CS 270 Systems Programming Spring 2021

Project 1: Defusing a Binary Bomb

Out: *12 February 2021* ***Due:*** *26 February 2021*

1 Introduction

The nefarious *Dr. Evil* 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`. 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 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. Good luck, and welcome to the bomb squad!

Step 1: Get Your Bomb

You can obtain your bomb by pointing your Web browser at the following URL **WHILE CONNECTED TO THE VPN**:

`http://calvert.cs.uky.edu:21270/`

This will display a binary bomb request form for you to fill in. Enter your user name and email address and hit the Submit button. The server will build your bomb and return it to your browser in a `tar` file called `bomb`k.tar, where k is the unique number of your bomb.

> **Note:** The request page **times out** if you don't submit the form within a limited time (about 20 seconds). If that happens, you will get a cryptic failure message from your browser. Then you will have to reload the page and start over. So be prepared to enter your username (e.g., `abcd129`) and email address quickly.

Save copy `bomb`k.tar file to a directory **ON YOUR VIRTUAL MACHINE**, in which you plan to do your work. Then give (again, on your virtual machine) the command: `tar xvf bomb`k`.tar`. This will create a subdirectory called `bomb`k, containing the following files:

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

If for some reason you request multiple bombs, this is not a problem. Choose one bomb to work on and delete the rest. Please be sure to enter your correct userid, or you may not get credit.

Step 2: Defuse Your Bomb

Your job for this lab is to defuse your bomb.

You must do the assignment on your VM. In fact, there is a rumor that Dr. Evil really is evil, and the bomb will always blow up if run on a machine that is not a CS 270 VM. There are several other 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 disassemble the binary and step through it.

Each time your bomb explodes it notifies the bomblab server, and you lose 1/2 point (up to a max of 20 points) in the final score for the lab. So there are consequences to exploding the bomb. You must be careful!

The first four phases are worth 10 points each. Phases 5 and 6 are a little more difficult, so they are worth 15 points each. So the maximum score you can get is 70 points.

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. Also, if you run your bomb with a command line argument, something like:

```
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 (and thus losing points), you will need to know how to single-step through the assembly code and how to set breakpoints (i.e., places where `gdb` halts execution and prompts for input). You will also need to learn how to inspect both the registers and the memory states. One of the nice effects of doing the lab is that you will get good at using a debugger. This is a crucial skill that will pay big dividends the rest of your career.

Logistics

This is an individual project. All handins are electronic. Clarifications and corrections will be announced viad Canvas.

Handin

There is no explicit handin. The bomb will notify the instructor automatically about your progress as you work on it. You can keep track of how you are doing by looking at the class scoreboard at:

```
http://calvert.cs.uky.edu:21270/scoreboard
```

(Again, you must be connected to the campus VPN to access this webpage.) The website is updated continuously to show the progress for each bomb.

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.

Whatever you do, *please do not use brute force!* You could write a program that will try every possible key to find the right one. But this is no good for several reasons:

- You lose 1/2 point (up to a max of 20 points) every time you guess incorrectly and the bomb explodes.
- Every time you guess wrong, a message is sent to the bomblab server. You could potentially overload the network/server with these messages, and cause the system administrators to revoke your computer access.
- We haven't told you how long the strings are, nor have we told you what characters are in them. Even if you made the (incorrect) assumptions that they all are less than 80 characters long and only contain lower-case letters, you would need 26^{80} guesses for each phase. This will take a very long time to run—the sun will burn out before you finish; even worse, you will not get the answer before the assignment is due.

There are many tools 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. Probably the most useful is `gdb`, the GNU debugger. Lab exercise 1 introduced you to this tool. (If you missed Lab 1 in class on Friday 12 Feb, you should do it on your own before starting this project.) You are **strongly encouraged** to play with it and become familiar with its capabilities. The CS 270 home page in Canvas has pointers to some resources about `gdb`, but there is no substitute for experimenting with it. You can use it to trace through the execution of a program line by line, examine memory and registers, look at the source code and assembly code (we are not giving you the source code for most of the bomb, so you won't be able to see that) set breakpoints, set memory watch points, and even write scripts.

Here are some additional tips for using `gdb`.

- To keep the bomb from blowing up every time you type in a wrong input, you'll want 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. If you use the `emacs` editor, it has a `gdb-mode` that some folks like to use.
- The `disassemble` ⟨*function name*⟩ command will print out a summary of the machine instructions of the given function.

Here are some other tools (separate from `gdb`) that may prove useful:

- `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 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 compiled into 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 as` 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 your instructor for help.