Problem set 2: Binary bomb

- Due Friday 10/4 11:59pm (1 day later for extension)
  - Each student gets their own bomb.
  - **This assignment is self-grading.** There is no explicit need to store your binary bomb in your Git repository, though it wouldn't hurt.

Introduction

A dark future awaits us unless we defuse a huge number of "binary bombs" an English showrunner has planted on our class server. A binary bomb is a program that consists of a sequence of phases. Each phase reads a line from the standard input. If the line is correct, then the phase is defused and the bomb proceeds to the next phase. Otherwise, the bomb explodes by printing `"BOOM!!!"`, deducting a half-point from your problem set grade, 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 different bomb to defuse. Your mission, which you have no choice but to accept, is to defuse your bomb before the due date. Good luck!

Download

Save the `bomb*k*.tar` file to a 64-bit Linux host, such as a CS61 VM. Move the file to the directory you want to do your work. Then run `tar -xvf bomb*k*.tar`. This will create a directory called `./bomb*k*` with the following files:

- `README.md`: Identifies the bomb and its owners.
- `bomb`: The executable binary bomb.
- `bomb.cc`: Source file with the bomb's main routine.

Defuse

Before running your bomb, read the entire assignment!

Your job is to defuse your bomb. This involves supplying it with just the right input. But though there is a `bomb.cc` file, it doesn't actually contain the code for the various phases. You're going to defuse the bomb by interpreting assembly language, lucky you.

The bombs are tamper-proofed in a couple ways. For one, they can only be defused when the computer is connected to the Internet. Running the bomb on a machine without Internet connectivity won't do anything.

You can use many tools to help you defuse your bomb. Probably the best way is to use your favorite debugger to step through the disassembled binary.

Each time your bomb explodes you lose 1/2 point (up to a max of 20 points) in the final score for the problem set. So there are consequences to exploding the bomb. Be careful!

The first four phases are worth 10 points each. Phases 5 and 6 are a little more difficult, so they are worth 15 points each. The maximum score you can get is 70 points.

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 everyone, 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,

```
./bomb psol.txt
```

then it will read the input lines from `psol.txt` until it reaches the end, and then switch over to stdin. Make sure to include a newline at the end of the file!

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 registers and memory state. One of the nice side effects of doing the lab is that you will get very good at using a debugger!

Turnin

The bomb notifies us automatically of your progress as you work on it. You can keep track of how you are doing, and compare (anonymously) with everyone else, by looking at the class scoreboard at: http://cs61.seas.harvard.edu:15213/scoreboard

This web page is kept updated to show the progress for each bomb. It may take up to a minute for new explosions and defusings to show up on the scoreboard. Also, the scoreboard displays time in UTC, so do not be alarmed if it appears that your bomb is reporting status for the future!

Hints

There are many ways of defusing your bomb. Hypothetically, you could even figure out the bomb without ever running the program, just from the machine code (and various tools like **objdump**). But it's much easier to run the bomb under a debugger, watch what it does step by step, and reverse-engineer the input it wants.

There are many tools that 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.

Understanding instructions

There are lots of ways to puzzle out instruction meanings: lecture, the book, even the examples distributed as part of our lectures. (For example, if you're curious about the `leaq` instruction, go to the cs61-lectures repositories, and try "`grep leaq */*.s`". Also try searching for the instruction name on Google: "`movq instruction`". But be careful. There are two syntaxes used for x86-64 assembly language. We use "AT&T syntax" in class and in the book, but many online references use "Intel syntax," which switches the order of arguments and is different in other annoying ways. For instance, Intel calls the `%rax` register `rax` (no percent). Read about the differences in syntaxes in the Aside on p177 of CS:APP3e, or here or here.

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. The CS:APP3e web site has a handy two-page gdb summary (TXT) 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**.
- Some critical gdb commands for this pset are `r` (of course), `c`, `b`, `disas`, `x` (for instance, try `x/5i $pc` and `x/20xw $rax`—gdb names registers with initial dollar signs), and `si`. The `s` command is sometimes useful and sometimes dangerous. Many of the related commands on these pages might also be useful. Check out, for example, `finish`, `info reg`, and `display`. And **do read the manual for these commands**! It contains lots of helpful time-saving hints. To exit gdb use `q`.

- Many students really like the TUI interface obtained by running `gdb -tui bomb`. Read up about the TUI `layout` command. Some people also like to run gdb under gdb-mode in emacs.
- Consider creating a file called `.gdbinit` in your `BOMBDIR`. `gdb` automatically executes all commands listed in this file every time it starts up. The command `set confirm off` is useful here (if you get tired of questions like "Quit anyway? (y/n)"). For this lab, a **breakpoint** or two would be super useful too!! But:
- Your `BOMBDIR/.gdbinit` may not be loaded by default. (Check this by reading gdb's startup messages. If you see an error about "`auto-loading has been declined`," then your `BOMBDIR/.gdbinit` was not loaded.) This is a security precaution. To get around it, create a file named `.gdbinit` *in your home directory* containing "`add-auto-load-safe-path [BOMBDIR]`". (More on auto-load-safe-path)
- For online documentation, type "help" at the gdb command prompt, or type "man gdb" or "info gdb" at a Unix prompt.

`objdump -d` (or `objdump -S`)

Use this to disassemble all of the code in the bomb. You can also just look at individual functions.

`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!

`strings`

This utility will display the printable strings in your bomb.

`man ascii`

A handy table of character encodings.

For more, don't forget your friends the commands `man` and `info`, and the amazing Google and Wikipedia. In particular, `info gas` has more than you might ever want to know about assembler.