PROJECT 1: BOMB PROJECT - CS261

Phase 1 is due Wednesday, Feb 21st at 11:59 PM
Phase 2 is due Tuesday, March 5th at 11:59 PM
Phase 3 and 4 are due Friday, March 8th at 11:59 PM
Phase 5, 6, and 7 are due Tuesday, March 12th at 11:59 PM

Disclaimer: this project is an adaptation of the bomblab from Bryant & O'hallaron, Computer Systems: A programmer's perspective. Third Edition.

1 Logistics

This is an individual project. All submissions are electronic. Clarifications and corrections will be posted on Piazza.

Points Breakdown

Phase 1 to 5: 20 Pts each phase.
Phase 6 and Secret Phase (Phase 7): 5 Extra Pts each phase
Maximum points: 100 points (+ 10 extra points).

Phases 1 through 5 are required. Phases 6 and 7 are for extra credit (EC), and there is no penalty for skipping them. You may still get credit for phases completed late, but only up to 24 hours after the deadline. You can only receive 85 percent of the points for a phase you submitted late. There is no partial credit for phases beyond this. If after the late submission period, you have still not solved a phase you can request the phase solution by posting a private post to all the instructors on Piazza with the bomb number and phase you would like the solution for.

Each time your bomb explodes you lose 1/2 point (up to a max of 20 points) in the final score for the project.

Read the entire PDF before start the project

2 Introduction

The nefarious *Dr. Evil* has planted a slew of "binary bombs" on our class 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 (This is not Phase 1)

The first step is to download your binary bomb from the course server (systems#). To download the bomb, we published a website accessible only from the University network. To reach this website, you have two options:

1. Connect to UIC's wifi if you are on campus.
2. Connect to the UIC's Virtual Private Network (VPN). If you have not ever used the UIC VPN, please view the instructions here. *The VPN is a secure connection that allows your computer to access the University network when you are off campus.*

Once in the network, paste the following link into your browser : http://cs261.cs.uic.edu:15213/. If you can't view the page, first try viewing it without your browser plugins in incognito mode, and finally, try using another browser. This will display a binary bomb request form for you to fill in. Enter your NetID and email address carefully (your grade will be reported based on the NetID you typed) and hit the Submit button. The server will build your bomb and return it to your browser in a tarball file called `bomb`**`k`**`.tar`, where k is the unique number of your bomb.

Note: Try this if you are having trouble connecting to the VPN or downloading the bomb in general

- SSH into cs261.cs.uic.edu
- Go to this Piazza post which contains a command to download a bomb, copy the command and paste the entire command into your terminal.
- Follow the instructions given to you on the terminal.

The second step is to send the `bombk.tar` file to one of the CS server via sftp (cs261.cs.uic.edu) and save the `bombk.tar` file into a (protected) directory in which you plan to do your work on.

The third step is to extract the files from the tarball. For this, type the following command on the terminal from the folder that contains the tarball:

```
tar -xvf bombk.tar
```

This will create a directory called `./bombk` with the following files (Read these files):

- `README`: Identifies the bomb and its owners.

- `bomb`: The executable binary bomb.
- `bomb.c`: Source file with the bomb's main routine and a friendly greeting from Dr. Evil.

Associated Sites:

- To setup 2-Factor Authentication: https://it.uic.edu/services/faculty-staff/uic-network/duo-2-factor-authentication-2/
- To download UIC's VPN (required): https://accc.uic.edu/services/infrastructure/network/virtual-private-network/
- To download a bomb: http://cs261.cs.uic.edu:15213
- To view the scoreboard (it updates every ten minutes): http://cs261.cs.uic.edu:15213/scoreboard. You must check that your score is updated after you solve a phase. If your score does not update after 10 minutes, it means you did not solve the phase.
- If you experience any technical difficulties with the VPN, request help here https://accc.uic.edu/forms/contact-us/

Additional notes

- You must use your UIC NetID when downloading a bomb to receive credit when graded!
- The project must be completed on cs261.cs.uic.edu and you might need to give permission to your bomb for execution (`chmod +x bomb`)
- You can download multiple bombs. However, your grade will be based on your bomb with the overall highest score. This can be helpful when getting started; however, say you're working on phase 4 and have a handful of explosions, downloading a new bomb may not be a wise idea, as your scores for phases 1, 2, and 3 will not carry over to the new bomb, and you will not be able to earn those points on the new bomb because the deadlines for those phases have passed. Basically, once you have the hang of things, you should complete phase 1 with a bomb you plan to finish all phases with (you can continue to practice with other bombs with no penalty).

Step 2: Learn how not to explode the bomb

Your job for this project is to defuse your bombs **before they explode**.

To avoid accidentally detonating the bomb, you will need to learn how to single-step through the assembly code and how to set breakpoints (see the Q&A at the end of this document). 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 your favorite debugger to step through the disassembled binary.

Each time your bomb explodes it notifies the server, and you lose 1/2 point (up to a max of 20 points) in the final score for the project. So there are consequences to exploding the bomb. You must be careful!

Students can download multiple bombs and each bomb is graded independently (the grade is computed per bomb, not per student). Your project grade is your bomb with the highest score. You might have exploded your bomb while you learn where to set the breakpoints. Once you know how to set the breakpoints, you can download a new bomb so you do not lose points for these first explosions.

