular string on `stdin` (standard input). If you type the correct string, then the phase is defused and the bomb proceeds to the next phase. Otherwise, the bomb explodes by printing "`BOOM!!!`" and then terminating. The bomb is defused when every phase has been defused.

There are too many bombs for us to deal with, so we are giving everyone a bomb to defuse. Your mission, which you have no choice but to accept, is to defuse your bomb before the due date. Good luck, and welcome to the bomb squad!

Code for this lab

Everyone gets a different bomb to diffuse! Substitute `<username>` in the URL below with your UWNetID in order to find yours.

Terminal: NOT SUPPORTED, 'wget' command will NOT work

Browser:
`https://courses.cs.washington.edu/courses/cse351/19sp/labs/lab2/<username>/lab2-bomb.tar` (sign in with your UW credentials)

Unzip: Running `tar xvf lab2-bomb.tar` from the terminal will extract the lab files to a directory called `bomb$NUM` (where $NUM is the ID of your bomb) with the following files:

- `bomb` - The executable binary bomb
- `bomb.c` - Source file with the bomb's main routine
- `defuser.txt` - File in which you write your defusing solution
- `lab2reflect.txt` - File for your Reflection answers

Lab 2 Instructions

You should do this assignment on a 64-bit CSE Linux VM or a CSE lab Linux machine or on attu. Be sure to test your solution on one of those platforms before submitting it, to make sure it works when we grade it! In fact, there is a rumor that Dr. Evil has ensured the bomb will always blow up if run elsewhere. There are several other tamper-proofing devices built into the bomb as well, or so they say.

Your job is to find to correct strings to defuse the bomb. Look at the Tools section for ideas and tools to use. Two of the best ways are to (a) use a debugger to step through the disassembled binary and (b) print out the disassembled code and step through it by hand.

The bomb has 5 regular phases. The 6th phase is extra credit, and rumor has it that a secret 7th phase exists. If it does and you can find and defuse it, you will receive additional extra credit points. The phases get progressively harder to defuse, but the expertise you gain as you move from phase to phase should offset this difficulty. Nonetheless, the latter phases are not easy, so please don't wait until the last minute to start! (If you're stumped, check the Hints section at the end of this document.)

The bomb ignores blank input lines. If you run your bomb with a command line argument, for example,

```
$ ./bomb defuser.txt
```

then it will read the input lines from `defuser.txt` until it reaches EOF (end of file), and then switch over to stdin (standard input from the terminal). In a moment of weakness, Dr. Evil added this feature so you don't have to keep retyping the

solutions to phases you have already defused, instead you can put them in `defuser.txt`.

Note: you will turn in your `defuser.txt` file as part of this lab. It is extremely important that your `defuser.txt` has a newline character at the end of the file. You can check this by using the Linux `cat` command from the terminal. If you type:

```
$ hexdump -C defuser.txt
```

and your `defuser.txt` contains the sentence `This is not the real answer.`, the output will look something like:

```
00000000  54 68 69 73 20 69 73 20  6e 6f 74 20 61 20 72 65  |This
 is not a re|
00000010  61 6c 20 61 6e 73 77 65  72 2e 0a                 |al an
swer..|
0000001b
```

This shows the contents of the file, printed out as bytes in hex. (Remember, each byte is two hex characters.) Notice that the last hex character in the file is `0x0a`, which is the newline character in ASCII. So this `defuser.txt` *does* end with a newline. However, if you type `hexdump -C defuser.txt` and your file *does not* end with a newline, you'll instead get something like this:

```
00000000  54 68 69 73 20 69 73 20  6e 6f 74 20 61 20 72 65  |This
 is not a re|
00000010  61 6c 20 61 6e 73 77 65  72 2e                    |al an
swer.|
0000001a
```

Notice that the last character is now `0x2e`, which corresponds to the `.` character in ASCII (the period in the sentence). To add a newline to the end of your file, you can use the command:

```
$ echo "" >> defuser.txt
```

The `echo` command prints out its argument, and the `>>` redirects that argument to the file `defuser.txt`. By default, `echo` appends a newline character to its output. So if you supply an empty string to `echo`, it will just add the newline character to the end of the file.

