Name:\_\_Xiang, Xin\_\_ Date: \_\_\_\_ \_April 10, 2020\_ \_ \_

Section: \_ 005\_ \_ \_ Github Username: \_\_FreyaXiang\_ \_

**Lab 2**

Total in points (100 points total):

Professor’s Comments:

Honor Pledge: I have neither given nor received aid on this assignment.

Signature: Xin Xiang

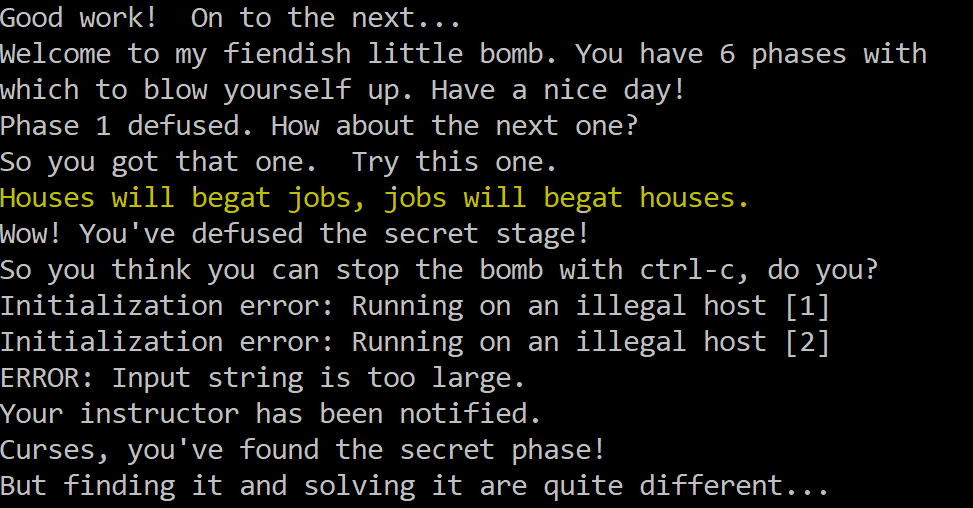
1. **Binary Bomb Refusal**

**Phase 1:**

1. Use command “strings bomb” to display all printable strings in the bomb.



1. Then we can find a “strange” sentence, which is “Houses will begat jobs, jobs will begat houses”. It seems that this sentence has no relationship with our bomb, compared to other sentences. So we can have a try.



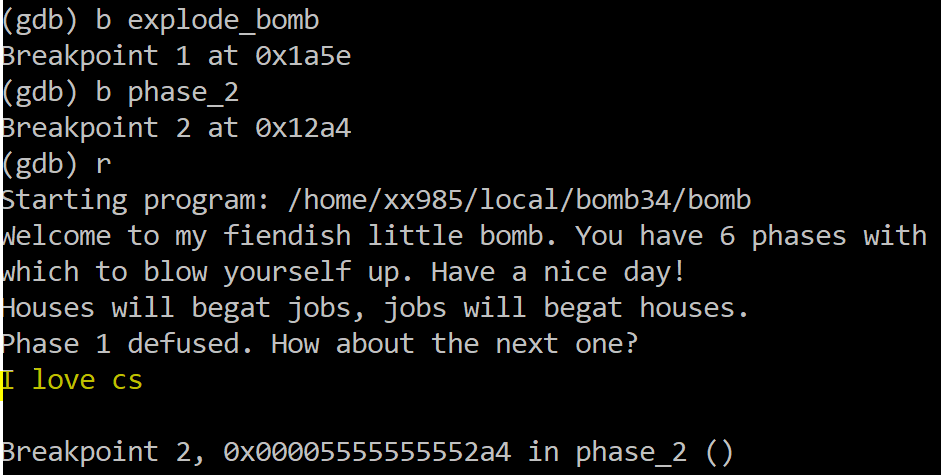
1. Use gdb to run the program. Before running, we should set a break point at function explode\_bomb in case the sentence we found is wrong.



We can see that phase 1 has been defused, indicating that our sentence is correct.

**Phase 2:**

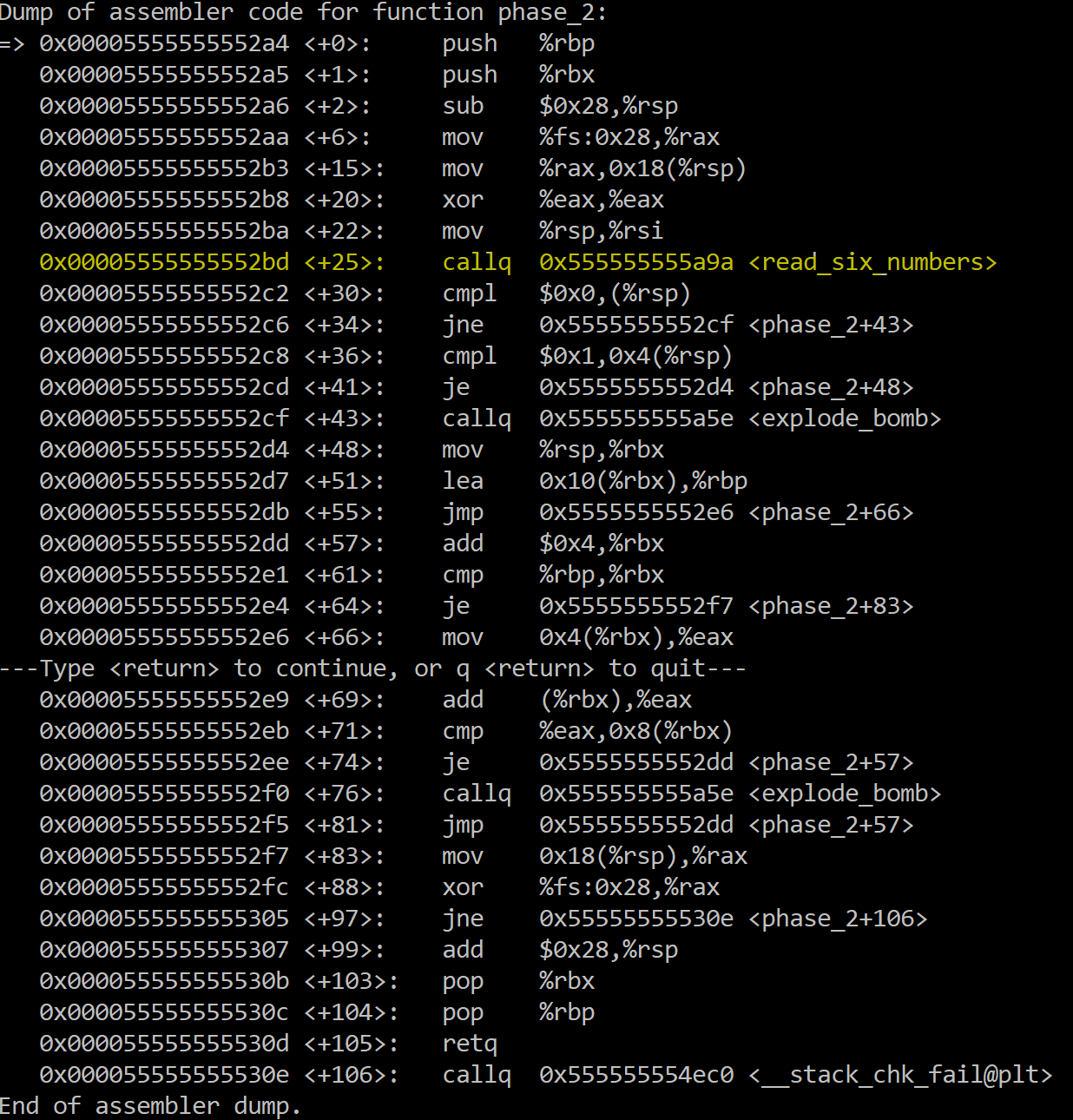
1. Enter gdb, set a break point at function phase\_2 and explode\_bomb, then type in a arbitrary string into phase 2.



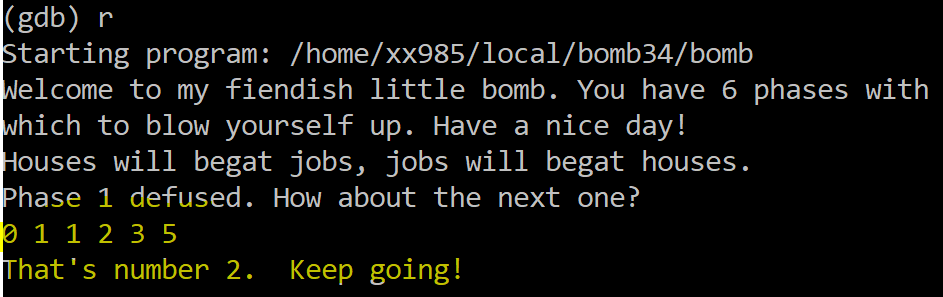
1. Then disassemble the phase\_2 function.



