Binary Bomb Phase 1 Defusing Worksheet

COMP2100 8 September 2021

Introduction

The binary bomb lab assignment is an important learning activity in COMP2100. To assist your learning, this worksheet supports you through a series of steps to analyse phase 1 of your binary bomb so as to work out a solution. As you work through these steps, you will learn a lot about machine code and the debugging tool gdb.

1. Before you do anything else

- Read the assignment specification. Completely.
- Have a copy of the *X64 Assembly and ASCII Reference Sheet* available, and look at it first for any details that you need to recall about assembly or ASCII.
- Quickly look through the document *Bomb Defusing with Gdb*. This worksheet assumes that you will use that document (or other sources) to find the appropriate `gdb` commands for tasks such as disassembly, stepping, breakpoints, etc.
- Download your binary bomb using the `lab` command (as described in the assignment specification).

2. Disassemble phase 1

Use `gdb` to disassemble the phase 1 function (it is called `phase_1`). Looking at the disassembly, answer the following items:

a. There is a function that explodes your bomb. What is it called?

b. What `gdb` breakpoint command could you use to halt execution of your bomb if that function is called?

3. Analyse the flow control of phase 1

Print out the assembly code for your phase 1, or put it into a word processing and drawing tool, and analyse it.

a. Sketch the conditional branches. Draw a red arrow if the branch takes you to code that makes the bomb explode, and a green arrow if it avoids exploding the bomb.

b. Look for a safe path through the code, where some conditional branches may be taken and others may not be taken. You want to find the simplest path that returns from the phase without exploding the bomb. There is more than one path – start with the simplest path.

4. Understand the parameter that is passed to phase 1

Look at the source code `bomb.c` and answer the following questions:

a. What is the type of the parameter that is passed to phase_1?

b. What register will contain the parameter? For help, see the *X64 Assembly and ASCII Reference Sheet*.

c. The parameter is a pointer, which is equivalent to an array. What type of array are you working with here?

   What type are the elements of the array?

5. Interpret memory address expressions

The code contains memory address expressions that are equivalent to array references in C. Identify them and translate them to pseudo-C expressions. Remember that the array index in C is multiplied by the size of the array element when it is written as an assembly address expression. (If this is not familiar to you then you need to watch lectures from week 9).

a. You will see `(%rdi)` This is equivalent to `*rdi` and therefore also equivalent to an array reference. Fill out the missing index in the pseudo-C expression:

(%rdi) rdi[_____]

b. You will also see `0x1(%rdi)` and `0x2(%rdi)`. Fill out the indices.

0x1(%rdi) rdi[_____]

0x2(%rdi) rdi[_____]

c. There are other similar expression in your code. List them here and show the array reference equivalences. Remember to convert hexadecimal to decimal where appropriate.

_____(%rdi) rdi[_____]

_____(%rdi) rdi[_____]

_____(%rdi) rdi[_____]

_____(%rdi) rdi[_____]

_____(%rdi) rdi[_____]

_____(%rdi) rdi[_____]

_____(%rdi) rdi[_____]

_____(%rdi) rdi[_____]

6. Compares and conditional branches

The conditional branches in your code probably all depend on compare instructions. Most of the compare instructions refer to array elements that you considered in part 5 – focus on those compare instructions now.

a. What size is the data that is being compared? i.e. is it cmpb, cmpw, cmpl or cmpq?

b. List the values that are being compared with each array element, along with the index number. The type of the array elements is characters, so use an ASCII table to convert the values to ASCII characters and write them below. Then look back at your sketch from part 3… Do you want to take each branch or not? So, do you want the array element to equal the constant value or to not equal it? Write that information in the table below also.

Index	ASCII	Equal?
0		
1		
2		

7. Sketch a solution string

Fill out the characters that you know in the string. Here is space to sketch the string. Make sure that you write the character you want to equal. For characters that should *not* be equal, write some other character. If a character does not have a specified value then you can probably put any character at all in that position.

Index	0	1	2	3	4	5	6	7	8	9
Char										

Index	10	11	12	13	14	15	16	17	18	19
Char										

Index	20	21	22	23	24	25	26	27	28	29
Char										

Index	30	31	32	33	34	35	36	37	38	39
Char										

Index	40	41	42	43	44	45	46	47	48	49
Char										

Index	50	51	52	53	54	55	56	57	58	59
Char										

Index	60	61	62	63	64	65	66	67	68	69
Char										

Index	70	71	72	73	74	75	76	77	78	79
Char										

8. Try your solution

a. Start `gdb` on your bomb program.

b. Carefully set the breakpoints that you need to protect yourself (see part 2).

c. Set a breakpoint at the start of `phase_1` so that you can check the parameter before executing the phase.

d. Set a breakpoint at the start of `phase_2` so that your program will stop if you succeed in defusing `phase_1`.

e. Run your bomb inside `gdb`.

f. Enter your solution string. (You can write it down here if you like)

g. When the program stops at the `phase_1` breakpoint, check the string that has been passed as the parameter. Here is a `gdb` command to examine a string pointed to by `%rdi`. (Explanation: `x` means examine; `/s` means examine as a string; and `$rdi` is gdb's name for the register `%rdi` which contains the memory address that you want to examine).

```
(gdb) x/s $rdi
```

h. Did `gdb` show the parameter string being what you typed in? It should be, unless you made a typing mistake.

i. Use the following `gdb` command to show you the next instruction that will be executed each time you step through the code:

```
(gdb) display/i $pc
```

j. Now step through the `phase_1` module. As you go, track the instructions that are being executed on your printout from part 3.

