CS 449
Bomb Lab: Disassembling and Defusing a Binary Bomb

1 Introduction

In this lab, you will gain basic familiarity with x86-64 assembly instructions and how they are used to implement typical C language features, including comparison, loops, recursion, etc. You will also gain experience using the `gdb` debugger to step through assembly code and other tools such as `objdump`.

For this lab, you will be working with a binary bomb, which 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. As always, *start early* to get it done on time. Good luck, and welcome to the bomb squad!

Before you start, please review the course policy on academic integrity and remember that you are **not allowed** to search for help online, and you are **not allowed** to ask other students for help beyond using the tools, show them your solution, or discuss its specifics.

Important: Be aware that you may be asked to explain your solution to a member of our course staff.

Step 1: Get Your Bomb

To obtain your bomb, you need to run the following command script (exactly as shown below) on the Thoth Linux machine:

```
$  /afs/pitt.edu/home/v/t/vtp4/public/download-bomb.sh
```

This script will show a binary bomb request form for you to fill in. Enter your PITT user name and email address. This will download your bomb and save it to your local folder in a `tar` file called `bombk.tar`, where k is the unique number of your bomb.

Save the `bombk.tar` file to a (protected) directory in which you plan to do your work on the Thoth Linux machine. 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

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 the Thoth Linux machine. In fact, 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 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 1/2 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! Since we round final score up to nearest integer, you get first explosion free. :-)

The bomb has 3 regular phases. Phase 1 is worth 10 points and Phases 2 and 3 are worth 25 points each. So the maximum score you can get is 60 points while solving the regular phases. Phase 4 is slightly more difficult and is considered optional extra credit (5 points). Although phases get progressively harder to defuse, 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,

```
$  ./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 `stdin`. This feature was added so 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.

During this lab, we **strongly recommend** that you keep notes of the steps you took in solving each stage. This will be immensely helpful in helping you to keep track of what's stored at important addresses in memory and in registers at different points in the program's execution. (A good strategy for this might be to keep a notetaking app open on your computer so you can copy and paste values between it and gdb.)

Logistics

This is an *individual* project. All submissions are electronic using an autograder service. Before you begin, please take the time to review the course policy on academic integrity at the course web site.

Submission

There is no explicit submission. 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 at:

`http://people.cs.pitt.edu/~vinicius/bomblab/`

This web page is updated continuously to show the progress for each bomb.

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.

Reviewing the x86-64 calling convention may be helpful (see Figure 3.28 in the textbook or this reference `https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf`). It is worth noticing that x86-64 passes the first six arguments to a function in the following registers (in order): `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`. The return value of a function is passed in `rax`.

You will notice that the binary code calls `sscanf` ("string scan format"), which is similiar to scanf but reads in data from a string instead of stdin:

```
char* mystring = "123, 456";
int a, b;
sscanf(mystring, "%d, %d", &a, &b);
```

The first argument, `mystring`, is the input string. The second argument, `"%d, %d"` is the format string that contains format specifiers to parse the input string with. After matching the input string to the format string, the extracted values are stored at the addresses given in the additional arguments. After this code is run, `a = 123` and `b = 456`. Reference information can be found online for `sscanf`, `scanf`, and `printf`.

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 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.

- `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.
  - 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`

  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 utility will display the printable strings in your bomb.

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 about these tools. **If you get stumped, feel free to ask your course staff for help.**

Where to start?

If you're still having trouble figuring out where to start and what your bomb is doing, follow these steps:

1. Make sure your working directory is `<YOUR_PRIVATE_FOLDER>/bombk`, where k is your bomb number. Start the debugger

   ```
   gdb ./bomb
   ```

2. Disassemble the first phase using

   ```
   disas phase_1
   ```

   You will see that this phase calls two functions, one called `strings_not_equal` and another one called `explode_bomb`. This latter function is the one you will want to avoid executing.

3. Set a breakpoint to the call to `strings_not_equal`

   ```
   break strings_not_equal
   ```

4. 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 $rip
   ```

5. 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.

6. Recall that, prior to the call to `strings_not_equal`, the arguments for `strings_not_equal` are pushed onto the registers `rsi` and `rdi` (in particular, these are the registers used to place the first and second function arguments, respectively). Inspect the contents of those registers to learn something about them:

   ```
   x /2wx $rdi
   ```

   Alternatively, you may find it more conveniently to inspect the register contents using a string format.

   ```
   x /s $rdi
   ```

This should reveal something. How about the other register? What is it used for? Now run the same commands to inspect the register `$rsi`.

7. 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 for 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. This 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.

   As mentioned earlier, 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`, then run your bomb using `bsol.txt` as an argument:

   ```
   run < bsol.txt
   ```

8. Once you've stopped at a breakpoint, say at 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)

9. Stay alert for calls `explode_bomb`! You never want to jump there! Use a breakpoint.

10. Recall that `$rax` always contains the result of function calls.