1. Then we can see the assembly code for phase\_2, which could help us figure out what the second string is.

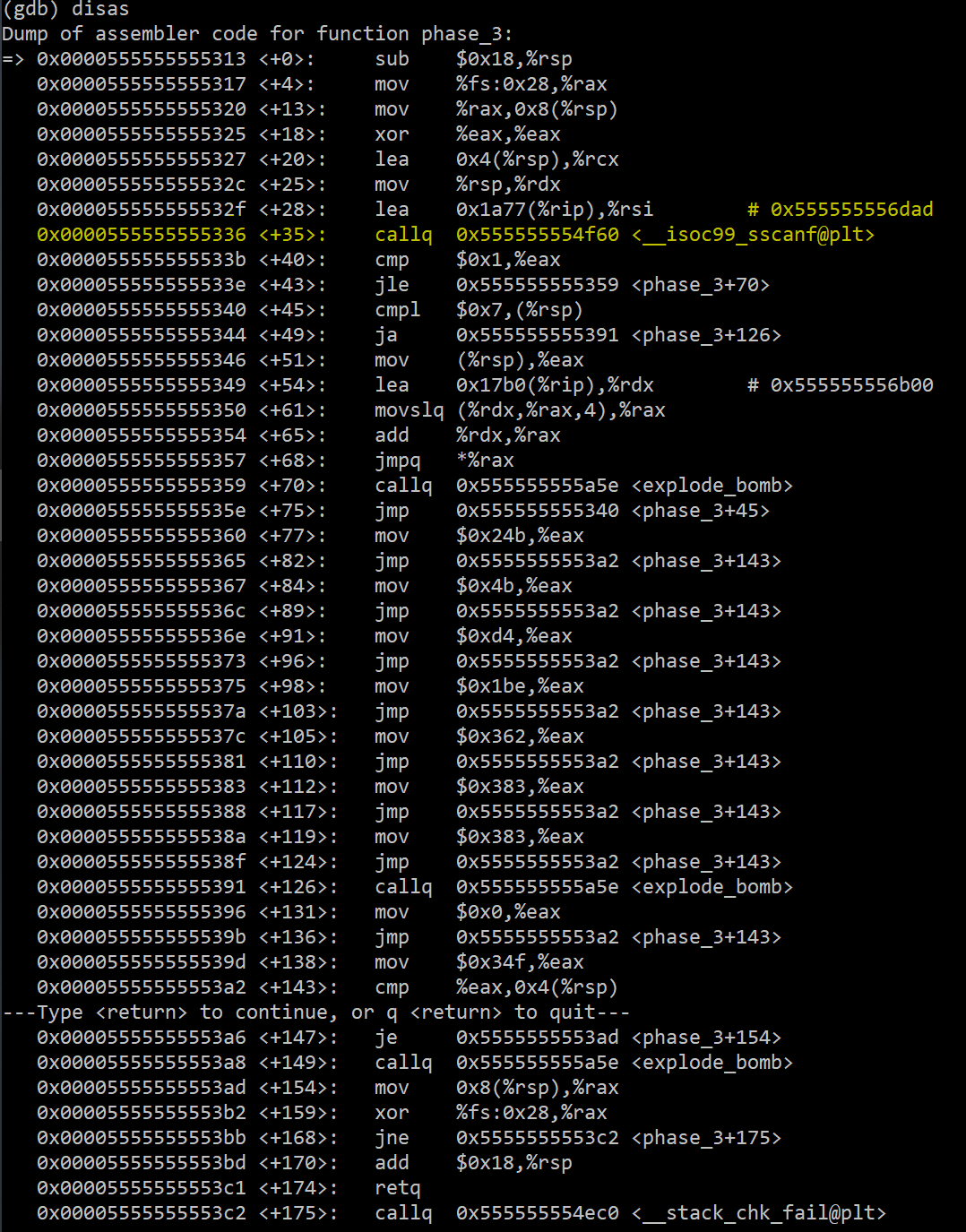


1. First, at line <+30>, we compare (%rsp) to 0x0, and if (%rsp) != 0, then the code jumps to line <+43>, which calls the explode\_bomb function. This means our first number must be 0.
2. Then if the first number is 1, we come to line <+36>, which compares 4(%rsp) with 1, and if 4(%rsp) == 1, the code will jump to line <+48>, otherwise it will call explode\_bomb ar line <+43>. This means that the second number must be 1.
3. After we jump to line <+48>, we know that now %rbx=%rsp. From step i, we know that (%rsp)=0, so (%rbx)=0.
4. Then at line <+55>, the code jumps to <+66>. At line <+66>, we see that %eax=4(%rbx)=4(%rsp). From step ii, 4(%rsp)=1, so now %eax=1.
5. Then we come to line <+69>, where we add (%rbx) to %eax. From step iii, we know (%rbx)=0, we %eax=1+0.
6. Then at line <+71>, the code is comparing %eax and 8(%rsp), which is the third number, and if they are equal, we jump to line <+57>, otherwise the explode\_bomb will be called at the next line. So the third number must be 1+0=1.
7. Then look at line <+74>. We can see that the code is jumping back to 57 and except for line <+64>, there is no jump between line <+57> and <+74>. This means that we have a loop here. As we can see, the loop is actually keeping letting the next number we entered equals to the sum of the previous two numbers, just like the Fibonacci numbers. And the condition of the loop is at line <+61>, which is to check whether the number we checked right now is the last number.
8. Because the name of the function is read\_six\_numbers, so we can guess the answer to this phase is 6 integers 0 1 1 2 3 5.
9. Enter these numbers into phase 2, we can see that we defused phase 2.

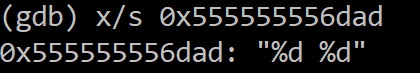


**Phase 3:**

1. First use the same command as phase 2 to see the assembly code of phase 3.

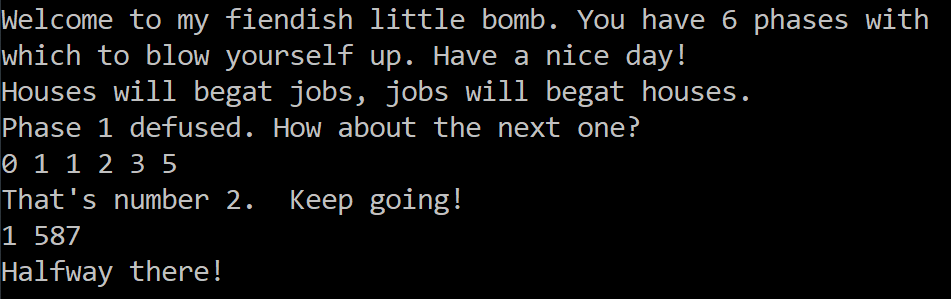


1. Then we notice that there is a comment at line <+28>, which may give some hints about what the answer is like. Use command “x/s” in the terminal.