To avoid accidentally detonating the bomb, you will need to learn how to single-step through the assembly code in `gdb` and how to set breakpoints. You will also need to learn how to inspect both the registers and the memory states. One of the

nice side-effects of doing the lab is that you will get very good at using a debugger. This is a crucial skill that will pay big dividends the rest of your career.

During this lab, we **strongly recommend** that you keep notes of the steps you took in solving each stage. This will be immensly helpful in helping you to keep track of what's stored at important addresses in memory and in registers at different points in the program's execution. (A good strategy for this might be to keep a notetaking app open on your computer so you can copy and paste values between it and `gdb`.)

Helpful Information

There are many online resources that will help you understand any assembly instructions you may encounter. In particular, the instruction references for x86-64 processors distributed by Intel and AMD are exceptionally valuable. They both describe the same ISA, but sometimes one may be easier to understand than the other.

Important Note: The instruction format used in these manuals is known as “Intel format”. This format is **very different** than the format used in our text, in lecture slides, and in what is produced by `gcc`, `objdump` and other tools (which is known as “AT&T format”. You can read more about these differences in our textbook (p.177) or on Wikipedia (https://en.wikipedia.org/wiki/X86_assembly_language#Syntax). ***The biggest difference is that the order of operands is SWITCHED.*** This also serves as a warning that you may see both formats come up in web searches.

- Intel Instruction Reference (https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf)
- AMD Instruction Reference (http://developer.amd.com/wordpress/media/2008/10/24594_APM_v3.pdf)

x86-64 Calling Conventions

The x86-64 ISA passes the first six arguments to a function in the following registers (in order): `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`.
The return value of a function is passed in `rax`.

Using sscanf

First let's look at `scanf` ("scan format"), which reads in data from `stdin` (the keyboard) and stores it according to the format specifier into the *locations* pointed to by the additional arguments:

```
int i;
printf("Enter a number: ");
scanf("%d", &i);
```

- The `printf` prints a prompt, once the user enters in a number and hits enter `scanf` will store the input from stdin into `i` with the format of an integer. Notice how `scanf` uses the address of `i` as the argument.

Lab 2 uses `sscanf` ("string scan format"), which is similiar to `scanf` but reads in data from a string instead of `stdin` :

```
char* mystring = "123, 456";
int a, b;
sscanf(mystring, "%d, %d", &a, &b);
```

- The first argument, `mystring` , is the input string.
- The second argument, `"%d, %d"` is the format string that contains format specifiers to parse the input string with.
- After matching the input string to the format string, the extracted values are stored at the addresses given in the additional arguments. After this code is run, `a = 123` and `b = 456` .

Reference information can be found online for sscanf (http://www.cplusplus.com/reference/cstdio/sscanf/), scanf (http://www.cplusplus.com/reference/cstdio/scanf/), and printf (http://www.cplusplus.com/reference/cstdio/printf/).

Tools (Read This!!)

There are many ways of defusing your bomb. You can print out the assembly and examine it in great detail without ever running the program, and figure out exactly what it does. This is a useful technique, but it not always easy to do. You can also run it under a debugger, watch what it does step by step, and use this information to defuse it. Both are useful skills to develop.

We do make one request, please do not use brute force! You could write a program that will try every possible key to find the right one, but the number of possibilities is so large that you won't be able to try them all in time.

There are many tools which 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 bomb, and hints on how to use them.

