CS201
REPORT: BOMB LAB

1551020 - Vo Tran Thanh Luong

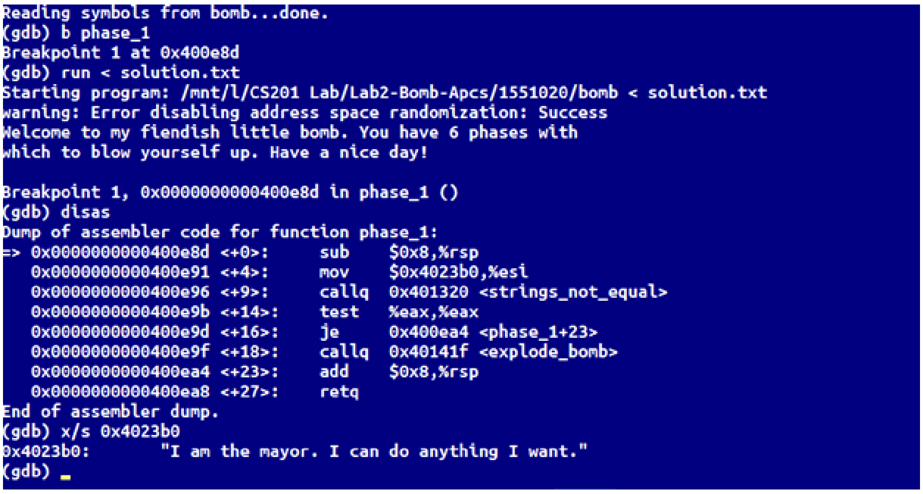
November 22, 2016

Contents

1 Phase 1

Phase 1

Phase 1 is very easy. We can follow Mr. Thang in class detailed instructions or simply just x/s the address on the second line. Because the command means moving something to esi (to serve something in <strings_not_equal>that we havent know yet) , we investigate it and the result surprisingly shows up.

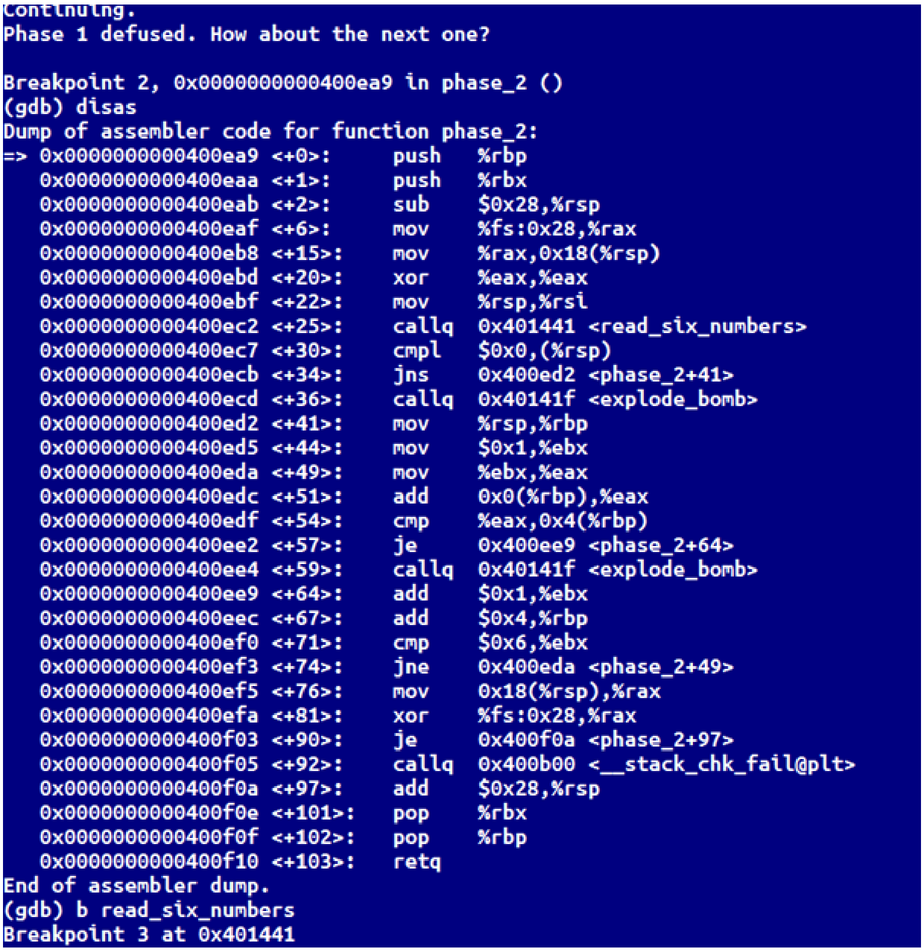


```
Reading symbols from bomb...done.
(gdb) b phase_1
Breakpoint 1 at 0x400e8d
(gdb) run < solution.txt
Starting program: /mnt/l/CS201 Lab/Lab2-Bomb-Apcs/1551020/bomb < solution.txt
warning: Error disabling address space randomization: Success
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!

Breakpoint 1, 0x0000000000400e8d in phase_1 ()
(gdb) disas
Dump of assembler code for function phase_1:
=> 0x0000000000400e8d <+0>:     sub    $0x8,%rsp
   0x0000000000400e91 <+4>:     mov    $0x4023b0,%esi
   0x0000000000400e96 <+9>:     callq  0x401320 <strings_not_equal>
   0x0000000000400e9b <+14>:    test   %eax,%eax
   0x0000000000400e9d <+16>:    je     0x400ea4 <phase_1+23>
   0x0000000000400e9f <+18>:    callq  0x40141f <explode_bomb>
   0x0000000000400ea4 <+23>:    add    $0x8,%rsp
   0x0000000000400ea8 <+27>:    retq
End of assembler dump.
(gdb) x/s 0x4023b0
0x4023b0:       "I am the mayor. I can do anything I want."
(gdb)
```

Figure 1: Phase1

2 Phase2

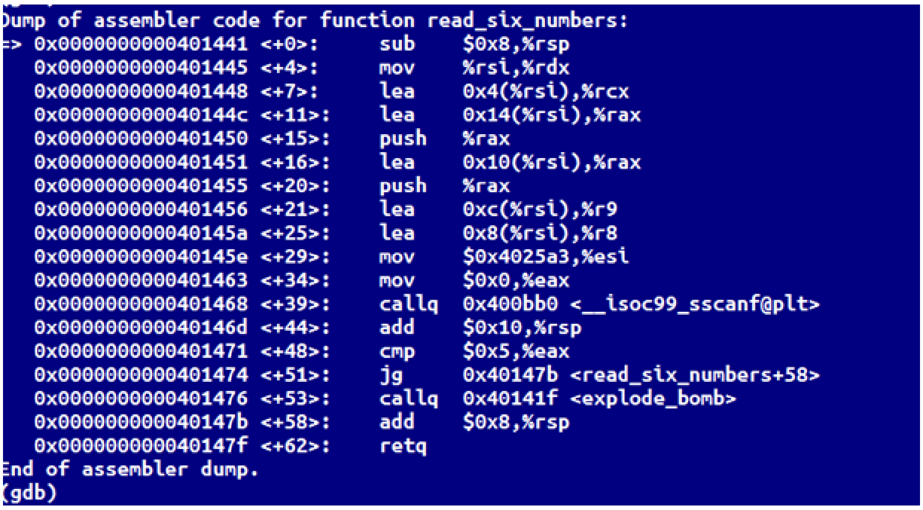
Set the breakpoint and jump into <strings_not_equal>.



```
Continuing.
Phase 1 defused. How about the next one?

Breakpoint 2, 0x0000000000400ea9 in phase_2 ()
(gdb) disas
Dump of assembler code for function phase_2:
=> 0x0000000000400ea9 <+0>:     push   %rbp
   0x0000000000400eaa <+1>:     push   %rbx
   0x0000000000400eab <+2>:     sub    $0x28,%rsp
   0x0000000000400eaf <+6>:     mov    %fs:0x28,%rax
   0x0000000000400eb8 <+15>:    mov    %rax,0x18(%rsp)
   0x0000000000400ebd <+20>:    xor    %eax,%eax
   0x0000000000400ebf <+22>:    mov    %rsp,%rsi
   0x0000000000400ec2 <+25>:    callq  0x401441 <read_six_numbers>
   0x0000000000400ec7 <+30>:    cmpl   $0x0,(%rsp)
   0x0000000000400ecb <+34>:    jns    0x400ed2 <phase_2+41>
   0x0000000000400ecd <+36>:    callq  0x40141f <explode_bomb>
   0x0000000000400ed2 <+41>:    mov    %rsp,%rbp
   0x0000000000400ed5 <+44>:    mov    $0x1,%ebx
   0x0000000000400eda <+49>:    mov    %ebx,%eax
   0x0000000000400edc <+51>:    add    0x0(%rbp),%eax
   0x0000000000400edf <+54>:    cmp    %eax,0x4(%rbp)
   0x0000000000400ee2 <+57>:    je     0x400ee9 <phase_2+64>
   0x0000000000400ee4 <+59>:    callq  0x40141f <explode_bomb>
   0x0000000000400ee9 <+64>:    add    $0x1,%ebx
   0x0000000000400eec <+67>:    add    $0x4,%rbp
   0x0000000000400ef0 <+71>:    cmp    $0x6,%ebx
   0x0000000000400ef3 <+74>:    jne    0x400eda <phase_2+49>
   0x0000000000400ef5 <+76>:    mov    0x18(%rsp),%rax
   0x0000000000400efa <+81>:    xor    %fs:0x28,%rax
   0x0000000000400f03 <+90>:    je     0x400f0a <phase_2+97>
   0x0000000000400f05 <+92>:    callq  0x400b00 <__stack_chk_fail@plt>
   0x0000000000400f0a <+97>:    add    $0x28,%rsp
   0x0000000000400f0e <+101>:   pop    %rbx
   0x0000000000400f0f <+102>:   pop    %rbp
   0x0000000000400f10 <+103>:   retq
End of assembler dump.
(gdb) b read_six_numbers
Breakpoint 3 at 0x401441
```