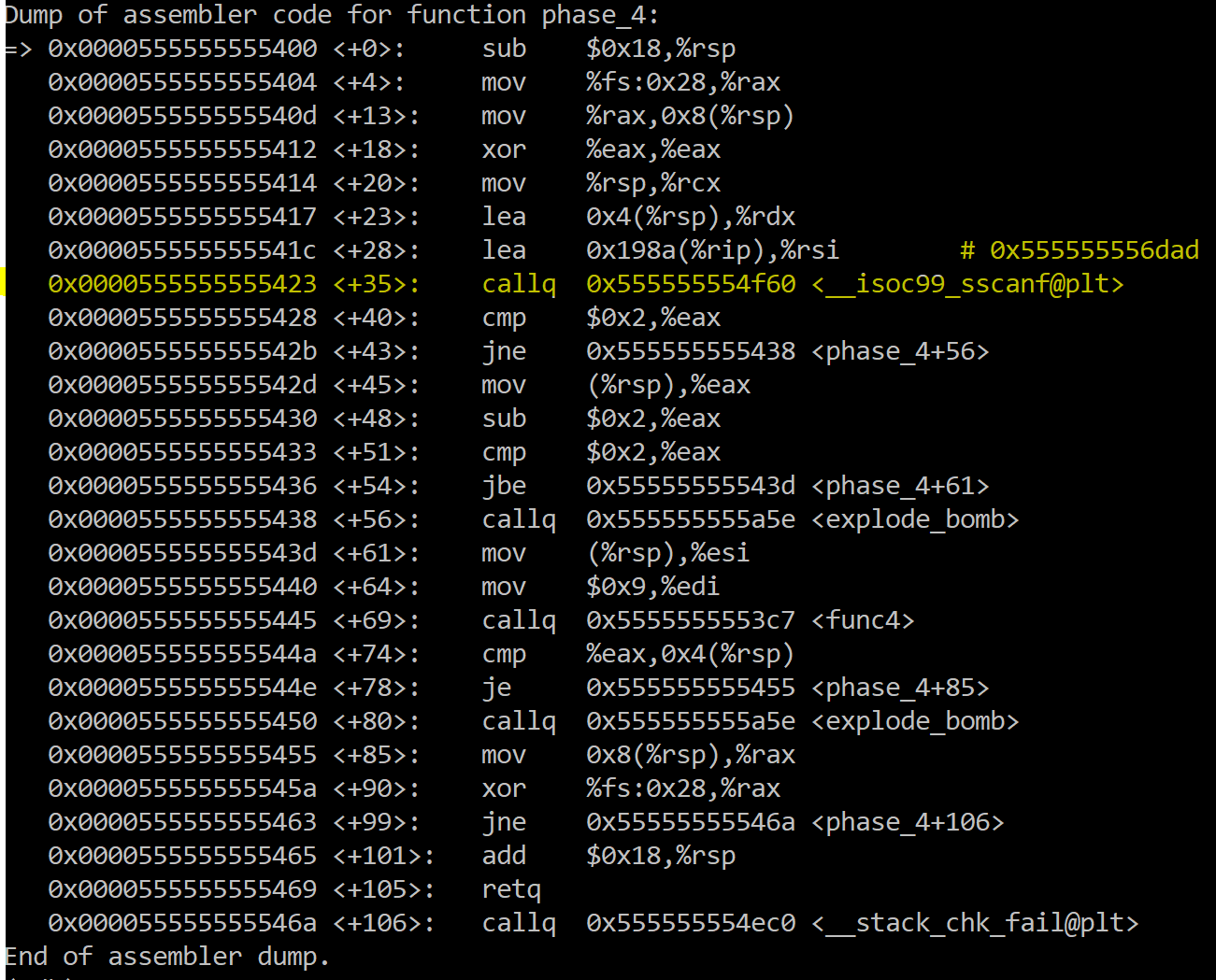
Then we know that the answer for phase 4 is two integers, so we can simply enter “1 2” into phase 3 and see how the code works.

1. Look at line <+40>, we compare %eax to 0x1, and if %eax<=0, then the code jumps to line <+70>, which calls the explode\_bomb function. This is checking that we have entered more than one numbers.
2. Then at line <+45>, we compare 0x7 with (%rsp), and if (%rsp)>7, the code will call explode\_bomb at line <+126>. This means that our first number must be less than or equal to 7.
3. Then from line <+51>, we know %eax=(%rsp)=the first number we entered=1. And at line <+68>, the code will jump to \*%eax. By looking at the code below, we can guess that the code here is a switch case.
4. Then by using gdb to trace the code, it will jump directly to line <+77>, which is the first case.
5. At line <+77>, the code is letting %eax=0x24b=587, and then jump to <+143>. Line <+143> is comparing %eax, which is 587, to 4(%rsp), which is the second number we entered. And if these two numbers are not equal, the code will call explode\_bomb. So the second number we entered must be 587. This jumps to line <+154>.
6. We can see that after line <+154>, we would not encounter any explode\_bomb. So one of the answers for phase 3 is “1 587”.

