

**class Procedure Reality test report tell**

**Course Title: Computer System Fundamentals**

**professional class: School cross 1601**

**student ID: U201612696**

**Name: Chen Haorui**

**mentor: Shi Xuanhua**

**Date of report: June 20 , 2018 \_ \_**

**School of Computer Science and Technology**

# 

# Experiment 2: Bomb Disposal Experiment

**2.1 Experiment overview**

Purpose of the experiment: To enhance the mastery of the principles and skills of machine-level representation of programs, assembly language, debugger, and reverse engineering.

Experiment goal: need to defuse as many bombs as possible.

Experimental requirements: use gdb debugger and objdump to disassemble the executable file of the bomb, and step through the machine code of each stage of debugging, understand the behavior or function of each assembly language code, and then try to "deduce" the bomb to be dismantled The desired target string.

Experimental language: c .

Experimental environment: linux

Report writing environment : windows

**2.2 Experimental content**

A "binary bombs" (binary bombs, hereinafter referred to as bombs) is a Linux executable C program, including 6 phases (phase1~phase6). Each stage of the bomb operation requires you to input a specific string. If your input matches the expected input of the program, the bomb in this stage will be "demolished", otherwise the bomb will "explode" and print out the words "BOOM!!!" . The goal of the experiment was to defuse as many bomb layers as possible.

Each bomb stage examines a different aspect of a machine-level language program, with increasing levels of difficulty:

\* Phase 1: String comparison

\* Phase 2: Loop

\* Phase 3: Condition/Branch

\* Phase 4: Recursive calls and stacks

\* Phase 5: Pointer

\* Stage 6: linked list/pointer/structure

There is also a hidden stage, but it only appears if you append a specific string to the solution of stage 4.

In order to complete the task of dismantling the binary bomb, you need to use the gdb debugger and objdump to disassemble the executable file of the bomb, and step through the machine code of each stage to understand the behavior or role of each assembly language code, and then Managed to "deduce" the target string needed to defuse the bomb. This may require you to set breakpoints before the start code of each stage and before the function that detonates the bomb to facilitate debugging.

**2.2.1 Phase 1 String Comparison**

1. Task description: find out the string to be input through the disassembly code of phase\_1.
2. Experimental Design: Use gdb combined with breakpoints to analyze dynamically.
3. experiment procedure:

Observe the disassembly code of phase\_1, as shown in Figure 2.1.1 :

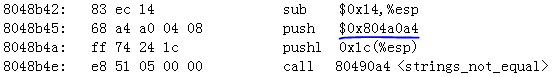


Figure 2.1.1

It was found that before calling strings \_not \_equal to compare the strings, an address was sent to the stack. I guessed that this address was the first address of the correct string, so I directly used the p command to view the string in gdb. As shown in Figure 2.1.2 :



Figure 2.1.2 \_

Therefore, it is guessed that " He is evil and fits easily into most overhead storage bins " is the required string. Re-execute the program, directly input the character string, and find that the bomb is successfully dismantled . As shown in Figure 2.1.3 :



Figure 2.1.3

**2.2.2 Phase 2 cycle**

1. Task description: Infer the data to be input in the second phase through the disassembly code of phase\_2
2. Experimental design: using gdb combined with breakpoint analysis
3. experiment procedure:

Observe the disassembly code of the previous part of phase\_1, as shown in Figure 2.2.1 :

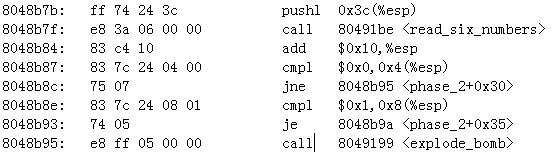


Figure 2.2.1

It is found that the input should be six numbers, and the first number is required to be 1, and the second number is 2. Continue to observe the code backwards, as shown in Figure 2.2.2 , and find that this is a cycle, and the length of ebx as a pointer is The pointer of the array of 6 first points to the first element 0, and then whether the sum of the next element and the first two elements is equal is used as the judgment condition, that is, 6 numbers form the Fibonacci sequence: 0,1,1,2,3,5 .