Figure 2: Phase 2

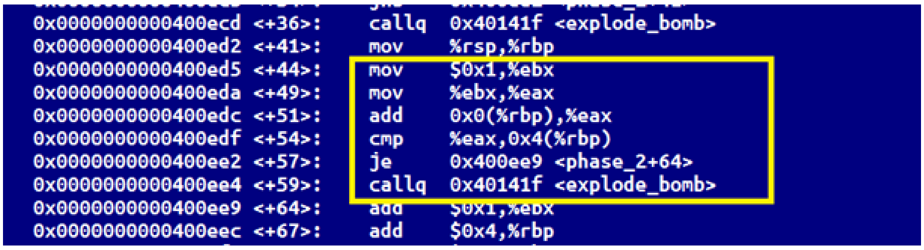
Here's the assembly version of read six numbers.



```
Dump of assembler code for function read_six_numbers:
=> 0x0000000000401441 <+0>:     sub    $0x8,%rsp
   0x0000000000401445 <+4>:     mov    %rsi,%rdx
   0x0000000000401448 <+7>:     lea    0x4(%rsi),%rcx
   0x000000000040144c <+11>:    lea    0x14(%rsi),%rax
   0x0000000000401450 <+15>:    push   %rax
   0x0000000000401451 <+16>:    lea    0x10(%rsi),%rax
   0x0000000000401455 <+20>:    push   %rax
   0x0000000000401456 <+21>:    lea    0xc(%rsi),%r9
   0x000000000040145a <+25>:    lea    0x8(%rsi),%r8
   0x000000000040145e <+29>:    mov    $0x4025a3,%esi
   0x0000000000401463 <+34>:    mov    $0x0,%eax
   0x0000000000401468 <+39>:    callq  0x400bb0 <__isoc99_sscanf@plt>
   0x000000000040146d <+44>:    add    $0x10,%rsp
   0x0000000000401471 <+48>:    cmp    $0x5,%eax
   0x0000000000401474 <+51>:    jg     0x40147b <read_six_numbers+58>
   0x0000000000401476 <+53>:    callq  0x40141f <explode_bomb>
   0x000000000040147b <+58>:    add    $0x8,%rsp
   0x000000000040147f <+62>:    retq
End of assembler dump.
(gdb)
```

Figure 3: Phase 2.2

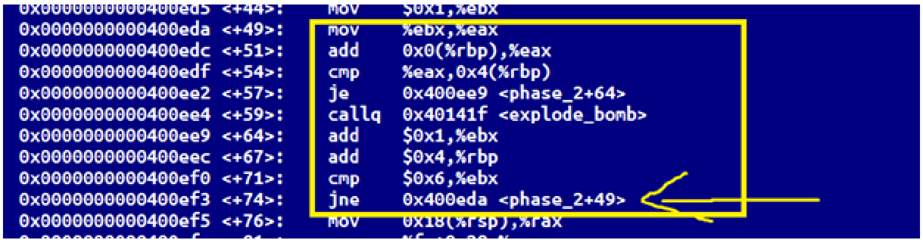
Nothing special, it just reads in 6 numbers. We continue looking at these steps :



```
   0x0000000000400ecd <+36>:    callq  0x40141f <explode_bomb>
   0x0000000000400ed2 <+41>:    mov    %rsp,%rbp
   0x0000000000400ed5 <+44>:    mov    $0x1,%ebx
   0x0000000000400eda <+49>:    mov    %ebx,%eax
   0x0000000000400edc <+51>:    add    0x0(%rbp),%eax
   0x0000000000400edf <+54>:    cmp    %eax,0x4(%rbp)
   0x0000000000400ee2 <+57>:    je     0x400ee9 <phase_2+64>
   0x0000000000400ee4 <+59>:    callq  0x40141f <explode_bomb>
   0x0000000000400ee9 <+64>:    add    $0x1,%ebx
   0x0000000000400eec <+67>:    add    $0x4,%rbp
```

Figure 4: Phase 2.3

These codes tell us one important information. It means that when we add 1 with the first number we enter, if the result does not equal to the second number, we will die. So we have to make sure the second number = the first number + 1. Let's call the 1 here a "check" value. If you continue looking at the code, everything from 49 to 74 is a loop. Moreover, this loop goes through the 6 numbers we enter. Keep an eye on the change of ebx and rbp, we can conclude that after everyturn, the "check" value got raised by 1, the current number will move to the next number. Thus, the pattern for our inputted 6 numbers is : <(N+1) position >number = <(N) position >number + check (check runs from 1-5). At last, the result is 1 2 4 7 11 16

