Project Bomb

Due: October 11, 2016

1 Introduction

Dory has swum across the ocean to find her parents; she's almost to the Marine Life Institute when she runs into yet another obstacle: a field of Memory Erasing Bombs! They must have fallen from a sinking ship. Faced with steep cliffs on either side, she realizes that she has to swim directly through the field, defusing each bomb so she doesn't lose her last memories of her parents.

You have been tasked with deactivating these bombs, so that Dory can finally be reunited with her parents. A Memory Erasing 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 Memory Erasing Bomb proceeds to the next phase.

Otherwise, the bomb *explodes*, causing your terminal to temporarily shut down for repairs[1] and then terminate. The Memory Erasing Bomb is defused when every phase has been defused.

There are too many Memory Erasing Bombs for Nemo and Dory to deal with, so we are giving each student their own Memory Erasing Bomb to defuse alone (see *Section 2.1: Collaboration*). 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!

2 Assignment

Your job for this project is to defuse your bomb. Doing so will require you to employ all of your knowledge of the x86-64 assembly language. The Marine Life Institute has certainly employed all of its brainpower in the design of these bombs, and it is not easily thwarted.

In this assignment, you will be providing input to a pre-compiled binary executable file. *Do not run your bomb on any computer outside of the computer science department.* These binary files were compiled to run on those machines and there is no guarantee that they will run correctly on a different machine. Using `ssh` or Portable Sunlab to log in remotely to run and defuse your bomb is fine.

2.1 Collaboration

As with Project Data, the nature of this assignment is quite different from the others in this course; consequently, the normal CS033 collaboration policy will be modified for this assignment.

For this assignment:

- You may not share any aspect of your bomb, including the binary or disassembled code, for any part of this assignment with other students in this course.

[1] All it really does is temporarily block off signals (which you will learn about in a few weeks), and sleep for a few seconds.

- You may not discuss the solutions to any part of this assignment with other students, even at a high level.
- You may not assist other students with defusing their bombs, or look at any other student's binary or disassembled code. Only you or a member of the course staff may examine any aspect of your code.
- You may not search for solutions to these phases on the internet.

2.2 Obtaining Your Bomb

You can obtain your bomb by running the command

```
cs033_bomb_get
```

This script will copy the following files into your cs033 directory:

- *README*: Identifies the bomb and its owner (you).
- *bomb*: The executable binary bomb.
- *bomb.c*: Source file with the bomb's main routine and a friendly greeting from Pixar.

2.3 Defusing Your Bomb

You can (and should!) use many tools to help you defuse your bomb. Please look at section 3 (the *Hints* section) for some tips and ideas. The best way is to use your favorite debugger (cough cough `gdb`) to step through the disassembled binary, frequently toggling breakpoints as you go.

Each bomb consists of five phases, with each phase more difficult than the one preceding it. Although phases get progressively harder to defuse, the expertise you gain as you move from phase to phase should offset this difficulty. Nevertheless, you should *certainly* avoid waiting until the last minute to start defusing your bomb.

Some of the phases call various support functions within the bomb. **These functions will not explode your bomb**. The only functions that can cause the bomb to explode are the `phase()` functions. Additionally, support routines with a name indicating a behavior do indeed exhibit that behavior - for example, a function named `read_three_characters()` does indeed read three characters. You will need to decipher the return values and side effects of such functions, but you will not need to scan them in detail to avoid explosions.

The bomb ignores blank input lines. Additionally, the bomb accepts input from files. If you run your bomb with a command line argument, for example,

```
./bomb solution.txt
```

then it will read the input lines from *solution.txt* until it reaches EOF (end of file), and then switch over to `stdin`. In a moment of vulnerability, The Marine Life Institute added this feature so you don't have to keep retyping the solutions to phases you have already defused.

3 Hints

There are many ways of defusing your bomb. You can disassemble the bomb, examine it in great detail, and figure out exactly what it does without ever running it. This is a useful technique, but is 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, and is our recommended approach.