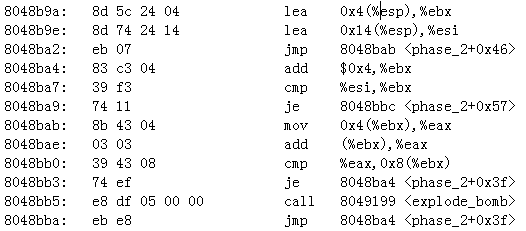


Figure 2.2.2

Re-execute the program, input "0 1 1 2 3 5", it was found that the bomb was successfully dismantled . As shown in Figure 2.2.3 :



Figure 2.2.3 \_

**2.2.3 Phase 3 condition/branch**

1. Task description: Infer the data to be input in the third phase through the disassembly code of phase\_3
2. Experimental design: use gdb combined with breakpoints to analyze dynamically
3. experiment procedure:

the experience of stage 1, the address 0x804a106 pushed onto the stack before calling the sscanf function will be used as the parameter of the function , as shown in Figure 2.3.1 :

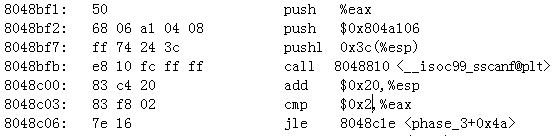


Figure 2.3.1

So put this address into gdb to view the content, the result is shown in Figure 2.3.2 , the function requires input of 1 int, 1 char and 1 int, and it will jump to explode\_bomb detonates the bomb, and the side confirmation parameters should be 3:



Figure 2.3.2

Continue to observe the latter part of the disassembly code of phase\_3, as shown in Figure 2.3.3 :

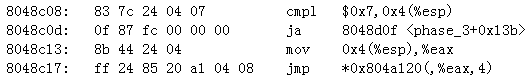


Figure 2.3.3

Apparently, a number cannot be greater than 0x 7 , and at the same time , the iconic statement of the jump table \*0x804a120(,%eax,4) appears , which proves that there is a switch structure after it . Continue to look down at the code . The following codes correspond to the second and third input when the first input is 0 to 7. The second number is regarded as ASCII code , and the third number should be converted to decimal. As shown in Figure 2.3.4 -2.3.11 .



Figure 2.3.4



Figure 2.3.5



Figure 2.3.6



Figure 2.3.7



Figure 2.3.8



Figure 2.3.9



Figure 2.3.10



Figure 2.3.11

Only when the first input is 0 and 1 are selected for verification, as shown in Figure 2.3.12 and 2.3.13, it proves that the third stage bomb disposal is successful:



Figure 2.3.12



Figure 2.3.13

**2.2.4 Phase 4 recursive call and stack**

1. Infer the data to be input in the fourth phase through the disassembly code of phase\_4 and func 4
2. Experimental design: use gdb combined with breakpoints to analyze dynamically
3. experiment procedure:

Observe the disassembly code of phase\_4, as shown in Figure 2.4.1 , and find that the address before sscanf appears again is pushed onto the stack. According to the experience of phase 3, the following code shows that the parameter for sscanf is 2.:

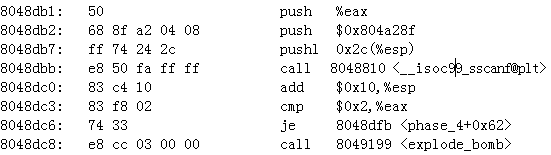


Figure 2.4.1

in gdb , the result is shown in Figure 2.4.2 , which shows two ints:



Figure 2.4.2

According to the jump of the program, skip the position where fnc4 appears and look back and find that the first int required to be input is ≤ 14, and the second int is determined to be 4, as shown in Figure 2.4.3 :

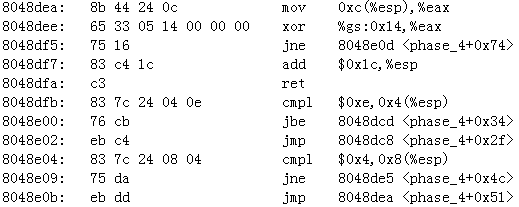


Figure 2.4.3

Since the range of the first input cannot be determined, it is advisable to directly use the enumeration to test the bomb here , and output the correct result when the input is 2 4, as shown in Figure 2.4.4 :



Figure 2.4.4

**2.2.5 Phase 5 pointers**

1. Task description: Infer the data to be input in the fifth stage through the disassembly code of phase\_5
2. Experimental design: use gdb combined with breakpoints to analyze dynamically
3. experiment procedure:

Observe the disassembly code of the previous part of phase\_5, as shown in Figure 2.5.1 , and find that the output of the string\_length function should be 6, that is, there should be 6 inputs at this stage :