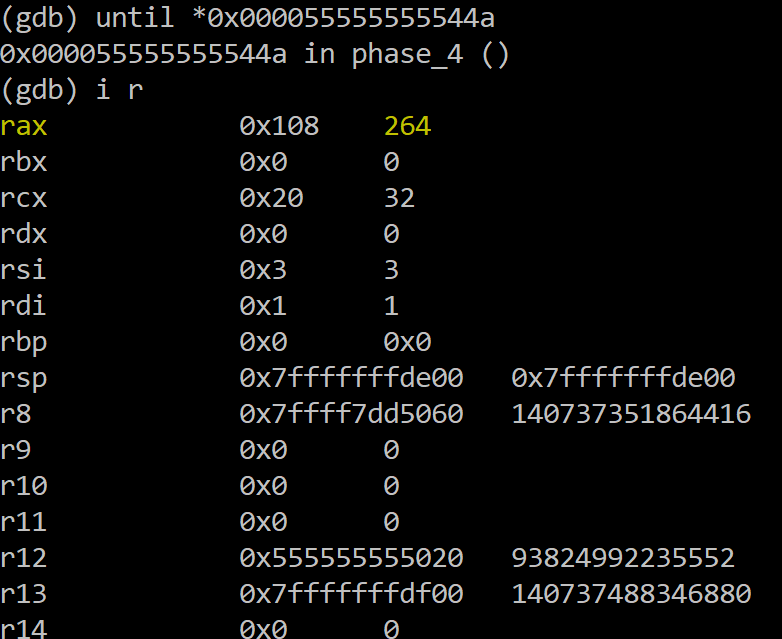


**Phase 4:**

1. First open the assembly code for phase 4:

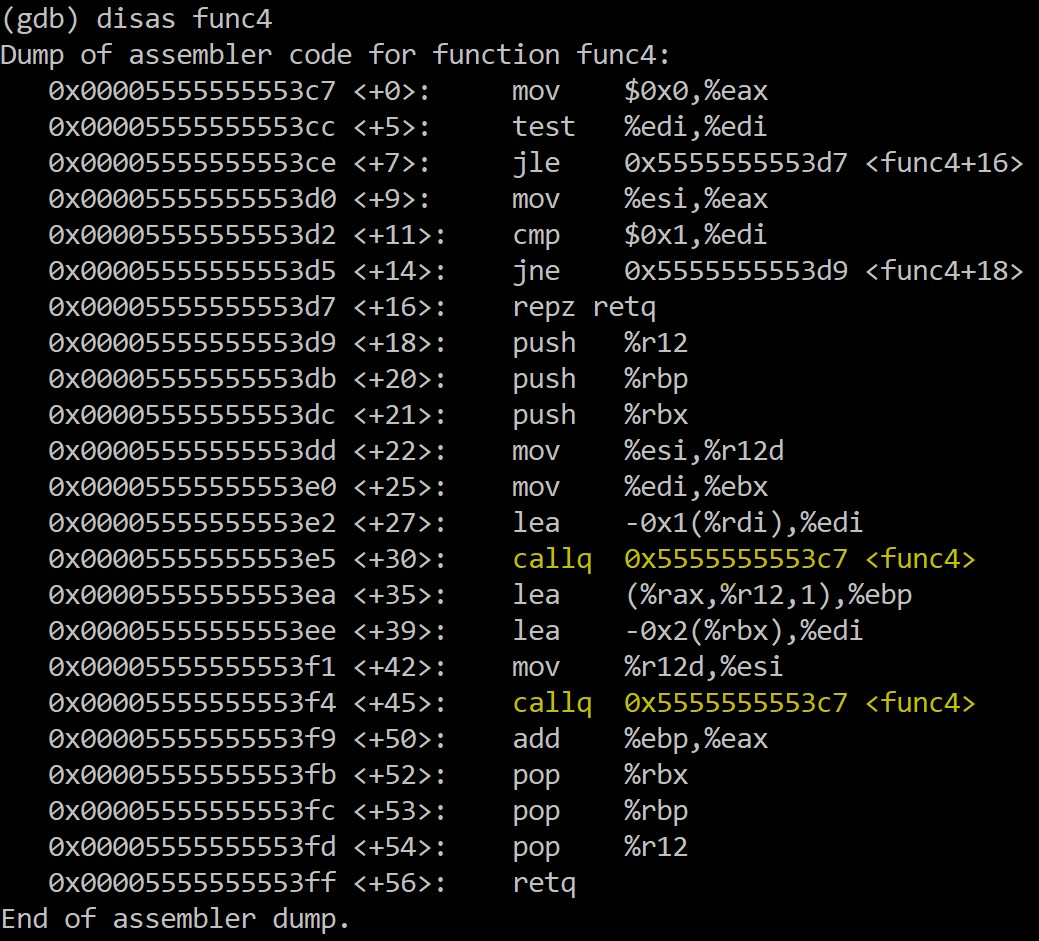


1. Again, we notice that there is a comment at line <+28>, which is the same as the one in phase 3, so we can know that the answer for phase 4 is also two integers. To make the number easy to recognize, we can try to enter ”2 3” and reproduce the assembly code.
2. Look at line <+40>, we compare %eax to 0x2, and if %eax != 0, then the code jumps to line <+56>, which calls the explode\_bomb function. This is checking that we have entered exactly two numbers.
3. Then at line <+45>, we know that %eax=(%rsp). And then %eax-=2 (line <+48>) and we compare %eax with 2. If %eax>2, the bomb will explode. This means that the second number we entered must be less than or equal to 4. (we can use the command “until \*[address of <+51>]” and “i r” to see that %eax is the second number we entered.)
4. At line <+61> and <+64>, we know that %esi=(%rsp)=the second number we entered, %edi=0x9. And we pass %edi and %esi as the first two parameters into function func4.
5. At line <+74>, we compare the return value %eax with 0x4(%rsp), which is the first number we have entered. And if these two numbers not equal, the bomb will explode (line <+80>). This means that the first number we entered must be equal to the return value of func4.
6. We need to know what %eax is at line <+74>. By gdb, we can use the command “until \*[address of <+74>]” to get to that line and use “i r” to see the value of all the registers.

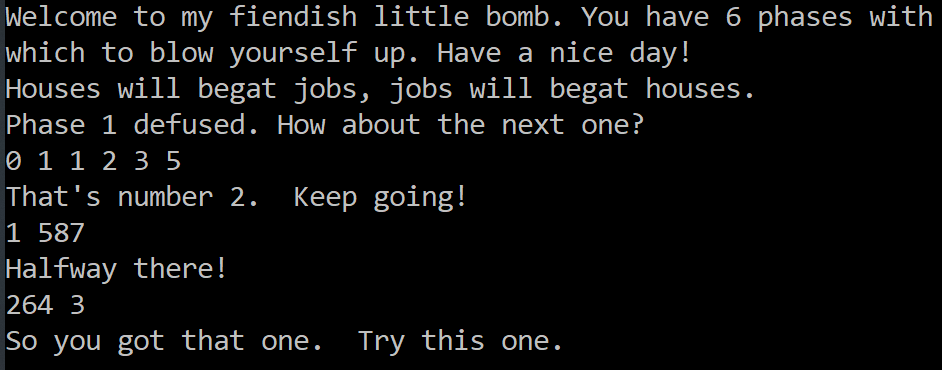


We can see that %eax now is 264, so the first number must be 264, provided that the second number is 3.

1. In fact, if we look into the code of func4, we can see that this is a recursion function.

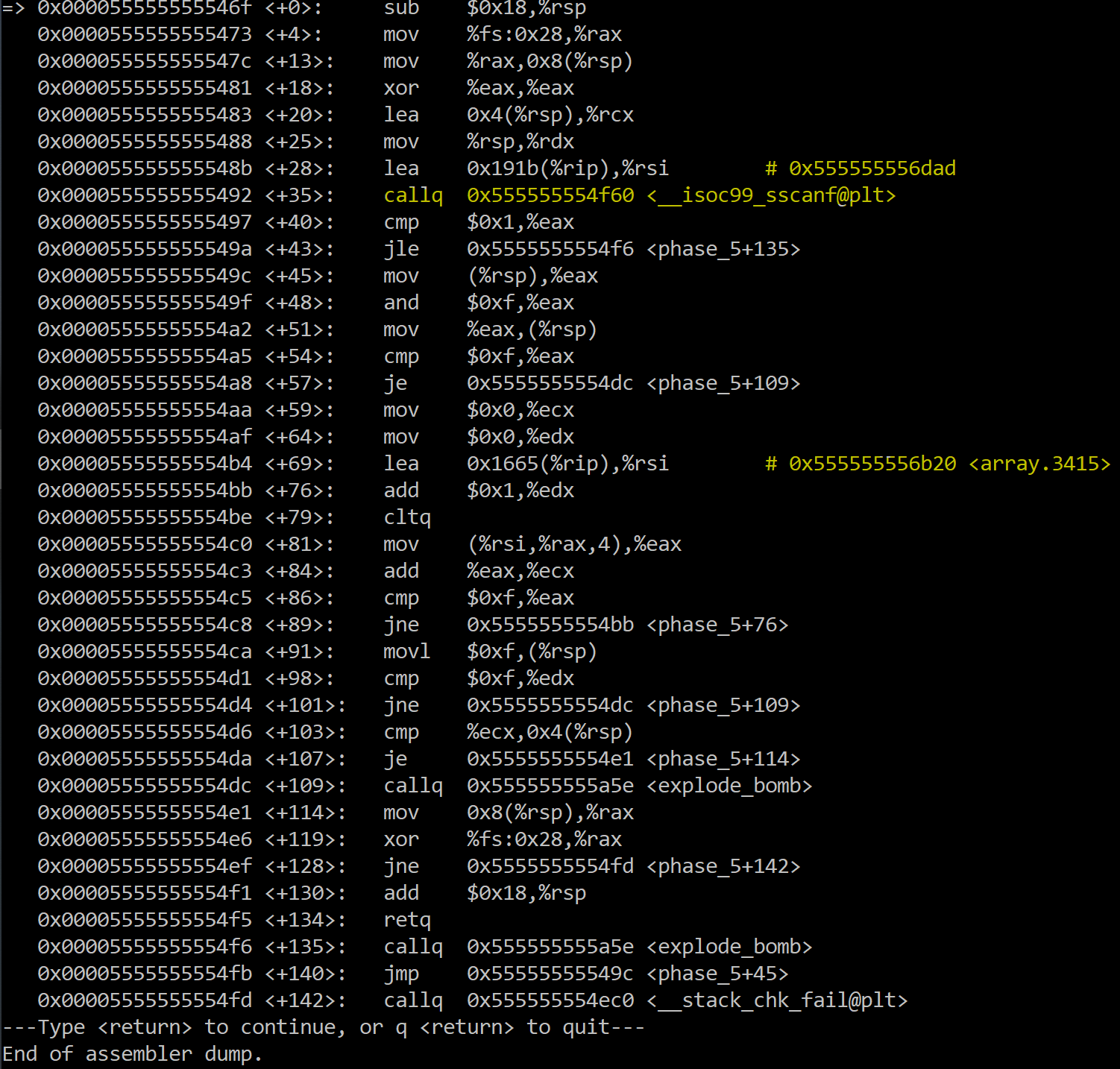


1. Therefore, one of the answers for phase 4 is “264 3”.

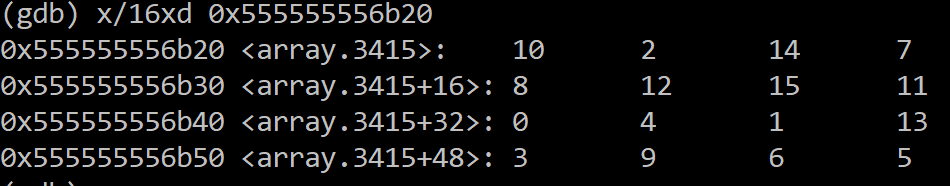


**Phase 5:**

1. First open the assembly code for phase 5:



1. Again, we notice that there is a comment at line <+28>, which is the same as the one in phase 3, so we can know that the answer for phase 4 is also two integers.
2. There is another comment at line <+69> with hints “array…”. So we can guess this is a array-related phase. Use x/16xd to see that the array is:



From here we know that the array we need to use in this phase is:

[10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 1, 3, 9, 6, 5]

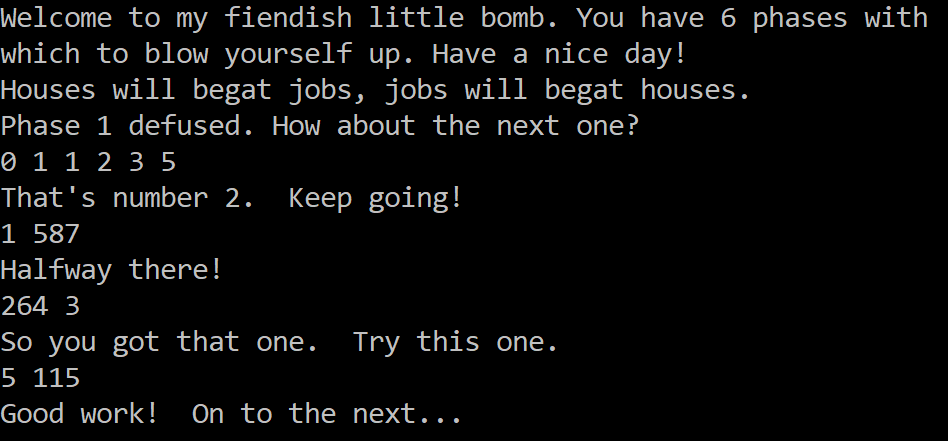
1. Then look at the assembly code. As the previous phases, line <+40> and <+43> is checking we have entered more than two numbers.
2. Line <+45> to <+57> means that if the first number & 0xf == 0xf, the bomb will explode. Thus, the first number cannot be 0xf=15.
3. At line <+86> and <+89>, we jump back to <+76> if %eax != 15. This means that we have a loop here, and the loop stops when %eax=15. And then keep looking down at line <+98>. We can see the code is comparing 0xf and %edx and they must be equal, otherwise at <+109> the explode function will be called. Then at line <+103>, 0x4(%rsp), which is the second number we entered, must be equal to %ecx, otherwise the bomb will also explode.