- Do you arrive at the call to explode your bomb? If you do, you must *not* continue execution or the bomb will explode. You probably should `quit` out of `gdb`, update your analysis, and try again.
- Are the branches being taken or not taken as you expected? If a branch is not responding as you expect, take note of which branch it is.

k. If your `phase_1` function returns, use the `gdb` command to continue execution so that your success is registered by the server. You will receive some congratulatory messages from `bomb.c` and a function will be called to record your success. Then, when you are

asked for the next line of input, you can interrupt the program by pressing control-C and you can `quit` out of `gdb` knowing that you defused a bomb phase! You can check your mark using the `lab` command.

l. If you did not defuse your bomb in the previous steps, go back to part 6 of this worksheet and check your analysis of the particular branch instruction that did not behave as you expected. Update your analysis and then update your solution string and try again.

9. Try for the Dr Evil solution

Dr Evil's solution is more difficult. You probably have not yet found that solution. Here's how to identify Dr Evil's solution:

a. Look at `bomb.c` source file. After `phase_1` returns, congratulatory messages are printed out. There's an extra congratulatory message that is only printed for Dr Evil's solution. What value do you need to return (or what condition applies to the return value) to print this extra congratulatory message?

b. Was that message printed in response to your solution? If so, then you have solved Dr Evil's solution and you can verify your mark with the `lab` command, and then move on to phase 2. But if you didn't get Dr Evil's solution, you can keep working on it.

10. Assembly understanding that you need for Dr Evil's puzzle

To solve Dr Evil's puzzle, you need to understand the following instructions, which were discussed in lectures. Also see the *X64 Assembly and ASCII Reference Sheet*.

Instruction	Meaning
movzbl	Move zero-extended byte to longword. Move a single byte from the source into the longword destination. The high bits of the destination are filled with zero bits.
movsbl	Move sign-extended byte to longword. Move a single byte from the source into the longword destination. The high bits of the destination are filled with copies of the sign bit (the highest bit) of the source byte.
lea	Load effective address. Computes the memory address expression which is the source, but does not access memory. Instead, it puts the computed memory address into the destination register. Lea can compute 64 bit expressions or 32-bit expressions – you can tell which is being done by looking at the destination register.
mov	Move. Takes source data and puts it in destination. Can move byte, word, longword or quadword. You can tell which by looking at the register name.

You also need to know the registers, including the 64-bit registers, and the embedded 32-bit, 16-bit and 8-bit registers. See the *X64 Assembly and ASCII Reference Sheet*.

You need to know which register is used for the return value of a function – see the *X64 Assembly and ASCII Reference Sheet* if you don't remember this.

11. Solve Dr Evil's puzzle

a. Part of Dr Evil's extra puzzle is similar to what you have already analysed, so that should be easy for you now. Use the table of part 6 to analyse any remaining compare instructions that compare an array element with a value.

b. Find a path through your disassembly that represents Dr Evil's solution. i.e. A path that returns a value that will print the extra congratulatory message. Which conditional branches do you want to take, and which ones do you want to not take? For each of your ASCII characters listed above, show whether you want your solution to be equal or not equal to that character. Update these parts of your solution.

However, Dr Evil really is tricky, and he has created a slightly more difficult part to his puzzle. You need to work out some expressions (such as `lea` or arithmetic instructions) that relate one ASCII character to another. You work out what the expression is by writing pseudo-C next to the instructions. Then find a solution for the pair of characters that satisfies the expression. Any solution will be sufficient.

c. Which elements of the array pointed to by `%rdi` are mentioned in the code, but not in the compare instructions that you analysed earlier?

d. Identify the instructions that work with these array elements. Use the pseudo-C to work out what the code computes from them and what equality or inequality is required to reach Dr Evil's solution. Remember the relationships between the embedded registers

(e.g. that `%dl` is the low byte of `%edx`) – you can remind yourself by looking at the *X64 Assembly and ASCII Reference Sheet*.

e. Update the solution string from part 7 with the new information that you have identified.

f. Try your new solution, using the steps in part 8.

12. Questions?

If you have queries about this worksheet, please ask on the forum or by email to richard.han@mq.edu.au.

Good luck!