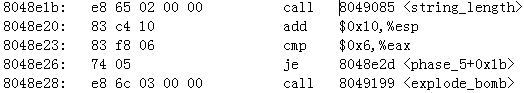


Figure 2.5.1

Observe the next part of the code , as shown in Figure 2.5.2 , it is found that this part is a loop with a count of 6 , and the 6 inputs are sequentially combined with 0xf and the last value as the offset address to find the element in the table with the first address of 0x80 4a140 and sum, which sums to 0x18:

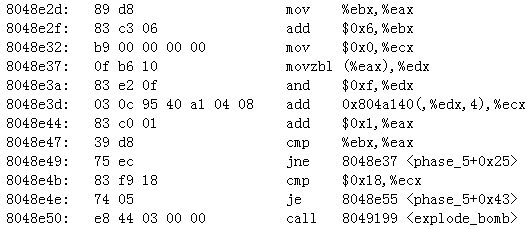


Figure 2.5.2

Observe the above address in gdb , as shown in Figure 2.5.3 . Since the sum of the 6 numbers is only required to be 24 and there is no requirement for the specific value of each number, the 6 numbers are all 4, and its offset address relative to 0x804a140 is 32, and the scale factor is 4, so the offset address is 8:

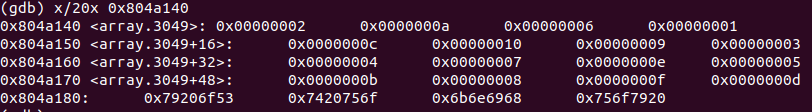


Figure 2.5.3

Check the ASCII code table, the ASCII code of h is 68 , input 6 h, and find that the bomb is successfully dismantled, as shown in Figure 2.5.4 :



Figure 2.5.4 \_

**2.2.6 Phase 6 linked list/pointer/structure**

1. Task description: Infer the data to be input in the fifth stage through the disassembly code of phase\_6
2. Experimental design: use gdb combined with breakpoints to analyze dynamically
3. experiment procedure:

the disassembly code of the first part of phase\_ 6 , as shown in Figure 2.6.1 , and find that 6 numbers need to be input at this stage , and at the same time, it is required that the subtraction must not be greater than 5, that is, it must be between 1, 2, 3, 4, 5, Only numbers in 6 are allowed:

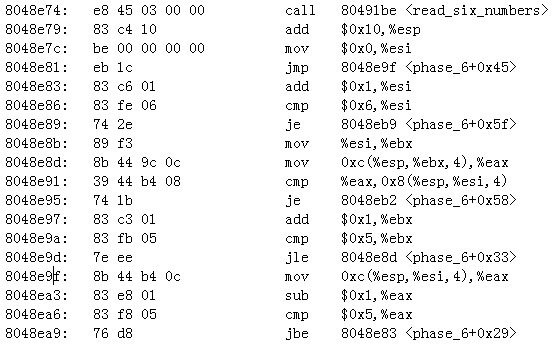


Figure 2.6.1

Continue to look at the next section of disassembly code of phase\_ 6, it is not difficult to find that this is a double loop realized by esi and ebx, and compare the input values one by one. As long as there is a pair of the same, the bomb will explode, so the input data should be It is a permutation of 1 to 6. As shown in Figure 2.6.2 :

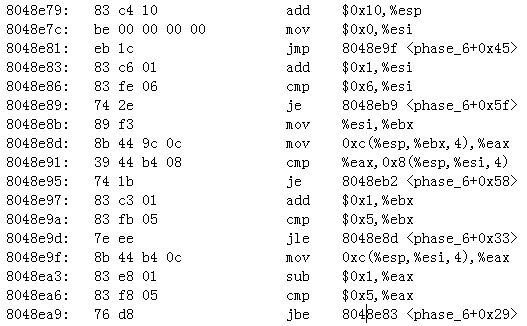


Figure 2.6.2 \_

Continue to look at the next piece of code. This section is another loop , changing the input x to 7 -x, as shown in Figure 2.6.3.

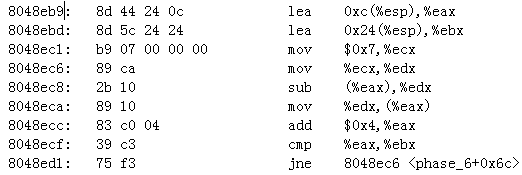


Figure 2.6.3

Continue to look at the next section of disassembly code, where the address appears again. As shown in Figure 2.6.4.

