Assignment #3: Bomb Lab (due on Wed, Feb. 27, 2019 by 11:59pm)

Introduction

This assignment gives you a binary program containing *“bombs”* which trigger a ping to our server (and make you lose points). Your goal is to set breakpoints and step through the binary code using `gdb` to figure out the program inputs that defuse the bombs.

The binary program 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.

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.

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 for the rest of your career.

Instructions

You must complete the assignment using the class VM. **Your virtual machine must be connected to the internet,** as the program will connect to our server when it explodes or when a phase is defused.

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 `gdb` to step through the disassembled binary.

You can start the debugger with:

```
$ gdb bomb
GNU gdb 7.11.1

(gdb) layout asm
```

Then, you can move between the two parts of the screen by pressing `CTRL+x` and then `o`. Notice that:

- When the focus is on the upper half of the screen, the `UP`/`DOWN` keys scroll through the disassembled code.
- When the focus in on the lower half of the screen, the same keys move through the history of previous commands.

You can set a breakpoint using the `b` command:

```
(gdb) b phase_1
Breakpoint 1 at 0x400f1f
(gdb)
```

Make sure to set breakpoints on the routine that causes a bomb explosion. (It’s up to you to guess its name in the assembly code.) You can start running the program with the `r` command, and step through its instructions using `ni` or `si` (pressing `ENTER` repeats the previous command).

When the program reads input from the console, you can type the solution (which you should figure out by inspecting the assembly code and memory/registers). If you enter the correct solution to a phase, the `bomb` program will save it at the end of a text file named `sol.txt`. You can use this file at startup by running your bomb with:

```
$ ./bomb sol.txt
```

or, inside of `gdb`, with:

```
(gdb) r sol.txt
```

This will use each line of `sol.txt` as input to a phase, so that you don't have to re-type previous solutions to get to the next phase.

Evaluation

When the bomb explodes, it notifies our server and you lose points in the final score. Nonetheless, **you will always gain points for completing a phase regardless of how many times the bomb has exploded.**

These are the precise rules:

- There are a total of 70 points: 10 for each of the first 4 phases, 15 for each of phases 5 and 6.
- There is 1 free explosion (no points lost) in each of phases 1-4, and 3 free explosions in each of phases 5-6.
- Each additional explosion costs you 0.5 points.
- These penalty points are not removed from your score until you complete the phase! When you complete the phase, you receive the *maximum* between `0.4 * phase_points` and `phase_points - penalty_points`.
- We don't count explosions in phases that you already completed (providing the correct input).

This means that:

- Losing points on the next phase does not affect your current score.
- When you complete the next phase, you always get at least 40% of the points.
- You get more points if you used few explosions to complete the phase.

For example:

- If you incur 13 or more explosions before you solve phase 1, but later solve it successfully, you still gain 4 points.
- If you have only 5 explosions while working on phase 1, and then solve it, you get eight points for that phase (1 free explosion and 4 explosions costing you 2 points).

Finally: keep in mind that you can avoid all explosions by setting breakpoints on "bomb explosion functions" using `gdb`.

Logistics

This is an individual project. All handins are electronic and automatically logged by the software. Clarifications and corrections will be posted on the course Piazza.

Handout Instructions

There is no explicit handin. The bomb will notify our server automatically each time it explodes or you defuse a phase.

We have included the ability for you to check your score and last completed phase. Running the script `check_grade.sh` provided in your repository will output your attempts at each phase, as well as the one you last completed. Make sure you are connected to the Internet when running the script.

Since our server logs your successful phase completions or bomb explosions, we will handle grace day usage by just looking at the timestamps. You do not need to do anything special to use grace days. However, don't attempt to pass any more phases after the due date as we will count those attempts as a grace day usage.

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 solving the assignment.

We do make one request: Please do not use brute force! You could write a program that tries every possible key to find the right one. But this is not good for several reasons:

- You could lose points every time you guess incorrectly and the bomb explodes.
- Every time you guess wrong, a message is sent to our server. You could very quickly saturate the network with these messages, and cause the system administrators to revoke our server's 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 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.

- The GNU debugger (`gdb`), 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 your bomb), set breakpoints, set memory watch points, and write scripts. The CS:APP web site has a very handy single-page gdb summary that you can print out and use as a reference.
- `objdump -t` 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` will 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. Keep in mind that, 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` will display the printable strings in your binary program.

Acknowledgements. This lab was developed by the authors of the course textbook and their staff. It has been customized for use by this course.

CS356