University of Miami, Electrical and Computer Engineering

Project 1: Defusing a Binary Bomb
Assigned: January 21st, 2014
Due: February 3rd, 2014 11:59pm

EEN 312, Spring 2014
Professor Eric W. D. Rozier

Maximum Score: 80pts

1 Pre-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 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. Good luck, and welcome to the bomb squad!

2 Obtaining Your Bomb

You can obtain your bomb by pointing your Web browser at:

```
http://een312.performalumni.org:15213/
```

This will display a binary bomb request form for you to fill in. For your user name, enter the last names of the people in your group, separated by hypens, e.g. `murat-yan` and the email address of one of your group

members, and 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.
You should download a single bomb per group. Each bomb is different.
Save the `bombk.tar` file to a (protected) directory in which you plan to do your work. 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, this is not a problem. Choose one bomb to work on and delete the rest.

3 Defuse Your Bomb

Your job for this lab is to defuse your bomb.
You must do the assignment on one of the class machines. 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 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 20 points) in the final score for the lab. So there are consequences to exploding the bomb. You must be careful!
The first four phases are worth 10 points each. Phases 5 and 6 are a little more difficult, so they are worth 15 points each. So the maximum score you can get from the lab portion is 70 points. An additional 10 points will be given for completing the post-lab write up.
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 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.

4 Logistics

This project should be completed in your groups. All handins are electronic.

5 Handin

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://een312.performalumni.org:15213/scoreboard`

This web page is updated continuously to show the progress for each bomb.
In addition to your solution which will be graded by the autograder, you should submit a short report on your solution individually. You should discuss how you solved each phase. Include disassembly of key functions with notes on their meaning. This report is worth 10 points.

6 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 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.

  The following web sites

  `http://www.ofb.net/gnu/gdb/gdb_toc.html`
  `http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html`
  `http://sourceware.org/gdb/onlinedocs/gdb/ARM.html`
  `http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf`
  `http://www.performalumni.org/erozier2/EEN312/armv6arch.pdf`

have good references for `gdb` and the ARMv6 instruction set that you can print out. 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 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`.

Some useful commands within `gdb`.

- `b LABEL` - set a breakpoint at the specified `LABEL`. Example `b phase_1`.
- `b *ADDRESS` - set a breakpoint at the specified `ADDRESS`. Example `b 0x8c00`.
- `disas LABEL` - disassemble the code under a given label. Example `disas phase_1`.
- `info r` - displays the current value of every register.
- `x/FMT ADDRESS` - displays the value stored at `ADDRESS` in the format specified by `FMT`. Address can be a memory location, or the name of a register (example `x/d $r0` displays the register `r0` as a decimal number).

- `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:

  ```
  95b0: ebfffd00  bl 89b8 <_init+0x134>
  ```

  To determine that the call was to `sscanf`, you would need to disassemble within `gdb`, where it appears as:

  ```
  0x000095b0 <+56>: bl 0x89b8 <__isoc99_sscanf>
  ```

- `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. If you get stumped, feel free to ask your instructor for help.

7 On-line Content

Several tutorial videos have been uploaded to the course website:
`http://www.performalumni.org/erozier2/een312.html`
Be sure to check them out, they contain many help examples.

Lab 1 – Bomb Defusing

Nathan Paternoster

Phase 1

The correct password for this phase is **"Crikey! I have lost my mojo!"**. In the function "strings_not_equal" another function "string_length" is called twice. In the first call our guess is loaded into $r0. After string_length exits the length of our guess is loaded into $r5 (in strings_not_equal+16). In the second call of string_length the correct password is loaded into $r0.

Phase 2

The correct password is **0 1 1 2 3 5**. The function "read_six_numbers" will check to make sure that our guess is formatted as six numbers and will also push our guess onto the stack. Back in phase_2 the first digit of our guess is loaded into $r3. It is compared with 0, so the first digit of the password must be 0 (instruction +20). The second digit is then put into $r3 and compared with 1 (instruction +32). Therefore the second digit must be 1.

A loop then occurs from +52 to +92. Every iteration of the loop checks that the current digit (in $r2) is equal to the previous digit added to the current one (in $r3, the summation happens at +60). This is essentially the Fibonacci series. The counter $r4 determines when the loop exits. It will exit after four iterations. So the next four digits of our six-digit password will be numbers of the Fibonacci series.

Phase 3

The correct password is **4 0,** although the first number can actually be any positive integer below 5. We chose 4. Something from the program counter is loaded into $r1. Examining $r1 we see the string "%d %d." This suggests that the correct password must be two integers. The function scanf puts the amount of integers of our guess into $r0. $r0 must be greater than 1, further confirming the previous speculation.

At instruction +208 our first integer guess is loaded from the stack into $r3. It is compared with 5 on the condition that it must be less than. This tells us our first digit may be any integer under 5. At instruction +220 the second digit of our guess is loaded into $r3. It must be equal to whatever is in register $r2 at this point to progress safely. The value 0 is in $r2 so the second digit of our guess should be 0.

