CS 1337
Lab Assignment L4: Defusing a Binary Bomb

1 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

A link by which you can obtain your bomb is available from the course website under the "Labs" menu.

The link will take you to a page that will display a binary bomb request form for you to fill in. Enter your full name and ISU email address and hit the Submit button. The server will build your bomb and return it to your browser in a `tar` file called `bombk.tar`, where k is the unique number of your bomb. (A tar file is just a compressed file, like zip.)

Copying Your Bomb to Your AWS Server

Once the `bombk.tar` file is saved to your computer, you will need to upload it to your server. First, on your server, create a directory for lab 4 using the command

```
mkdir /home/your_user_name/CS_1337/assignments/lab_4
```

Next, from your **local machine** (this means exiting your server or opening a different terminal window where you are not logged in to the server), you need to copy the `bombk.tar` file to the directory you just created on your AWS server. **Linux and Mac users**, you can do this from the commandline using the following command:

```
scp /local/path/to/bombk.tar user_name@ip_address:/remote/path/to/lab_4
```

where `user_name` and `ip_address` are the same username and address you use when you ssh into your server (if you have an ssh config shortcut as recommended in Lab 1, you can just put that in instead). Note that you if you drag and drop a file from your desktop into the terminal, it will automatically insert the `/path/on/local/machine/to/your/bombk.tar` into the commandline for you. The remote path to your lab_4 directory is the path returned by `pwd` when in the `lab_4` directory on your server.

Windows users, your commandline likely also has the `scp` command (just try it and see), in which case you can follow the same instructions above. If that doesn't work, try the same command but with `pscp` instead of `scp`. If that doesn't work, then you need to install PSCP to your local machine. See instructions here: `http://xray.rutgers.edu/~matilsky/documents/pscp.htm`

Troubleshooting/understanding scp

`scp` is just like the `mv` or `cp` commands in Linux or the mov instruction in assembly. `scp` copies a `source` file on one machine to a `target` presumably on a different machine using the syntax

```
scp source target
```

It does this (per the description given in the `man scp` entry) using "ssh(1) for data transfer, and uses the same authentication and provides the same security as a login session". The only question then is what should the form of `source` and `target` be?

Answer: `source` and `target` are both paths. In the case of this lab, the `source` is a file path to the `tar` file on your computer. For example, if my `tar` file is in my Downloads folder, then (either by drag and drop of the file into a terminal OR by tab completion in the terminal window) I find that the path to my `tar` file looks like this on my computer:

```
/Users/bodipaul/Downloads/bomb243.tar
```

You have to figure out what that path is on your computer.

This is what goes in the place of `source`.

Note: This is what is called an **absolute** path because it specifies the full path starting from the root directory. You can alternatively pass a **relative** path based on the current working directory from which you run the `scp` command.

`target` refers to the path to where you want the file to "land" on your server. To find this, I suggest logging into your server and navigating (i.e., using the `cd` command to change directories) to the directory where you want the file to "land". When you get there, you can print the working directory with the `pwd` command. When I navigate to my lab_4 directory and enter `pwd` I get back

```
/home/bodipaul/CS_1337/assignments/lab_4
```

Now, this is not quite the `target` yet. You need to run the `scp` command from on your computer (since that's where the tar file is), not from on your server. If you give that target filepath as the target for the `scp` command from your computer, it will interpret it as a filepath on your computer (i.e., a **local** path). To specify this filepath is on your AWS server (i.e., that it is a **remote** path), you have to prepend the "user@host:". My username is bodipaul and my host IP address is 3.95.179.128. So for me, prepending my username@host information before the filepath looks like this:

`bodipaul@3.95.179.128:/home/bodipaul/CS_1337/assignments/lab_4`

This is what goes in the place of `target`.

So in all, for me the `scp` command would look like

```
scp /Users/bodipaul/Downloads/bomb243.tar bodipaul@3.95.179.128:/home/bodipaul/CS_1337/assignments/lab_4
```

But it's going to be different for you depending on the file directory structure on your local machine, the name of the file you're copying, your username and IP address for your server and the file directory structure on your remote machine.

Untarring Your Bomb on Your AWS Server

Back on your server, navigate to the lab_4 directory. Then give the command: `tar -xvf bombk.tar`. This will create a directory called `./bombk` with 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.
- `writeup.{pdf,ps}`: The lab writeup.

If for some reason you request multiple bombs, this is not a problem. Choose one bomb to work on and delete the rest.

Step 2: Defuse Your Bomb

Your job for this lab is to defuse your bomb.

You must do the assignment on your server (the bomb is specifically designed for x86-64). 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.

You can use many tools to help you defuse your bomb. Please look at the **hints** section below 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 bomblab server, and you lose 1/4 point (up to a max of 20 points) in the final score for the lab. 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 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 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 can store solutions to phases in a file `psol.txt` (using vim) so that you don't have to keep retyping the solutions to phases you have already defused.

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 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.

Logistics

This is an individual project. All handins are electronic. Clarifications and corrections will be posted on the Discord channel for the lab.

Handin

There is no explicit handin. The bomb will notify your instructor automatically about your progress as you work on it. You can keep track of how you are doing by looking at the class scoreboard which is available via a link from the "Labs" menu on the course website. The linked web page is updated continuously to show the progress for each bomb for every student in the class (bombs are anonymized).

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 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 1/4 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 bomblab 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) assumptions 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.

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.

- `ctl-c`

  You should pretty much always run your bomb executable in gdb (more below), but should you start your bomb without gdb, you will want to kill the program rather than set of the bomb by failing a phase. In Linux, you can kill any process at any point simply by typing `ctl-c`.

- `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.

  The CS:APP web site

  `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. (Word to the wise: set a breakpoint on the function that makes the bomb explode as a failsafe!)
  - 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`.
  - Make gdb work for you the way you want it to. Figure out how to execute multiple commands at once (i.e., define a function). Figure out how to save a list of breakpoints so you don't have to reenter them each time.

- `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 Linux command line utility will display the printable strings in any file that is passed as an argument.

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 ask the TAs or the instructor for help. You are welcome to discuss the lab and ask questions on Discord, but please do not do other people's thinking and learning for them.

Good luck!