PKU-ICS
Bomb Lab: Defusing a Binary Bomb

1 Introduction

The nefarious *Mr. Gin* has planted a slew of "binary bombs" on our 64-bit linux 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. We believe that you, like Conan, have also learned bomb disposal in Hawaii. Good luck, and welcome to the bomb squad!

Step 1: Get Your Bomb

You can obtain your bomb from the Autolab site

```
http://162.105.31.232/
```

After logging in to Autolab, select `Bomblab -> Download your bomb`. The Autolab 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.

Save the `bombk.tar` file to a (protected) working directory in which you plan to do your work. Then login to a linux machine and give the command: `tar -xvf bombk.tar`. This will create a directory called `./bombk` with the following files:

- `README`: Identifies the bomb and its owner.
- `bomb`: The executable binary bomb.
- `bomb.c`: Source file with the bomb's main routine and a friendly greeting from Mr. Gin.

You should only download one bomb. If for some reason you download multiple bombs, choose one bomb to work on and delete the rest.

Warning: If you expand your `bombk.tar` file on a PC, you'll risk resetting the bomb's execute bit. You can undo this with the `chmod +x bomb` command.

Step 2: Defuse Your Bomb

Your job for this lab is to defuse your bomb. You can do the assignment on any 64-bit linux machine.

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 Autolab server, and you lose 1/2 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! However, this year we have introduced a small safety net: if your bomb happens to go off, you may want to take a careful look at the very last part of the writeup. It may help you get your points back.

The first four phases are worth 15 points each. Phases 5 and 6 are a little more difficult, so they are worth 20 points each. So the maximum score you can get is 100 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 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 can safely exit your bomb at any time by typing `ctrl-c` (simultaneously pressing the ctrl and c keys).

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.

Warning: You should never use your debugger to jump directly to a particular phase. Doing so can cause your bomb to explode silently.

Handin

There is no explicit handin. The bomb will notify Autolab automatically about your progress as you work on it. You can keep track of how you are doing by looking at the Autolab class scoreboard (From Autolab, follow `Bomblab -> View scoreboard`). 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.

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 1/2 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 Autolab 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 120 characters long and only contain letters, then you will have 26^{120} 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.
- The answer of *phase5* should only consist of digits and letters! Input string with not supported characters may cause strange results which we won't bear the responsibility.

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 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 textbook Web page at

  `http://csapp.cs.cmu.edu/3e/students.html`

  has a handy 1-page `gdb` command summary for x86-64 that you can print out and use as a reference. It also contains a link to Prof. Norm Matloff's `gdb` GDB tutorial.

  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 need 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`.

  In the section 2, we have provided some useful gdb commands for reference.

- `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! For example, you could discover a function called "`explode_bomb`", which would be a good place to set a breakpoint to keep the bomb from blowing up.

- `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. If you get stumped, feel free to ask the teaching staff for help.

2 Introduction to `gdb` Commands

For a detailed introduction to specific `gdb` commands, we recommend reading Section 3.10.2 in the CS:APP textbook before starting the bomb lab.

Here, we will introduce some of the commands we used while working on this lab.

Command	Full Name	Description
`r`	`run`	Start executing the program until the next breakpoint or the end of the program.
`q`	`quit`	Exit the GDB debugger.
`ni`	`next instruction`	Execute the next instruction but do not step into functions.
`si`	`step instruction`	Execute the current instruction; if it is a function call, step into the function.
`b`	`break`	Set a breakpoint at a specified location.
`c`	`continue`	Continue executing the program from the current position until the next breakpoint or the end of the program.
`p`	`print`	Print the value of a variable.
`x`	`examine`	Examine memory at a specified location.
`j`	`jump`	Jump to a specified location in the program.
`disas`	`disassemble`	Disassemble the current function or a specified code region.
`layout asm`	-	Display the assembly code view.
`layout regs`	-	Display the current register states and their values.

Table 1: Common `gdb` Commands

Usually, after starting `gdb bomb`, we first use `layout asm` and `layout regs` to open views for easier analysis.

To close the `layout` view, press `Ctrl + x`, then press `a`.

