CS 367 Spring 2023
Project 4: Defusing a Binary Bomb
Due: Friday, May 5th, 2023 (11:59pm EST)

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.

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

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 **zeus-1.cec.gmu.edu**. For this project, you **must** connect to **zeus-1**. In case you have problems connecting with your bomb, double-check the server.

Logon to **zeus-1.cec.gmu.edu** using ssh and your GMU userid/password. Remember that you need to be on a VSE lab machine, on zeus or mason servers, or be connecting remotely through the VPN software.

The bottom line is that the technique you've been using to connect to zeus in the previous assignment should also work for this project.

You can obtain your bomb by pointing your Web browser at:

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

This will display a binary bomb request form for you to fill in. Enter your user name and GMU 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.

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-1.cec.gmu.edu
- `bomb.c`: Source file with the bomb's main routine.

Important Notes:

1. Due to the heavy load on zeus, sometimes it may take up to 60-90 seconds before the bomb file is loaded on your browser. Please be patient. If you are appropriately connected (VSE lab or via VPN) and the site seems to be down, please send a message on Piazza immediately so others can chime in or an instructor can restart the server.

2. As some students postpone working on the project to the last couple of days, zeus may become slower and slower towards the deadline. To avoid the frustration, we strongly recommend that you complete as many phases as possible as early as possible.

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 you need to let us know your old bomb and new bomb (send an e-mail to your recitation GTA and CS 367 section instructor).

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 project. So there are consequences to exploding the bomb. You must be careful!

Step 2: Defuse Your Bomb

You must do the assignment only on zeus (specifically, zeus-1; not any other machine or zeus-2!). 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 we hear.

Your job for this project is to defuse your bomb. You can use several tools available to you on Zeus 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 **gdb** to step through the disassembled binary. (You may **not** use external tools to solve the bomb.)

Each time your bomb explodes it notifies the bomb server, and you lose 1/2 point (up to a max of 10 points) in the final score for the project. So there are consequences to exploding the bomb. You must be careful!
The first four phases are worth 15 points each. Phases 5 and 6 are a little more difficult, so they are worth 20 points each. So the maximum score you can get is 100 points. (Some of you may have heard about a "secret phase"... What secret phase?)

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. 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 project 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

This is an **individual** project. Clarifications and corrections will be posted on the Piazza forum if needed.

Hand-In

There is no explicit hand-in. The bomb will notify your instructor automatically after you have successfully defused it on `zeus`. You can keep track of how you are doing by looking at:

http://zeus-1.cec.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 for several reasons:

- You lose 1/2 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 bomb server. 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 (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.

- `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/3e/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 our experience defusing bombs, we have the following advice:

- **Don't bother trying to step through helper functions.** This is generally a waste of time. The helper functions are the ones with descriptive names, like **input_strings**, **read_numbers**, **strings_not_equal**, **read_six_numbers**, etc. You can assume that these functions do what their name implies. Looking at what is being passed in to them and what is being returned from them is more important than their code.

- I print out all of the code associated with a particular **phase** when I start to defuse it. As I work, I annotate this code with information as I figure out what is going on. The phase functions you want to step through and understand the code on are generic named, like **phase_1**, **phase_2**, etc, or **func4**, etc.

- 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.

- As you will notice from examining `bomb.c`, the calls to your phases are all given to you in their original C as well. In here, you can see the calls to each of the six phases, which are your primary functions to disassemble for this project.

- 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. Examples would be like <**sscanf**@plt> for sscanf.

- 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. A better approach is to use **nexti** to step over that call instead.

- The bomb functions to reverse engineer all have a **generic** name like `fn7` or `phase2`. 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. Do not waste your time trying to reverse engineer functions like **read_six_numbers**.

- Understanding parameter passing is critical to this assignment. Whenever you see a reference to `%rsp` 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.

- It may not be important to understand exactly what everything does - the goal is to avoid `explode_bomb` calls. I 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.

- **Breakpoint guards are crucially important to success in this project**. If I put a breakpoint at the start of the `explode_bomb` function, I can ensure that I always give myself the option not to explode. Always use breakpoint guards before attempting to run any code, even if you think you are only performing "safe" operations.

- If I use a text file as input, it will help me save time because I will not have to retype the solutions to phases which I have already solved. The text file is sensitive to file format (e.g. character encoding, system-dependent newlines, or terminating newline), so be sure to use a breakpoint guard whenever you update your solution file.

- Remember the compiler will work with **different sizes**. `%eax`, `%ecx`, and other registers will show up often!

GDB Layout Modes:
When using gdb, there are two layout modes that you may find helpful.

layout asm

This layout mode will display the assembly source code on the screen with a border around it. Sometimes this border may look garbled, but it is trying the best it can to serve everyone up a nice pretty display. If the screen looks too garbled, you can issue a refresh command in gdb to redraw the screen.

refresh

If you hit a problem where every time you use stepi, it scrolls the screen and continues to do so after a refresh, you will need to quit and restart gdb. gdb wasn't designed for graphical modes, but it does a pretty good job at it on the average.

The second layout mode is to show the registers at the top of the screen.

layout regs

This one will show registers based on the size of the screen.

Warnings on GDB layout modes: There have been past reports of crashing gdb (and setting off bombs) when using layout modes and resizing the window (eg. expanding to full-screen while running gdb, or shrinking/growing the font while in gdb). While everything looks like it's running well, you should take care by getting a good size for you window before running gdb.

Tips on GDB layout modes: Previous students reported that setting your breakpoints and start running the program before running the layout commands worked more reliably than setting layouts before beginning to run the code. If you have any crashes of gdb itself while using layout, please post on Piazza and we'll submit trouble tickets. Please document exactly what you were doing and how you started gdb and we can look into it.

The deadline for this project is Friday, May 5th, 2023 11:59 PM EST.

All your interactions with the server are recorded in a log – you can continue to work beyond this deadline without a penalty as long as you have available late tokens. Once you have used all of your available late tokens, there will be 25% ceiling penalty for each extra 24-hour period. At most two days of late work are accepted beyond the deadline (regardless of the number of late tokens that you have). If you continue to work past the initial deadline, the additional work will only factor into your grade if it results in a net improvement to your grade.

The CS 367 GTAs and UTAs can help you with the use of tools such as gdb and objdump. You are encouraged to use the Piazza forum for your questions. However, please do not post long Assembly code sections from your binary bomb to Piazza and ask what the “logic” of that code is – it is your responsibility to figure out that logic. But of course, questions about the semantics of individual assembly instructions and the use of specific tools (e.g., gdb) are welcome.

Good luck and have fun defusing your binary bomb!