p5: Binary Bombs

1. Logistics

1. This project must be done individually.

   It is academic misconduct to share your work with others in any form including posting it on publicly accessible web sites, such as GitHub.

   It is academic misconduct for you to copy or use some or all of a program that has been written by someone else.

2. All work for this project is to be done on one of the CS department's instructional Linux machines. You're welcome to remotely login using ssh or MobaXterm as was shown in lecture (or any other way that you figure out on your own).
3. All projects in this course are graded on CS department's instructional Linux machines. It is your responsibility to make sure that your code runs as you expect on these machines.

2. Learning Goals

There are two main objectives of this project. The first is to quickly become familiar with x86 assembly language. The second relates to the first: to gain some familiarity with powerful tools that help with this process, namely **gdb** (the debugger) and **objdump** (the disassembler).

3. Binary Bombs

In this assignment, you will be defusing four binary bombs. The idea is simple: each bomb is an executable program that prompts the user for five inputs via the stdin console, one at a time, in order to defuse the bomb. If you type in the right values, you successfully defuse the bomb. If not, the bomb explodes! (Don't worry, it just prints that the bomb explodes; no real harm is done to you or your computer)

3.1 Getting the Required Files

The four bombs are unique for every student and are located in the following directory:

```
/p/course/cs354-skrentny/public/students/<your-cs-login-ID>/p5/
```

Replace <your-cs-login-ID> with your actual cs login and copy the contents of the above directory into your working directory. There should be 4 executable files named **b1**, **b2**, **b3**, and **b4**.

Please copy your executable bombs to your own private directory, and work towards finding your solutions in your own directory. That way, you will have the original executables in your student directory if you accidentally overwrite an executable.

3.2 Defusing the Bombs

The challenge is to figure out the correct set of 5 inputs expected by each of the four bombs. You can run each bomb interactively, and type in your guesses, one at a time. This will be useful in defusing each bomb with a debugger, as described in the next section. Take a look at the sample run below:

```
[haseeb@espresso] (55)$ ls
b1* b2* b3* b4*
[haseeb@espresso] (56)$ ./b1
input 1 (of 5)? 951905
input 2 (of 5)? 1234
BOMB EXPLODED
[haseeb@espresso] (57)$ ./b1
input 1 (of 5)? 951905
input 2 (of 5)? 994563
input 3 (of 5)? 493693
input 4 (of 5)? 828695
input 5 (of 5)? 278566
success!
[haseeb@espresso] (58)$
```

Your task is to figure out all the inputs and create four files, **b1.solution**, **b2.solution**, **b3.solution** and **b4.solution**, where each file contains the five lines of input demanded by its associated bomb. We will test your solution files as show below.

```
[haseeb@espresso] (70)$ ls
b1*  b1.solution  b2*  b2.solution  b3*  b3.solution  b4*  b4.solution
[haseeb@espresso] (71)$ cat b1.solution
951905
994563
493693
828695
278566
[haseeb@espresso] (72)$ ./b1 < b1.solution
input 1 (of 5)? input 2 (of 5)? input 3 (of 5)? input 4 (of 5)? input 5 (of 5)? succe
ss!
[haseeb@espresso] (73)$
```

All 5 inputs need to be correct to defuse a bomb. As long as the bomb explodes, no points will be given no matter how many inputs were correct before the explosion.

Make sure you create this file with a text editor (vim/gedit/nano) on a Linux Machine. If you are using **Windows or Mac**, editing locally followed by uploading will fail for **b2**. Use remote accessing tools like ssh or MobaXterm instead. Make sure your solution file contains five non-empty lines. **Remember to press enter or return after the last line in the solution file**. The bomb will be trapped into an infinite loop if the solution file contains less than 5 lines. If that happens, press ctrl-c to break.

This testing also uses another shell skill that you have already used: IO redirection. In this case the contents of a file are redirected as stdin to the program.

4. Tools: gdb and objdump

To figure out how to defuse your binary bombs, you will use two powerful tools: **gdb** and **objdump**. Both are critical in reverse engineering each binary bomb to understand what it does.

4.1 objdump

objdump is a command in linux to display information about object files. For our purposes the two important command line options are:

- **`-d`** which disassembles a binary
- **`-s`** which displays the full binary contents of the executable

For example, to see the assembly code of bomb b1, you might type:

```
objdump -d b1
```

This will show an assembly listing of each function in the bomb. Your first task then might be to look at **`main()`** and figure out what the code is doing.

The **`-s`** flag is also quite useful, as it shows the contents of each segment of the executable. This may be needed when looking for the initial value of a given variable.

