**Ezgi Akkaya**

**comp201**

**fall23**

**BOMB LAB**

**Phase 1**

First I created psol.txt to save my answers.

I disas’d the phase 1. Then I saw that there is a function call named “strings\_not\_equal”.. I understood that I needed to find the text corresponding to this memory address.

Dump of assembler code for function phase\_1:

=> 0x00005555555555b5 <+0>: repz nop %edx

0x00005555555555b9 <+4>: sub $0x8,%rsp

0x00005555555555bd <+8>: lea 0x1bbc(%rip),%rsi # 0x555555557180

0x00005555555555c4 <+15>: callq 0x555555555b3a <strings\_not\_equal>

(gdb) x/s 0x555555557180

0x555555557180: "Verbosity leads to unclear, inarticulate things."

I found the text. Then I added this text inside my psol.txt and run.

metin, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

**PHASE 2**

I observed the phase\_2 asm code by disas command and see :

mov %rsp,%rsi callq 0x555555555e23 <read\_six\_numbers>

The read\_six\_numbers function is likely reading six numbers into the location pointed to by %rsp.

cmpl $0x1,(%rsp) jne 0x55555555560b <phase\_2+50>

This checks if the first number is 1. If not, it jumps to the bomb explode function.

mov (%rbx),%eax add %eax,%eax cmp %eax,0x4(%rbx) je 0x555555555617 <phase\_2+62> jmp 0x555555555612 <phase\_2+57>

This loop seems to be comparing each number with the next, potentially to see if each subsequent number is double the previous one.

So I wrote 1 2 4 8 16 32 as my input.

ekran görüntüsü, yazı tipi, metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Phase 3: Conditional Logic and Memory Address Calculation**

* Started by disassembling the phase\_3 function to analyze its control flow.
* Noted that the function compares the input value with 2 and triggers the bomb if the input is less than 2.
* Chose 5 as the first input. This value is greater than 2 and less than 7, which avoids triggering the bomb and continues execution. At the instruction 0x000055555555568a <+67>: mov 0x10(%rsp),%eax, the program moves to the next significant part.
* Using nexti until the lea instruction is executed, then calculating the memory address used by the code.
* Calculated address: 0x555555557200 + 5\*4 = 0x555555557214.
* Used (gdb) x/1xw 0x555555557214 to determine the value stored at this address, which was saved in %eax. The value from the calculated address (0xffffe546) added to 0x555555557200 results in the next jump address.
* Further observed that %eax (now holding 0x64 or 100) was compared to 981 (0x3d5), which is the second argument entered by the user.
* The input character was then compared to a character stored in %al (retrieved by p/x $al, which was 0x5 or enquiry in ASCII).
* SO i wrote 5 d 981

**Commands & Code Used**:

scssCopy code

(gdb) disas phase\_3

(gdb) x/7xw 0x555555557200

(gdb) x/1xw 0x555555557214

(gdb) p/x $al

ekran görüntüsü, metin, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**PHASE 4**

nitially, I disassembled the function to understand its flow:

* The function uses **scanf** to read two integers.
* It validates that two integers are input. If not, it triggers the bomb.
* The first integer input is manipulated in a series of operations: it is first moved to the EAX register and then reduced by 2.
* The adjusted value of the first input is then compared to 2, indicating the input must be greater than or equal to 4.

The core of this phase was understanding **func4**:

* **func4** is a recursive function.
* It involves two recursive calls with decremented arguments.
* The result of these calls is combined and used in a comparison back in **phase\_4**.

Through testing in GDB, I discovered that an input of **36** for the first integer and **3** for the second integer led the program to the desired outcome:

* The program compares the result from **func4** (which was **36** in my testing) with the second input argument (which I set as **3**).
* If these values match, the program continues without triggering the bomb.

**GDB Commands Used**

* **disas phase\_4** and **disas func4** for disassembling the functions.
* **b \*0x0000555555555851** to set a breakpoint.
* **next** and **i r** to step through the code and inspect register values.
* Checking memory addresses and values.

ekran görüntüsü, metin, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**PHASE 5**

In Phase 5, I disas’d the assembly code to determine the proper six-character string input. The aim was understanding how each character of the input string was processed and added to produce a specific sum.

