2023 Fall CSED 211 Lab Report

Lab number: 3 (Bomb lab)

Student number: 20220302

Name: 김지현

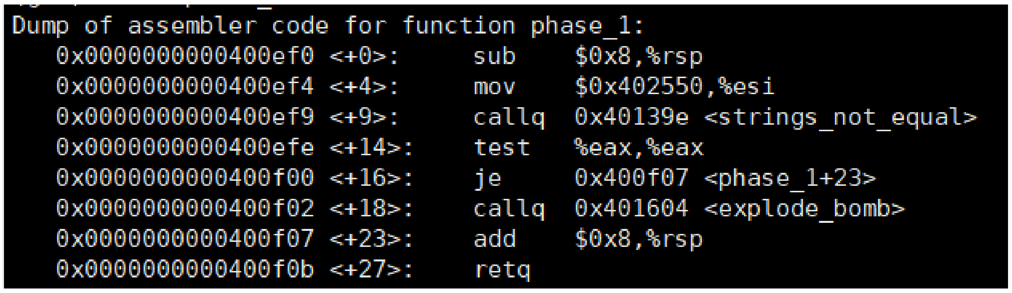
1. Introduction

The objective of this lab is to defuse the "bomb" by avoiding the function call "explode_bomb." There are 6 phases, each more challenging than the last, and we should analyze the disassembled code and come up with the correct input to proceed to the next phase. Each phase may contain loops, recursion, or switch cases.

2. System Design/ Algorithm

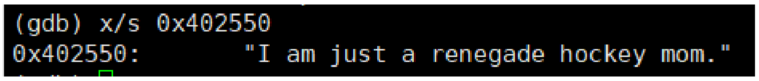
Before disassembling each phase, I made a breakpoint at the function <explode_bomb>.

1. Phase 1



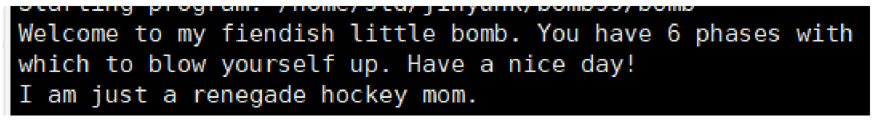
```
Dump of assembler code for function phase_1:
   0x0000000000400ef0 <+0>:     sub    $0x8,%rsp
   0x0000000000400ef4 <+4>:     mov    $0x402550,%esi
   0x0000000000400ef9 <+9>:     callq  0x40139e <strings_not_equal>
   0x0000000000400efe <+14>:    test   %eax,%eax
   0x0000000000400f00 <+16>:    je     0x400f07 <phase_1+23>
   0x0000000000400f02 <+18>:    callq  0x401604 <explode_bomb>
   0x0000000000400f07 <+23>:    add    $0x8,%rsp
   0x0000000000400f0b <+27>:    retq
```

From <+9>, we know the <strings_not_equal> function is called. And to avoid the bomb, the register %eax must be 0. When you do a "test", the result is 0 if and only if the register is 0. With this information, I used "x/s" to reveal the string in the address 0x402550 from the <+9>.

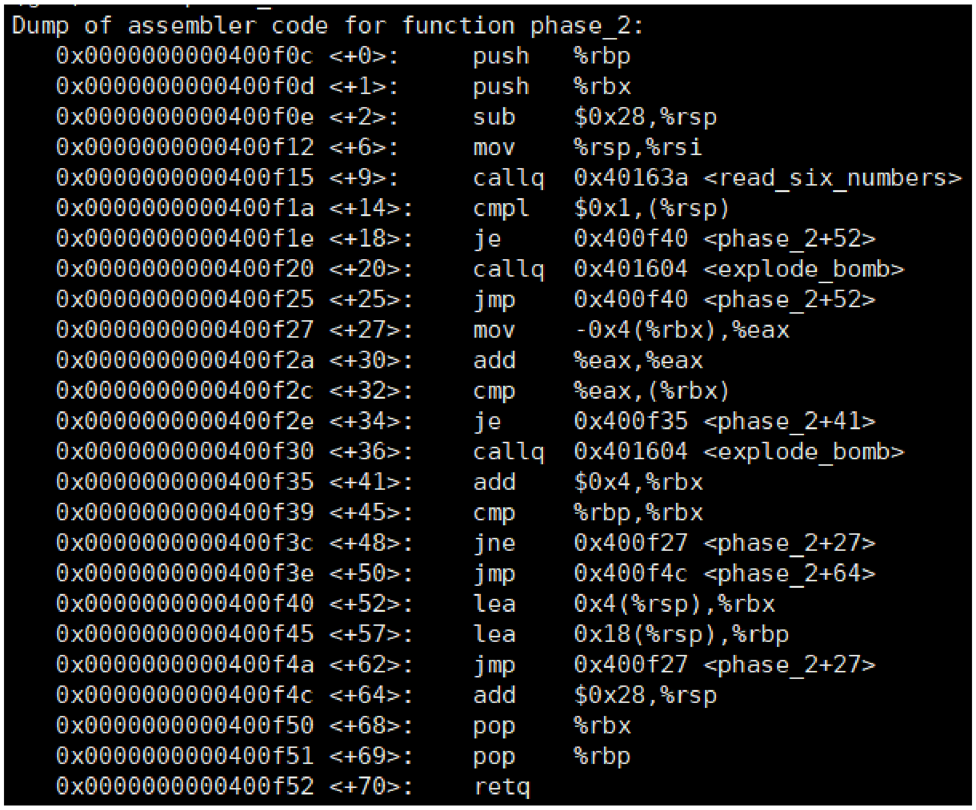


```
(gdb) x/s 0x402550
0x402550:       "I am just a renegade hockey mom."
```

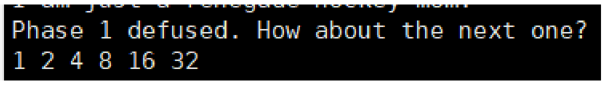
As a result, the answer for the first phase is "I am just a renegade hockey mom."


```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am just a renegade hockey mom.
```

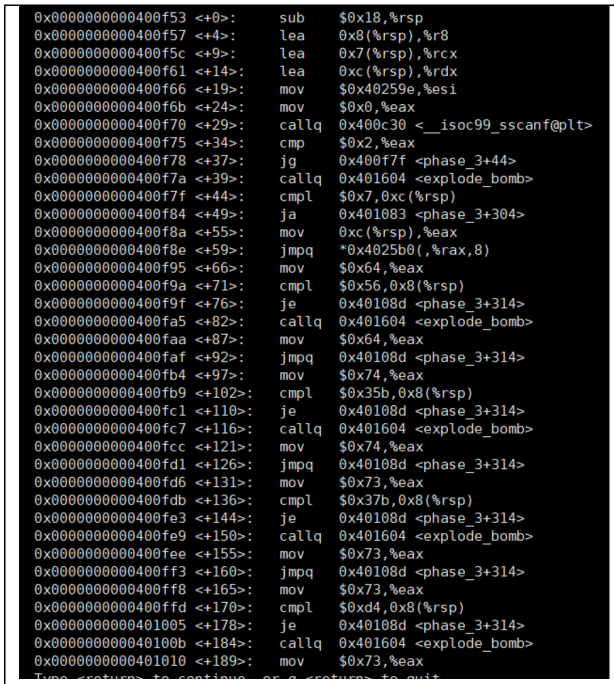
2. Phase 2


```
Dump of assembler code for function phase_2:
   0x0000000000400f0c <+0>:     push   %rbp
   0x0000000000400f0d <+1>:     push   %rbx
   0x0000000000400f0e <+2>:     sub    $0x28,%rsp
   0x0000000000400f12 <+6>:     mov    %rsp,%rsi
   0x0000000000400f15 <+9>:     callq  0x40163a <read_six_numbers>
   0x0000000000400f1a <+14>:    cmpl   $0x1,(%rsp)
   0x0000000000400f1e <+18>:    je     0x400f40 <phase_2+52>
   0x0000000000400f20 <+20>:    callq  0x401604 <explode_bomb>
   0x0000000000400f25 <+25>:    jmp    0x400f40 <phase_2+52>
   0x0000000000400f27 <+27>:    mov    -0x4(%rbx),%eax
   0x0000000000400f2a <+30>:    add    %eax,%eax
   0x0000000000400f2c <+32>:    cmp    %eax,(%rbx)
   0x0000000000400f2e <+34>:    je     0x400f35 <phase_2+41>
   0x0000000000400f30 <+36>:    callq  0x401604 <explode_bomb>
   0x0000000000400f35 <+41>:    add    $0x4,%rbx
   0x0000000000400f39 <+45>:    cmp    %rbp,%rbx
   0x0000000000400f3c <+48>:    jne    0x400f27 <phase_2+27>
   0x0000000000400f3e <+50>:    jmp    0x400f4c <phase_2+64>
   0x0000000000400f40 <+52>:    lea    0x4(%rsp),%rbx
   0x0000000000400f45 <+57>:    lea    0x18(%rsp),%rbp
   0x0000000000400f4a <+62>:    jmp    0x400f27 <phase_2+27>
   0x0000000000400f4c <+64>:    add    $0x28,%rsp
   0x0000000000400f50 <+68>:    pop    %rbx
   0x0000000000400f51 <+69>:    pop    %rbp
   0x0000000000400f52 <+70>:    retq
```

