CS105 – Computer Systems Spring 2020

Assignment 3: Bomb Lab

Due: Friday, February 21, 2020 at 5pm

Part A: Assembly-level Debugging

In C Lab we saw how to use `gdb` to run and debug C code by taking advantage of the fact that `gdb` understands your program at the source level: it knows about strings, source lines, call chains, and even complicated C++ data structures. But sometimes it's necessary to inspect the assembly code.

For this part, we will use the program `q3b.c` from clab. To be sure we're all on the same page, assemble that program with optimization level zero ("`gcc -g -O0 -o parta q3b.c`") and bring it up with `gdb`.

Create a file called `PartA.txt` and put your answers to the following questions there. Be sure to put both your names at the top of the file!

1. Set a breakpoint in `main`. Run the program with arguments of `1 42 2 47 3`. Where does it stop? Type "list" to see what's nearby, then type "`b 29`" and "`c`". Where does it stop now?

2. *Existing breakpoints at* `main` *lines 26 and 29.*
   So since that's the start of the loop, typing "`c`" will take you to the next iteration, right? Oops. Good thing we can start over by just typing "`r`". Continue past that first breakpoint to the second one, which is what we care about. But why, if we're in the `for` statement, didn't it stop the second time? Type "`info b`" (or "`info breakpoints`" for the terminally verbose). Lots of good stuff there. The important thing is in the "address" column. Take note of the address given for breakpoint 2, and then type "`disassem main`". You'll note that there's a helpful little arrow right at breakpoint 2's address, since that's the instruction we're about to execute. Looking back at the corresponding source code, what part of the `for` statement does this assembly code correspond to?

3. *Existing breakpoints at* `main` *lines 26 and 29.*
   The code at `+44` jumps to `main+104`, which has three instructions that jump back to `main+46`. This is all part of the loop pattern we covered briefly in class (in this case, a `for`). We've successfully breaked ("broken?" "Set a breakpoint?") at the initialization of the loop. But we'd like to have a breakpoint *inside* the for loop, so we could stop on every iteration. The jump to `main+46` tells us that we want to stop there. But that's not a source line; it's in the middle clause of the `for` statement. No worries, though, because `gdb` will let us set a breakpoint on *any* instruction even if it's in the middle of a statement. Just type "`b *(main+46)`" or "`b *0x40069f`" (assuming that's the address of `main+46`, as it was when I wrote these instructions). The asterisk tells `gdb` to interpret the rest of the command as an address in memory, as opposed to a line number in the source code. What does "`info b`" tell you about the line number you chose? (Fine, we could have just set a breakpoint at that line. But there are more complicated situations where there isn't a simple line number, so it's still useful to know about the asterisk.)

4. *Existing breakpoints at* `main` *lines 26 and 29, and instruction* `main+46`*.*
   We can look at the current value of the array by typing "`p array[0]@argc`". But the current value isn't interesting. Let's continue a few times and see what it looks like then. Typing "`c`" over and over

is tedious (especially if you need to do it 10,000 times!) so let's continue to breakpoint 3 and then try "`c 4`". What are the full contents of `array`?

5. *Existing breakpoints at `main` lines 26 and 29, and instruction `main+46`.*
   Perhaps we wish we had done "`c 3`" instead of "`c 4`". We can rerun the program, but we really don't need all the breakpoints; we're only working with breakpoint 3. Type "`info b`" to find out what's going on right now. Then use "`d 1`" or "`delete 1`" to completely get rid of breakpoint 1. But maybe breakpoint 2 will be useful in the future, so type "`disable 2`". Use "`info b`" to verify that it's no longer enabled ("Enb"). Continue past breakpoint 3, where we're stopped. Where do we stop next? (Hopefully that wasn't too much of a surprise!)

6. Sometimes, instead of stepping through a program line by line, we want to see what the individual instructions do. Of course, instructions manipulate registers. Quit `gdb` and restart it, setting a breakpoint in `fix_array`. Run the program with arguments of `1 42 2 47 3`. Type "`info registers`" to see all the processor registers in both hex and decimal. What flags are set right now? Note: sometimes it is not necessary to see all the registers. You can use the print commands `p` or `p/x` to print the value of an individual register.

7. *Existing breakpoint at `fix_array`.*
   In C Lab, we looked at lots of different ways to interpret data, but there is one that we didn't use: `x/i`. I particularly like "`x/16i $rip`". Try this command: what do you see? Compare that to the result of "`disassem fix_array`".

8. *Existing breakpoint at `fix_array`.*
   Finally, we mentioned stepping by instructions. That's done with "`stepi`" ("step one instruction"). Type that now, and note that `gdb` gives a new instruction address but still says that you're in the for loop. Hit return to `stepi` again, and keep hitting return until the displayed line `doesn't` contain a hexadecimal instruction address. Where are you?

9. *Existing breakpoint at `fix_array`.*
   It's useful to use "`x/16i $rip`" or `disassem fix_array` here to make sure we understand what's about to happen. You should see three `mov` instructions followed by a `call`. Use `stepi 3` to get past the `movs`. What instruction address will be executed next?