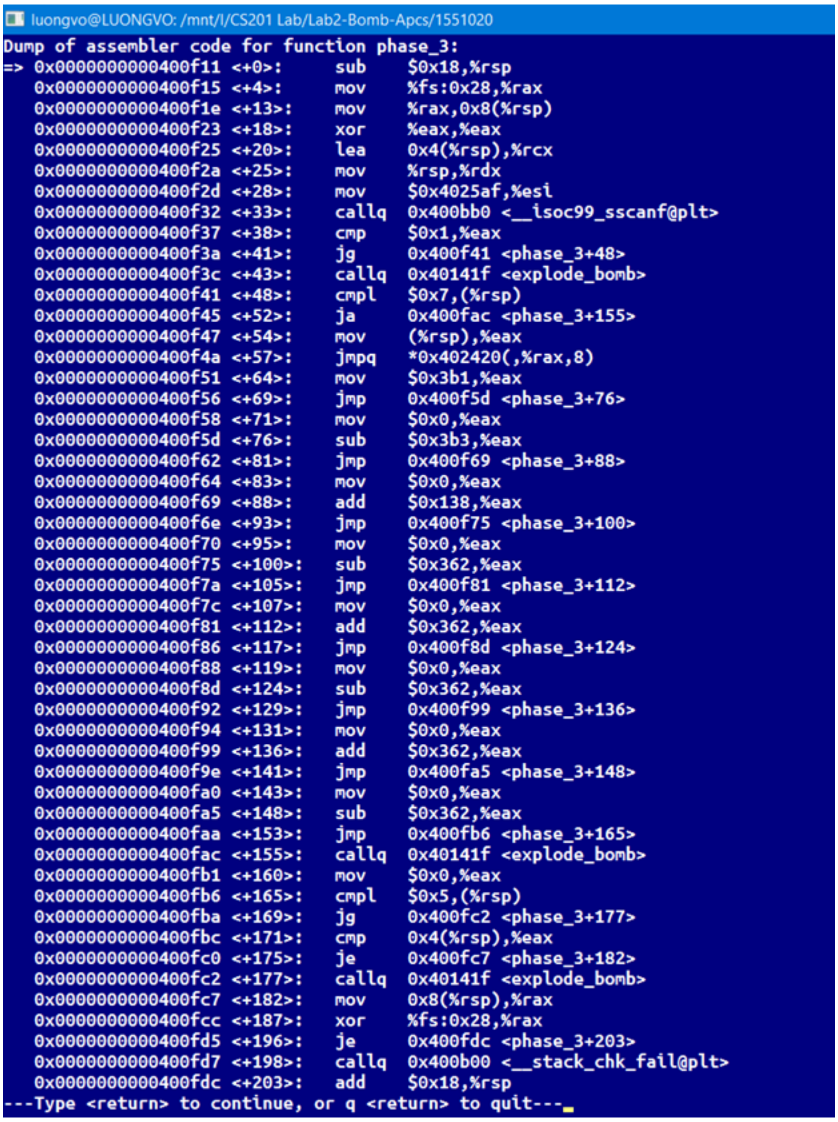


```
0x0000000000400ed5 <+44>:    mov    $0x1,%ebx
0x0000000000400eda <+49>:    mov    %ebx,%eax
0x0000000000400edc <+51>:    add    0x0(%rbp),%eax
0x0000000000400edf <+54>:    cmp    %eax,0x4(%rbp)
0x0000000000400ee2 <+57>:    je     0x400ee9 <phase_2+64>
0x0000000000400ee4 <+59>:    callq  0x40141f <explode_bomb>
0x0000000000400ee9 <+64>:    add    $0x1,%ebx
0x0000000000400eec <+67>:    add    $0x4,%rbp
0x0000000000400ef0 <+71>:    cmp    $0x6,%ebx
0x0000000000400ef3 <+74>:    jne    0x400eda <phase_2+49>
0x0000000000400ef5 <+76>:    mov    0x18(%rsp),%rax
```

Figure 5: Phase 2.4

3 Phase3

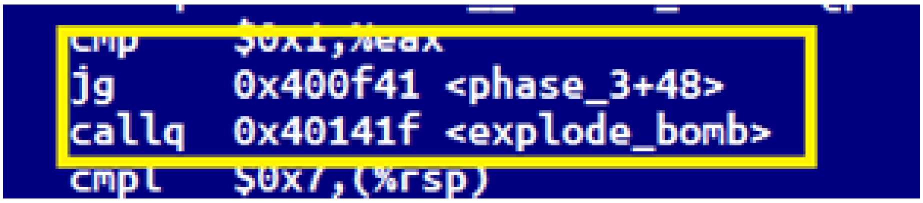
Here's the assembly code of phase_3 :



```
Dump of assembler code for function phase_3:
=> 0x0000000000400f11 <+0>:     sub    $0x18,%rsp
   0x0000000000400f15 <+4>:     mov    %fs:0x28,%rax
   0x0000000000400f1e <+13>:    mov    %rax,0x8(%rsp)
   0x0000000000400f23 <+18>:    xor    %eax,%eax
   0x0000000000400f25 <+20>:    lea    0x4(%rsp),%rcx
   0x0000000000400f2a <+25>:    mov    %rsp,%rdx
   0x0000000000400f2d <+28>:    mov    $0x4025af,%esi
   0x0000000000400f32 <+33>:    callq  0x400bb0 <__isoc99_sscanf@plt>
   0x0000000000400f37 <+38>:    cmp    $0x1,%eax
   0x0000000000400f3a <+41>:    jg     0x400f41 <phase_3+48>
   0x0000000000400f3c <+43>:    callq  0x40141f <explode_bomb>
   0x0000000000400f41 <+48>:    cmpl   $0x7,(%rsp)
   0x0000000000400f45 <+52>:    ja     0x400fac <phase_3+155>
   0x0000000000400f47 <+54>:    mov    (%rsp),%eax
   0x0000000000400f4a <+57>:    jmpq   *0x402420(,%rax,8)
   0x0000000000400f51 <+64>:    mov    $0x3b1,%eax
   0x0000000000400f56 <+69>:    jmp    0x400f5d <phase_3+76>
   0x0000000000400f58 <+71>:    mov    $0x0,%eax
   0x0000000000400f5d <+76>:    sub    $0x3b3,%eax
   0x0000000000400f62 <+81>:    jmp    0x400f69 <phase_3+88>
   0x0000000000400f64 <+83>:    mov    $0x0,%eax
   0x0000000000400f69 <+88>:    add    $0x138,%eax
   0x0000000000400f6e <+93>:    jmp    0x400f75 <phase_3+100>
   0x0000000000400f70 <+95>:    mov    $0x0,%eax
   0x0000000000400f75 <+100>:   sub    $0x362,%eax
   0x0000000000400f7a <+105>:   jmp    0x400f81 <phase_3+112>
   0x0000000000400f7c <+107>:   mov    $0x0,%eax
   0x0000000000400f81 <+112>:   add    $0x362,%eax
   0x0000000000400f86 <+117>:   jmp    0x400f8d <phase_3+124>
   0x0000000000400f88 <+119>:   mov    $0x0,%eax
   0x0000000000400f8d <+124>:   sub    $0x362,%eax
   0x0000000000400f92 <+129>:   jmp    0x400f99 <phase_3+136>
   0x0000000000400f94 <+131>:   mov    $0x0,%eax
   0x0000000000400f99 <+136>:   add    $0x362,%eax
   0x0000000000400f9e <+141>:   jmp    0x400fa5 <phase_3+148>
   0x0000000000400fa0 <+143>:   mov    $0x0,%eax
   0x0000000000400fa5 <+148>:   sub    $0x362,%eax
   0x0000000000400faa <+153>:   jmp    0x400fb6 <phase_3+165>
   0x0000000000400fac <+155>:   callq  0x40141f <explode_bomb>
   0x0000000000400fb1 <+160>:   mov    $0x0,%eax
   0x0000000000400fb6 <+165>:   cmpl   $0x5,(%rsp)
   0x0000000000400fba <+169>:   jg     0x400fc2 <phase_3+177>
   0x0000000000400fbc <+171>:   cmp    0x4(%rsp),%eax
   0x0000000000400fc0 <+175>:   je     0x400fc7 <phase_3+182>
   0x0000000000400fc2 <+177>:   callq  0x40141f <explode_bomb>
   0x0000000000400fc7 <+182>:   mov    0x8(%rsp),%rax
   0x0000000000400fcc <+187>:   xor    %fs:0x28,%rax
   0x0000000000400fd5 <+196>:   je     0x400fdc <phase_3+203>
   0x0000000000400fd7 <+198>:   callq  0x400b00 <__stack_chk_fail@plt>
   0x0000000000400fdc <+203>:   add    $0x18,%rsp
---Type <return> to continue, or q <return> to quit---_
```