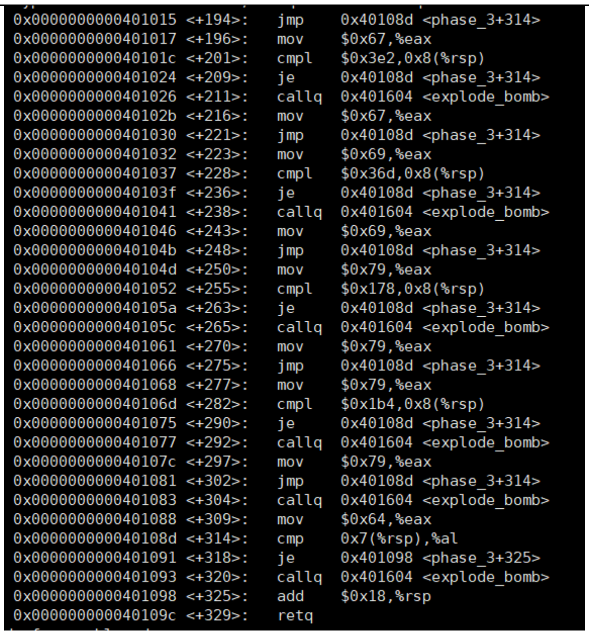
When you disassemble the <read_six_numbers>, we know that the inputs are the six numbers. From the lines <+14> and <+18>, we know that the first number must be 1 because the top of the stack, which is stored in the address of %rsp, must be 1. From the rest of the lines, We know that the next number must be the addition of the previous numbers. For example, the second input will be 2 because 2 = 1 + 1. In the same sense, the answer for phase 2 will be 1 2 4 8 16 32.


```
Phase 1 defused. How about the next one?
1 2 4 8 16 32
```

3. Phase 3

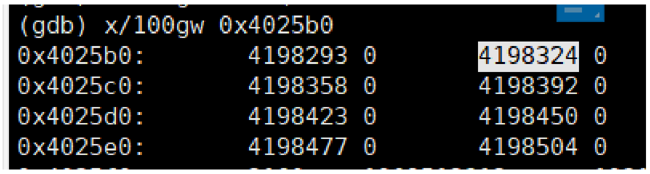


```
0x0000000000400f53 <+0>:     sub    $0x18,%rsp
0x0000000000400f57 <+4>:     lea    0x8(%rsp),%r8
0x0000000000400f5c <+9>:     lea    0x7(%rsp),%rcx
0x0000000000400f61 <+14>:    lea    0xc(%rsp),%rdx
0x0000000000400f66 <+19>:    mov    $0x40259e,%esi
0x0000000000400f6b <+24>:    mov    $0x0,%eax
0x0000000000400f70 <+29>:    callq  0x400c30 <__isoc99_sscanf@plt>
0x0000000000400f75 <+34>:    cmp    $0x2,%eax
0x0000000000400f78 <+37>:    jg     0x400f7f <phase_3+44>
0x0000000000400f7a <+39>:    callq  0x401604 <explode_bomb>
0x0000000000400f7f <+44>:    cmpl   $0x7,0xc(%rsp)
0x0000000000400f84 <+49>:    ja     0x401083 <phase_3+304>
0x0000000000400f8a <+55>:    mov    0xc(%rsp),%eax
0x0000000000400f8e <+59>:    jmpq   *0x4025b0(,%rax,8)
0x0000000000400f95 <+66>:    mov    $0x64,%eax
0x0000000000400f9a <+71>:    cmpl   $0x56,0x8(%rsp)
0x0000000000400f9f <+76>:    je     0x40108d <phase_3+314>
0x0000000000400fa5 <+82>:    callq  0x401604 <explode_bomb>
0x0000000000400faa <+87>:    mov    $0x64,%eax
0x0000000000400faf <+92>:    jmpq   0x40108d <phase_3+314>
0x0000000000400fb4 <+97>:    mov    $0x74,%eax
0x0000000000400fb9 <+102>:   cmpl   $0x35b,0x8(%rsp)
0x0000000000400fc1 <+110>:   je     0x40108d <phase_3+314>
0x0000000000400fc7 <+116>:   callq  0x401604 <explode_bomb>
0x0000000000400fcc <+121>:   mov    $0x74,%eax
0x0000000000400fd1 <+126>:   jmpq   0x40108d <phase_3+314>
0x0000000000400fd6 <+131>:   mov    $0x73,%eax
0x0000000000400fdb <+136>:   cmpl   $0x37b,0x8(%rsp)
0x0000000000400fe3 <+144>:   je     0x40108d <phase_3+314>
0x0000000000400fe9 <+150>:   callq  0x401604 <explode_bomb>
0x0000000000400fee <+155>:   mov    $0x73,%eax
0x0000000000400ff3 <+160>:   jmpq   0x40108d <phase_3+314>
0x0000000000400ff8 <+165>:   mov    $0x73,%eax
0x0000000000400ffd <+170>:   cmpl   $0xd4,0x8(%rsp)
0x0000000000401005 <+178>:   je     0x40108d <phase_3+314>
0x000000000040100b <+184>:   callq  0x401604 <explode_bomb>
0x0000000000401010 <+189>:   mov    $0x73,%eax
```



```
0x0000000000401015 <+194>:   jmp    0x40108d <phase_3+314>
0x0000000000401017 <+196>:   mov    $0x67,%eax
0x000000000040101c <+201>:   cmpl   $0x3e2,0x8(%rsp)
0x0000000000401024 <+209>:   je     0x40108d <phase_3+314>
0x0000000000401026 <+211>:   callq  0x401604 <explode_bomb>
0x000000000040102b <+216>:   mov    $0x67,%eax
0x0000000000401030 <+221>:   jmp    0x40108d <phase_3+314>
0x0000000000401032 <+223>:   mov    $0x69,%eax
0x0000000000401037 <+228>:   cmpl   $0x36d,0x8(%rsp)
0x000000000040103f <+236>:   je     0x40108d <phase_3+314>
0x0000000000401041 <+238>:   callq  0x401604 <explode_bomb>
0x0000000000401046 <+243>:   mov    $0x69,%eax
0x000000000040104b <+248>:   jmp    0x40108d <phase_3+314>
0x000000000040104d <+250>:   mov    $0x79,%eax
0x0000000000401052 <+255>:   cmpl   $0x178,0x8(%rsp)
0x000000000040105a <+263>:   je     0x40108d <phase_3+314>
0x000000000040105c <+265>:   callq  0x401604 <explode_bomb>
0x0000000000401061 <+270>:   mov    $0x79,%eax
0x0000000000401066 <+275>:   jmp    0x40108d <phase_3+314>
0x0000000000401068 <+277>:   mov    $0x79,%eax
0x000000000040106d <+282>:   cmpl   $0x1b4,0x8(%rsp)
0x0000000000401075 <+290>:   je     0x40108d <phase_3+314>
0x0000000000401077 <+292>:   callq  0x401604 <explode_bomb>
0x000000000040107c <+297>:   mov    $0x79,%eax
0x0000000000401081 <+302>:   jmp    0x40108d <phase_3+314>
0x0000000000401083 <+304>:   callq  0x401604 <explode_bomb>
0x0000000000401088 <+309>:   mov    $0x64,%eax
0x000000000040108d <+314>:   cmp    0x7(%rsp),%al
0x0000000000401091 <+318>:   je     0x401098 <phase_3+325>
0x0000000000401093 <+320>:   callq  0x401604 <explode_bomb>
0x0000000000401098 <+325>:   add    $0x18,%rsp
0x000000000040109c <+329>:   retq
```