However, *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:

- Every time you guess wrong, a message is saved in the filesystem. You could very quickly saturate the filesystem with these messages and cause the system administrators to revoke your computer access.
- You have no way of knowing (without examining the bomb) how long the strings are, or what characters are in them. Even if you made the senseless 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 (without even accounting for the delay caused by exploding the bomb) and you will not get the answer before the assignment is due.

There are many tools designed to help you figure out both how programs work and what is wrong when programs 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, `gdb`, 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[2] and assembly code, set breakpoints, set memory watch points, and write scripts.

  There are many `gdb` resources available to you, including the `gdb` Cheatsheet located on the course website. 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`.

  – **Toggle breakpoints liberally**. It will be very difficult for you to decipher a phase without being able to evaluate the state of the program during execution. Setting frequent breakpoints, and `print`ing different parts of the program state while stopped, will help you understand what a phase is doing throughout its execution, and save you a lot of time. You can set a breakpoint on a specific instruction with the command `break *0x<address>`. For example, to break on the instruction stored at address `0804b19d`, use the command `break *0x0804b19d`.

  – Use `print` frequently. This command will allow you to both examine the state of the bomb, and examine data stored within the bomb that you will need to defuse it. In particular, if you see a memory access to an absolute address, try `print`ing the data stored at that address in different formats so you can determine what it is. You can

[2]Although not in this assignment.

print a string this way with the command `print (char *) <address>`. The `gdb print` statement can cast and dereference values the same way you would when writing C.

  - 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`.
  - You can cast values and dereference pointers in `gdb` with the C operators `(<type>)` and `*` respectively. You can glean a lot more information about the program state by using these operations in `gdb print` statements, as described above.
  - You can use the `disassemble` command to disassemble a function within `gdb`, displaying its instructions and object code.
  - You can use the `source` command to pass in a command file. This will run all the commands, line separated, in the specified file, which might be useful when defusing later stages. You must be careful, though, since a typo in the command file might cause your bomb to explode.

- `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` using the `disassemble` command.

  For most convenient use, send the output of `objdump` to a file and annotate that file, keeping it open alongside you as you work.

- `strings` This utility will display the printable strings (sequences of at least 4 consecutive printable characters) in your bomb.

If you are looking for a particular tool or for documentation, the commands `apropos`, `man` and `info` are your friends. In particular, `man ascii` might come in handy. `info gas` will give you more than you ever wanted to know about the GNU Assembler.

Lastly, some other tips:

- Break each function down into parts, and annotate those parts with what they do. You shouldn't try to understand each instruction individually, but instead groups of instructions.

- Write out in prose what the jump instructions do. Jump instructions in x86_64 are not usually intuitive, so it will help you to think of how they work in a different way.

4 Grading

Your grade for this assignment is determined by the number of phases you successfully defuse.

This assignment is worth 50 points in total. Each phase is worth 10 points. If you do not solve every phase, you will receive credit for the phases you do solve. Your bomb can be considered completely solved when you get through all 5 phases of the bomb without modifying the program's state using gdb.

To clarify the role of bomb explosions as part of your grade: we do keep a log of the explosions, and will use this to detect brute forcing and other such nasty techniques. However, there is no hard and fast "limit" on the number of explosions you may have, nor will we explicitly take off points for explosions. Don't abuse this policy. And don't sweat the few times you forget to set breakpoints in gdb.

4.1 Late Policy

If you continue to work on your bomb after the due date, you should know that `cs033_bomb_grade` (discussed below) does not take into account late days; however, when the TAs/HTAs are assigning grades, late days will be taken into account. The important thing is to try to defuse all the phases before the due date, but any work done before the due date will not be overwritten by work done after the due date. All explosions and defusals are recorded.

5 Handing In

There is no explicit handin for this project, as the bomb continuously updates your score. You can check your current score by running

```
cs033_bomb_grade
```

from the terminal.