Figure 6: Phase 3.1

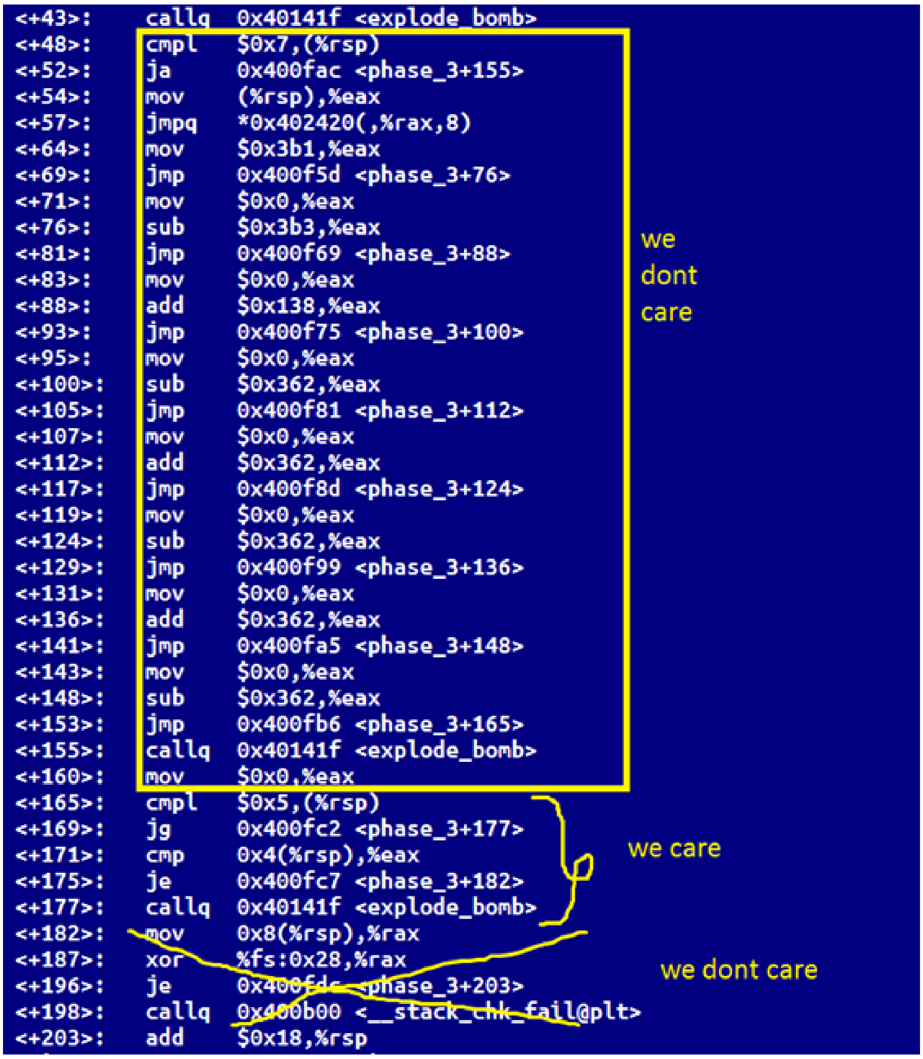
This lines tells that we must have more than 1 input :



```
cmp     $0x1,%eax
jg      0x400f41 <phase_3+48>
callq   0x40141f <explode_bomb>
cmpl    $0x7,(%rsp)
```

Figure 7: Phase 3.2

Now let's split the remaining lines into 3 parts, in which only 1 part we do care about.



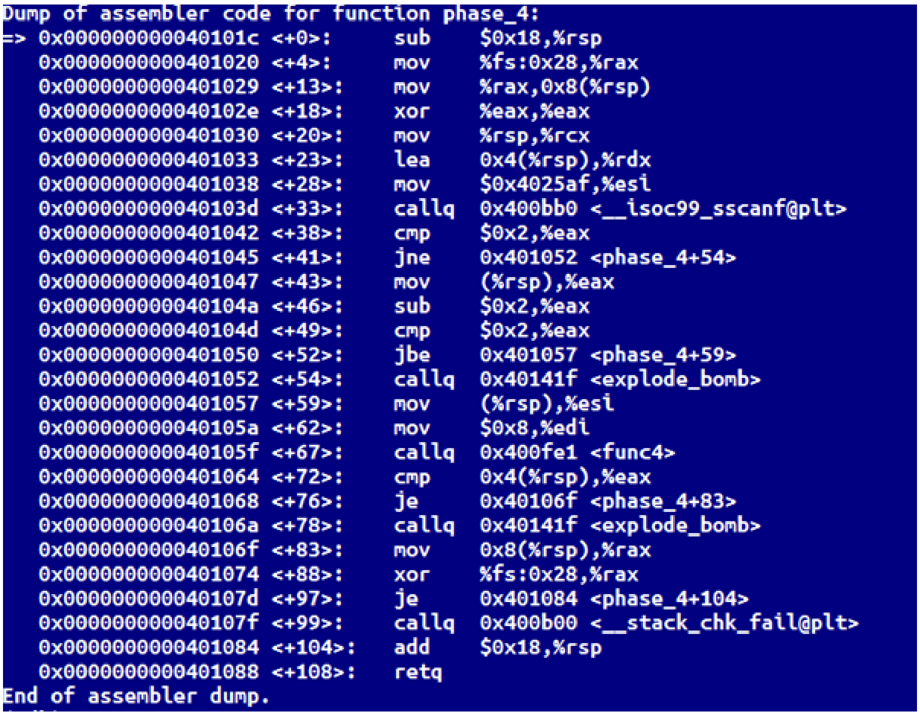
```
<+43>:    callq  0x40141f <explode_bomb>
<+48>:    cmpl   $0x7,(%rsp)
<+52>:    ja     0x400fac <phase_3+155>
<+54>:    mov    (%rsp),%eax
<+57>:    jmpq   *0x402420(,%rax,8)
<+64>:    mov    $0x3b1,%eax
<+69>:    jmp    0x400f5d <phase_3+76>
<+71>:    mov    $0x0,%eax
<+76>:    sub    $0x3b3,%eax
<+81>:    jmp    0x400f69 <phase_3+88>
<+83>:    mov    $0x0,%eax
<+88>:    add    $0x138,%eax
<+93>:    jmp    0x400f75 <phase_3+100>
<+95>:    mov    $0x0,%eax
<+100>:   sub    $0x362,%eax
<+105>:   jmp    0x400f81 <phase_3+112>
<+107>:   mov    $0x0,%eax
<+112>:   add    $0x362,%eax
<+117>:   jmp    0x400f8d <phase_3+124>
<+119>:   mov    $0x0,%eax
<+124>:   sub    $0x362,%eax
<+129>:   jmp    0x400f99 <phase_3+136>
<+131>:   mov    $0x0,%eax
<+136>:   add    $0x362,%eax
<+141>:   jmp    0x400fa5 <phase_3+148>
<+143>:   mov    $0x0,%eax
<+148>:   sub    $0x362,%eax
<+153>:   jmp    0x400fb6 <phase_3+165>
<+155>:   callq  0x40141f <explode_bomb>
<+160>:   mov    $0x0,%eax
<+165>:   cmpl   $0x5,(%rsp)
<+169>:   jg     0x400fc2 <phase_3+177>
<+171>:   cmp    0x4(%rsp),%eax
<+175>:   je     0x400fc7 <phase_3+182>
<+177>:   callq  0x40141f <explode_bomb>
<+182>:   mov    0x8(%rsp),%rax
<+187>:   xor    %fs:0x28,%rax
<+196>:   je     0x400fdc <phase_3+203>
<+198>:   callq  0x400b00 <__stack_chk_fail@plt>
<+203>:   add    $0x18,%rsp
```

Figure 8: Phase 3.3

Why we don't care the first part. Because it's jump command everywhere. Eventually, it will end up at <+165>we don't have to worry at all. Now look at the part that we care. Obviously, if rsp >5 then boom, it jumps to the bomb. So our first inputted number must be <=5. Then we compare the next inputted number with eax (which is 0 because of "mov 0x0, eax"). If the second inputted number is not equal to 0 then boom again as it will call the explode_bomb. Therefore, my inputted solution is "4 0".