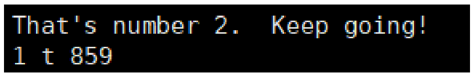
Phase 4 is about the jump table. When you examine the scan function, the inputs are "%d %c %d" (integer char integer). We know that the first input is smaller than 7 from the line <+44>. Then, we jump to a certain address depending on the first input.



```
(gdb) x/100gw 0x4025b0
0x4025b0:       4198293 0       4198324 0
0x4025c0:       4198358 0       4198392 0
0x4025d0:       4198423 0       4198450 0
0x4025e0:       4198477 0       4198504 0
```

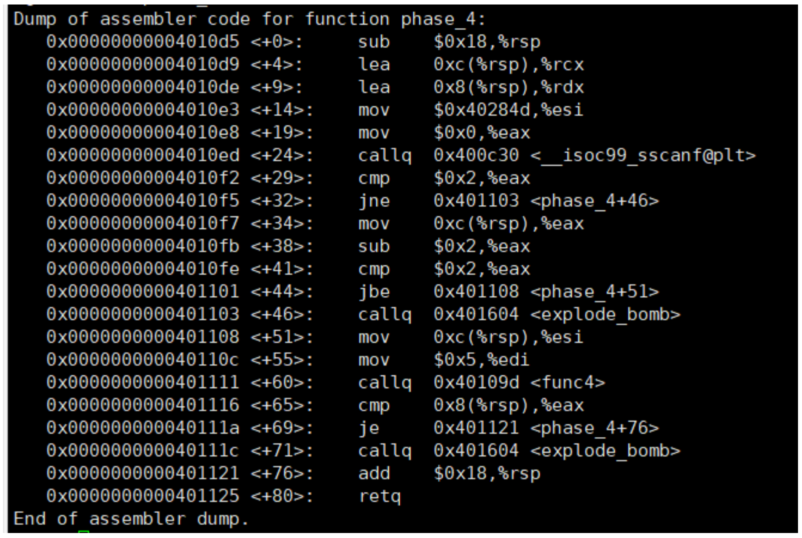
Let's assume that the first input is 1. The above picture shows the address for jump instruction based on the first input (between 0 and 7). Previously, we assumed the first input to be 1, so we jumped to the following address highlighted which is in decimal form. As a result, we jump to 0x400fb4 <+97>. At the line <+97>, the register eax, which is the char input, is 0x74. When we check the ASCII code, 0x74 is 't'.

Finally, at the line <+102>, we can figure out that the second integer input is 0x35b, which is 859 in decimal. To sum up, one of the answers of phase 3 is 1 t 859.

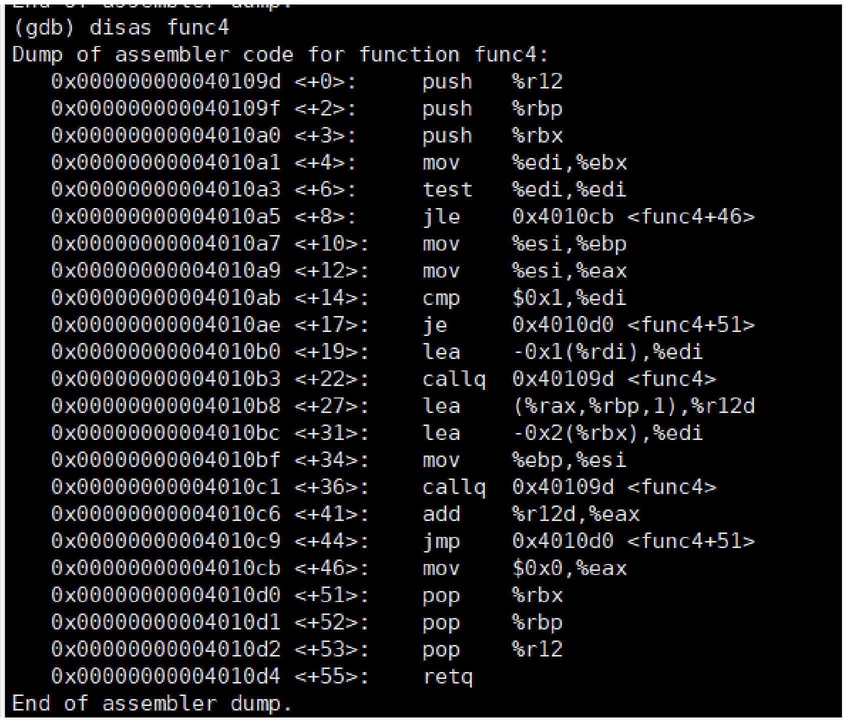


```
That's number 2.  Keep going!
1 t 859
```

4. Phase 4

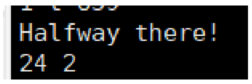

```
Dump of assembler code for function phase_4:
   0x00000000004010d5 <+0>:     sub    $0x18,%rsp
   0x00000000004010d9 <+4>:     lea    0xc(%rsp),%rcx
   0x00000000004010de <+9>:     lea    0x8(%rsp),%rdx
   0x00000000004010e3 <+14>:    mov    $0x40284d,%esi
   0x00000000004010e8 <+19>:    mov    $0x0,%eax
   0x00000000004010ed <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x00000000004010f2 <+29>:    cmp    $0x2,%eax
   0x00000000004010f5 <+32>:    jne    0x401103 <phase_4+46>
   0x00000000004010f7 <+34>:    mov    0xc(%rsp),%eax
   0x00000000004010fb <+38>:    sub    $0x2,%eax
   0x00000000004010fe <+41>:    cmp    $0x2,%eax
   0x0000000000401101 <+44>:    jbe    0x401108 <phase_4+51>
   0x0000000000401103 <+46>:    callq  0x401604 <explode_bomb>
   0x0000000000401108 <+51>:    mov    0xc(%rsp),%esi
   0x000000000040110c <+55>:    mov    $0x5,%edi
   0x0000000000401111 <+60>:    callq  0x40109d <func4>
   0x0000000000401116 <+65>:    cmp    0x8(%rsp),%eax
   0x000000000040111a <+69>:    je     0x401121 <phase_4+76>
   0x000000000040111c <+71>:    callq  0x401604 <explode_bomb>
   0x0000000000401121 <+76>:    add    $0x18,%rsp
   0x0000000000401125 <+80>:    retq
End of assembler dump.
```

From the scan function at line <+24> and the following compare and jump instruction, we can figure out that there are two inputs, both being integers. Then, from the following lines from <+34> to <+44>, we can figure out that the second input must be smaller than 4. Next, we call the <func4> with arguments being edi (=5) and esi.


```
(gdb) disas func4
Dump of assembler code for function func4:
   0x000000000040109d <+0>:     push   %r12
   0x000000000040109f <+2>:     push   %rbp
   0x00000000004010a0 <+3>:     push   %rbx
   0x00000000004010a1 <+4>:     mov    %edi,%ebx
   0x00000000004010a3 <+6>:     test   %edi,%edi
   0x00000000004010a5 <+8>:     jle    0x4010cb <func4+46>
   0x00000000004010a7 <+10>:    mov    %esi,%ebp
   0x00000000004010a9 <+12>:    mov    %esi,%eax
   0x00000000004010ab <+14>:    cmp    $0x1,%edi
   0x00000000004010ae <+17>:    je     0x4010d0 <func4+51>
   0x00000000004010b0 <+19>:    lea    -0x1(%rdi),%edi
   0x00000000004010b3 <+22>:    callq  0x40109d <func4>
   0x00000000004010b8 <+27>:    lea    (%rax,%rbp,1),%r12d
   0x00000000004010bc <+31>:    lea    -0x2(%rbx),%edi
   0x00000000004010bf <+34>:    mov    %ebp,%esi
   0x00000000004010c1 <+36>:    callq  0x40109d <func4>
   0x00000000004010c6 <+41>:    add    %r12d,%eax
   0x00000000004010c9 <+44>:    jmp    0x4010d0 <func4+51>
   0x00000000004010cb <+46>:    mov    $0x0,%eax
   0x00000000004010d0 <+51>:    pop    %rbx
   0x00000000004010d1 <+52>:    pop    %rbp
   0x00000000004010d2 <+53>:    pop    %r12
   0x00000000004010d4 <+55>:    retq
End of assembler dump.
```

