CS61, Fall 2011
Assignment 2: Defusing a Binary Bomb

Assigned: Thursday, Sept. 8th, Due: Thursday, Sept. 22, 11:59PM

Important Note – Academic Honesty

As with all of your work in this course, academic honesty is expected at all times. In particular, this means a few specific things:

1. You must work alone on this lab (partners will be allowed on the programming assignments later in the course.) You may discuss hints and techniques for solving the assignment with other students, but all the actual work on your assignment must be your own. Explicit sharing of answers is not permitted (and, in this assignment, will not be useful, as the code for the assignment is generated uniquely for each student.)
2. You may draw on any resources provided to you by the CS61 course staff, and linked to the CS61 website. However, use of materials you find on other websites, or software tools that we do not authorize, is *not permitted*. (An example would be a reverse compiler that translates assembler to C. That would kind of defeat the purpose of this lab!)
3. Any attempts to circumvent the automated checking and security mechanisms in the lab code distributed to you (see below) will also be construed as cheating.

If you have questions about whether a given source of information or approach to this lab is permissible, contact the course staff.

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: Set up a CS61 Virtual Machine

You will complete this and the remaining labs for this course on your own virtual machine on the SEAS cloud. The first step in this lab is to set up and connect to your virtual machine for this lab. Instructions for this are provided on the CS61 website at `http://cs61.seas.harvard.edu/wiki/Infrastructure`.

Each student will attempt to defuse his or her own personalized bomb. Each bomb is a Linux binary executable file that has been compiled from a C program. Your bomb has been placed in your CS61 home directory, which you will see upon logging into your virtual machine. Your bomb is in a folder called `hw1`. This folder contains the following files:

- `README`: Identifies the bomb and its owner. This file contains the unique ID of your bomb, which you will need to track your progress online (see below.)
- `bomb`: The executable binary bomb.
- `bomb.c`: Source file with the bomb's main routine.

If you cannot find this folder or its contents are not correct, please notify the course staff immediately.

Step 2: Defuse Your Bomb

Your job is to defuse the bomb.

You can use many tools to help you with this; 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.

Each time your bomb explodes it notifies the staff, and you lose 1/4 point (up to a max of 10 points) in the final score for the lab. So there are consequences to exploding the bomb. You must be careful!

Each phase is worth 10 points, for a total of 60 points.

The phases get progressively harder to defuse, but 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 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.

Logistics

Any clarifications and revisions to the assignment will be posted on the class web page.

You should do the assignment on your CS61 VM. In fact, there is a rumor that Dr. Evil really is evil, and 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.

Hand-In

There is no explicit hand-in. The bomb will notify course staff automatically after you have successfully defused it (and also each time it explodes!) You can keep track of how you (and others) are doing by visiting `http://cs61.seas.harvard.edu/lab2/fa11.html`. This web page is updated continuously to show your progress.

Late Days

Each student has five "late days" which can be applied to any of the labs. A late day extends the due date by 24 hours. At most three late days can be used on any single lab. You must email course staff before the due date to request late days. In cases of medical or other emergencies that interfere with your work, have your resident dean or proctor contact the instructor.

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.

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 this is no good for several reasons:

- You lose 1/4 point (up to a max of 10 points) every time you guess incorrectly and the bomb explodes.
- Every time you guess wrong, a message is sent to the staff. You could very quickly saturate the network 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 (wrong) assumptions that they all are less than 80 characters long and only contain letters, then you will have 26^{80} guesses for each phase. This will take a very long time to run, and you will not get the answer before the assignment is due.

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 (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`.

  - To keep the bomb from blowing up every time you type in a wrong input, you'll want to learn how to set breakpoints.
  - You may also want to review the lecture notes from the first day to see how to examine memory during program execution and why doing so may be useful. The `gdb` resources below will also be helpful for this.
  - The CS:APP Student Site at `http://csapp.cs.cmu.edu/public/students.html` has a very handy single-page `gdb` summary (look at Prof. Norm Matloff's GDB tutorial and the Quick GDB Reference).
  - 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`

  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 in your bomb.

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