* The function verifies that the input string is exactly six characters long.
* Each character in the string undergoes bitwise operations and is used as an index in an array located at **0x555555557220**.
* The values retrieved from this array are then summed up.
* The final sum is compared against the decimal value **65** (**0x41** in hexadecimal).
* The array at **0x555555557220** contained hexadecimal values ranging from **0x01** to **0x0f**.
* To determine a correct input, you would need to know the values stored in array[] at 0x555555557220. The goal is to find six characters whose masked indices (last four bits of each character) sum up to 65 when added according to the values in array. Array [0x02, 0x0a, 0x06, 0x01, 0x0c, 0x10, 0x09, 0x03, 0x04, 0x07, 0x0e, 0x05, 0x0b, 0x08, 0x0f, 0x0d]
* As decimals [2,10,6,1,12,16,9,3,4,7,14,5,11,8,15,13]
* asm code indicates that each character of the input string is taken, its last four bits are extracted (using the AND $0xf, %edx operation), and this value is used as an index into an array at 0x555555557220. The array value at that index is then added to %ecx.
* The final value in %ecx is compared against 0x41 (65 in decimal). If the sum is not 65, the bomb explodes.
* The challenge was to find a combination of six characters whose last four bits (after the AND operation) corresponded to array indices that summed up to **65**. After analyzing the array values, I chose characters whose last four bits were **3**, **8**, and four times **14** in hexadecimal.
* These corresponded to **S** (**0x53**), **X** (**0x58**), and **N** (**0x4e**).
* The string **NNNNSX** met the criteria, with its characters translating to the required indices in the array.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**SECRET PHASE**

The secret phase of the bomb challenge was uncovered using the **objdump -d** command. It revealed that the **secret\_phase** is invoked from the **phase\_defused** function. Then I observd the phase\_defused function to reach where the secret\_phase func is called, I noticed that there is string equality check with the scanf function in %d %d %s type so the missing string should be in the fourth phase since it had %d %d input. I disas’d and see that the secret is DearDrEvil. I added the secret key to input of phase 4, **secret\_phase** is called.

* The function begins by reading a line of input, then converting this string into an integer.
* The input is restricted to be less than **1000** (**0x3e8**).
* Following this, the function **fun7** is called with the input value. **fun7** is designed to traverse a binary tree structure, starting at a specific memory address (**0x555555559150**).
* The function manipulates the **eax** register based on the tree traversal.
* When moving to the left child of a node, **eax** remains the same; when moving to the right child, **eax** is incremented by 1 and then doubleThe tree's structure was examined using the **x/60gx** command.

x/60gx 0x555555559150

0x555555559150 <n1>: 0x0000000000000024 0x0000555555559170

0x555555559160 <n1+16>: 0x0000555555559190 0x0000000000000000

0x555555559170 <n21>: 0x0000000000000008 0x00005555555591f0

0x555555559180 <n21+16>: 0x00005555555591b0 0x0000000000000000

0x555555559190 <n22>: 0x0000000000000032 0x00005555555591d0

0x5555555591a0 <n22+16>: 0x0000555555559210 0x0000000000000000

0x5555555591b0 <n32>: 0x0000000000000016 0x00005555555590b0

0x5555555591c0 <n32+16>: 0x0000555555559070 0x0000000000000000

0x5555555591d0 <n33>: 0x000000000000002d 0x0000555555559010

0x5555555591e0 <n33+16>: 0x00005555555590d0 0x0000000000000000

0x5555555591f0 <n31>: 0x0000000000000006 0x0000555555559030

0x555555559200 <n31+16>: 0x0000555555559090 0x0000000000000000

0x555555559210 <n34>: 0x000000000000006b 0x0000555555559050

36

8 50

6 22 45 107

1 7 20 25 40 47 99 1001

* The goal is for **fun7** to return a value of **6**.

he traversal required to achieve a return value of **6** from **fun7** would be "right, right, left," based on how **eax** is altered during traversalBased on this analysis, I hypothesized that the input value **99** would lead to the desired path. This input corresponds to the value **0x63**, which matches the right, right, left traversal required to achieve an **eax** of **6**. Because eax=, right 2\*0+1 = 1, right 2\*1+1=3, then left 2\*3=6.

when we traverse right right left, starting from root node, we come to node with value 99.Despite my theoretical analysis indicating that **99** would be the correct input, the bomb exploded when this value was used.

The input 47 gives 5, input 1001 gives 7 as expected. But 99 should give 6 here. However ,it returns 3.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

ps. i even tried brute forcing:

40 1

20 2

7 4

1 0

47 5

45 1

50 1