Therefore, from <+76> to <+84>, we can conclude:

1. Every time the loop iterates, %edx ++, %eax = array[%eax], %ecx is summing up the array elements we have visited.
2. When exiting, %eax=%edx=15.
3. The second number we entered=%ecx.

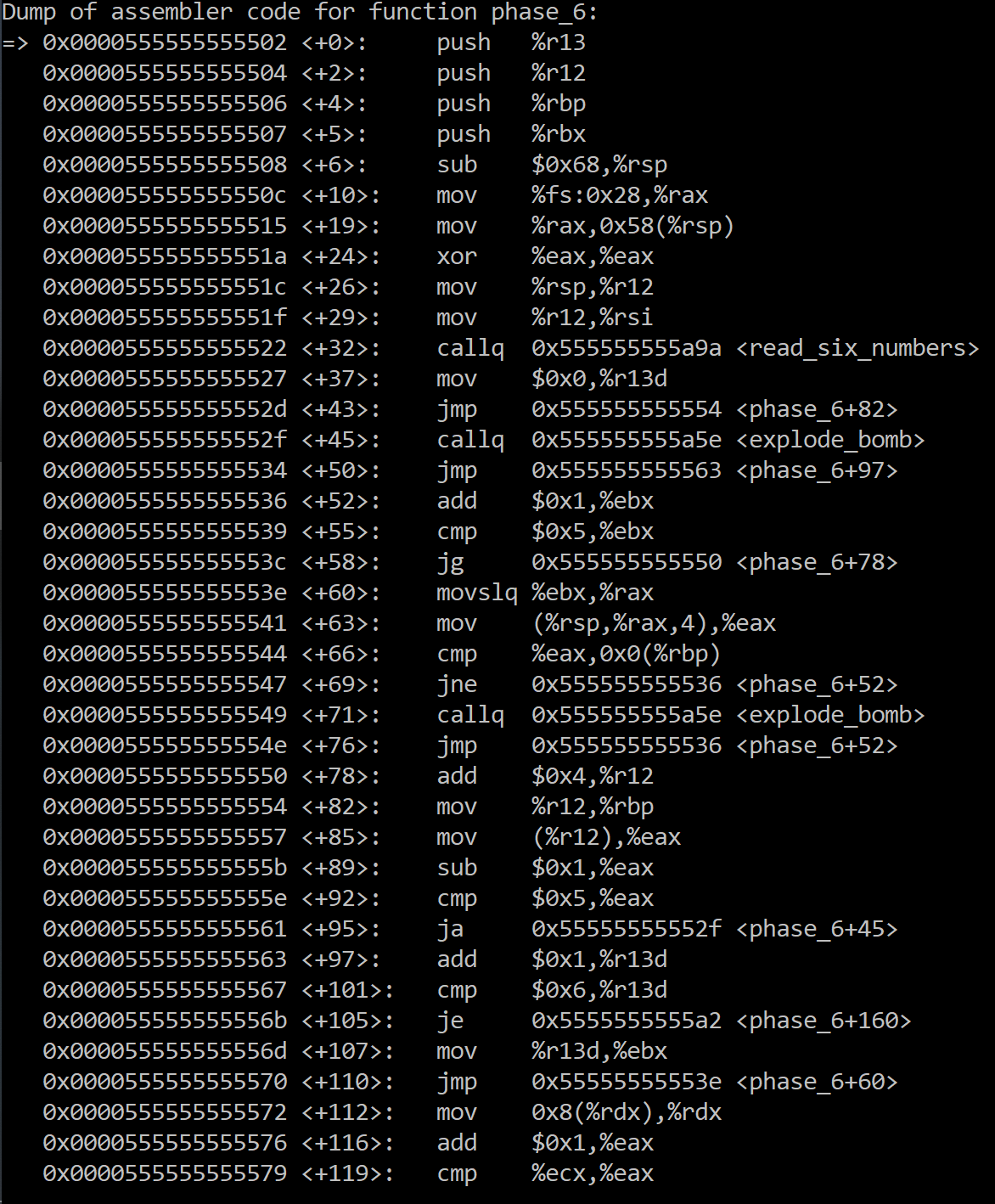
This means that the first number we entered should make the loop iterate 15 times and the 15th %eax should be 15. So we can let the first number be 5, and then the 15th %eax=15. The second number is therefore 12+3+7+11+13+9+4+8+0+10+1+2+14+6+15=115.

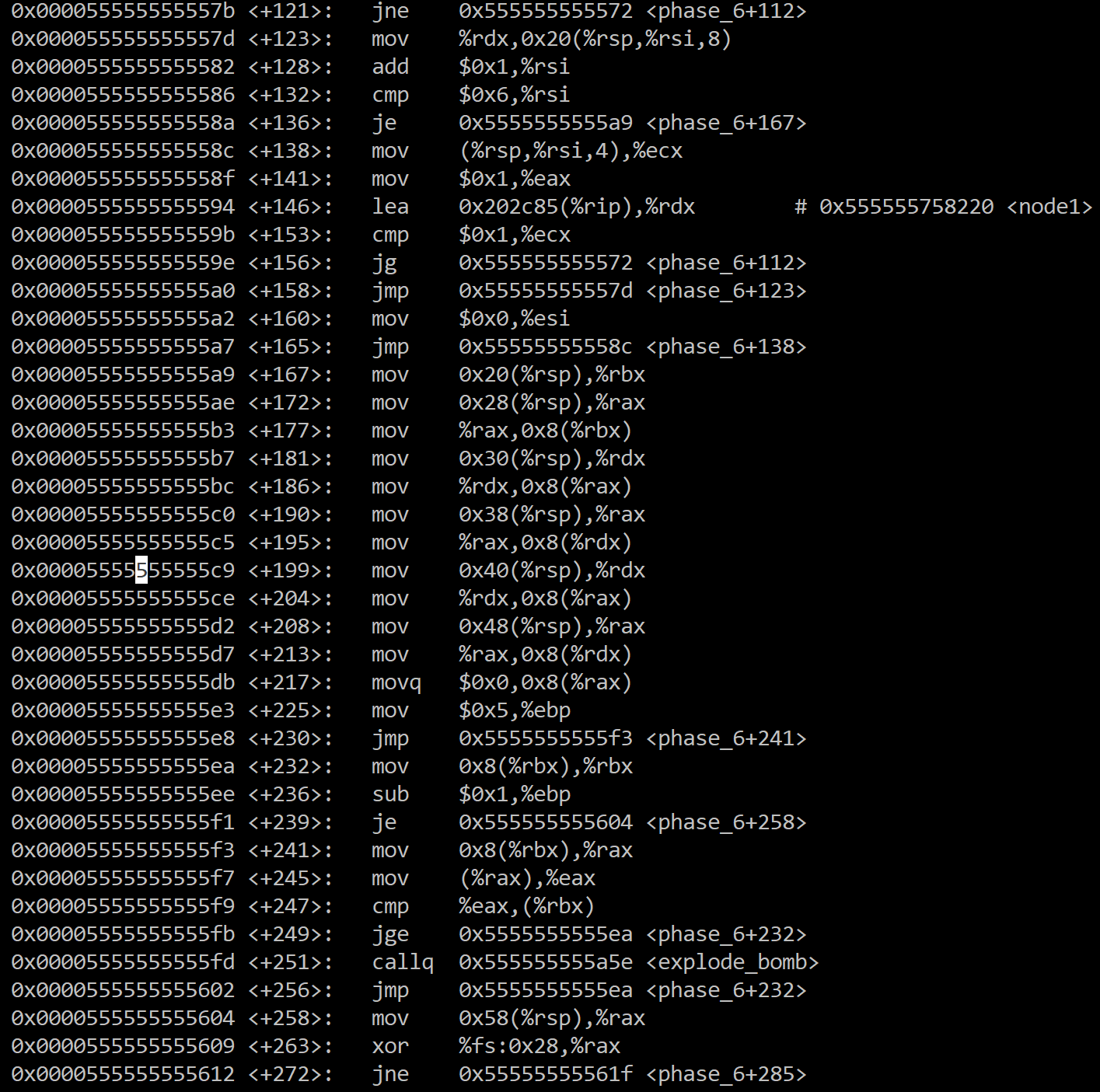
1. Therefore, the answer for phase 5 is “5 115”.