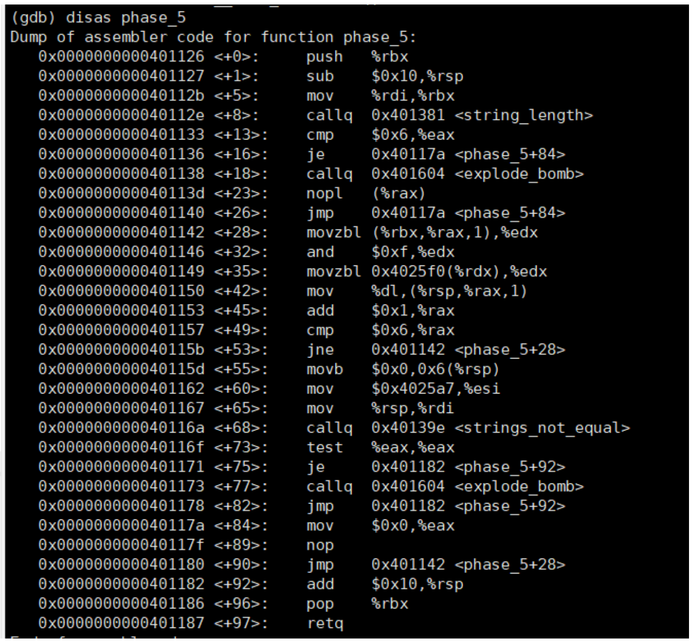
Now, when we look at the disassembled code, we know that func4 is a recursive function because the function itself is called within the function. If the first argument (edi) is 0, the function returns 0. If the first argument (edi) is 1, the function returns eax, which will be the second argument(esi). Else, as we reduce the edi by 1 and do the recursion. In two different places. The first recursion occurs in line <+22> with edi = 5, 4, 3, 2, 1, 0. Then the second recursion occurs in the line <+36> with edi = 2, 1, 0. If edi is neither 1 nor 0, the 2 * second argument is added to the rax based on the lines <+27> and <+41>. Let's assume that the second input from the above <phase_4> code is 2. Then we add 4 to eax each time recursion occurs, therefore the func4 returns 24.

Then, when we look at the phase_4 function, the return value must be the first input based on the lines <+65> and <+69>. To sum up, one of the answers for phase 4 is 24 2.



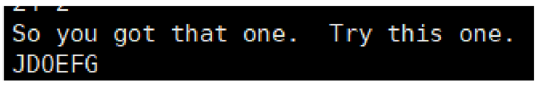
```
Halfway there!
24 2
```

5. Phase 5



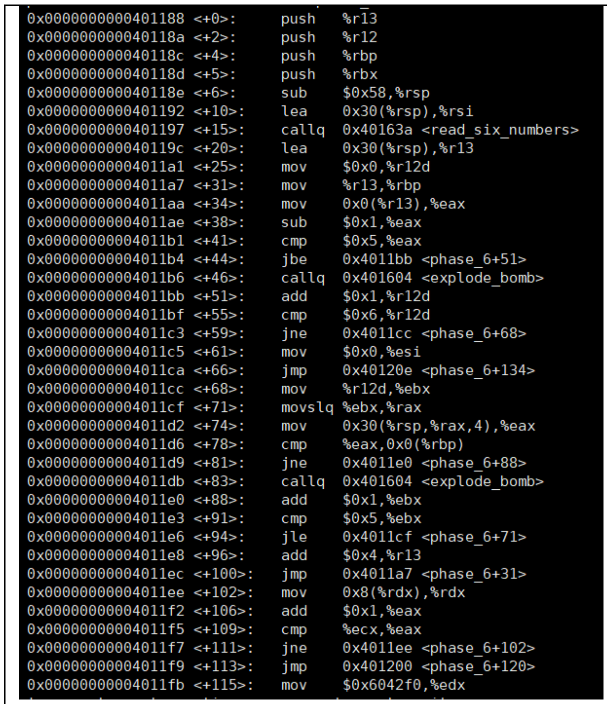
```
(gdb) disas phase_5
Dump of assembler code for function phase_5:
   0x0000000000401126 <+0>:     push   %rbx
   0x0000000000401127 <+1>:     sub    $0x10,%rsp
   0x000000000040112b <+5>:     mov    %rdi,%rbx
   0x000000000040112e <+8>:     callq  0x401381 <string_length>
   0x0000000000401133 <+13>:    cmp    $0x6,%eax
   0x0000000000401136 <+16>:    je     0x40117a <phase_5+84>
   0x0000000000401138 <+18>:    callq  0x401604 <explode_bomb>
   0x000000000040113d <+23>:    nopl   (%rax)
   0x0000000000401140 <+26>:    jmp    0x40117a <phase_5+84>
   0x0000000000401142 <+28>:    movzbl (%rbx,%rax,1),%edx
   0x0000000000401146 <+32>:    and    $0xf,%edx
   0x0000000000401149 <+35>:    movzbl 0x4025f0(%rdx),%edx
   0x0000000000401150 <+42>:    mov    %dl,(%rsp,%rax,1)
   0x0000000000401153 <+45>:    add    $0x1,%rax
   0x0000000000401157 <+49>:    cmp    $0x6,%rax
   0x000000000040115b <+53>:    jne    0x401142 <phase_5+28>
   0x000000000040115d <+55>:    movb   $0x0,0x6(%rsp)
   0x0000000000401162 <+60>:    mov    $0x4025a7,%esi
   0x0000000000401167 <+65>:    mov    %rsp,%rdi
   0x000000000040116a <+68>:    callq  0x40139e <strings_not_equal>
   0x000000000040116f <+73>:    test   %eax,%eax
   0x0000000000401171 <+75>:    je     0x401182 <phase_5+92>
   0x0000000000401173 <+77>:    callq  0x401604 <explode_bomb>
   0x0000000000401178 <+82>:    jmp    0x401182 <phase_5+92>
   0x000000000040117a <+84>:    mov    $0x0,%eax
   0x000000000040117f <+89>:    nop
   0x0000000000401180 <+90>:    jmp    0x401142 <phase_5+28>
   0x0000000000401182 <+92>:    add    $0x10,%rsp
   0x0000000000401186 <+96>:    pop    %rbx
   0x0000000000401187 <+97>:    retq
```

The lines <+13> to <+18> check whether there are 6 inputs or not. If so, jump to <+84>, or else the bomb explodes. Then, we check the last four bits (or last bytes) of the edx and replace it with some other char based on the lines <+32> and <+35>. So, I tried to examine the char using the instruction “x/s 0x4025f0,” where the address comes from the line <+35>. And I got “maduiersnfotvbylSo you …”. Since we are dealing with one byte, I matches each character from 0 to F(15). Then, after looping for all 6 letters, the <strings_not_equal> function is called. Just like phase 1, I checked the arguments the string is checked for which is esi. I used the instruction “x/s 0x4025a7” to check the string stored in the address, which comes out to be “oilers”. Combining the above two results, the answer will be some characters where the last 4 bits will be A, 4, F, 5, 6, 7. I tried J(0x4A) D(0x44) O(0x4F) E(0x45) F(0x46) G(0x47) and it was the bomb is solved.