4 Phase4

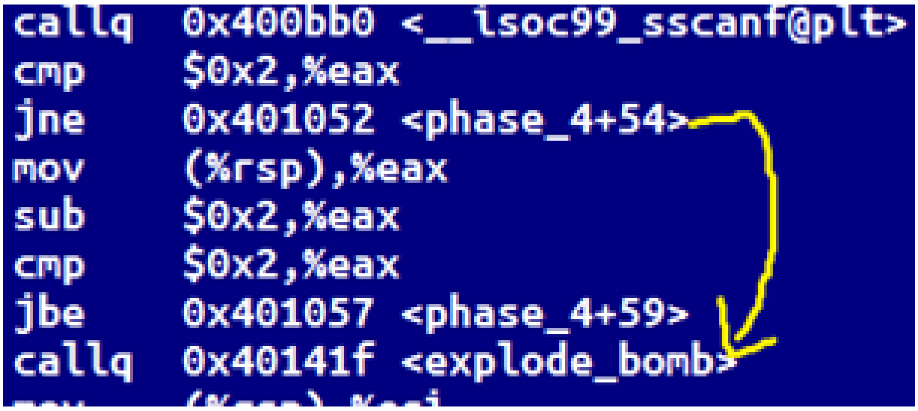
Here is the assembly code of Phase_4 :



```
Dump of assembler code for function phase_4:
=> 0x000000000040101c <+0>:     sub    $0x18,%rsp
   0x0000000000401020 <+4>:     mov    %fs:0x28,%rax
   0x0000000000401029 <+13>:    mov    %rax,0x8(%rsp)
   0x000000000040102e <+18>:    xor    %eax,%eax
   0x0000000000401030 <+20>:    mov    %rsp,%rcx
   0x0000000000401033 <+23>:    lea    0x4(%rsp),%rdx
   0x0000000000401038 <+28>:    mov    $0x4025af,%esi
   0x000000000040103d <+33>:    callq  0x400bb0 <__isoc99_sscanf@plt>
   0x0000000000401042 <+38>:    cmp    $0x2,%eax
   0x0000000000401045 <+41>:    jne    0x401052 <phase_4+54>
   0x0000000000401047 <+43>:    mov    (%rsp),%eax
   0x000000000040104a <+46>:    sub    $0x2,%eax
   0x000000000040104d <+49>:    cmp    $0x2,%eax
   0x0000000000401050 <+52>:    jbe    0x401057 <phase_4+59>
   0x0000000000401052 <+54>:    callq  0x40141f <explode_bomb>
   0x0000000000401057 <+59>:    mov    (%rsp),%esi
   0x000000000040105a <+62>:    mov    $0x8,%edi
   0x000000000040105f <+67>:    callq  0x400fe1 <func4>
   0x0000000000401064 <+72>:    cmp    0x4(%rsp),%eax
   0x0000000000401068 <+76>:    je     0x40106f <phase_4+83>
   0x000000000040106a <+78>:    callq  0x40141f <explode_bomb>
   0x000000000040106f <+83>:    mov    0x8(%rsp),%rax
   0x0000000000401074 <+88>:    xor    %fs:0x28,%rax
   0x000000000040107d <+97>:    je     0x401084 <phase_4+104>
   0x000000000040107f <+99>:    callq  0x400b00 <__stack_chk_fail@plt>
   0x0000000000401084 <+104>:   add    $0x18,%rsp
   0x0000000000401088 <+108>:   retq
End of assembler dump.
```

Figure 9: Phase 4.1

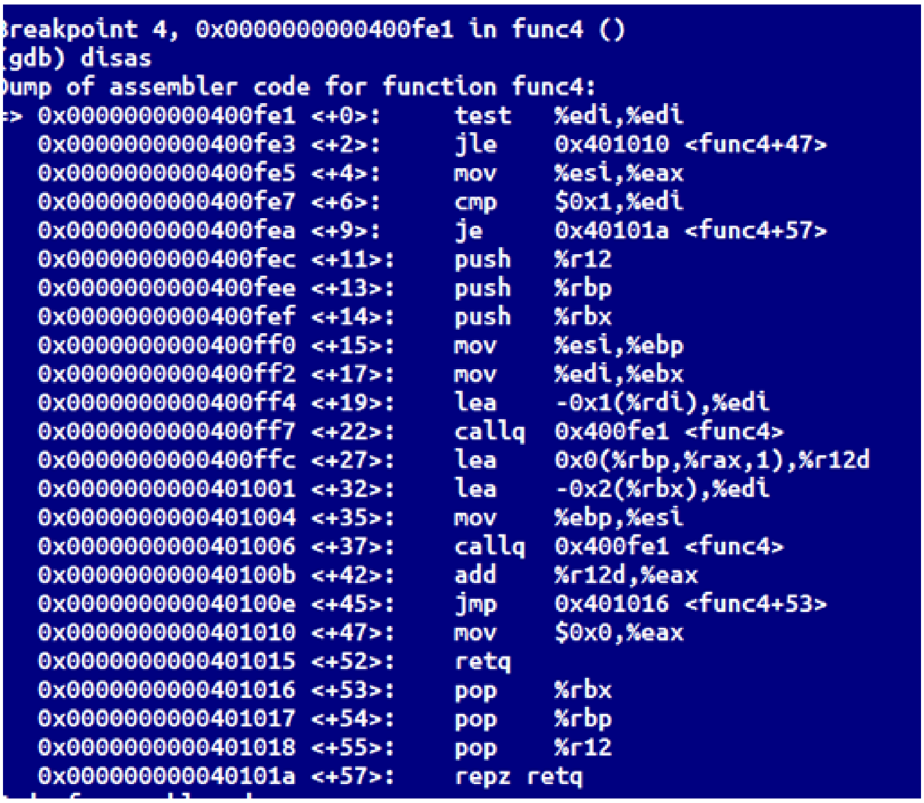
Firstly, this part here tells us that we should have 2 inputted numbers or the bomb will explode. Also, the inputted number must be smaller or equal to 4.



```
callq  0x400bb0 <__isoc99_sscanf@plt>
cmp    $0x2,%eax
jne    0x401052 <phase_4+54>
mov    (%rsp),%eax
sub    $0x2,%eax
cmp    $0x2,%eax
jbe    0x401057 <phase_4+59>
callq  0x40141f <explode_bomb>
mov    (%rsp),%esi
```

Figure 10: Phase 4.2

Then look at line <+67>, it calls func4 so we need to disassemble func4 code to see what it is doing inside.

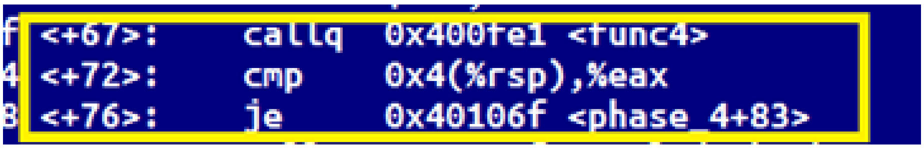


```
Breakpoint 4, 0x0000000000400fe1 in func4 ()
(gdb) disas
Dump of assembler code for function func4:
=> 0x0000000000400fe1 <+0>:     test   %edi,%edi
   0x0000000000400fe3 <+2>:     jle    0x401010 <func4+47>
   0x0000000000400fe5 <+4>:     mov    %esi,%eax
   0x0000000000400fe7 <+6>:     cmp    $0x1,%edi
   0x0000000000400fea <+9>:     je     0x40101a <func4+57>
   0x0000000000400fec <+11>:    push   %r12
   0x0000000000400fee <+13>:    push   %rbp
   0x0000000000400fef <+14>:    push   %rbx
   0x0000000000400ff0 <+15>:    mov    %esi,%ebp
   0x0000000000400ff2 <+17>:    mov    %edi,%ebx
   0x0000000000400ff4 <+19>:    lea    -0x1(%rdi),%edi
   0x0000000000400ff7 <+22>:    callq  0x400fe1 <func4>
   0x0000000000400ffc <+27>:    lea    0x0(%rbp,%rax,1),%r12d
   0x0000000000401001 <+32>:    lea    -0x2(%rbx),%edi
   0x0000000000401004 <+35>:    mov    %ebp,%esi
   0x0000000000401006 <+37>:    callq  0x400fe1 <func4>
   0x000000000040100b <+42>:    add    %r12d,%eax
   0x000000000040100e <+45>:    jmp    0x401016 <func4+53>
   0x0000000000401010 <+47>:    mov    $0x0,%eax
   0x0000000000401015 <+52>:    retq
   0x0000000000401016 <+53>:    pop    %rbx
   0x0000000000401017 <+54>:    pop    %rbp
   0x0000000000401018 <+55>:    pop    %r12
   0x000000000040101a <+57>:    repz retq
```

Figure 11: Phase 4.3

Some important informations about this func4 is that it is a recursion (as on line <+22>it is trying to call itself) and all it does is adding to our first inputted number an equal value for 53 times then check if it matches the second inputted number. In short, 54 times multiply the first number must equals to the second number. (and don't forget the first inputted number must <=4).

After func4, this block here shows that the program try to compare the second inputted number with the result of func4 (which is the first inputted number mutiplied by 54 times.