By redirecting stdout, you can capture the output of **objdump** in a file, such that you can look at this output without having to regenerate it every time. And, you can use both command line options at the same time to create a full dump of the contents of the executable as well as the disassembled contents.

4.2 gdb

The debugger, gdb, is an even more powerful ally in your search for clues as to how to defuse each binary bomb. To run **gdb** on a particular bomb say **b1**:

```
gdb b1
```

which will launch the debugger and ready you for a debugging session. Ignore the '`no debugging symbols found`' warning on the last line. This is intended. The command **run** causes the debugger to run the program, in this case prompting you for input.

However, before running the debugger, you likely need to first set some breakpoints. Breakpoints are places in the code where the debugger will stop running and let you take control of the debugging session. For example, a common thing to do before typing run in the debugger is:

```
break main
```

to set a breakpoint at the **`main()`** routine of the program, and then type:

```
run
```

to run the program. When the debugger enters the **`main()`** routine, it will then stop running the program and pass control back to you, the user.

You will need to learn some basic commands in gdb in order to set breakpoints at various functions and addresses in your code, step through instructions, and examine the contents of memory addresses and registers in order to figure out the inputs that each bomb is expecting. We've listed some commands for you to get started with but you will have to read up more about gdb on your own to explore these and other commands fully.

- **`break <location>`**: sets up a breakpoint at the location which can be a function name or the address of an instruction. So for instance "**`break *0x804861a`**" will set a breakpoint at the instruction address `0x804861a`. Note when specifying an address in break it has to be of the form **`*addr`**.
- **`continue`**: resumes the execution until any breakpoint is reached again
- **`stepi`**: steps through the code one instruction at a time
- **`info registers`**: shows you the contents of all of the registers of the system
- **`x/nfu <addr>`**: The examine command, which shows you the contents of memory. **`n`**, **`f`**, and **`u`** are all optional parameters that specify how much memory to display and how to format it. **`addr`** is the hexadecimal address you want to look at. So for instance "**`x/3ub 0x54320`**" is a request to display 3 bytes (b) of memory formatted as unsigned decimal integers (u) starting at the address 0x54320.

Getting good with gdb will make this project go smoothly, so spend the time and learn! One thing to notice: using the keyboards up and down arrows (or ctrl-p and ctrl-n for previous and next, respectively) allows you to go through your gdb history and easily re-execute old commands. Getting good at using your history, whether in gdb or more generally in the shell you use, is a good idea.

There are plenty of good tutorials and resources online to get started with gdb. We will list a few to get you started off:

- Basic gdb example (Links to an external site.)Links to an external site.
- A nice introduction (Links to an external site.)Links to an external site. for those who like videos
- Handy gdb cheatsheet (Links to an external site.)Links to an external site.
- Gdb text user interface(TUI) (Links to an external site.)Links to an external site. mode. This is worth taking the time to learn, because it lets you look at the assembly code and registers side by side as you step through it. Type **`layout asm`** in gdb to try it out.

5. Hints

- x86 cheat sheet
- Function **`strtol()`** corresponds to the use of **`atoi()`** in C source code.
- Every C program has a **`main()`** function. Figure out how to locate it.
- A loop in **`main()`** iterates five times. Remember that each bomb requires five inputs.
- On a wrong input, function **`bomb()`** is called. This results in an explosion.
- If all five inputs are correct, function **`success()`** is called.
- Function arguments are set up in the call stack just prior to the function call.
- The two parameters to **`strcmp()`** are addresses to 2 C strings.

6. Requirements

We will test your solution files by running them as shown in the sample output in section 3.2. It is your responsibility to ensure that your solutions correctly defuse each of the four binary bombs on the CS Linux lab machines.

7. Submitting Your Work

Submit the following **source files** under Project 5 in Assignments on Canvas on or before the deadline:

1. b1.solution
2. b2.solution
3. b3.solution
4. b4.solution

It is your responsibility to ensure your submission is complete with the correct file names having the correct contents. The following points will seem obvious to most, but we've found we must explicitly state them otherwise some students will request special treatment for their carelessness:

- **You will only receive credit for the files that you submit.** You will not receive credit for files that you do not submit. Forgetting to submit, not submitting all the listed files, or submitting executable files or other wrong files will result in you losing credit for the assignment.
- **Do not zip, compress or submit your files in a folder.** Submit only the text files as listed.
- **Make sure your file names exactly match those listed.** If you resubmit your work, Canvas will modify the file names by appending a hyphen and a number (e.g., b1-1.solution) and these Canvas modified names are accepted for grading.

Repeated Submission: You may resubmit your work repeatedly until the deadline has passed. **We strongly encourage you to use Canvas as a place to store a back up copy of your work.** If you resubmit, you must resubmit all of your work rather than updating just some of the files.