Regarding `p` and `x`, it is crucial to remember that the `p` command is used to print the value of an expression, while the `x` command is mainly used to examine the contents of memory. Here are some commonly used examples:

Command	Description
p $rax	Print the value of the %rax register
p/x $rsp	Print the value of the %rsp register in hexadecimal
p/d $rsp	Print the value of the %rsp register in decimal
x/2x $rsp	Examine 2 units of memory at %rsp in hexadecimal format.
x/2c $rsp	Examine 2 units of memory at %rsp in character format.
x/s $rsp	Examine memory at %rsp as a C-style string.
x/b $rsp	Examine 1 byte of memory at %rsp.
x/h $rsp	Examine 1 half-word (2 bytes) of memory at %rsp.
x/w $rsp	Examine 1 word (4 bytes) of memory at %rsp.
x/g $rsp	Examine 1 giant (8-byte) unit of memory at %rsp.
j 0x00004869	Jump to the address 0x00004869.
j *(main + 40)	Jump to the address of the main function plus a 40-byte offset.
info registers	Print the values of all registers
info breakpoints	Print information about all breakpoints
delete breakpoints 1	Delete the first breakpoint, can be abbreviated as d 1

Table 2: Common `gdb` Commands

The suffixes after the `/` in these commands (such as `2x`, `2d`, `s`, `g`, `20c`) specify the way and quantity of memory to be examined. Specifically:

- The first number (e.g., `2`, `20`) specifies the number of units to be examined
- The letter `c/d/x` indicates that the memory should be examined in character/decimal/hexadecimal format
- The letter `s` indicates that the memory should be examined as a C-style string
- The letter `b/h/w/g` indicates that the memory should be examined as a byte/half-word(2 bytes)/word(4 bytes)/giant(8 bytes) unit

  When using `x/b`, `x/h`, `x/w`, `x/g`, the `unit` will be changed after you use those commands until you use those commands again.

2.1 Using the `.gdbinit` File for Configuration

The `.gdbinit` file is a powerful feature of `gdb` that allows users to set default configurations upon startup, eliminating the need to repeatedly enter commands manually. This can significantly streamline the debugging process.

To set this up, follow these steps:

- Create a `.gdbinit` file in the current directory:

  ```
  touch .gdbinit
  ```

- Create the configuration directory:

  ```
  mkdir -p /home/ubuntu/.config/gdb
  ```

- Allow `gdb` to preload files from the root directory

  ```
  mkdir -p /home/ubuntu/.config/gdb
  echo "set auto-load safe-path /" >> /home/ubuntu/.config/gdb/gdbinit
  ```

Next, open the `.gdbinit` file and add the following configurations:

```
# Set default input file, avoiding manual input each time
set args psol.txt

# Set breakpoints at each phase function to monitor execution
b phase_1
b phase_2
b phase_3
b phase_4
b phase_5
b phase_6
```

By employing the `.gdbinit` file, you can automate the setup of breakpoints and other commands, allowing for a more efficient and less error-prone debugging experience.

2.2 Breakpoint Scripting with `.gdbinit`

Utilizing the `.gdbinit` file not only simplifies the initialization of `gdb` but also allows for advanced breakpoint programming. This feature enables users to automate responses when the program hits a specified breakpoint, enhancing the debugging workflow.

For example:

```
b phase_1
command
p $rsp  # Print the value of the stack pointer
end
```

This script will automatically print the value of the stack pointer when the breakpoint is hit.

(Maybe you can think of some other useful scripts, awa.)

3 Secret phase

As you may have heard, *Mr. Gin* is also a fan of ”Harry Potter”, he always has a card up his sleeve, and it is indeed the case this time. But he left some words most elusive to you, which we quoted verbatim as follows:

> Never could a muggle senses the magic. It’s purely beyond their ken. Do they really think alohomora could open all the doors in the world? No way! Neither would they have any idea that I further secured it with some abracadabra. Mua ha ha ha! Just bother with these most impregnable spells ever!

Can you solve this conundrum? But we would be remiss if we did not mention that this task earns you no credits. Since the administrator’s schedule is crammed, we can only count on you now to figure these all out and safeguard our vulnerable little linux machines. Please do help us!