CSE 351, Autumn 2010
Lab 3: Disassembling and Defusing a Binary Bomb
Due: Wednesday October 27, 11 PM

No late assignments accepted

Questions? Problems? See the course website for contact info:
`cs.washington.edu/351`

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 everyone 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!

Instructions

You should do the assignment on a lab Linux machine or a CSE Linux VM and be sure it works there or on `attu` (at least test your solution there before submitting it!), to make sure it works when we grade it. In fact, there is a rumor that Dr. Evil has ensured the bomb will always blow up if run elsewhere. There are several other tamper-proofing devices built into the bomb as well, or so they say.

Everyone gets a unique bomb to defuse. On `attu` or a lab machine, get yours by extracting the relevant tar file:

```
tar xvf /projects/instr/10au/cse351/$USER/lab3-bomb.tar
```

Your shell will automatically replace `$USER` with your username when it runs the command (or you can replace it with your username yourself if you want). This command will create a directory called `bomb`i (where i is the ID of your bomb) with the following files:

- `bomb`: The executable binary bomb.
- `bomb.c`: Source file with the bomb's main routine.
- `defuser.txt`: File where you will write your defusing solution.

- `Makefile`: A Makefile for submitting your solution.

Your job is to defuse the bomb. You can use many tools to help you with this; please look at the **tools** section for some tips and ideas. The best way is to use a debugger to step through the disassembled binary.

The bomb has 5 regular phases, each worth 10 points, for a total of 50 points. The 6th phase is extra credit (worth an extra 5 points), and rumor is that a secret 7th phase exists. If it does and you can find and defuse it, you will receive another 5 extra credit points. The phases get progressively harder to defuse, but the expertise you gain as you move from phase to phase should offset this difficulty. Nonetheless, the latter phases are not easy, so please don't wait until the last minute to start. (If you're stumped, check the hints section at the end of this document.)

The bomb ignores blank input lines. If you run your bomb with a command line argument, for example,

```
./bomb defuser.txt
```

then it will read the input lines from `defuser.txt` until it reaches EOF (end of file), and then switch over to `stdin` (standard input from the terminal). 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.

To avoid accidently 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 the rest of your career.

Submitting Your Work

To submit your solution, make sure your solution is saved in `defuser.txt` and then run `make submit` in your `bomb`i directory. That will create an archive file that you should turn in using the regular Catalyst assignment drop box. (Or you can just turn in the `defuser.txt` file which is the only thing in the archive, but we kept the `make submit` option for consistency with other assignments.)

Tools *(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 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 the number of possibilities is so large that you won't be able to try them all in time.

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, this 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/public/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 bomb`

  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 bomb`

  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 may look cryptic. 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 -t x bomb`

  This utility will display the printable strings in your bomb and their offset within the bomb.

Looking for a particular tool? How about documentation? Don't forget, the commands `apropos` and `man` are your friends. In particular, `man ascii` might come in useful. Check the course web page for Linux basics. If you get stumped, feel free to ask the TA or instructor for help.

Hints

If you're still having trouble figuring out what your bomb is doing, here are some hints for what to think about at each stage: (1) comparison, (2) loops, (3) switch statements, (4) recursion, (5) pointers and arrays, (6) sorting linked lists.