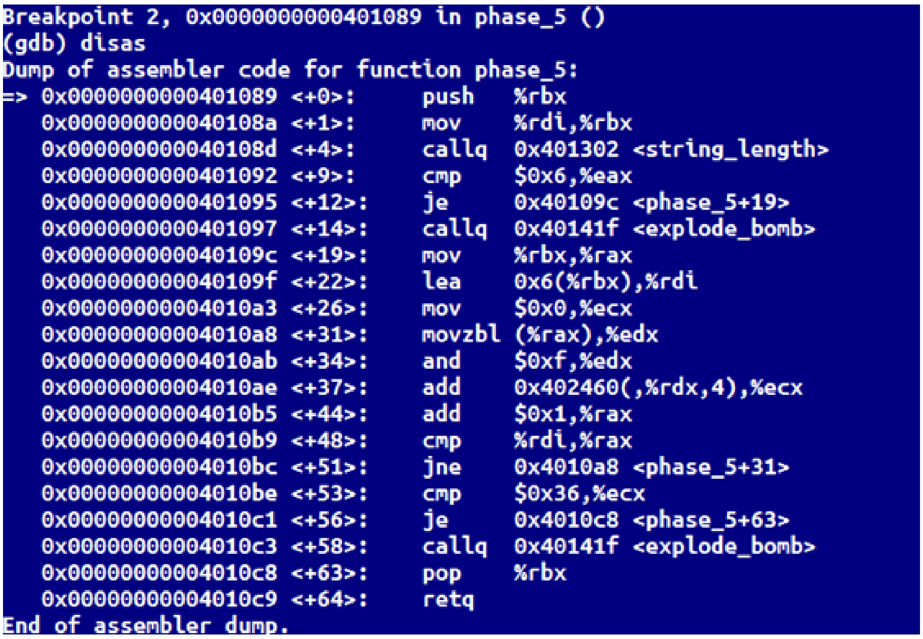
```
<+67>:     callq   0x400fe1 <func4>
<+72>:     cmp     0x4(%rsp),%eax
<+76>:     je      0x40106f <phase_4+83>
```

Figure 12: Phase 4.4

Therefore the result is 162 3

5 Phase5

Here's the assembly code of phase_5

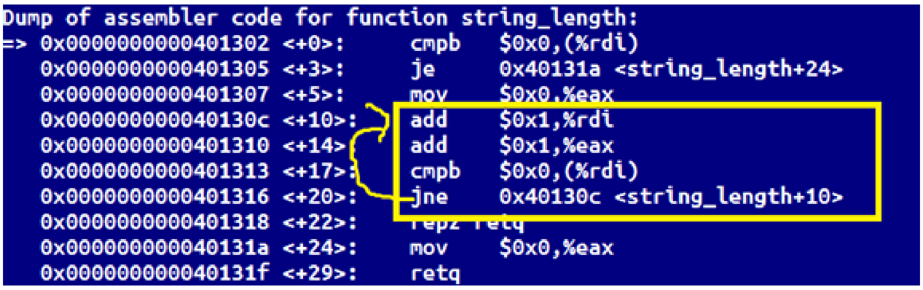


```
Breakpoint 2, 0x0000000000401089 in phase_5 ()
(gdb) disas
Dump of assembler code for function phase_5:
=> 0x0000000000401089 <+0>:     push   %rbx
   0x000000000040108a <+1>:     mov    %rdi,%rbx
   0x000000000040108d <+4>:     callq  0x401302 <string_length>
   0x0000000000401092 <+9>:     cmp    $0x6,%eax
   0x0000000000401095 <+12>:    je     0x40109c <phase_5+19>
   0x0000000000401097 <+14>:    callq  0x40141f <explode_bomb>
   0x000000000040109c <+19>:    mov    %rbx,%rax
   0x000000000040109f <+22>:    lea    0x6(%rbx),%rdi
   0x00000000004010a3 <+26>:    mov    $0x0,%ecx
   0x00000000004010a8 <+31>:    movzbl (%rax),%edx
   0x00000000004010ab <+34>:    and    $0xf,%edx
   0x00000000004010ae <+37>:    add    0x402460(,%rdx,4),%ecx
   0x00000000004010b5 <+44>:    add    $0x1,%rax
   0x00000000004010b9 <+48>:    cmp    %rdi,%rax
   0x00000000004010bc <+51>:    jne    0x4010a8 <phase_5+31>
   0x00000000004010be <+53>:    cmp    $0x36,%ecx
   0x00000000004010c1 <+56>:    je     0x4010c8 <phase_5+63>
   0x00000000004010c3 <+58>:    callq  0x40141f <explode_bomb>
   0x00000000004010c8 <+63>:    pop    %rbx
   0x00000000004010c9 <+64>:    retq
End of assembler dump.
```

Figure 13: Phase 5.1

Clearly, we can see that it does something in the <string_length>function. If the return of that function is smaller than 6 or bigger than 6, then the callq on

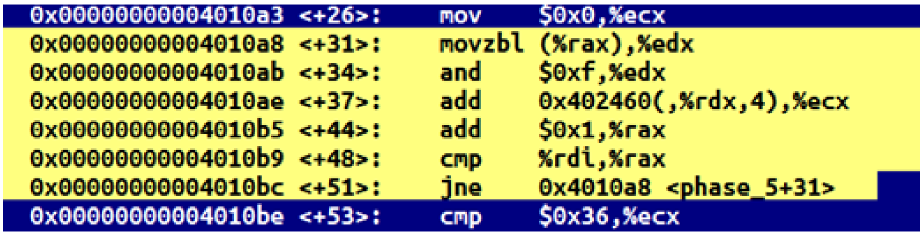
<+14>will execute, which leads to the bomb. Therefore, we predict that we must enter a string with 6 numbers. Take a closer look into string_length :



```
Dump of assembler code for function string_length:
=> 0x0000000000401302 <+0>:    cmpb   $0x0,(%rdi)
   0x0000000000401305 <+3>:    je     0x40131a <string_length+24>
   0x0000000000401307 <+5>:    mov    $0x0,%eax
   0x000000000040130c <+10>:   add    $0x1,%rdi
   0x0000000000401310 <+14>:   add    $0x1,%eax
   0x0000000000401313 <+17>:   cmpb   $0x0,(%rdi)
   0x0000000000401316 <+20>:   jne    0x40130c <string_length+10>
   0x0000000000401318 <+22>:   repz retq
   0x000000000040131a <+24>:   mov    $0x0,%eax
   0x000000000040131f <+29>:   retq
```

Figure 14: Phase 5.2

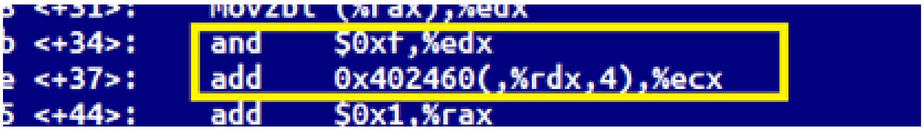
From line 10 to line 20, it is trying to do a loop in which it checks if the string length is equal to 6. Nothing special here.
Now we proceed to the next important block . We can detect a loop at <+31>



```
0x00000000004010a3 <+26>:    mov    $0x0,%ecx
0x00000000004010a8 <+31>:    movzbl (%rax),%edx
0x00000000004010ab <+34>:    and    $0xf,%edx
0x00000000004010ae <+37>:    add    0x402460(,%rdx,4),%ecx
0x00000000004010b5 <+44>:    add    $0x1,%rax
0x00000000004010b9 <+48>:    cmp    %rdi,%rax
0x00000000004010bc <+51>:    jne    0x4010a8 <phase_5+31>
0x00000000004010be <+53>:    cmp    $0x36,%ecx
```

