EAST TENNESSEE STATE UNIVERSITY

CSCI 3160 – Computer Systems

Bomb Lab

Note: this lab adapted from Bryant and O'Hallaron's *Computer Systems: A Programmer's Perspective (3E)***.** The acronym "CS:APP" refers to this text (which serves as our primary source for the back half of the course).

Introduction

The nefarious ***Dr. Evil*** has planted a slew of "binary bombs" on D2L. 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 me to deal with, so I am giving each student a bomb to defuse. Your mission, which you have no choice but to accept, is to defuse your bomb before the due date and time posted on D2L. Good luck, and welcome to the bomb squad!

Individual Assignment

While I am OK with students collaborating to figure out general workflows for defusing each stage of the bomb, students have their own unique bombs attached to their individual D2L Dropbox. Each student should submit a UNIX-friendly text file containing their unique solutions. While there's not a "minimum" number of required phases to solve, the more phases a student can disarm, the more evidence the student possesses to argue towards meeting and exceeding learning outcomes 1., 2., and 4.

Lab Procedure

Get Your Bomb

1. Download `<your last name>.tar.gz` from D2L (***Dr. Evil*** attached your unique bomb to the Dropbox—try viewing your feedback) and copy it to a directory on your Linux VM that you plan to work out of (e.g., `/home/<your username>/bomblab`).

2. Run the tar command to un-tar (like unzipping a .zip file):

```
$ tar xzvf <your bomb>.tar.gz
```

3. After un-zipping/un-tarring, look in your bomblab directory and note the files—you will **not** modify any files:
   - `README` and `ID`: identifies the bomb and its owner
   - `bomb`: the executable binary bomb
   - `bomb.c`: the source file with the bomb's main routine and a "friendly" greeting from ***Dr. Evil***

Defuse Your Bomb

Your job for this lab is to defuse your bomb. 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.

Defusing the bomb occurs in "6" phases. 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,

```
$ ./bomb defuse.txt
```

then it will read the input lines from `defuse.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 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.

Evaluation

I will check if you successfully defused ***Dr. Evil***'s binary bomb (or not). I do not have a set number of phases each person should get through, but, as mentioned in the Introduction, more phases solved is more evidence towards meeting/exceeding learning outcomes.

Deliverable

Submit your defuse solution (as a UNIX text file) to the Bomb Lab Dropbox on D2L by the posted due date.

Hints

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.

Please do not use brute force! ***Dr. Evil*** knows you could write a program that will try every possible key to find the right one; however, this will not work out since ***Dr. Evil*** uses (at least) alphanumeric strings with (up to, or more) 80 characters each, which means you would need to guess ***at least*** 62^{80} times per phase (in the worst case) to find the string (that assumes no punctuation, spaces, …).

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, this 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 (I am not giving you the source code for most of your bomb, since ***Dr. Evil*** did not give it to me), set breakpoints, set memory watch points, and write scripts.
    - Check out https://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html for a great guide and tutorial on using `gdb`.
- `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!

- `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 does not 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. `info gas` will give you more than you ever wanted to know about the GNU Assembler. Also, the web may also be a treasure trove of information. If you get stumped, feel free to ask me for help.