4190.308

Computer Architecture

Spring 2020

Bomb Lab Report

05/04 ~ 05/18

Name: Kim Gideok

Student ID: 2018-13627

Introduction

This bomb lab is a useful tool for learning x_86 assembly code. I should understand bomb code written in x_86 assembly and follow the instructions to find out the keys that defuse the bombs in bomb code (in my case, bomb69). It is so difficult to analyze large assembly code only using text reading application like vim. Fortunately, gdb helps me to read such a long assembly code. With gdb, I can pause bomb69 at any moment while running by marking break points, check all registers, and find out the value(string, integer) in any address if I want to know. Furthermore, processing one instruction or function are available. The purpose of bomb lab is making me accustomed to x_86 instructions. I learned what operations in instructions do and what happens in registers and memories that result from the operations by using gdb. Also, with gdb, I can obtain knowledge about the way how to communicate between caller and callee when a function call occurs. Finally, I found out that what functions do in bomb69 by analyzing instructions, registers and memories.

Solution

Phase 1

In phase_1, register rdi contains the address that is the start of input string. It soon moves to rbx. After init, register esi is going to be filled with 0x402470, which is the start address of “They got there early, and they got really good seats.”. Also, rbx’s value which contains input string’s address moves to register rdi. Esi and rdi is two parameters of function ‘strings_not_equal’.
In strings_not_equal function, first, it saves rbp and rbx in stack. And it calls function ‘string_length’ for each parameter. It returns length of the target string. And compare them. If they are different, return 1. If two strings have same length, check string1.charAt(i)==string2.charAt(i). If it pass this test, return 0. Else 1. Before return, they restore rbp, rbx value by poping from the stack. Back to phase_1, now, eax contains 0 if two strings are the same and the bomb will be defused by test. If not, it contains 1 and the bomb will explode. So the answer is “They got there early, and they got really good seats.”

Phase 2

In phase_2, stack point decreases 28 which is 4 times 7. So, I thought that it receives 7 inputs. Moving rbx which is my input, to rdi that is a parameter of ‘read_seven_numbers’.
In read_seven_numbers, it also makes stack point go down by 28. Then, copy addresses where input values will be stored. Then, write esi with 7 ‘%d’s. It means seven integers. And calls scanf. It returns number of inputs. So, right after the end of scanf, eax contains the number of inputs.(first seven inputs are valid) If it is less of equal than 6, the bomb booms! So I realized that I should write seven integers. Furthermore, input values are stored in stack.
Back to phase_2, Mem[rsp+0x18]=last input integer. It is moved to eax. Not to boom a bomb, following should be required : unsigned(eax-0x400)>unsigned(0x3fc00). In other words, eax should not be 0x400<=eax<=0x3fc00+0x400. So, I took 1 as the last input which does not explode bomb. After that, there is a for loop. To be more specific, I translate it to pseudo code.
For i is 5 to 0:
    if(input[i]!=input[i+1]+i) explode bomb!!
So, the value should be 16 16 15 13 10 6 1 because I took the last value as 1.
There would be so many answers because I can take many last value following the condition.
So, my answer is : 16 16 15 13 10 6 1

Phase 3

In phase_3, r8 = rsp+8, rcx = rsp+4, rdx = rsp+12. With esi = “%d %d %d”, go to scanf. So, I realize that first, second, third input will be saved in rsp+12, rsp+4, rsp+8 respectively. By {cmp 2, eax} and jg instruction, inputs should not be less than 2.
Then, Mem[rsp+12] = first input is moved to eax. And unsigned(eax-0xe3)<=unsigned(0x8). So, it should be more or equal than 227 and less than 235. Therefore, I chose 230.
Similarly, next, eax receives Mem[rsp+8]=third input. It should be required that unsigned(eax-0x371)<unsigned(0x8) not to touch bomb. Therefore, the third value would be anything which does not in [881, 889). So, I took 1.
Last, second input should be equal to 0x1fd which is 509 in decimal.
Therefore, my answer is : 230 509 1

Phase 4

This phase was the most complex among phase_1 to phase_4. First of all, go scanf with esi = “%d %d”. So, I thought that 2 integer inputs are needed. Two inputs are saved in rsp+12, rsp+8 respectively. First, check that the number of inputs is not less than two.
Then, eax = Mem[rsp+12] = first input’s last four digit in binary. It means eax = first input(mod 16). And reset Mem[rsp+12] = eax. In other words, it means Mem[rsp+12] %=16. If it is equal to 15, the bomb explodes.
Next, ecx = 0x64 = 100, edx = 0. Now, there is a loop. Also, eax contains first input mod 16.
Do
    edx++
    rax=eax
    eax = Mem[4*rax+0x4024e0]
    ecx-=eax
While eax!=15

The reason why compiler didn’t do eax = Mem[4*eax+0x4024e0] is making eax to 64bit because of the instruction form. Anyway, make these pseudo code more understandable,
i=0, remain=100, val = first input mod 16.
Do
    i++
    val = arr[val]
    remain-=val
While val!=15
Then, let’s look the array which starts at 0x4024e0.
By using x/d tool, I found that the array looks like following.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	2	14	7	8	12	15	11	0	4	1	13	3	9	6	5

Also, it can be represented like that with reflecting the situation.
0 → 10 → 1 → 2→ 14 → 6 → 15 → 5 → 12 → 3 → 7 → 11 → 13 → 9 → 4 → 8 → 0
The loop breaks when the value reaches 15. Remain contains 100 – sum of values of every val it was. For example, if the first input is 2, val follows 14 → 6 → 15, remain would be 100 – (14+6+15), i would be 3. Anyway, after break the loop, i should be 15, remain should be the same as second input.
Eventually, first input should be 5 (mod 16) to make i 15.(I took 5 as first input) Then, remain would be -15 which the second input should be.
Therefore, my answer is : 5 15

Conclusion

Until before this lab, I’ve learned x_86 instructions only with letters. Bomb lab was my first experiment about x_86 assembly. I’ve already known what instructions do in code. But there were some instructions that I had not known exactly. I obtained knowledge about memory form which can be written like D(Rb, Ri, S). Not only this acquisition, I learned what happens when using function call. From parameters and return register, above all things, I realized that stack plays an important role in a program. It saves and restores values like storage.
I also have difficulty with this lab. I totally didn’t know about how many inputs should be needed. I solved this problem with searching some strange values. I investigated that value with x/d and x/s. Then, I found that x/s with that value refers a string that is some number of %d. I recognized that it indicates the number of inputs I should type in. I first saw cltq instruction so that I confused. I conquered this problem with googling. Cltq means moving 32 bit to 64 bit with sign extension. It looks like meaningless but there are various bit form. Furthermore, every form is essential to compose x_86. Instructions are sensitive to the number of bits of a value. I also learned that how many bits the value has is so important.
Condition codes and branching operations struggled me too. Unlike RISC-V, it uses condition code also called flags. Branch operation reacts with these flags. There are so many branching operations and it is so unfamiliar with me that setting condition codes. I overcame with googling but I think I need more study about condition codes.
Now, I become getting more senses in assembly coding than before.