Figure 15: Phase 5.3

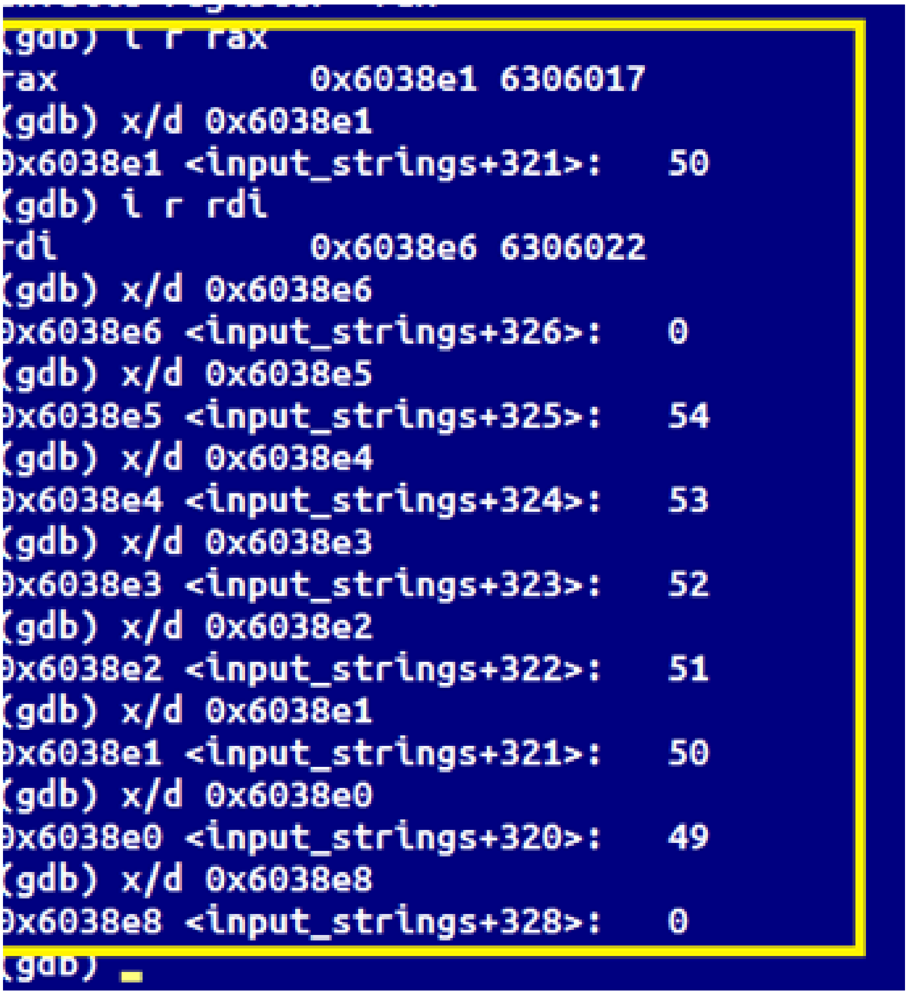
We notice that the loop tries to accomplish something with ecx since it is the address that doesn't move but keep being added up.



```
8 <+31>:    movzbl (%rax),%edx
b <+34>:    and    $0xf,%edx
e <+37>:    add    0x402460(,%rdx,4),%ecx
5 <+44>:    add    $0x1,%rax
```

Figure 16: Phase 5.4

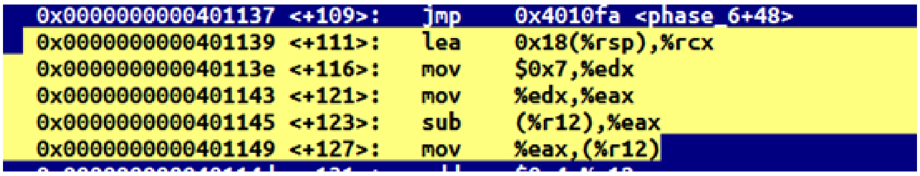
After some investigations, we will discover rax is the first element in our inputted string and rdi is the last element in our inputed string. (I inputted 123456, 49 to 54 is their elements' ascii code).



```
(gdb) i r rax
rax            0x6038e1 6306017
(gdb) x/d 0x6038e1
0x6038e1 <input_strings+321>:   50
(gdb) i r rdi
rdi            0x6038e6 6306022
(gdb) x/d 0x6038e6
0x6038e6 <input_strings+326>:   0
(gdb) x/d 0x6038e5
0x6038e5 <input_strings+325>:   54
(gdb) x/d 0x6038e4
0x6038e4 <input_strings+324>:   53
(gdb) x/d 0x6038e3
0x6038e3 <input_strings+323>:   52
(gdb) x/d 0x6038e2
0x6038e2 <input_strings+322>:   51
(gdb) x/d 0x6038e1
0x6038e1 <input_strings+321>:   50
(gdb) x/d 0x6038e0
0x6038e0 <input_strings+320>:   49
(gdb) x/d 0x6038e8
0x6038e8 <input_strings+328>:   0
(gdb) 
```

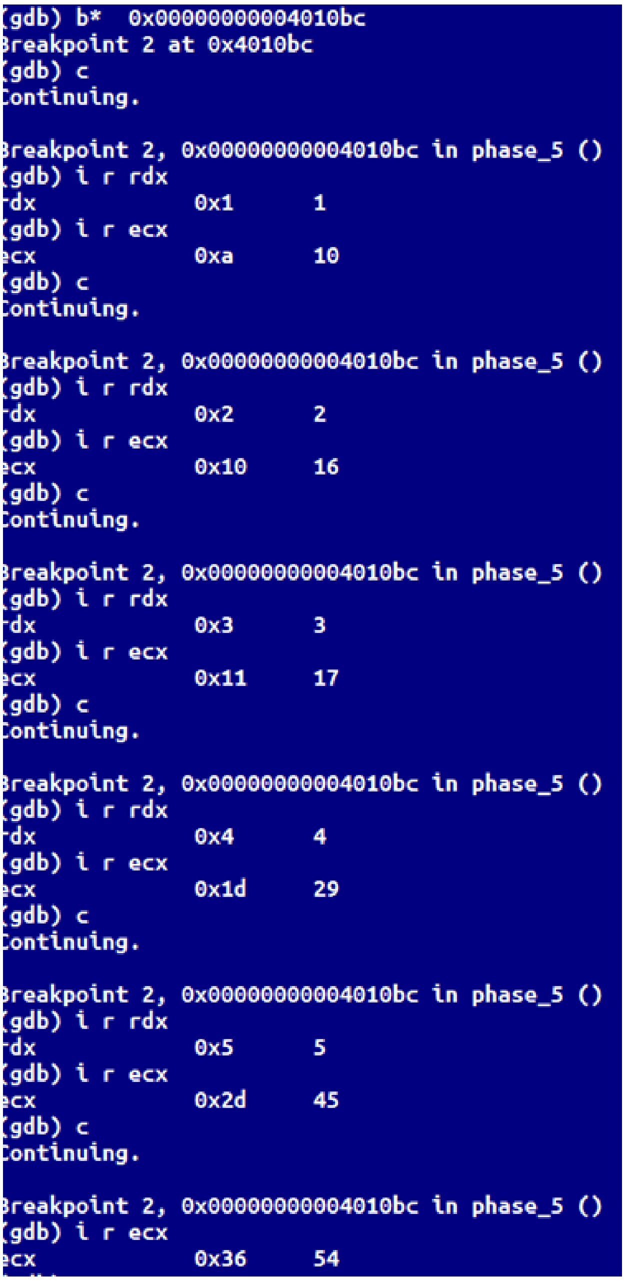
Figure 17: Phase 5.5

Also, by looking deeply in how rdx and rax change, we can conclude that the loop tries to change the address of eax. Now look at this .



```
   0x0000000000401137 <+109>:   jmp    0x4010fa <phase_6+48>
   0x0000000000401139 <+111>:   lea    0x18(%rsp),%rcx
   0x000000000040113e <+116>:   mov    $0x7,%edx
   0x0000000000401143 <+121>:   mov    %edx,%eax
   0x0000000000401145 <+123>:   sub    (%r12),%eax
   0x0000000000401149 <+127>:   mov    %eax,(%r12)
```

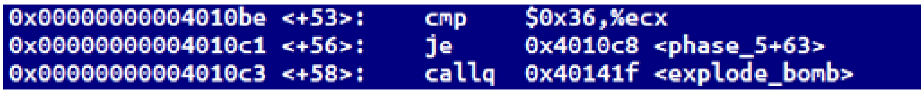
Figure 18: Phase 5.6



```
(gdb) b*  0x00000000004010bc
Breakpoint 2 at 0x4010bc
(gdb) c
Continuing.

Breakpoint 2, 0x00000000004010bc in phase_5 ()
(gdb) i r rdx
rdx            0x1      1
(gdb) i r ecx
ecx            0xa      10
(gdb) c
Continuing.

Breakpoint 2, 0x00000000004010bc in phase_5 ()
(gdb) i r rdx
rdx            0x2      2
(gdb) i r ecx
ecx            0x10     16
(gdb) c
Continuing.

Breakpoint 2, 0x00000000004010bc in phase_5 ()
(gdb) i r rdx
rdx            0x3      3
(gdb) i r ecx
ecx            0x11     17
(gdb) c
Continuing.

Breakpoint 2, 0x00000000004010bc in phase_5 ()
(gdb) i r rdx
rdx            0x4      4
(gdb) i r ecx
ecx            0x1d     29
(gdb) c
Continuing.

Breakpoint 2, 0x00000000004010bc in phase_5 ()
(gdb) i r rdx
rdx            0x5      5
(gdb) i r ecx
ecx            0x2d     45
(gdb) c
Continuing.

Breakpoint 2, 0x00000000004010bc in phase_5 ()
(gdb) i r ecx
ecx            0x36     54
```

Figure 19: Phase 5.7

But how much do we need to change eax ? It's 54 in decimal or 36 in hexadecimal. Therefore, my input ”123456” meets the requirement.