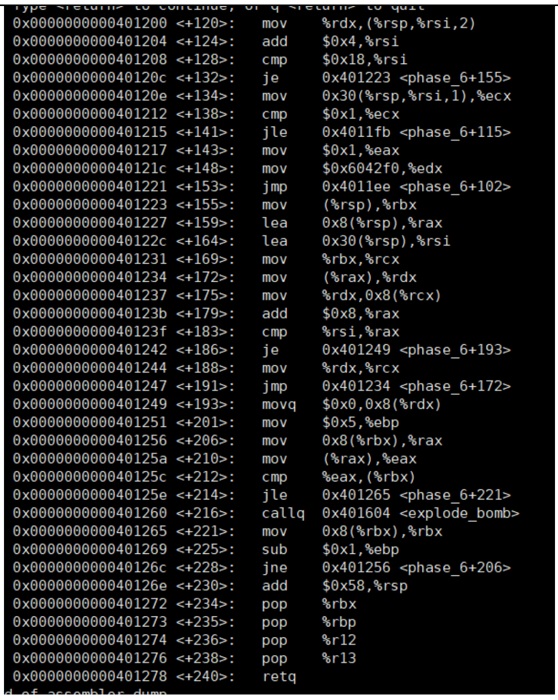


```
So you got that one.  Try this one.
JDOEFG
```

6. Phase 6



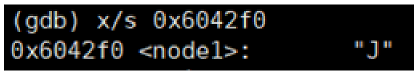
```
0x0000000000401188 <+0>:     push   %r13
0x000000000040118a <+2>:     push   %r12
0x000000000040118c <+4>:     push   %rbp
0x000000000040118d <+5>:     push   %rbx
0x000000000040118e <+6>:     sub    $0x58,%rsp
0x0000000000401192 <+10>:    lea    0x30(%rsp),%rsi
0x0000000000401197 <+15>:    callq  0x40163a <read_six_numbers>
0x000000000040119c <+20>:    lea    0x30(%rsp),%r13
0x00000000004011a1 <+25>:    mov    $0x0,%r12d
0x00000000004011a7 <+31>:    mov    %r13,%rbp
0x00000000004011aa <+34>:    mov    0x0(%r13),%eax
0x00000000004011ae <+38>:    sub    $0x1,%eax
0x00000000004011b1 <+41>:    cmp    $0x5,%eax
0x00000000004011b4 <+44>:    jbe    0x4011bb <phase_6+51>
0x00000000004011b6 <+46>:    callq  0x401604 <explode_bomb>
0x00000000004011bb <+51>:    add    $0x1,%r12d
0x00000000004011bf <+55>:    cmp    $0x6,%r12d
0x00000000004011c3 <+59>:    jne    0x4011cc <phase_6+68>
0x00000000004011c5 <+61>:    mov    $0x0,%esi
0x00000000004011ca <+66>:    jmp    0x40120e <phase_6+134>
0x00000000004011cc <+68>:    mov    %r12d,%ebx
0x00000000004011cf <+71>:    movslq %ebx,%rax
0x00000000004011d2 <+74>:    mov    0x30(%rsp,%rax,4),%eax
0x00000000004011d6 <+78>:    cmp    %eax,0x0(%rbp)
0x00000000004011d9 <+81>:    jne    0x4011e0 <phase_6+88>
0x00000000004011db <+83>:    callq  0x401604 <explode_bomb>
0x00000000004011e0 <+88>:    add    $0x1,%ebx
0x00000000004011e3 <+91>:    cmp    $0x5,%ebx
0x00000000004011e6 <+94>:    jle    0x4011cf <phase_6+71>
0x00000000004011e8 <+96>:    add    $0x4,%r13
0x00000000004011ec <+100>:   jmp    0x4011a7 <phase_6+31>
0x00000000004011ee <+102>:   mov    0x8(%rdx),%rdx
0x00000000004011f2 <+106>:   add    $0x1,%eax
0x00000000004011f5 <+109>:   cmp    %ecx,%eax
0x00000000004011f7 <+111>:   jne    0x4011ee <phase_6+102>
0x00000000004011f9 <+113>:   jmp    0x401200 <phase_6+120>
0x00000000004011fb <+115>:   mov    $0x6042f0,%edx
```



```
0x0000000000401200 <+120>:   mov    %rdx,(%rsp,%rsi,2)
0x0000000000401204 <+124>:   add    $0x4,%rsi
0x0000000000401208 <+128>:   cmp    $0x18,%rsi
0x000000000040120c <+132>:   je     0x401223 <phase_6+155>
0x000000000040120e <+134>:   mov    0x30(%rsp,%rsi,1),%ecx
0x0000000000401212 <+138>:   cmp    $0x1,%ecx
0x0000000000401215 <+141>:   jle    0x4011fb <phase_6+115>
0x0000000000401217 <+143>:   mov    $0x1,%eax
0x000000000040121c <+148>:   mov    $0x6042f0,%edx
0x0000000000401221 <+153>:   jmp    0x4011ee <phase_6+102>
0x0000000000401223 <+155>:   mov    (%rsp),%rbx
0x0000000000401227 <+159>:   lea    0x8(%rsp),%rax
0x000000000040122c <+164>:   lea    0x30(%rsp),%rsi
0x0000000000401231 <+169>:   mov    %rbx,%rcx
0x0000000000401234 <+172>:   mov    (%rax),%rdx
0x0000000000401237 <+175>:   mov    %rdx,0x8(%rcx)
0x000000000040123b <+179>:   add    $0x8,%rax
0x000000000040123f <+183>:   cmp    %rsi,%rax
0x0000000000401242 <+186>:   je     0x401249 <phase_6+193>
0x0000000000401244 <+188>:   mov    %rdx,%rcx
0x0000000000401247 <+191>:   jmp    0x401234 <phase_6+172>
0x0000000000401249 <+193>:   movq   $0x0,0x8(%rdx)
0x0000000000401251 <+201>:   mov    $0x5,%ebp
0x0000000000401256 <+206>:   mov    0x8(%rbx),%rax
0x000000000040125a <+210>:   mov    (%rax),%eax
0x000000000040125c <+212>:   cmp    %eax,(%rbx)
0x000000000040125e <+214>:   jle    0x401265 <phase_6+221>
0x0000000000401260 <+216>:   callq  0x401604 <explode_bomb>
0x0000000000401265 <+221>:   mov    0x8(%rbx),%rbx
0x0000000000401269 <+225>:   sub    $0x1,%ebp
0x000000000040126c <+228>:   jne    0x401256 <phase_6+206>
0x000000000040126e <+230>:   add    $0x58,%rsp
0x0000000000401272 <+234>:   pop    %rbx
0x0000000000401273 <+235>:   pop    %rbp
0x0000000000401274 <+236>:   pop    %r12
0x0000000000401276 <+238>:   pop    %r13
0x0000000000401278 <+240>:   retq
```

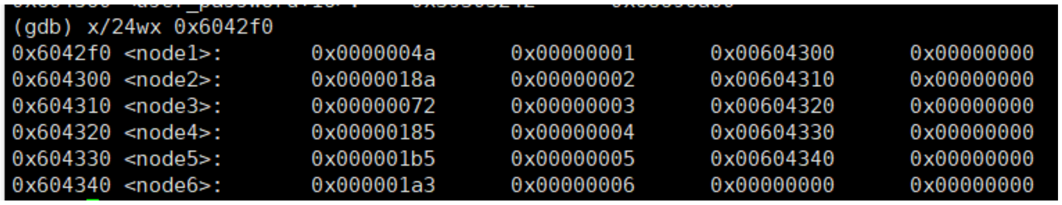
Based on the function shown in line <+15>, we know that the input must be 6 integers. And from line <+38> to <+46>, we can figure out that each 6 number should be less than equal to 6. Then the next few lines check that the first input is different from the third, fourth, …, and sixth input. These steps are repeated for all 6 inputs and as a result, we can figure out that the 6 inputs are all different numbers that are smaller than or equal to 6.

