4190.308

Computer Architecture

Spring 2020

Bomb Lab Report

05/04 ~ 05/18

Name: Yang Kichang

Student ID: 2015-13078

Introduction

The objective of this lab was to figure out the input that needs to be given to each phase. Each phase takes in a single input, and there exists one correct input which can defuse the phase, while other inputs make the bomb explode. In order to figure out the correct input, I needed to disassemble the given executable file (bomb) and analyse the code with the help of debugger tool.

Solution

Phase1~Phase5: bomb56, Phase6: bomb2

Phase 1

First I noticed that %rdi holds the address of the string I enter (by x/s command). Then, from the function name <strings_not_equal>, I could infer that it compares two strings, which are given in %rdi and %rsi. Since %rsi points to 0x402490, by examining that address, I found out the string saved in there is “Iguanas were falling out of the trees.” The return value goes to %eax and phase_1 is defused when the value in %eax is 0.

Just to check whether <strings_not_equal> really operates as I think, I looked at the code. %rbx and %rbp has the address of the strings to compare. It first compares the lengths of two strings. If they’re not equal, it returns 1. If equal, it compares each byte of two strings until the byte is 0(null) or the bytes from two strings are different. In short, <strings_not_equal> returns 0 if two strings are equal, 1 if not, which is what I expected.

Therefore, the key was to pass the same string as the string in 0x402490.

Phase 2

Two inputs are given to <read_seven_numbers>: %rdi which is the input string I give, and %rsi which is the position of stack pointer (while in phase_2). From the function name, I could infer that this function reads seven numbers, and I needed to find out where those numbers are saved.

Looking at the code of <read_seven_numbers>, I found out that the function calls <sscanf>, and the arguments are 1) rdi: input string, 2) esi: 0x4026e1 3)~9) %rsi, %rsi+0x4, … ,%rsi+0x18. After examining 0x4026e1 which points to the format string “%d %d %d %d %d %d %d”, I knew that seven integers are going to be saved in addresses %rsi, %rsi+0x4, …, %rsi+0x18. I checked this by giving random seven numbers and examining those addresses at the end of <read_seven_numbers>. Also, <sscanf> returns the number of items that are successfully read, and <read_seven_numbers> tests if the return value is greater than 6.

Back to phase_2, let’s call the seven integers i1, i2… i7. Phase_2 first tests if i7-0x400 > 0x3fc00 by unsigned comparison. So i7>262144. Then it tests the remaining numbers with a loop where %ebx is a counter that starts from 5 and decreases until -1. At the first iteration, %rbp points to i6, at the second iteration i5, then i4 and so on. To avoid explosion, the following condition should be satisfied.
(Number that %rbp+4 points to) + Counter == (Number that %rbp points to).
This means that i7+5==i6, i6+4==i5… i2+0==i1.

Therefore, I set i7=262145, i6=262150, i5=262154, i4=262157, i3=262159, i2=262160, i1=262160.

* I had difficulty when checking values in memory address because the command “x $rsi” kept printing different results. It was because if I don’t specify SIZE or FORMAT(ex. b/h/w/g, d/x/o), the last value that I specified is used.

Phase 3

Phase 3 was almost same as phase 2. First, it calls <sscanf>, with format string “%d %d %d %d” in 0x4026ea and the four integers are saved in addresses %rsp+0xc, %rsp+0x8, %rsp+0x4, %rsp. I could check that this is true by giving random inputs and examining the addresses, just like phase 2.

Let’s call the four integers i1, i2, i3, i4. It first checks if i1 - 0x69 <= 0x8 by unsigned comparison. Next it checks if i2 <= 0x1bc by signed comparison. Lastly, it checks if i3 == 0x226. It has no restriction about i4.

Therefore, I set i1=113, i2=0, i3=550, i4=0. Of course, there are lots of other options with i1, i2 and i4.

Phase 4

First, it reads the string and checks if the length of the string is 7.

Then, it goes through a loop where %rax is a counter, starting from the address of the first character of the string, increasing until the address of the end of the string.
At each iteration, MEM[0x4024e0 + 4 * %rdx] is getting added to %ecx. This form is identical to how array is accessed where each element is 4 bytes, so I could infer that this is an array access. Examining the address 0x4024e0, I found out it is actually an address of an array of 15 integers, as I expected.
Thus the code is using the %rdx as the index of the array, and adding the elements at the corresponding indices. %rdx is each character of the string, ANDed with 0xf.

