ENS Security course 2019
Bomb Lab: Defusing a Binary Bomb

Carnegie Mellon Univ. 15213 staff, adapted by Guillaume DIDIER

Assigned:	Fri, Oct. 4
Report Due:	Fri, Nov. 29, 11:59PM
Presentation:	Fri, Dec. 6
Final Report:	Fri, Dec. 13, 11:59PM

1 Introduction

The nefarious *Dr. Evil* has planted a slew of "binary bombs" on our 64-bit 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`. 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 you a bomb to defuse. Your mission, which you have no choice but to accept, is to defuse your bomb before the due date (and as soon as possible). Good luck, and welcome to the bomb squad!

Step 1: Get Your Bomb

You should have recieved a link to download a VM with the bomb alongside those instructions.

Login to the virtual machine using the provided crdentials and run: `tar -xvf bomb`k`.tar`. This will create a directory called `./bomb`k with the following files:

- `README`: Identifies the bomb and its owner.
- `bomb`: The executable binary bomb.
- `bomb.c`: Source file with the bomb's main routine and a friendly greeting from Dr. Evil.

Warning: If you let your browser expand your `bomb`k`.tar` file automatically, you risk resetting the bomb's execute bit. Always expand an archive file using the `tar` command on a Linux Machine.

Step 2: Defuse Your Bomb

Your job for this lab is to defuse your bomb.

There are several tamper-proofing devices built into the bomb as well, or so we hear.

You can use many tools to help you defuse your bomb. Please look at the **Hints** section for some tips and ideas. The best way is to use your favorite debugger to step through the disassembled binary.

Although phases get progressively harder to defuse, the expertise you gain as you move from phase to phase should offset this difficulty. However, the last phase will challenge even the best students, so please don't wait until the last minute to start.

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

```
linux>  ./bomb psol.txt
```

then it will read the input lines from `psol.txt` until it reaches EOF (end of file), and then switch over to `stdin`. 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.

To avoid accidentally detonating the bomb, you will need to learn how to single-step through the assembly code and how to set breakpoints. You can safely exit your bomb at any time by typing `ctrl-c` (simultaneously pressing the ctrl and c keys).

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.

Warning: You should never use your debugger to jump directly to a particular phase. Doing so can cause your bomb to explode silently.

Logistics

This is an group project for the team assigned to the bomb squad. Please see the project instruction on the course website.

Note: Your defusing strings may not contain the single quote character `'`. There is a known bug in the backend autograder that causes it to get confused by strings with these characters. The bomb program will reject any such string.

Hints *(Please read this!)*

There are many ways of defusing your bomb. You can 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. This is probably the fastest way of defusing it.

However, do not use a script to automatically brute-force solutions to your bomb. You are intelligent students, and you can probably figure out how to write a program to try every possible key to find the right one. But doing so won't teach you anything about assembly, and we reserve the right to deduct points if we suspect you of using a script in this manner.

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.

  The CS:APP textbook Web page at

  `http://csapp.cs.cmu.edu/3e/students.html`

  has a handy 1-page `gdb` command summary for x86-64 that you can print out and use as a reference. It also contains a link to Prof. Norm Matloff's `gdb` GDB tutorial.

  Here are some other tips for using `gdb`.

  - To keep the bomb from blowing up every time you type in a wrong input, you'll need to learn how to set breakpoints.
  - For online 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`.
  - Many of you have asked about what some functions inside of bomblab do, such as `sscanf`. For these functions, we suggest that you do not step into these functions, and instead, go onto the next instruction. You can assume the sscanf works as expected, and you are allowed to search what generic functions do, such as `printf`, `sscanf`, `malloc`, `free`, etc. (Typically you can use the manual page or the C section of `cppreference.com`) Stepping into sscanf can be a learning experience, but it's extremely long and tedious. Your mileage may vary. A common sign of a library function is that they have `@plt` at the end of the function call name in assembly. This is due to library address space randomization, which you may learn by looking at the relevant lectures of 213 or chapter of the book. You do need to undestrand this to do the lab.

- `objdump -t`

  This will print out the bomb's symbol table. 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! For example, you could discover a function called "`explode_bomb`", which would be a good place to set a breakpoint to keep the bomb from blowing up.

- `objdump -d`

  Use this to disassemble all of the code in the bomb. You can also just look at individual functions. Reading the assembler 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 are displayed in a cryptic form. 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`

  This utility will display the printable strings in your bomb.

Looking for a particular tool? How about documentation? Don't forget, the commands `apropos`, `man`, and `info` are your friends. In particular, `man ascii` might come in useful. If you get stumped, feel free to ask the teaching staff for help.