Then, we move to <+134> from <+66>. From there, we loop each 6 inputs and all of the inputs jump to <+115> by the line <+141>. At the line <+115>, edx stores the address of 0x6042f0. I checked the value stored in the address using the instruction “x/s 0x6042f0.”



```
(gdb) x/s 0x6042f0
0x6042f0 <node1>:       "J"
```

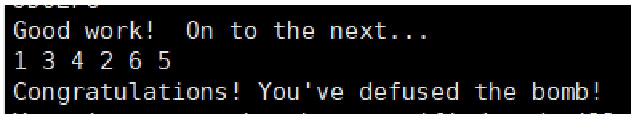
That “node 1” tells that the values are stored in a linked list. Therefore, I tried to check which values are stored in the total linked list by using the instruction “x/24wx 0x6042f0.” This instruction shows 24 bits of information in the form of 4 bytes in hexadecimal form.



```
(gdb) x/24wx 0x6042f0
0x6042f0 <node1>:       0x0000004a      0x00000001      0x00604300      0x00000000
0x604300 <node2>:       0x0000018a      0x00000002      0x00604310      0x00000000
0x604310 <node3>:       0x00000072      0x00000003      0x00604320      0x00000000
0x604320 <node4>:       0x00000185      0x00000004      0x00604330      0x00000000
0x604330 <node5>:       0x000001b5      0x00000005      0x00604340      0x00000000
0x604340 <node6>:       0x000001a3      0x00000006      0x00000000      0x00000000
```

The first four bytes are what we need to determine the order of the input, the second four bytes are the order of the nodes, and the last 4 bytes are the address of the next nodes (pointer).

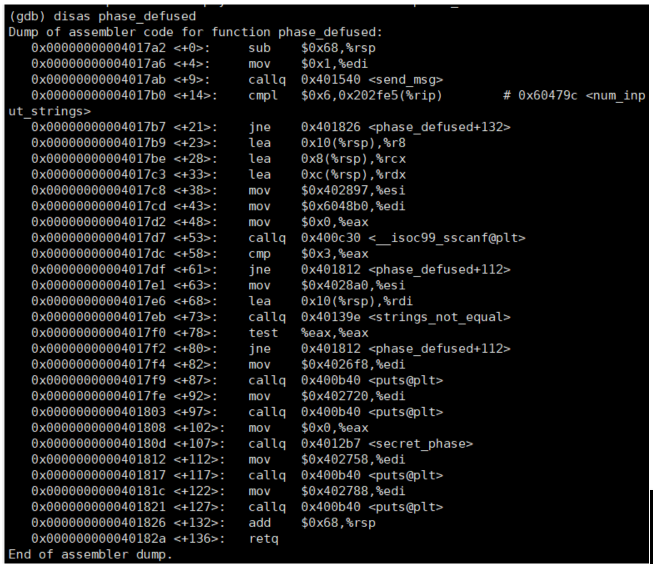
Looking at the end of the function, there is another loop from <+206> to <+228>, and the ebp must equal to 1 to end the loop. Also, eax must be less than or equal to rbx to avoid the bomb. When looking at the line <+206>, rax has a higher address, meaning that the rbx will be a former input and rax will be the latter input. Therefore, I should put the inputs in ascending order. From the above node values, when we list them in ascending order, the answer will be “1 3 4 2 6 5.”



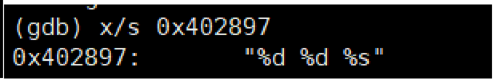
```
Good work!  On to the next...
1 3 4 2 6 5
Congratulations! You've defused the bomb!
```

7. Phase 7 (Secret Phase)

If you try “(gdb) disas phase_defused,” you’ll be able to see a path to the secret phase.

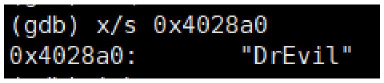


```
(gdb) disas phase_defused
Dump of assembler code for function phase_defused:
   0x00000000004017a2 <+0>:     sub    $0x68,%rsp
   0x00000000004017a6 <+4>:     mov    $0x1,%edi
   0x00000000004017ab <+9>:     callq  0x401540 <send_msg>
   0x00000000004017b0 <+14>:    cmpl   $0x6,0x202fe5(%rip)        # 0x60479c <num_inp
ut_strings>
   0x00000000004017b7 <+21>:    jne    0x401826 <phase_defused+132>
   0x00000000004017b9 <+23>:    lea    0x10(%rsp),%r8
   0x00000000004017be <+28>:    lea    0x8(%rsp),%rcx
   0x00000000004017c3 <+33>:    lea    0xc(%rsp),%rdx
   0x00000000004017c8 <+38>:    mov    $0x402897,%esi
   0x00000000004017cd <+43>:    mov    $0x6048b0,%edi
   0x00000000004017d2 <+48>:    mov    $0x0,%eax
   0x00000000004017d7 <+53>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x00000000004017dc <+58>:    cmp    $0x3,%eax
   0x00000000004017df <+61>:    jne    0x401812 <phase_defused+112>
   0x00000000004017e1 <+63>:    mov    $0x4028a0,%esi
   0x00000000004017e6 <+68>:    lea    0x10(%rsp),%rdi
   0x00000000004017eb <+73>:    callq  0x40139e <strings_not_equal>
   0x00000000004017f0 <+78>:    test   %eax,%eax
   0x00000000004017f2 <+80>:    jne    0x401812 <phase_defused+112>
   0x00000000004017f4 <+82>:    mov    $0x4026f8,%edi
   0x00000000004017f9 <+87>:    callq  0x400b40 <puts@plt>
   0x00000000004017fe <+92>:    mov    $0x402720,%edi
   0x0000000000401803 <+97>:    callq  0x400b40 <puts@plt>
   0x0000000000401808 <+102>:   mov    $0x0,%eax
   0x000000000040180d <+107>:   callq  0x4012b7 <secret_phase>
   0x0000000000401812 <+112>:   mov    $0x402758,%edi
   0x0000000000401817 <+117>:   callq  0x400b40 <puts@plt>
   0x000000000040181c <+122>:   mov    $0x402788,%edi
   0x0000000000401821 <+127>:   callq  0x400b40 <puts@plt>
   0x0000000000401826 <+132>:   add    $0x68,%rsp
   0x000000000040182a <+136>:   retq
End of assembler dump.
```



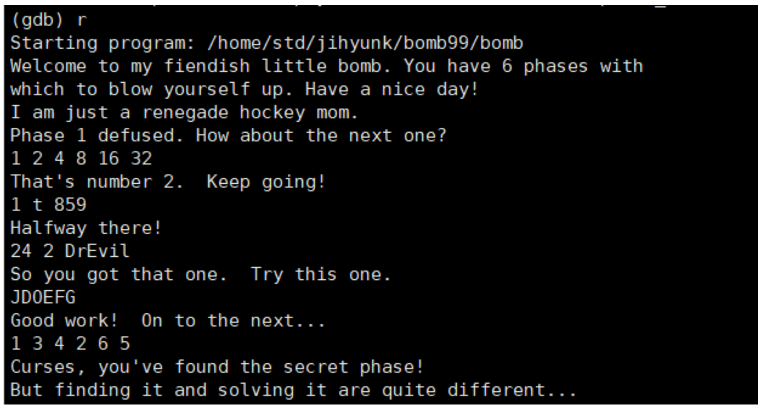
```
(gdb) x/s 0x402897
0x402897:       "%d %d %s"
```

On line <+38>, esi register is passed to scanf function. Therefore, when you check the value stored in 0x402897 by the instruction x/s, “%d %d %s” is stored. From the above phase 1 to 6, there is only one phase which has two integers as an answer: phase 4. So far, we know that if the user types a string after the answer of phase 4, the secret phase will open. Then, on line <+73>, the “strings not equal” function is called. So I checked esi value using x/s. “DrEvil” was stored in the address 0x4028a0.

