CSC 373 Winter 2020 – Professor Lytinen
Homework Assignment 5: Defusing a Binary Bomb
Due date/time as specified on D2L

1 Introduction

The goal of this assignment is for you to improve your understanding of how programs run at a systems level. In this assignment, you will examine an executable file with low-level tools such as the Gnu debugger (`gdb`) in order to understand what the executable does and how it does it. You will **not** have access to the C source code for parts of the program.

The assignment will be graded on a scale of 0 to 10 There are 5 phases in the assignment, each worth 2 points. There is also an opportunity for up to 2.5 points of extra credit.

2 The bomb program

You will find a file called `bomb.tar` in the D2L submissions folder for this assignment. the .tar file contains an executable file called `bomb`. (Using tar fooled D2L into thinking I was not placing an executable into a submissions box) The executable was created using gcc version 4.4.7 with the `-O2` option. The executable runs on `cdmlinux.cdm.depaul.edu`. A student who uses Ubuntu told me that it also executes in that environment, but it is unlikely to run on a Mac. If it does not run on the platform that you normally use, then you will have to use `cdmlinux.cdm.depaul.edu` for the assignment.

As a reminder, the way to uncompress a tar file is to use the Linux `tar` command, with the flags `xvf`. in this case:

```
tar xvf bomb.tar
```

After you run this command, you should see the `bomb` executable in the same directory as the tar file. You will see a couple of other C files, which may help you to understand how the bomb works and thereby defuse it. Since the bomb itself is an executable, you should be able to run it once you've transferred it to your own Linux environment by typing:

```
./bomb
```

The bomb program reads a sequence of inputs from the console or a .txt file. The first input is interpreted to be the user's ID. Each subsequent input is fed into one of the 5 "phases" of the bomb. The phases are implemented as functions `phase_1`, `phase_2`, etc. For each phase, a user's input will first be read by the bomb as a string (using the getline C function). Any further tokenization of parsing of this string will be done by the phases, or their supporting functions. You may notice use of verb!sscanf!! throughout many of the phases of the code. The `sscanf` function works the same as `scanf`, except it receives a string as though it were user input. The string is passed as the first parameter to `sscanf`. For example:

```
int x;
char str1[100], str2[100];
double humid;
char input[] = "120 degrees 87.5 humidity";
sscanf(input, "%d %s %f %s", &x, str1, &humid, str2);
```

In each phase, certain inputs will be "accepted", in which case the phase is "defused", and the program will proceed to the next phase. However, most inputs will cause the phase to "explode", as indicated by this output:

```
BOOM!!!
The bomb has blown up.
```

If you type 6 lines of input (your D2L user ID, plus one line of input for each phase) without seeing the above output, then you have defused the whole bomb. Note that for some phases, the user ID will have an effect on what input(s) defuse a phase.

3 Your task

For this assignment, you must do the following:

1. Determine a sequence of 6 inputs, the first being your D2L user ID (the ID name, not the number), that will defuse the bomb. Each correct input string for each of the phases will be worth 1 points (there is no credit for typing your user ID).

2. Determine a sequence of 6 inputs, this time the first being your D2L ID number (not the name), that will defuse the bomb. Each correct input string will be worth 1 point (again, there is no credit for typing your ID number).

3. For **extra credit**, write 5 C functions (`phase_1`, `phase_2`, ...) which emulate the phases in the `bomb` executable. Each C function will be worth .5 points, for a total of 2.5 possible extra credit points. Note that I will test your phases by using a different user ID than you will use in (1) and (2) above, so you should make sure that your C code emulates the 5 phases for a variety of user IDs.

All 5 functions `phase_1`, `phase_2`, `phase_3`, etc. are passed two parameters: the input you typed, and an integer. The integer is calculated based on your user ID, and will affect the proper input for the phases 5. Therefore not all students will have the same solutions for this assignment.

4 Using the debugger

You may choose to run the bomb through the GNU debugger. This will help you figure out what the assembly language code is doing. Because of the way I compiled the bomb executable, when `gdb` starts up it may display some error messages which you can ignore. Eventually, you should see this prompt:

```
(gdb)
```

In debug mode, you can type commands after the `(gdb)` prompt to disassemble a function (e.g., `disas phase_1`), set a breakpoint (e.g., `break *phase_1`), inspect a register (e.g., `print/d $rax`, `x/s $rax`). Note that `print` and `x` do different things; print displays the contents of a register, whereas x assumes the register contains a memory address and in essence dereferences this pointer. Once you have set breakpoints or examined code, you can start the program by either typing

```
(gdb) run
```

Since the phases are somewhat more complex than the sample assembly language functions we've seen so far, it is likely that you will not be able to determine the required inputs just by reading the functions. You will probably spend most of your time on this assignment runing the bomb under the debugger (gdb) to examine the disassembled binary, set breakpoints, examine registers and memory, etc. I will say more about `gdb` as we progress toward the assignment due date.

5 Extra credit

As stated above, you can earn extra credit points by writing C code to emulate each of the 5 phases. I will release additional files that will assist you in writing this C code early next week.