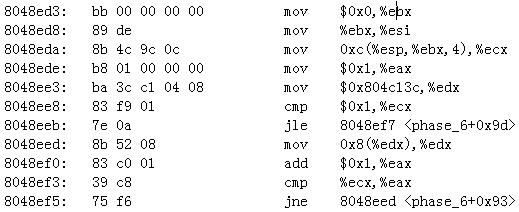


Figure 2.6.4

in gdb , it is found that the space after it is in the form of a linked list, which conforms to the prompt of this stage. Each node in the linked list has three elements, which are value, serial number and pointer to the next node, as shown in Figure 2.6.5:

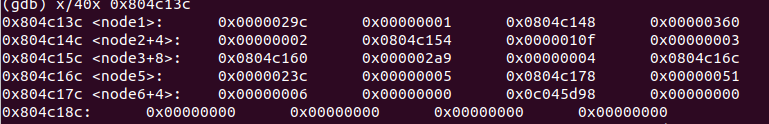


Figure 2.6.5

the last piece of code at this stage , as shown in Figure 2.6.6 , and find that its operation is to check whether the elements in the linked list are sorted in descending order, so the 6 numbers we input that are less than 7 should be subtracted by 7 to form a new sequence, which is the same as the The numerical order of the nodes that the sequence matches is in descending order. The descending order is 2(0x360 ), 4( 0x2a9 ) , 1(0x29c), 5(0x23c), 3(0x10f), 6(0x51), so the input should be 5 3 6 2 4 1.

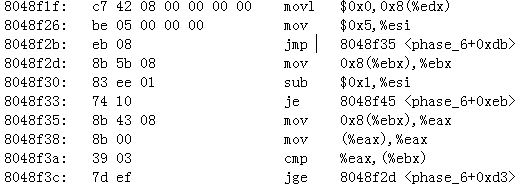


Figure 2.6.6

Re-run the program and enter the string for testing, as shown in Figure 2.6.7:



Figure 2.6.7

**2.2.7 Stage 7 hidden stage**

1. Task description: Find out how to open the hidden stage and defuse the bomb in the hidden stage.
2. Experimental design: use gdb combined with breakpoints to analyze dynamically
3. experiment procedure:

Observing directly in the disassembly code, it is found that only the phase\_defused function has a call to the secret\_phase function, so observe the disassembly code of the phase\_defused function, as shown in Figure 2.7.1, and find that there is an address here , and the meaning can be seen in gdb To enter six character strings , that is, to meet the conditions here after completing the bomb disposal of the six phases , as shown in Figure 2.7.2:

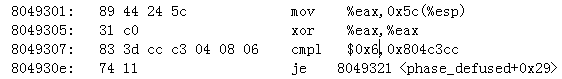
 Figure 2.7.1



Figure 2.7.2

Press Jump to look down, and find that sscanf needs three parameters to jump, and combined with the prompt, it is known that its specific form is the answer of phase4 followed by a string, as shown in Figure 2.7.3 :

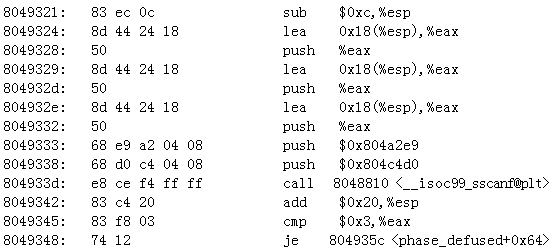


Figure 2.7.3

Continue to observe and find that after pushing an address on the stack, strings\_not\_euqal is called for string comparison, as shown in Figure 2.7.4 , it is guessed that the target string is stored in the address .

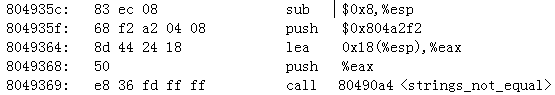


Figure 2.7.4

Put this address into gdb for observation , and get the string D rEvil, as shown in Figure 2.7.5 :



Figure 2.7.5

Add this string when decrypting phase4. After completing phase6 , you will find a pop-up prompt : find secret\_phase, as shown in Figure 2.7.6 :

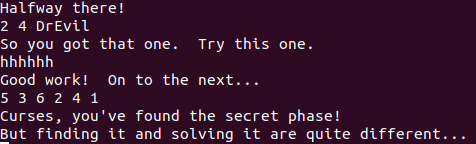


Figure 2.7.6

Observe the code of the secret phase , and find that the input of the phase is a value, and it can not be greater than 0x3e8+1 =1001 . In Phase phase, fun7 is called , one of the two parameters is the input value, and the other is the address ; the return value must be 2, as shown in Figure 2.7.7 :