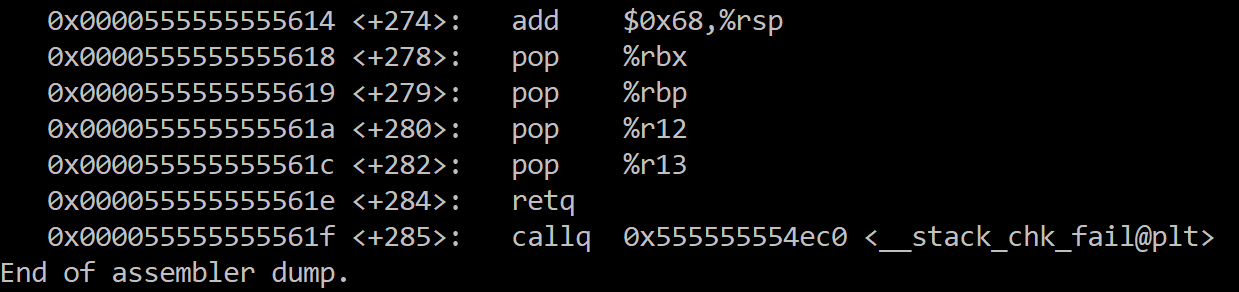


**Phase 6:**

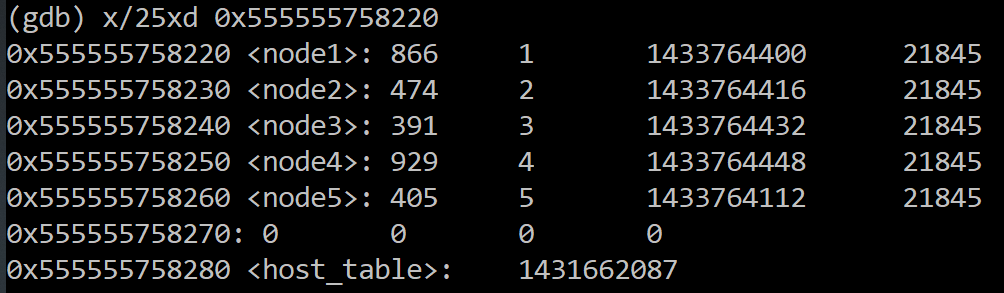
1. First open the assembly code for phase 6:



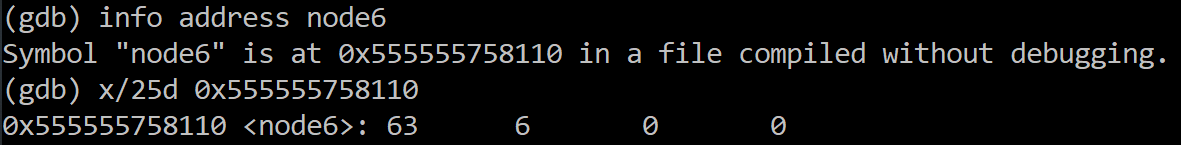




1. From the function name, we can know that we need to enter 6 numbers. Also, there is a comment with address that we can use x/25xd to see the nodes.



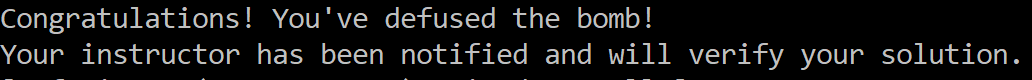
Find node6:



1. Then analyze the assembly code. After the function is called, we let %r13d=0, which is the counter of the number of integers we have entered.
2. At line <+43>, the code jumps to <+82>. And then at line <+89>, <+92> and <+95>, we know that %eax-1 must be less than 5, otherwise the bomb will explode. So %eax must be less than or equal to 6. And if %eax<=6, we will add one to %r13d. If %r13d is less than 6, we will jump to <+60>. So here is a loop letting us know that all the numbers are less than or equal to 6.
3. From line <+60> to <+71>, we can know that the value of %eax is changed to the next number we entered and the code is comparing %eax to the other numbers. If any of these two numbers are equal, the bomb will explode. Assume we enter distinct 6 numbers. And then the code continues to <+52>. From <+52> to <+58>, we can see that this is another loop that iterate 5 times. Therefore, we can conclude that all these 6 numbers are distinct to each other. So we know that the number we entered must contain 123456 in a specific order.
4. Assume that these 6 numbers are distinct and greater than/equal to 6. Then we jump to <+160> (at line <+105>). And then jump to <+138>.
5. From <+138> to <+158>, we can see that the code is comparing the first number with 1. If the first number>1, jump to <+112>, otherwise jump to <+123>. Wherever it goes, the code will always come back to <+123> unless the counter %rsi==6. This means that the loop is going to iterate 6 times. And from <+112> to <+153>, we can see that this loop builds a list of pointers to elements.
6. Then if %rsi==6, we go to <+167>. From <+167> to <+230> we can know that this is a linked list with 6 elements. The head node is 0x555555758220 and the last pointer is NULL.
7. Then we jump to <+241>. From <+232> to <+249> the code is checking the values in the nodes we entered are in reverse order. Actually by looking at the values of these 6 nodes, we can know that 929>866>474>405>391>63, which means the order of our six nodes is:

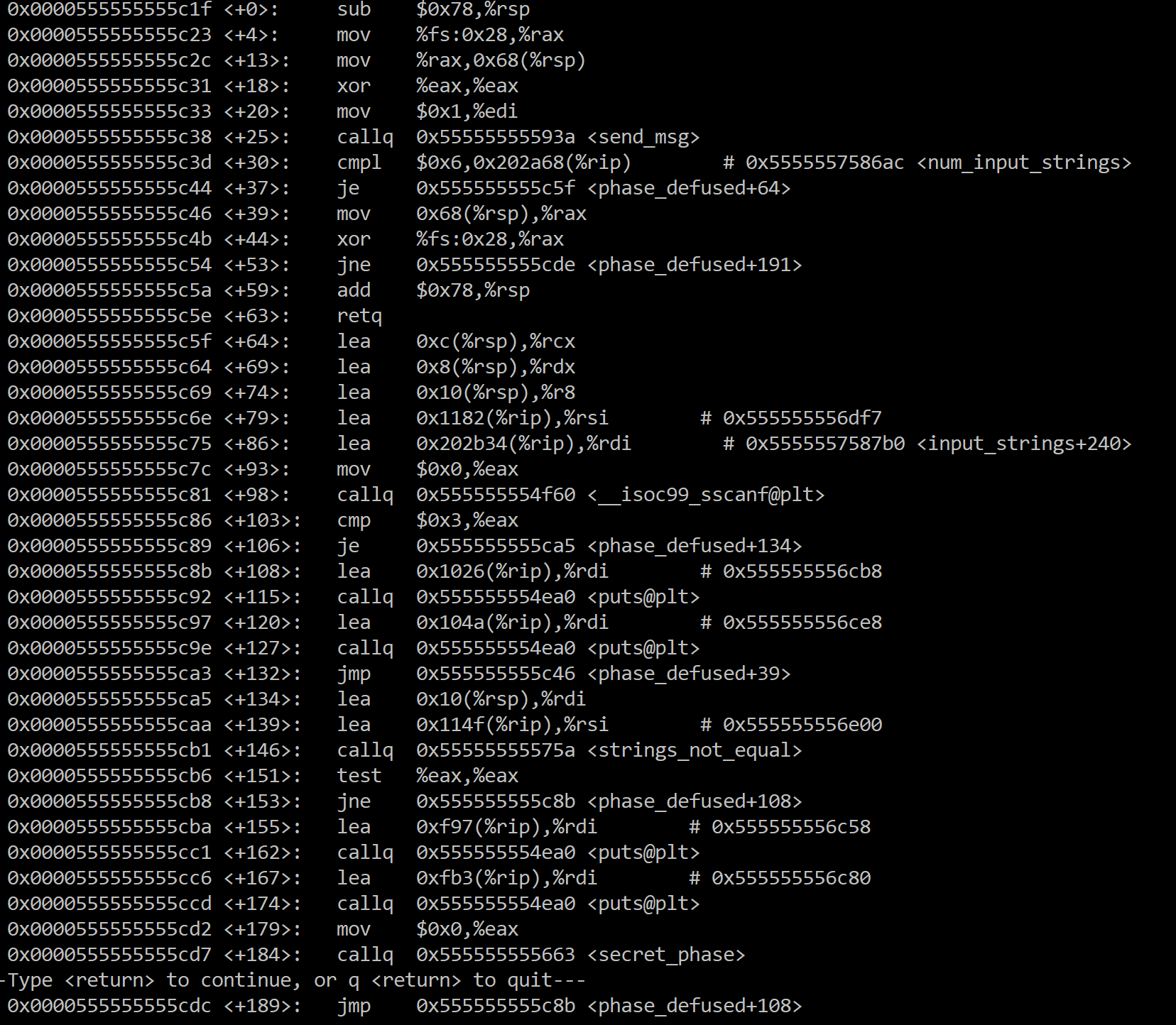
4 1 2 5 3 6

1. Therefore, the answer for phase 6 is “4 1 2 5 3 6”.

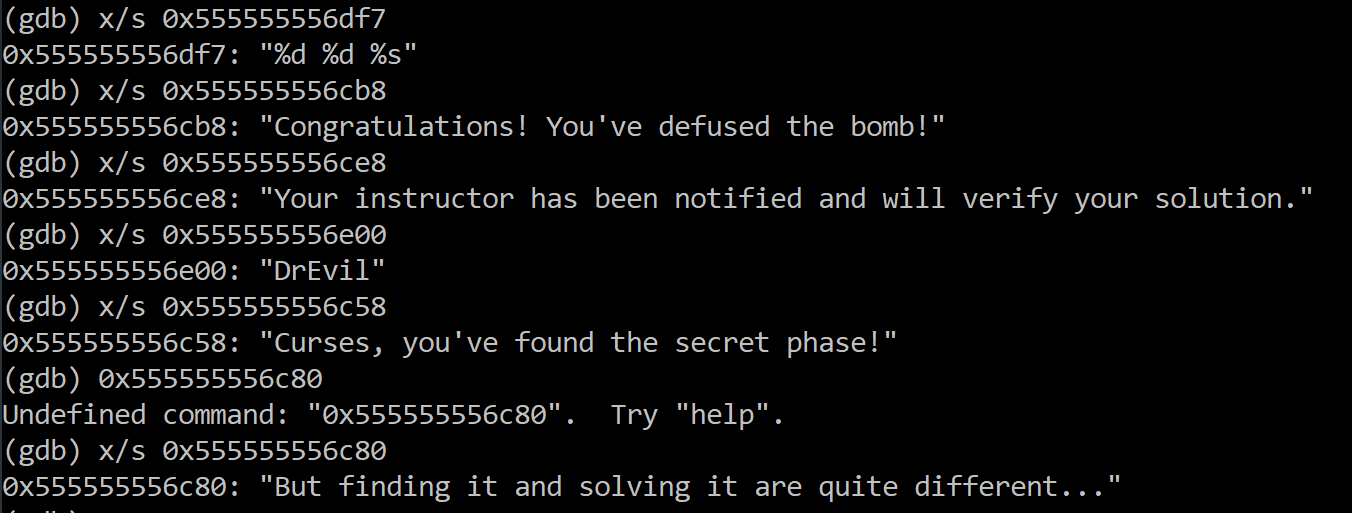


**Secret Phase:**

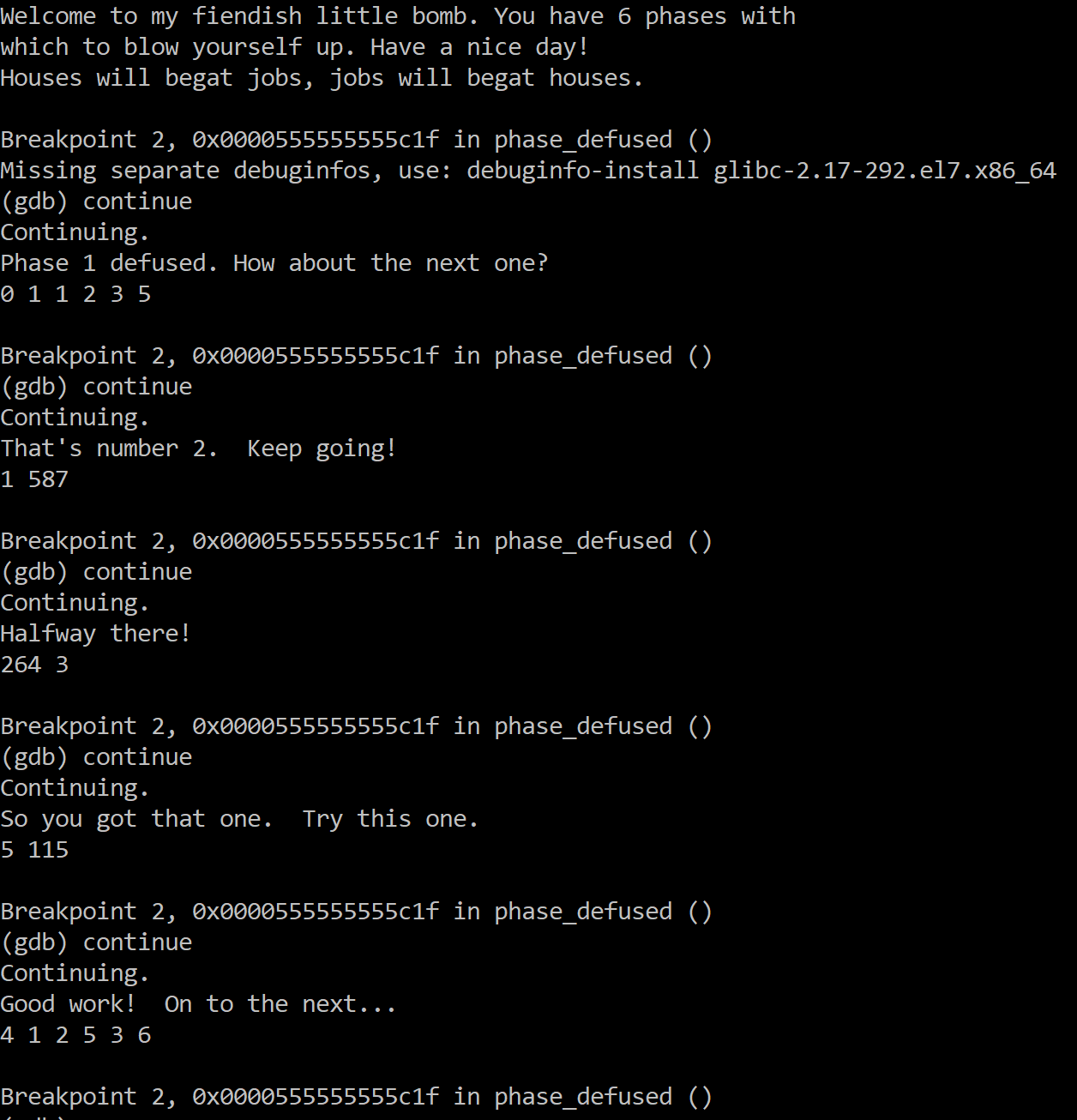
1. First look at the code of the function phase\_defused:



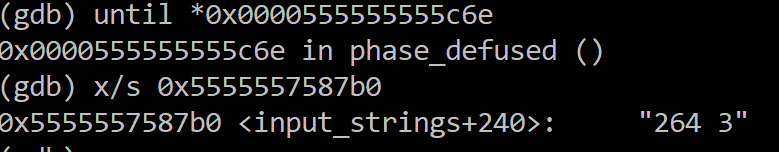
1. There are many comments with addresses in the right. Let us use “x/s” to see what they are.



1. From <+30> we can see that if the strings we entered is equal to 6, the code will jump to <+64>. We can set a breakpoint at phase\_defused and secret\_phase. Then run the program until we go into phase\_defused for the last time.