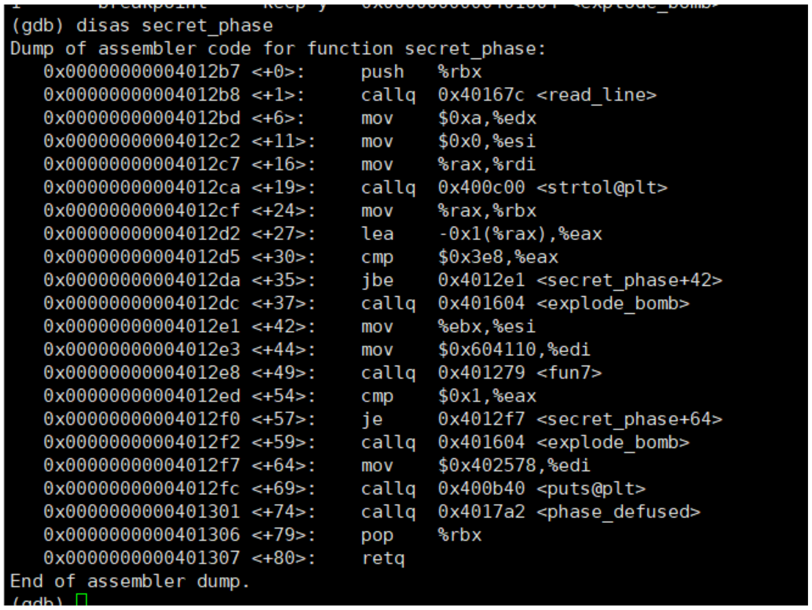


```
(gdb) x/s 0x4028a0
0x4028a0:       "DrEvil"
```

If you put in other keys for each phase as well as “24 2 DrEvil” for phase 4, the secret phase will be open and you’ll be able to disassemble the secret phase.