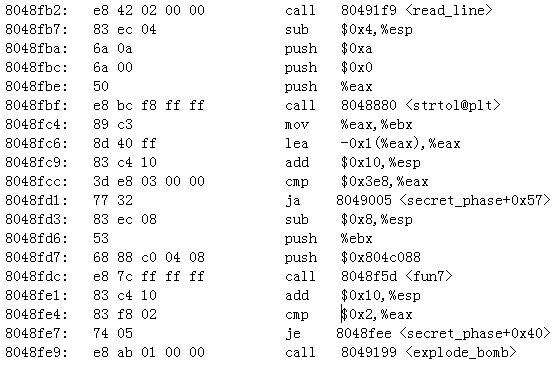


Figure 2.7.7

f un7 is a bit messy, but after analysis, it can be found that there are recursive calls. When calling for the first time, edx and ecx are 0x 804c088 , x (the input number). Then compare edx and 0, if they are equal, return -1, otherwise ebx= [edx], then compare ebx and ecx, that is, according to the value at the first parameter passed into fun 7 and the comparison result of the second parameter to correspond processing.

When the value at the first parameter is smaller than the second parameter, the original first parameter plus the value at 0x8, the original second parameter are used as parameters to call fun 7, and then the return value is multiplied by 2 and added to 1 is output as the return value. That is, the original parameters are a and b in sequence, and the incoming parameters are [a +0x8] and b in sequence;

When the value at the first parameter is greater than the second parameter, the original first parameter plus the value at 0x 4 and the original second parameter are used as parameters to call fun 7 , and then the return value is multiplied by 2 is output as the return value. That is, the original parameters are a and b in sequence, and the incoming parameters are [a +0x4] and b in sequence;

When the first parameter is equal to the average value of the two parameters, the average value is directly set to 0 and output. As shown in Figure 2.7.8 :

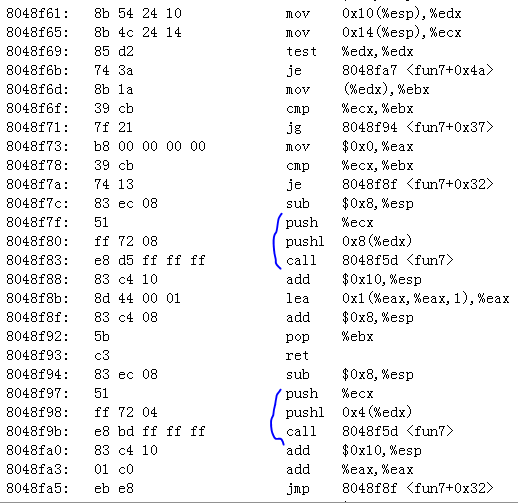


Figure 2.7.8

Since the return value is 2, the recursive method of multiplying by 2 is used , and it should be checked in 0x 804c088+0x4= 0x 804c08c . Observe the content starting from 0x 804c088 in gdb , as shown in Figure 2.7.9 , and find that 0x 804c08c is still an address :

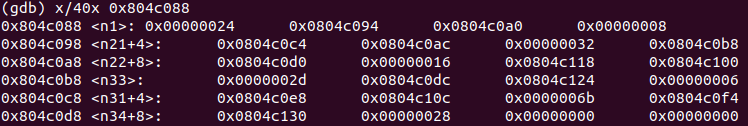


Figure 2.7.9

Put this address into gdb for observation , and find that its value is octal \ 026 , as shown in Figure 2.7.10 . Converted to decimal is 22.



Figure 2.7.10

Therefore, the input value should be 22 , so enter 2 2 after entering the secret phase , and the cracking is successful, as shown in Figure 2.7.1 1 :

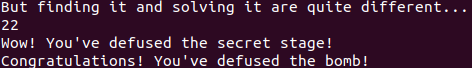


Figure 2.7.11

**1.3 Experimental summary**

Compared with the previous experiments, this binary bomb experiment is more interesting and novel, and of course the difficulty is also significantly improved. This experiment mainly uses the technology of reverse engineering. The forgetting of some details in the assembly language and the strangeness brought about by the different form of the assembly language in the experimental environment and the class have brought certain obstacles to my bomb disposal . The dense occurrence of passing parameters is also not a small challenge for me.

I think the key point of this experiment is to find the flow of input data, where it exists after input, and what operations are performed afterwards. This is the core of solving bomb disposal.

Through this experiment, I have a deeper understanding of the means of recursive calls. Although the recursive function requires a basic understanding of the recursive function, it took a lot of time, but this is also a kind of training for my patience. Patience is never good enough.

In short, the overall process of this experiment is still very pleasant. Although it takes a long time, the rewards are also great!