CS 2400 – Defusing a Binary Bomb

1 Introduction

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 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.

Each group (one or two students) gets 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

Each group of students will attempt to defuse their own personalized bomb. Each bomb is a Linux binary executable file that has been compiled from a C program. To obtain your group's bomb, one (and only one) of the group members should point your Web browser to the bomb request daemon at

```
http://felix.csc.villanova.edu:23456
```

Fill out the HTML form with the email addresses and names of your team members, and then submit the form by clicking the "Submit" button. The request daemon will build your bomb in a `tar` file called `bombk.tar`, where k is the unique number of your bomb. The daemon will send you an email with instructions on how to get your bomb. The email instructions essentially ask you to log on to felix and type in the following commands:

```
cp /tmp/bombs/bombk.tar ~/csc2400
cd ~/csc2400
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.

If you change groups, simply request another bomb and we'll sort out the duplicate assignments later on when we grade the lab.

Also, if you make any kind of mistake requesting a bomb (such as neglecting to save it or typing the wrong group members), simply request another bomb.

Step 2: Defuse Your Bomb

Your job is to defuse the bomb.

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

Each time your bomb explodes it notifies the staff, and you lose 1/4 point (up to a max of 10 points) in the final score for the lab. So there are consequences to exploding the bomb. You must be careful! In a moment of weakness however, Dr. Evil decided that the first 5 bomb explosions will be free of charge.

Each phase is worth 10 points, for a total of 60 points.

The phases get progressively harder to defuse, but the expertise you gain as you move from phase to phase should offset this difficulty.

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

```
bash$  ./bomb bsol.txt
```

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

Logistics

You may work in a group of up to 2 people. You should do the assignment on `felix.csc.villanova.edu`. 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 they say.

Hand-In

There is no explicit hand-in. The bomb will notify your instructor automatically after you have successfully defused it. You can keep track of how you (and the other groups) are doing by looking at

`http://www.csc.villanova.edu/~mdamian/csc2400/bomblab.html`

This web page is updated continuously to show the progress of each group.

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 the `gdb` 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 suggest that you use the following tools to help you analyze your bomb.

- `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 and set memory watch points. 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 and step through machine instructions.
  - Use the handy `gdb` quick reference handed to you in class (also available on the class page).

- `strings`

  This utility will display the printable strings in your bomb.

Other available tools of interest are:

- `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`.

Where to start?

1. Start the debugger

   ```
   gdb ./bombk
   ```

   Disassemble the first phase using

   ```
   disas phase_1
   ```

   You will see that this phase calls a function called `strings_not_equal`.

2. Set a breakpoint to `call strings_not_equal`

   ```
   break *(phase_1+???)
   ```

   Display the next machine instruction to be executed each time the debugger suspends execution, so you know where you are at all times:

   ```
   display /i $eip
   ```

   Then run the program up to the first breakpoint – you will need to type in your own string (type in an arbitrary string, such as `123456`) to get past the point where the program is waiting for user input.

   We all know that, prior to the `call <strings_not_equal>` machine instruction, the arguments for `strings_not_equal` are pushed into the stack (in particular, the addresses of the two strings that are to be compared). Inspect the two words at the top of the stack to learn these addresses:

   ```
   x /2xw $esp
   ```

   Next inspect the memory contents at the addresses you've learned.

   ```
   x /s address
   ```

   where `address` is what gets pushed onto the stack before the call to `strings_not_equal`. This should reveal the secret string for you.

3. Suppose that you are finished with phase_1, and are ready to start phase_2. After defusing phase_1, the program will wait for input from you to be used in phase_2. In this case, it will wait for six numbers separated by spaces. As before, disassemble the code for phase_2 using

   ```
   disas phase_2
   ```

   You will see that phase_2 seeks 6 integers that satisfy certain conditions. Focus on the `cmp` instructions and inspect the arguments being compared.

To print out the memory contents at an address given in the form

```
K(%ebp,%ebx,4)
```

use the debugger command

```
x /d ($ebp+4*$ebx+K)
```

Inspecting the arguments of `cmp` functions should give you a hint on what the next number in the sequence should be. Once you've learned that number, restart the program execution from the beginning, this time entering the correct information you've learned so far when prompted for input.

To avoid the hassle of typing in the input each time (and avoid typos that would trigger the bomb to explode), you may want to enter your solution in a text file, say `bsol.txt`, and run your bomb using `bsol.txt` as an argument:

```
(gdb) run bsol.txt
```

Once you've stopped at a breakpoint (usually a `cmp` machine instruction), step through machine instructions using `stepi` (`si`) or `nexti` (`ni`) and try to understand what happens (what each number is compared against, what conditions are being checked).

4. Stay alert for calls to `explode_bomb`! You never want to jump there!
5. Please keep in mind that `$eax` always contains the result of function calls.

HAVE FUN!