CS 367 Fall 2018
Project 2: Binary Bomb!
Due: October 30 at end of the day (midnight)

This is an individual assignment.

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.

Step 1: Get Your Bomb

Each student will attempt to defuse their own personalized bomb. Each bomb is a Linux binary executable file that has been compiled from a C program. While each bomb is unique, the phases follow basic patterns. To obtain your bomb, you need to be on a machine that can connect to the machine `zeus-1.vse.gmu.edu`

Note that `zeus-1` is a machine within the `zeus` server cluster. You can access `zeus-1.vse.gmu.edu` just like you did with `zeus`. Once you are able to connect to `zeus-1.vse.gmu.edu`, point your Web browser to the bomb request daemon at

http://zeus-1.vse.gmu.edu:15225

Fill out the HTML form with your name and GMU email address, and then submit the form by clicking the "Submit" button. The request daemon will build your bomb and return it immediately to your browser in a tar file called `bomb`$_k$`.tar`, where `k` is the unique number of your bomb.
Save the `bombk.tar` file on zeus and unpack it (`tar -xvf bombk.tar`).

This will create a directory called `./bombk` with the following files:

- `README`: Identifies the bomb and its owner.
- `bomb`: The executable binary bomb. This executable will only run on zeus.vse.gmu.edu
- `bomb.c`: Source file with the bomb's main routine.

> When you defuse a phase of the bomb (as described below), the program notifies the instructor. However, each time your bomb explodes on `zeus` it notifies the instructor, and you lose `0.5` points (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!

Also, if you make any kind of mistake requesting a bomb (such as neglecting to save it), simply request another bomb. We see who checks out every single bomb, so let us know your old bomb and new bomb.

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 (`gdb`) to step through the disassembled binary.

Each of the first four phases is worth 10 points, for a total of 40 points. The 5th and 6th phases are more difficult and are worth 15 points. Defusing all six phases gives you 70 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**. We 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 use **gdb** 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.

Hand-In

There is no explicit hand-in. The bomb will notify your instructor automatically after you have successfully defused it on **zeus-1.vse.gmu.edu**. You can keep track of how you (and other students) are doing by looking at

http://zeus-1.vse.gmu.edu:15225/scoreboard

This web page is updated continuously to show the progress of the class.
Note that this web page is only accessible from a machine in the VSE labs or if you have connected to the VSE labs using a VPN.

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 is 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. You may not modify your bomb in any way to defuse it; your inputs are attempted on a fresh copy of your bomb for grading purposes so you can't circumvent it this way.

We 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 because 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 2680 guesses for each phase. This will take a 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 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.
    - The CS:APP Student Site at http://csapp.cs.cmu.edu/public/students.html has a very handy single-page gdb summary.
    - 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 command **man** is your friend. You can use this to look up information on system functions that the binary executable might use. In particular, **man ascii** might come in useful.

Advice

Based on my experience defusing bombs, I have the following advice:

- I print out all of the code associated with a particular phase when I start to diffuse it. As I work, I annotate this code with information as I figure out what is going on.
- As you will notice from examining **bomb.c**, the user input is sent into the phase as a single string. One of the first things the phase will do is try to 'parse' it into the form it is expecting.
- When you are trying to figure out what values are needed for a phase (particularly if the values are integers), try 'interesting' numbers like **42**, **-17**, ... first. The reason for this is that if you see one of these numbers appear in a register or in a memory location, chances are pretty good that was your input.
- Don't bother trying to step through system functions. This is generally a waste of time. If you don't know what the function does, use **man** to investigate the parameters and return information. Then examine the parameters that are sent and the return value.
- If you end up inside a function that you don't want to step through, the **gdb** command **finish** will take you to the end.
- The bomb functions typically have a generic name like **fn7** or **phase2**. It has been my experience that if a function has a descriptive name, you can assume that the function behaves as expected and that they are NOT trying to trick you.
- Understanding parameter passing is critical to this assignment. Whenever you see a reference to **%esp** in the lines before a call, this involves placing a parameter on the stack. It is never a bad idea to figure out what the value of the parameter is. In the called procedure, references to **%ebp** involve use of the parameters in the code.
- It may not be important to understand exactly what every statement is doing - the goal is to avoid the **explode_bomb** calls. I tend to step through the code until just before one of these calls and then focus on what conditions are not being met. You can also set a breakpoint at **explode_bomb** to ensure you don't accidently execute it.