10. *Existing breakpoint at `fix_array`.*
    As with source-level debugging, at the assembly level it's often useful to skip over function calls. At this point you have a choice of typing "`stepi`" or "`nexti`". If you type "`stepi`", what do you expect the next instruction to be (hexadecimal address)? What about "`nexti`"? (By now, your debugging `gdb` skills should be strong enough that you can try one, restart the program, and try the other if you want to.)

Now you know everything you need to know about using `gdb` with assembly. Time to put your new skills to work!

Part B: Bomb lab

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 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 pair 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 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 their Web browser to the bomb request daemon at

`http://pom-itb-cs2.campus.pomona.edu:15213/`

This will display a binary bomb request form for you to fill in.

- Where it says "User Name," enter the CAS userids for *both* team members. Separate them by a single hyphen—no spaces!
- Enter the email address of just one team member.
- Then 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.

Copy the `bombk.tar` file to a (protected) directory on the VM in which you plan to do your work. You should be able to use something like:

```
scp ~/Downloads/bomb1.tar username@pom-itb-cs2.campus.pomona.edu:~/cs105
```

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.

If for some reason you request multiple bombs,just 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 course VM. 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 at the end of this document for some tips and ideas. You will no doubt use `gdb` 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 10 points) in the final score for the lab. So you might want to think about how to avoid exploding the bomb!

Phases 1 through 4 are each worth 10 points, and Phases 5 and 6 are each worth 15 points. You will also receive 5 points for Part A and 3 points for submitting a feedback file, for a total of 77 points.

Although phases get progressively harder to defuse, the expertise you gain as you move from phase to phase should offset this difficulty. But there last phase can be a bit tricky, so please do not 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,

```
% ./bomb PartB.txt
```

then it will read the input lines from `PartB.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 do not have to keep retyping the solutions to phases you have already defused. Please take advantage of this vulnerability by creating a file named `PartB.txt` and recording your solutions in this file.

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.

Practical Details

You must work with your assigned partner.

Clarifications and corrections will be posted on Piazza.

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://pom-itb-cs2.campus.pomona.edu:15213/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 not always easy. 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 maximum of 10 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 come find you.

- We have not 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. Or the universe ends.

There are many tools which are designed to help you figure out both how programs work, and what is wrong when they do not 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`

  As you saw in C Lab and in Part A, 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.

  You will likely find your notes and/or the gdb references from last week to be helpful. Here are some other tips for using `gdb`.

  – To keep the bomb from blowing up every time you type in a wrong input, you will want to remember how to set breakpoints.
  – For other 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 does not 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` (possibly after partially running the program).

- `strings`

  This utility will display the printable strings in your bomb.

Looking for a particular tool? How about documentation? Do not 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, keep in mind that you may not look for solutions on the web. And remember that the mentors (and I) are here to help you!

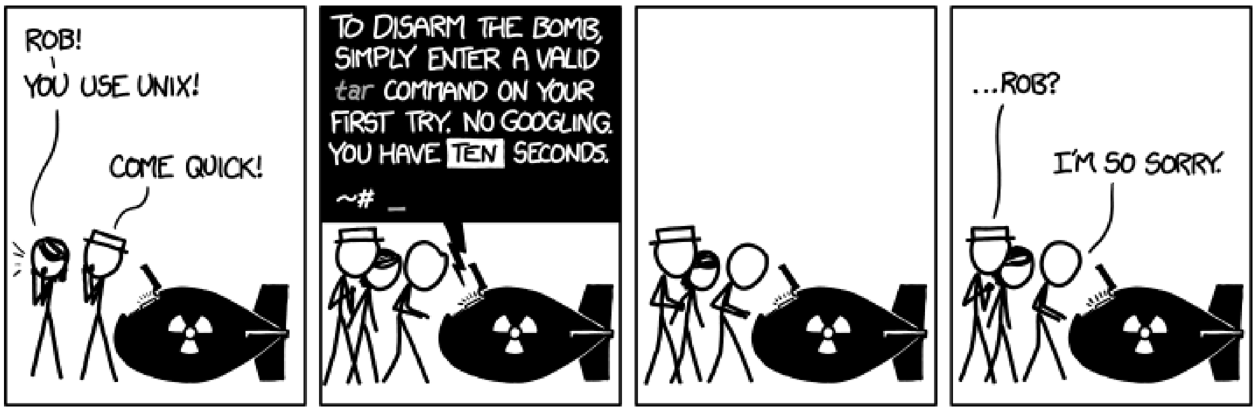
One other useful fact: you will find that the bomb has many functions with descriptive names. All functions do what their names say. Also remember that `sscanf` is a built-in library function. Do not try to reverse-engineer it unless you have several months to spend; instead read the manual page.

Step 3: Upload your files

Please upload to submit.cs your files `PartA.txt` and `PartB.txt` together with a file called `feedback.txt` that answers the following questions:

1. How long did each of you spend on this assignment?
2. Any comments on this assignment?

As always, how you answer these questions **will not affect your grade**, but whether you answer them will.


ROB!
YOU USE UNIX!
COME QUICK!
TO DISARM THE BOMB, SIMPLY ENTER A VALID tar COMMAND ON YOUR FIRST TRY. NO GOOGLING. YOU HAVE TEN SECONDS.
~# _
...ROB?
I'M SO SORRY.

`https://xkcd.com/1168/`