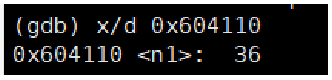


```
(gdb) r
Starting program: /home/std/jihyunk/bomb99/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am just a renegade hockey mom.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
1 t 859
Halfway there!
24 2 DrEvil
So you got that one.  Try this one.
JDOEFG
Good work!  On to the next...
1 3 4 2 6 5
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```



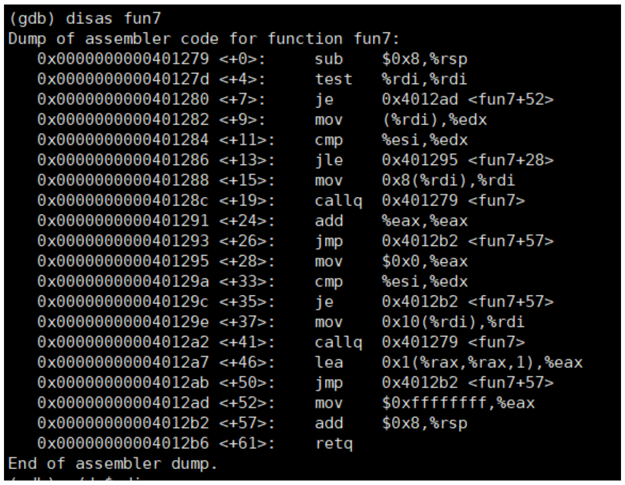
```
(gdb) disas secret_phase
Dump of assembler code for function secret_phase:
   0x00000000004012b7 <+0>:     push   %rbx
   0x00000000004012b8 <+1>:     callq  0x40167c <read_line>
   0x00000000004012bd <+6>:     mov    $0xa,%edx
   0x00000000004012c2 <+11>:    mov    $0x0,%esi
   0x00000000004012c7 <+16>:    mov    %rax,%rdi
   0x00000000004012ca <+19>:    callq  0x400c00 <strtol@plt>
   0x00000000004012cf <+24>:    mov    %rax,%rbx
   0x00000000004012d2 <+27>:    lea    -0x1(%rax),%eax
   0x00000000004012d5 <+30>:    cmp    $0x3e8,%eax
   0x00000000004012da <+35>:    jbe    0x4012e1 <secret_phase+42>
   0x00000000004012dc <+37>:    callq  0x401604 <explode_bomb>
   0x00000000004012e1 <+42>:    mov    %ebx,%esi
   0x00000000004012e3 <+44>:    mov    $0x604110,%edi
   0x00000000004012e8 <+49>:    callq  0x401279 <fun7>
   0x00000000004012ed <+54>:    cmp    $0x1,%eax
   0x00000000004012f0 <+57>:    je     0x4012f7 <secret_phase+64>
   0x00000000004012f2 <+59>:    callq  0x401604 <explode_bomb>
   0x00000000004012f7 <+64>:    mov    $0x402578,%edi
   0x00000000004012fc <+69>:    callq  0x400b40 <puts@plt>
   0x0000000000401301 <+74>:    callq  0x4017a2 <phase_defused>
   0x0000000000401306 <+79>:    pop    %rbx
   0x0000000000401307 <+80>:    retq
End of assembler dump.
```

Near where fun7 is called, we know that the input values are esi which stores the input key, and edi which directs to the address 0x604110. Then, the return value of fun7 should equal 1 to avoid the bomb.



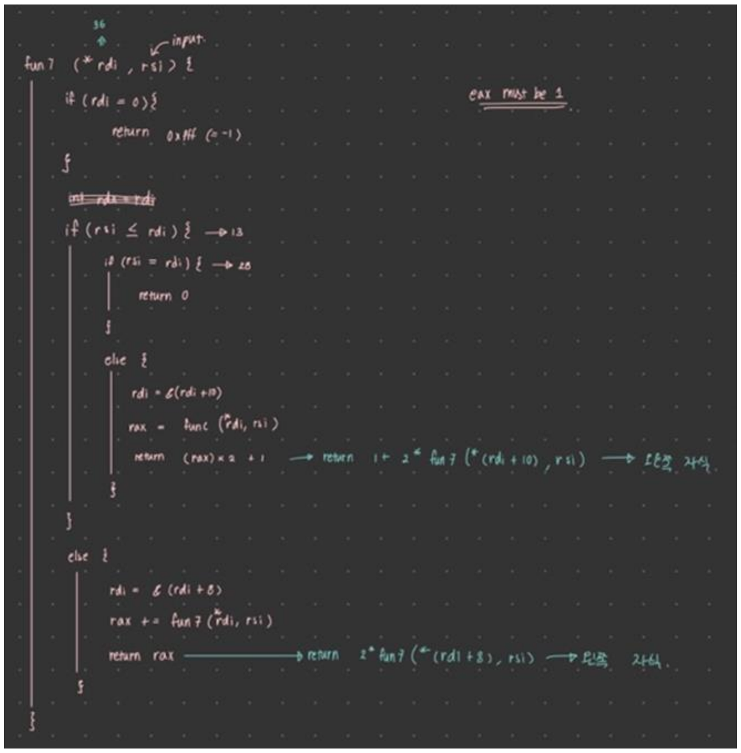
```
(gdb) x/d 0x604110
0x604110 <n1>:  36
```

A decimal value of 36 is stored in edi (or 0x604110) which will be the value stored in the root node. Explanation about the node will be on the next page with explanation of fun 7.



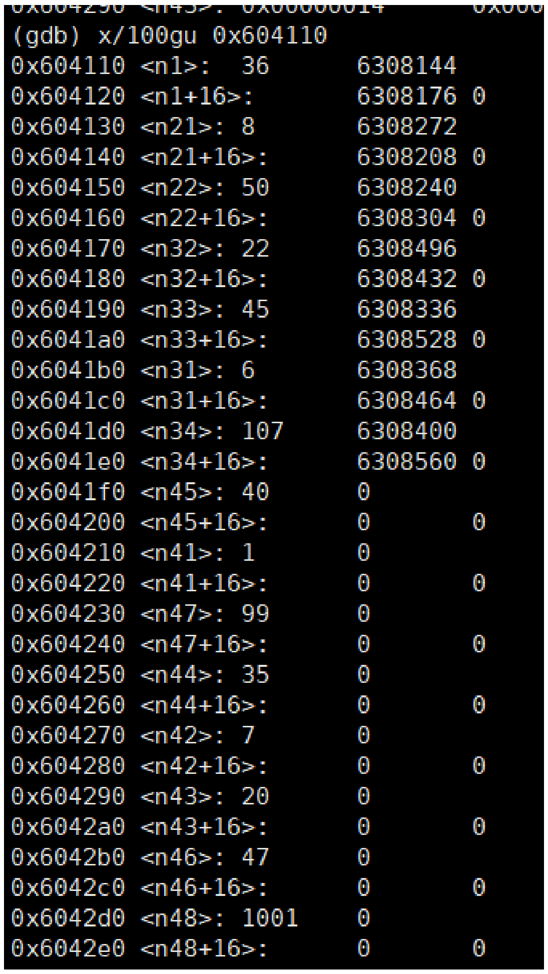
```
(gdb) disas fun7
Dump of assembler code for function fun7:
   0x0000000000401279 <+0>:     sub    $0x8,%rsp
   0x000000000040127d <+4>:     test   %rdi,%rdi
   0x0000000000401280 <+7>:     je     0x4012ad <fun7+52>
   0x0000000000401282 <+9>:     mov    (%rdi),%edx
   0x0000000000401284 <+11>:    cmp    %esi,%edx
   0x0000000000401286 <+13>:    jle    0x401295 <fun7+28>
   0x0000000000401288 <+15>:    mov    0x8(%rdi),%rdi
   0x000000000040128c <+19>:    callq  0x401279 <fun7>
   0x0000000000401291 <+24>:    add    %eax,%eax
   0x0000000000401293 <+26>:    jmp    0x4012b2 <fun7+57>
   0x0000000000401295 <+28>:    mov    $0x0,%eax
   0x000000000040129a <+33>:    cmp    %esi,%edx
   0x000000000040129c <+35>:    je     0x4012b2 <fun7+57>
   0x000000000040129e <+37>:    mov    0x10(%rdi),%rdi
   0x00000000004012a2 <+41>:    callq  0x401279 <fun7>
   0x00000000004012a7 <+46>:    lea    0x1(%rax,%rax,1),%eax
   0x00000000004012ab <+50>:    jmp    0x4012b2 <fun7+57>
   0x00000000004012ad <+52>:    mov    $0xffffffff,%eax
   0x00000000004012b2 <+57>:    add    $0x8,%rsp
   0x00000000004012b6 <+61>:    retq
End of assembler dump.
```

Looking at the disassembled code of fun7, we know that this function forms a path to the binary tree.



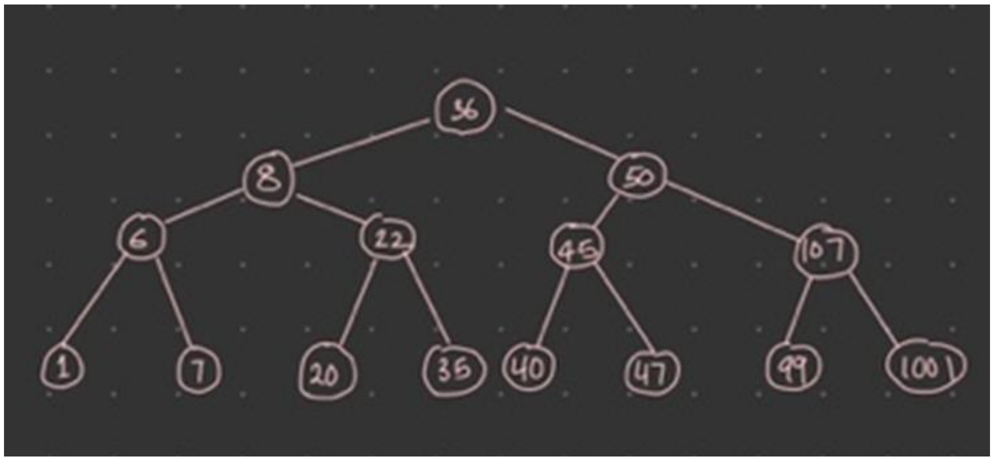
```
          36
          ↑     ↙ input
fun7  (*rdi , rsi) {                                        eax must be 1
    if (rdi = 0){
        return 0xfff (= -1)
    }
    ~~int rdx = rdi~~
    if (rsi ≤ rdi) {  → 13
        if (rsi = rdi) {  → 28
            return 0
        }
        else {
            rdi = &(rdi +10)
            rax = func (*rdi, rsi)
            return (rax) x 2 + 1   → return 1+ 2* fun7 (*(rdi + 10), rsi)  → 오른쪽 자식.
        }
    }
    else {
        rdi = & (rdi + 8)
        rax += fun7 (*rdi, rsi)
        return rax  ——————→ return 2* fun7 (*(rdi + 8), rsi)  → 왼쪽 자식.
    }
}
```

To check the value of the rest of the nodes, I used the instruction "x/100gu" to see the decimal values. As a result, I got the following node values and the following picture depicts the binary tree.



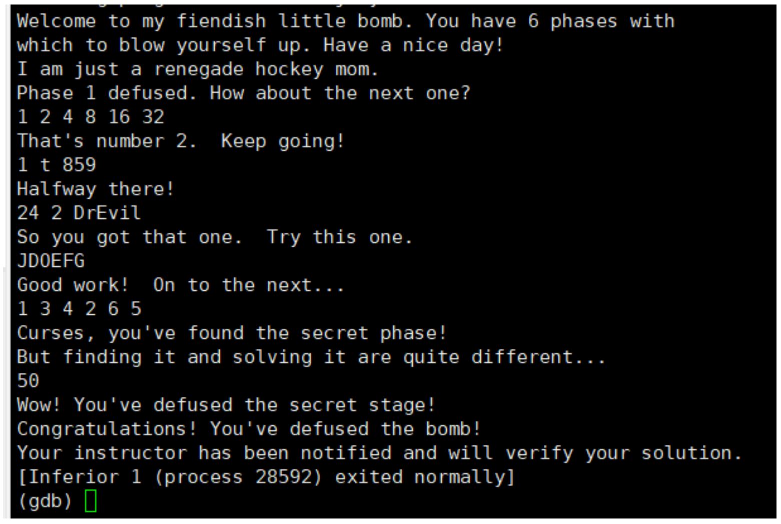
```
0x604290 <n43>: 0x00000014      0x000
(gdb) x/100gu 0x604110
0x604110 <n1>:  36      6308144
0x604120 <n1+16>:       6308176 0
0x604130 <n21>: 8       6308272
0x604140 <n21+16>:      6308208 0
0x604150 <n22>: 50      6308240
0x604160 <n22+16>:      6308304 0
0x604170 <n32>: 22      6308496
0x604180 <n32+16>:      6308432 0
0x604190 <n33>: 45      6308336
0x6041a0 <n33+16>:      6308528 0
0x6041b0 <n31>: 6       6308368
0x6041c0 <n31+16>:      6308464 0
0x6041d0 <n34>: 107     6308400
0x6041e0 <n34+16>:      6308560 0
0x6041f0 <n45>: 40      0
0x604200 <n45+16>:      0       0
0x604210 <n41>: 1       0
0x604220 <n41+16>:      0       0
0x604230 <n47>: 99      0
0x604240 <n47+16>:      0       0
0x604250 <n44>: 35      0
0x604260 <n44+16>:      0       0
0x604270 <n42>: 7       0
0x604280 <n42+16>:      0       0
0x604290 <n43>: 20      0
0x6042a0 <n43+16>:      0       0
0x6042b0 <n46>: 47      0
0x6042c0 <n46+16>:      0       0
0x6042d0 <n48>: 1001    0
0x6042e0 <n48+16>:      0       0
```

N1	36
N21	8
N22	50
N31	6
N32	22
N33	45
N34	107
N41	1
N42	7
N43	20
N44	35
N45	40
N46	47
N47	99
N48	1001



Since we need the return value to be 1, when we insert 50, fun7(50) = 2 * fun7(36) + 1 = 2 * (0) + 1 = 1. Therefore 50 will be the final answer for the secret phase.

The final answer will be as follows:



```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am just a renegade hockey mom.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
1 t 859
Halfway there!
24 2 DrEvil
So you got that one.  Try this one.
JDOEFG
Good work!  On to the next...
1 3 4 2 6 5
Curses, you've found the secret phase!
But finding it and solving it are quite different...
50
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
[Inferior 1 (process 28592) exited normally]
(gdb) 
```