After the loop, it tests if the sum equals to 0x3c (=60). Therefore, I needed to pick 7 elements of the array which add up to 0x3c, and enter their indices as the input string.
Also, since the input is treated as string and each character is used one by one, the index I choose should be less than 10. Because the array is {13, 9, 14, 8, 12, 11, 8, 3, 10, 7, 3, 1, 9, 15, 6}, I chose the input “3333334”, which adds up to 8*6+12*1 = 60.

Phase 5 (optional)

First, it reads 3 integers through <sscanf> and save those in address %rsp+4, %rsp+8, %rsp+12, just like before. Let’s call them i1, i2, i3. It checks if 4<=i3<=16.
Looking at the big picture, phase 5 calls func5 twice, which has 3 arguments. Specifically, it calls func5(rdi=rsp, esi=8, edx=i3) twice with same argument values, and checks if the sum of two return values equals to i2. Thus func5(%rdi=%rsp, %esi=8, %edx=i3) should return i2/2.

Then I looked at the code of func5. It calls func5 recursively inside the function, so I needed to find the base case. The base case was (1) if %esi<=0, return 0 (2) if %esi==1, return edx.
Else, it returns (func5(%rdi=%rsp, %esi=%esi-1, %edx=i3) + func5(%rdi=%rsp, %esi=%esi-2, %edx=i3) + %edx)
I couldn’t understand what exactly the purpose of this function is, but I could calculate the return values based on the recurrence relation. For example, if %esi==2, it calls func5 with %esi=1

and %esi=0, add them, then add %edx to it. So the return value is %edx+0+%edx = 2*%edx. The rest of the values are as follows.

%esi	0	1	2	3	4	5	6	7	8
Return	0	%edx	2*%edx	4*%edx	7*%edx	12*%edx	20*%edx	33*%edx	54*%edx

Going back to phase 5, if I set i3=5, return value of func5(%rdi=%rsp, %esi=8, %edx=i3) is 54*5 = 270 and this should be equal to i2/2. Therefore, i2=540. (There is no restriction about i1.)

Phase 6 (optional)

(bomb 2)
First it reads seven integers by <read_seven_numbers> as before, and the seven numbers i1, i2… i7 are saved in addresses %rsp+0x40, %rsp+0x44…%rsp+0x58.

Then it goes through a nested loop. In the outer loop, it tests if each number is in the range 1~7. Then for each number, it tests if the number is different from the other numbers. Thus, the seven numbers should be one of 1~7, and all of them should be different.
Next, it changes i1 to 8-i1, i2 to 8-i2… so on. The seven numbers are still in the range 1~7.

Next, by examining the address 0x6042e0, I figured out there is an array of 7 structs(node1, node2… node7) which has an integer “value”, “index”(1,2…7) and a pointer to “next node” as members.
Knowing that accessing MEM[address of node(i) + 8] is accessing the next node of node(i), I could recognize that the code is saving the address of the node(8-i1) in MEM[%rsp], node(8-i2) in MEM[%rsp+8] and so on.
Also, it changes the “next node” member of the nodes. Specifically, it assigns node(8-i1).next = node(8-i2), node(8-i2).next = node(8-i3) and so on.
Finally it tests node(8-i1).value >= node(8-i1).next.value >= node(8-i1).next.next.value…

Therefore, what I had to do was to order the “value” members of the nodes and provide their indices. The “value” member of node_1~node_7 are 391, 663, 346, 824, 858, 715, 942.
So if I give 1 3 4 2 6 7 5 as input, 8-i1 ~ 8-i7 will be 7 5 4 6 2 1 3, which is the correct order.

* Phase 6 code was very long and complicated so I almost had to stop line by line to examine what that line is actually doing. Figuring out how the struct looked like and how its members are accessed was the key to finding the answer.

Conclusion

After spending lots of time reading the code again and again, I got familiar with x86-64 instructions and got used to reading complicated assembly codes, like nested loops, recursive function and accessing linked list (although I’m not a fast reader of assembly code yet).

Especially, I learned that when I have to figure out what the given code is doing, ‘just giving some random inputs and examining the result values’ is much faster than ‘looking at the code only with my eyes and trying to analyse it without running it’.

I also got familiar with the gdb debugger tool, which I think will also be useful when I’m coding with higher level language. However I think that more graphical debugger tool would be much more helpful.

Overall, reading a very long assembly code was actually quite difficult thing to do, but I think I learned a lot and it was worth it.