Individual Unassessed Coursework 1: Defusing a Binary Bomb

Due date: 4 PM, 18th January 2024

Value: Unassessed (mark given but not part of module mark)

Introduction

The nefarious *Dr. Evil* has planted a slew of "binary bombs" on our class server. A binary bomb is a program that consists of a sequence of phases. Each phase expects you to type a particular string on the standard input `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 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!

Get Your Bomb

To obtain your bomb, you must first register for the 0019 grading server. You received email from the instructors with a link to a handout providing instructions on how to do so. Please follow the instructions in that email before proceeding further.

After you register successfully, you will find your bomb within your registration GitHub repo in a `tar.gz` file called `bomb`*k*`.tar.gz`, where *k* is the unique number of your bomb.

CW1 involves your stepping through and analyzing the execution of an x86-64 binary program. It therefore only functions on machines with an x86-64 CPU.[1] We provide several remotely ssh-accessible machines running a standard 0019 Linux environment on x86-64 hardware at UCL CS for you to use to run your bomb while working on CW1. You can find instructions on how to access these 0019 Linux machines via ssh on the 0019 courseworks web page: `http://www.cs.ucl.ac.uk/staff/B.Karp/0019/s2024/cw.html`.

Once you are logged into an x86-64 Linux machine, obtain an updated copy of your registration git repo. If you already have your registration repo on an x86-64 Linux host, just issue the command `git pull` in the shell while working in the directory for your local copy of the repo. Otherwise, clone the repo from GitHub by typing

```
git clone https://github.com/UCLCS0019/YOUR-REGREPO-NAME
```

in the shell, where `YOUR-REGREPO-NAME` is the name of your registration repo from when you completed the CW0 grading server registration process.

Copy the `bomb`*k*`.tar.gz` file to the directory where you wish to do your work. If you're working on a UCL CS-hosted VM, make sure the directory is readable only by your user ID. Then run `tar -xzvf bomb`*k*`.tar.gz` in that directory. Doing so will extract the following 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.

If you lose the `.tar.gz` file for your bomb, no problem; just clone the GitHub copy of your grading server registration repo again.

[1]Note that you cannot run the CW1 bomb binary on a Mac with an ARM CPU using Rosetta.

Defuse Your Bomb

Before running your bomb, read this entire coursework handout!

Your job is to defuse your bomb. Doing so involves supplying it with just the right input. You will rapidly notice that while there is a `bomb.c` file, it doesn't actually contain the code for the various phases. You're going to be defusing the bomb using x86 assembly language (lucky you!).

The bombs seem to be tamper-proofed in a couple ways. For one, they can only be defused when run on a machine connected to the Internet. Running the bomb on a machine without Internet connectivity won't do anything. There are several other tamper-proofing devices built into the bomb as well, we hear!

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 it notifies Dr. Evil, who informs us, and you lose 1/2 point (up to a maximum of 20 points) in the final score for the coursework. So there are consequences to exploding the bomb. You must 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. So the maximum score 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 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. **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!

Submission

There is no explicit submission step. The bomb automatically notifies the class staff of your progress as you work on it. You can keep track of how you are doing and see where you stand (anonymously) against the rest of the class by looking at the class scoreboard at:

```
https://studcw2.cs.ucl.ac.uk:15213/scoreboard
```

This web page is 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.

While the scoreboard will show your score on CW1, CW1 is unassessed—your score is not included in your overall mark for 0019 in any way. The score is only provided as feedback for you to evaluate your own performance on CW1. Note, however, that the entire content of CW1 is examinable: you can expect questions on the final exam that draw on the knowledge gained solving CW1. So CW1 is definitely not optional: you stand to lose marks on the final exam if you do not do it. Note further that while we do not enforce academic honesty rules on completing unassessed CW1 without assistance from others (apart from the 0019 teaching staff), you will significantly impair your learning if you do not complete CW1 on your own, without assistance. That again will lead to reduced performance on the final exam, whose CW1-related questions will not probe memorized details about the solution to CW1, but instead check that you understand the nuances of x86-64 assembly that you learn primarily by solving CW1 yourself.

Hints

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

There are many tools designed to help programmers 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 assembly instructions** There are many ways to puzzle out the meaning of an assembly instruction, starting with lecture and the class textbook.

  Also try searching for the instruction's name with Google, *e.g.*, with `movl instruction`. But be careful: there are two syntaxes used for x86 assembly language. In 0019, we use what is known as the "AT&T Syntax," as does the textbook. But many online references use "Intel syntax," which inverts the order of arguments and differs in other (annoying) ways. For instance, Intel calls the `%rax` register `rax` (no percent sign). You can find a summary of these syntax differences in the Aside on p. 213 of CS:APP/3e (the class textbook), at `http://en.wikipedia.org/wiki/X86_assembly_language#Syntax`, or `http://asm.sourceforge.net/articles/linasm.html`.

- `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 (remember, however, that we are not giving you the source code for most of your bomb), set breakpoints, set memory watch points, and write scripts.

  The CS:APP/3e web site offers a very handy two-page `gdb` summary that you can print out and use as a reference. 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 coursework are `r` (of course), `c`, `b`, `disas`, `x` (for instance, try `x/5i $pc` and `x/20xw $eax`—note that gdb names registers with leading 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_req`, and `display`. And *do read the manual for these commands!* It contains lots of helpful, time-saving hints. To exit gdb use `q`.
  - Consider creating a `.gdbinit` file in the directory where you have your bomb executable. `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 coursework, a *breakpoint* or two would be super-useful, too! But:

  - On some Linux distros, your `.gdbinit` file may not be loaded by default. (You can see if it hasn't been by reading gdb's startup messages. If you see an error "`auto-loading has been declined`," then your `.gdbinit` file hasn't been loaded.) This default is a security precaution. To get around it, create a file named `.gdbinit` *in your home directory*, containing either "`add-auto-load-safe-path [BOMBDIR]`" (where `BOMBDIR` is the absolute pathname of the directory where you have placed your bomb), or, if the bomb is in your home directory, "`add-auto-load-safe-path ~`". 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. 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` (or `objdump -S`)

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

- `strings`

  This utility will display the printable strings in your bomb.

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

Read the Ed Discussion Web Site

You will find it useful to monitor the 0019 Ed web site as you work on this (and all!) 0019 courseworks. Any announcements (*e.g.,* helpful tips on how to work around unexpected problems encountered by others) will be posted there. And you may ask questions there. *Please remember that if you wish to ask a question that reveals some aspect of your solution, you must mark your post on Ed as private, so that only the instructors may see it.* Questions about the interpretation of the coursework text, or general questions about, *e.g.,* gdb that do not reveal aspects of how you are defusing your bomb, however, may be asked publicly—and we encourage you to do so, so that the whole class benefits from the discussion.

Acknowledgments

This coursework is derived from the CS:APP course materials, as further modified by Eddie Kohler at Harvard. Further enhancements have been made by the 0019 course staff at UCL, particularly Ahmed Awad.