- `gdb` : The GNU debugger is a command line debugger tool available on virtually every platform. You can trace through a program line by line, examine memory and registers, look at both the source code and assembly code (we are not giving you the source code for most of your bomb), set breakpoints, set memory watch points, and write scripts. Here are some tips for using `gdb` .
  - See our GDB page (../gdb/) for guides and resources.
  - For other documentation, type `help` at the `gdb` command prompt, or type `man gdb` , or `info gdb` ; at a Unix prompt. Some people also like to run `gdb` under gdb-mode in emacs

- `objdump -t bomb > bomb_symtab` : This will print out the bomb's symbol table into a file called `bomb_symtab` . The symbol table includes the names of all functions and global variables in the bomb, the names of all the functions the bomb calls, and their addresses. You may learn something by looking at the function names!
- `objdump -d bomb > bomb_disas` : Use this to disassemble all of the code in the bomb into a file called `bomb_disas` . You can also just look at individual functions. If you would like to print out the assembly you can use this command from a linux machine in the CSE lab (or attu) to print to the printer in 002 in two column, two-sided format:

  ```
  $ enscript -h -2r -Pps002 -DDuplex:true bomb_disas
  ```

  Reading the assembly code can tell you how the bomb works. Although `objdump -d` gives you a lot of information, it doesn't tell you the whole story. Calls to system-level functions may look cryptic. For example, a call to `sscanf` might appear as: `8048c36: e8 99 fc ff ff call 80488d4 <_init+0x1a0>` To determine that the call was to `sscanf` , you would need to disassemble within `gdb` .
- `strings -t x bomb > bomb_strings` : This utility will print the printable strings in your bomb and their offset within the bomb into into a file called `bomb_strings` .

Looking for a particular tool? How about documentation? Don't forget, the commands `apropos` and `man` are your friends. In particular, `man ascii` is more useful than you'd think. If you get stumped, use the course's discussion board.

Hints

If you're still having trouble figuring out what your bomb is doing, here are some hints for what to think about at each stage:

1. Comparison
2. Loops
3. Switch statements
4. Recursion
5. Pointers and arrays
6. Linked lists

Lab 2 Reflection

REMINDER: You will need to use the CSE Linux environment in order to get addresses that are consistent with our solutions.

Start with a *fresh* copy of `lab0.c` and examine `part2()` using the following commands:

```
$ wget https://courses.cs.washington.edu/courses/cse351/19sp/labs/
lab0.c
$ gcc -g -std=c99 -o lab0 lab0.c
$ gdb lab0
(gdb) layout split
(gdb) break fillArray
(gdb) break part2
(gdb) run 2
```

Now answer the following questions. You will find the following GDB commands useful: `nexti`, `finish`, `print`, and `refresh`.

1. At which memory addresses are the variables `value` and `array` from `part2()` stored? [2 pt]
2. Which two registers (be specific as to the register width) determine if the `assert()` call in `fillArray()` fails? [2 pt]
3. Give the *relative* addresses (*i.e.* of the form `<function+#>`) of the instructions that perform the initalization and update statements for the for-loop in `fillArray`. [2 pt]
4. Consider the `lea` instruction at the relative address `<part2+18>`. Give an equivalent/replacement `lea` instruction that does not use `%rbp`. [2 pt]
5. What address is the string `"*** LAB 0 PART 2 ***\n"` stored at in memory? Which part of the memory layout is this? [2 pt]

Submission

Submit `defuser.txt` and `lab2reflect.txt` to the assignments page (../submit.php).

It is important to make sure that `defuser.txt` obeys the folowing formatting rules, otherwise our grading script is likely to conclude you defused zero bombs:

- Put your answer for each phase in one line. Your answer for phase 1 should be in the first line, answer for phase 2 on the second line, and so on.
- Do ***not*** put your name or other information at the top of the file. Again, you want the first line in the file to be your answer for phase 1.
- Do ***not*** add numbering or other “comments” for your answers (e.g. `1. This is my answer for phase 1`).
- Make sure all your answers, *including the last one*, have a newline character afterwards, so even your last-phase answer is a complete line *with* a newline. This last newline is important for our grading script even though you will not notice the difference in your own testing.