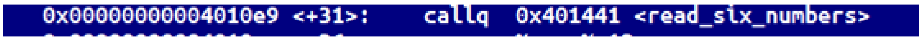


```
0x00000000004010be <+53>:    cmp    $0x36,%ecx
0x00000000004010c1 <+56>:    je     0x4010c8 <phase_5+63>
0x00000000004010c3 <+58>:    callq  0x40141f <explode_bomb>
```

Figure 20: Phase 5.8

6 Phase6

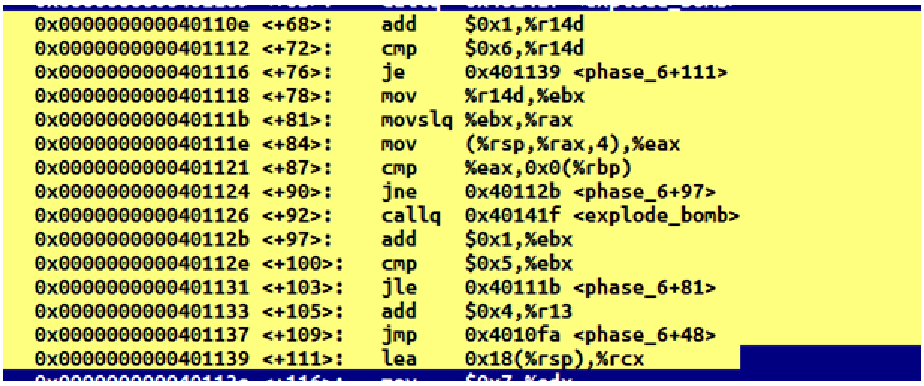
Phase 6's assembly code is too long to be put all in here. Let's just dig into some important parts only. At first glance, the program begins to read in 6 numbers we inputed.



```
   0x00000000004010e9 <+31>:    callq  0x401441 <read_six_numbers>
```

Figure 21: Phase 6.1

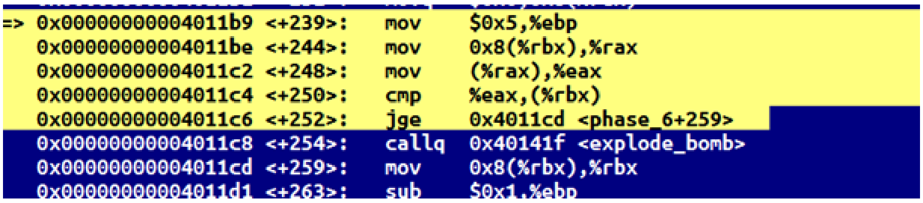
This loops tell us that the input must be 6 distinct numbers and smaller or equal to 6. So that's 1 2 3 4 5 6.



```
   0x000000000040110e <+68>:    add    $0x1,%r14d
   0x0000000000401112 <+72>:    cmp    $0x6,%r14d
   0x0000000000401116 <+76>:    je     0x401139 <phase_6+111>
   0x0000000000401118 <+78>:    mov    %r14d,%ebx
   0x000000000040111b <+81>:    movslq %ebx,%rax
   0x000000000040111e <+84>:    mov    (%rsp,%rax,4),%eax
   0x0000000000401121 <+87>:    cmp    %eax,0x0(%rbp)
   0x0000000000401124 <+90>:    jne    0x40112b <phase_6+97>
   0x0000000000401126 <+92>:    callq  0x40141f <explode_bomb>
   0x000000000040112b <+97>:    add    $0x1,%ebx
   0x000000000040112e <+100>:   cmp    $0x5,%ebx
   0x0000000000401131 <+103>:   jle    0x40111b <phase_6+81>
   0x0000000000401133 <+105>:   add    $0x4,%r13
   0x0000000000401137 <+109>:   jmp    0x4010fa <phase_6+48>
   0x0000000000401139 <+111>:   lea    0x18(%rsp),%rcx
```

Figure 22: Phase 6.2

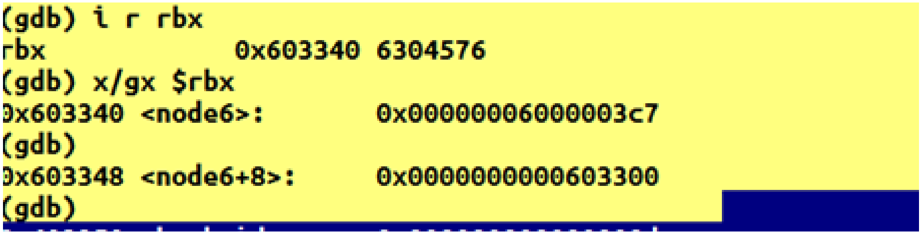
These lines here told us that the code subtract our inputted number by 7, then use the new value. After that these lines here



```
=> 0x00000000004011b9 <+239>:   mov    $0x5,%ebp
   0x00000000004011be <+244>:   mov    0x8(%rbx),%rax
   0x00000000004011c2 <+248>:   mov    (%rax),%eax
   0x00000000004011c4 <+250>:   cmp    %eax,(%rbx)
   0x00000000004011c6 <+252>:   jge    0x4011cd <phase_6+259>
   0x00000000004011c8 <+254>:   callq  0x40141f <explode_bomb>
   0x00000000004011cd <+259>:   mov    0x8(%rbx),%rbx
   0x00000000004011d1 <+263>:   sub    $0x1,%ebp
```

Figure 23: Phase 6.3

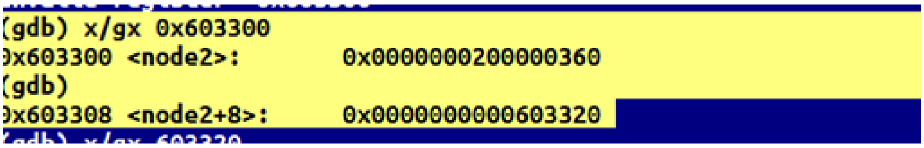
Told us that they are using the new values to compare. If these new values (7 - each inputted number) follows ascending order, it would be fine. (which means that our input must be in ascending order). However it is not as simple as 1 2 3 4 5 6. Every number got their own code so we got to use x/gx + address to decode all the number. For example :



```
(gdb) i r rbx
rbx            0x603340 6304576
(gdb) x/gx $rbx
0x603340 <node6>:       0x00000006000003c7
(gdb)
0x603348 <node6+8>:     0x0000000000603300
(gdb)
```

Figure 24: Phase 6.4

Rbx is the way to all the inputted number, so we use x/gx to find out the address of each member in it. Firstly, it is x/gx $rbx. Here we have ¡node6¿ is the hex value of 6. And Node 6+8 is the address of the next node. Keep doing that and we will find the next node hex value and the next next node .



```
(gdb) x/gx 0x603300
0x603300 <node2>:       0x0000000200000360
(gdb)
0x603308 <node2+8>:     0x0000000000603320
(gdb) x/gx 603320
```

Figure 25: Phase 6.5

After all, sort all the hex value and rearrange the input . We get the answer 1 5 3 6 2 4