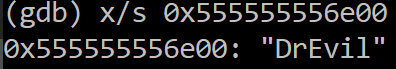


1. At this time, when we use “until \*[address of <+79>]” and “x/s [address of input strings+240]”, we can get:



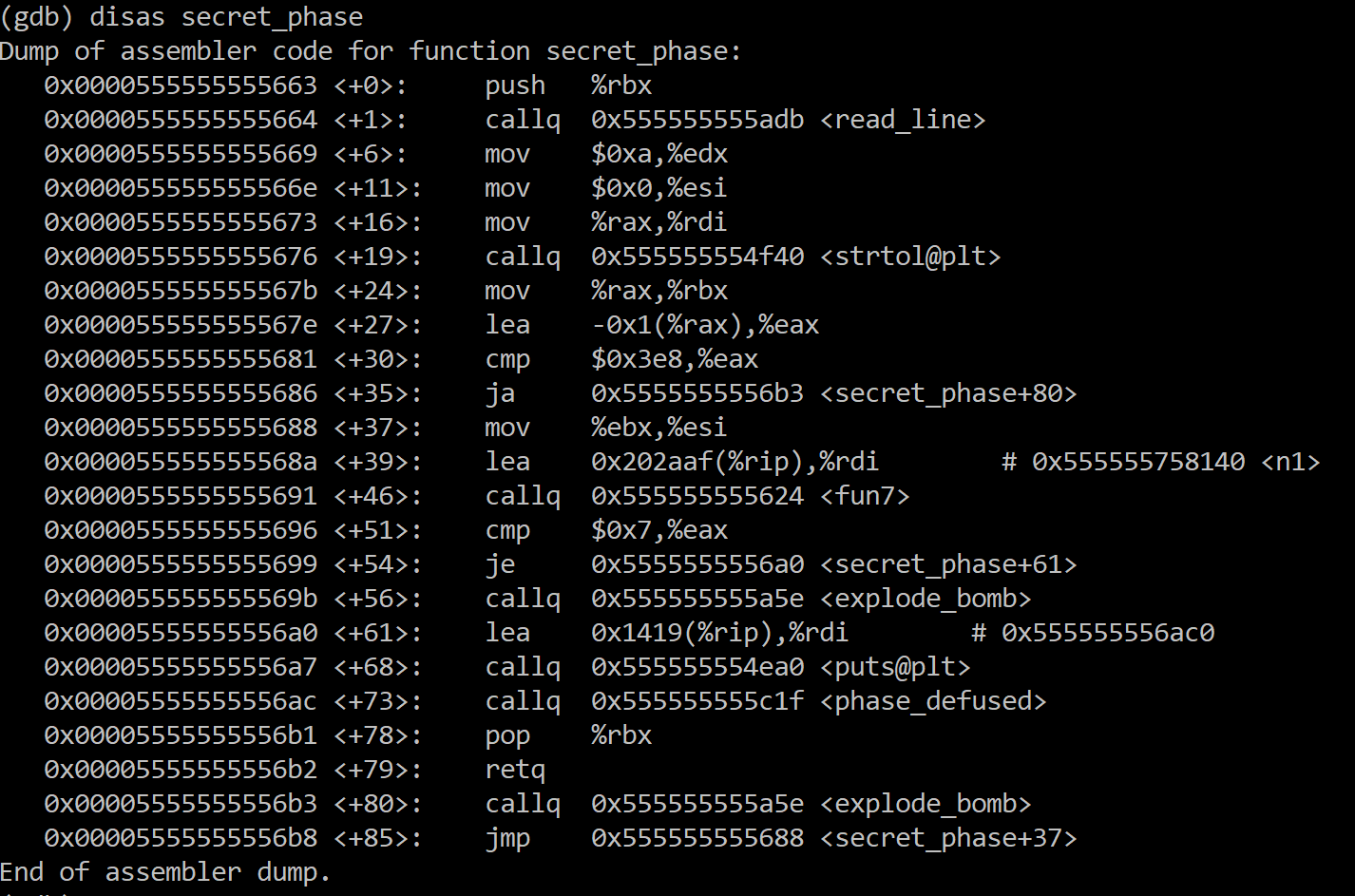
That is the answer of phase 4! Also, because we get from previous steps that the answer should be in format “%d,%d,%s”, so we are sure that there must be another string behind “264 3” so that we can enter the secret phase. This can also be assured by line <+103> and <+106>. These code is confirming we have 3 “things” in phase\_4. If so, it will jump to <+134>, and the code after <+134> will print “You find the secret phase!”,etc.

1. To find what should be add to phase 4, we should notice that:



And this is what we want.

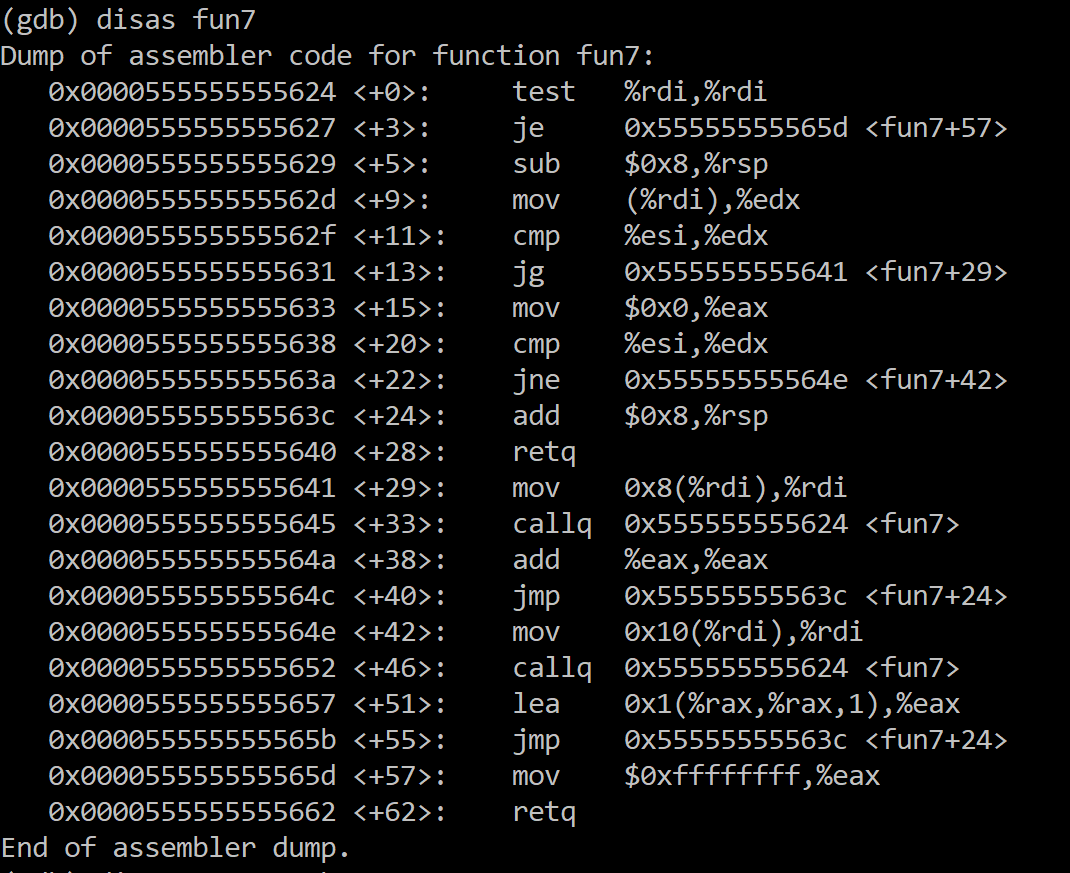
1. Then we add “DrEvil” after “264 3” for phase 4 and run the program again. Then the code indicates that we have found the secret phase.
2. Then we need to solve the secret phase.



From line <+27>, we can know that the number we entered minus one should be less than or equal to 1000, otherwise the bob will explode.

From <+51> and <54>, we want fun7() to return 7.

Code for fun7:



|  |  |
| --- | --- |
|  | 7 = 2\*3+1 = 2\*(2\*1+1)+1. |
|  | f(0x24) = 0 |
|  | f(0x32) = 2\*f(0x24)+1 = 1 |
|  | f(0x6b) = 2\*f(0x32)+1 = 3 |
|  | f(0x3e9) = 2\*f(0x6b)+1 = 7 |
|  |  |
|  | 0x3e9 is 1001 decimal, and is accepted by the first check (1001-1<=1000). |

1. Therefore, the answer for secret phase is 1001.