Phase 4

The correct password is **11 18**. Once again scanf puts the amount of integers in our guess into $r0. $r0 must be equal to two (instruction +24), so the correct password must contain two integers.

At instruction +32 the first integer of our guess is loaded into $r3. According to the two following conditions it must be greater than 0 and less than or equal to 14 (instructions +40 and +48 respectively). In func4 our goal is to exit the function with $r0 = 18 so that the check in phase_4 at instruction +72 will be satisfied. Because we had a small number of possibilities to work with we used trial and error to determine that the correct value for our first integer is 11.

At instruction +56 the second integer of our guess is loaded into $r0. At instruction +72 it is compared to the immediate 18. Therefore the second integer of our guess must be 18.

Phase 5

The correct password for this phase is **5 115**. The same beginning instructions are used as the previous phase to check that we have two integers in our guess. $r3 is loaded with our first guess at +36 and then AND'd with the immediate 15 at +40 to insure that the resulting number would be 15 or less. At instruction +48 a check is performed to make sure this number does not equal 15. Our first guess must be in the range 0-14.

0	10
+4	2
+8	14
+12	7
+16	8
+20	12
+24	15
+28	11
+32	0
+36	4
+40	1
+44	13
+48	3
+52	9
+56	6
+60	5

At instruction +64 $r0 is loaded with an array of 15 elements. The array is shown on the right. $r1 counts how many times we go through a loop spanning from instruction +68 to +84 (incremented at +68). The loop must go through 15 times (instruction +92). $r3 acts as the offset of the $r0 array and is also the value of our first guess. At instruction +72 $r3 is multiplied by 4 and then used to access that value in the array. That value in the array then overwrites into $r3. The next iteration of the loop will use this new value of $r3 (multiplied by 4) to be the next location accessed in the array. The loop will be broken out of when $r3 = 15 (instruction +80). In order to go through the loop the required number of 15 times we must insure that the $r3 will go through every other value of the array before becoming 15. The correct starting position of $r3 should be the value 5 at the last position in the array.

Meanwhile, register $r2 acts as the summation of all the values reached in the array (minus the first value). Our second integer guess is loaded into $r3 at instruction +100. It is compared to this summed result in the next instruction to check for equality. Therefore our second guess must be equal to the sum of the values in the array (minus the first value entered, 5). This means our second guess must be 115.

Phase 6

The correct password for this phase is **5 1 4 3 2 6**. The solution for this phase consisted of six numbers that each represented a link in a link list. The link list had to eventually be ordered in such a way that the values were in descending order. The values of the links couldn't be affected but the addresses to the next link (essentially the order of the list) could be.

Instructions 0 to +12 checked to see that there were six numbers. Instructions +16 to +108 consisted of a loop within a loop. In the outside loop every digit had 1 subtracted from it and compared to 5 (instruction +36) to make sure every digit was in the range 1-6. In the inside loop each digit was compared to every other digit to make sure there were no identical digits (instruction +80).

The loop from instruction +112 to +132 would subtract all of our six input numbers from 7. We had to account for this switch in the final ordering of the links.

0	85
+4	1
+8	MEM
+12	922
+16	2
+20	MEM
+24	434
+28	3
+32	MEM
+36	194
+40	4
+44	MEM
+48	90
+52	5
+56	MEM
+60	850
+64	6
+68	MEM

Instructions +136 to +212 consisted of another loop within a loop that wrote a set of values (instruction +176) to the stack depending on the order of our input values. The values we got were set up in an order close to this array shown on the right (the first link appeared to be in a different location in memory from the other links).

MEM means that the value there was a value of some other location in memory (specifically one of the other link's data value). This value is equivalent to the "next" parameter of a link in a link list. Some example values we got for this parameter are: 78468, 78228, 78456, etc. Each link is given three spaces in memory. The first is its value (e.g. 85, 922, 434), the second its link number (e.g. 1, 2, 3), and the last its last is the next link it points to.

Instructions +216 to +264 loaded these values from the stack onto register $r4. The values of the links will be accessed by $r4. The loop from instruction +272 to +374 will execute five times. Each time, the value of the current link $r4 is on (specified by our input sequence and represented by $r2) is compared to the value of the next link in the list (represented by $r3). The current link's value must be

greater than the next link's. Because of this we know that the link list must be ordered in descending order.

Working out the values of each link the correct solution would be to build the link list in the order 2 → 6 → 3 → 4 → 5 → 1 (values: 922 → 850 → 434 → 194 → 90 → 85). Conveniently, the numbers we entered match up to the order of the link list. However this is not the correct input sequence. The function earlier that subtracts our numbers from seven will end up ruining this sequence. The correct sequence is therefore 7 minus each value here. So the final correct password is 5 1 4 3 2 6.