Step 3: Defuse Your Bomb

Your job for this project is to defuse your bomb.

Every time you defuse a phase you will see a printed message saying so. If you do not see the message, your phase has not been defused yet, even i you see in gdb that the next instruction will call the function `defuse_bomb`. You should also see the score updated on the scoreboard (it updates every ten minutes) http://cs261.cs.uic.edu:15213/scoreboard

You will need to learn how to inspect both the registers and the memory states using your favorite debugger. 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 for the rest of your career.

You must do the assignment on one of the class machines. 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.

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`. 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.

Project Submissions

There is no explicit submission. The bomb will record your progress as you work on it, assuming you entered your UIC email address when downloading the bomb. You can keep track of how you are doing by looking at the class scoreboard (while on the UIC network, or connected via the VPN):

http://cs261.cs.uic.edu:15213/scoreboard.

This web page is updated every 2 minutes to show the progress of each bomb. You should check to make sure that your scores are recorded correctly on the server. The server has occasionally crashed, so checking that your scores were received is always a good idea.

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.

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 1/2 point (up to a max of 20 points) every time you guess incorrectly and the bomb explodes.
- Every time you guess wrong, a message is sent to the 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) assumption 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.

Many tools 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 watchpoints, and write scripts.

  The CS:APP website

  `http://csapp.cs.cmu.edu/public/students.html`

  has a very handy single-page `gdb` summary that you can print out and use as a reference. Here are some other 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.
  - For online 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`, `man`, and `info` are your friends. In particular, `man ascii` might come in useful. `info gas` will give you more than you ever wanted to know about the GNU Assembler. Also, the web may also be a treasure trove of information. If you get stumped, feel free to post a question to Piazza.

Q & A

1. How to run the next assembly instruction?

You can debug the bomb by using commands like 'si' and 'ni' to move between different **assembly** instructions and function calls.

Consider using 'ni' to step over a single assembly instruction. This is different from 's', which will also step inside, and 'n' alone, which is meant for single lines of C code.

2. How to set breakpoints at assembly instructions?

You can place breakpoints in all of the following ways:

- This will break at the first line of the function "function_name":

```
b function_name
```

- This will break at the address *address_here*, (only if you started the program before you got the address)

```
b *address_here
```

1. Example of a **good** command (program was run first):

```
b *0x00005555555555b5
```

2. Example of a **bad** command (program was not run before setting the breakpoint). *Note that this command is bad (and will not work) because the address value is way too low!*:

```
b *0x000000000000159d
```

- This is the **easiest** way to break at a specific line of a function.

```
b *function_name + offset_number
```

In this command, ‘function_name‘ is the function name, and ‘offset_number‘ is the offset (in bytes) that you would like to break. These offsets can easily be found with `disas disassemble`. They are listed on the right-hand side of the assembly dump (see example below).

```
(gdb) disas function_name

Dump of assembler code for function function_name:
    0x0000000000001489 <+0>: endbr64
    0x000000000000148d <+4>: push  %rbx
    0x000000000000148e <+5>: cmp $0x1,%edi
    0x0000000000001491 <+8>: je 0x158f <main+262>
    0x0000000000001497 <+14>: mov %rsi,%rbx
```

Using the assembly dump above (from ‘disas‘), if we want to break on the "je" instruction, we just type `b function_name+8` (because the "je" instruction has the $< +8 >$ to the left of it)

- You can also run the program until a particular line using the `until` command. For example,

```
until *0x0000000000001489
```

3. Advanced GDB tips (optional)

After solving a couple of phases you might have noticed that you are setting the same breakpoints and similar processes every time you debug. Here you find a few useful tricks you can use to avoid repeating the same steps (like typing the answer for phase 1) and make debugging your bomb more agile.

1. it is suggested to store your answer in a file (for example, a new file named ‘answer.txt‘) so that you do not need to type in your answer every time you open gdb.
2. you can set gdb safe path by editing ‘ /.gdbinit‘ and adding (more info here):
3. you can create a ‘.gdbinit‘ file under the bomb directory and put the following into that file:

```
set args answer.txt
tui enable
layout asm
focus cmd
```

These configs will set up your answer file as the first argument every time you run the bomb inside gdb. It also enables gdb tui with asm layout so that the source code in assembly (as well as your current position) will be shown on the top window (interactively as you are debugging). (Detailed usage of gdb tui can be found here)

4. You can also put breakpoints inside the '.gdbinit' file. For example, when you're working on phase 2, you can set breakpoints for phases 2-6 (as well as the explode function, maybe). When phase 2 is defused, simply remove the break point for that phase and you can continue working on the next phase.

5. For a more advanced customization of gdb, you can also take a look at the gdb-dashboard project. However, it is not a requirement to finish the bomb project. https://github.com/cyrus-and/gdb-dashboard

4. A useful small document for bomb project

This is a link to a small document that introduces the mapping between register and function arguments, which might be helpful when stepping through different functions in the bomb project.

https://web.archive.org/web/20230621084529/http://6.s081.scripts.mit.edu/sp18/x86-64-architecture-guide.html