CSED211 LAB3 REPORT

20220778 표승현

Phase_1

phase_1 의 코드를 disassemble 해본다.

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x0000000000400ef0 <+0>:     sub    $0x8,%rsp
   0x0000000000400ef4 <+4>:     mov    $0x4024e0,%esi
   0x0000000000400ef9 <+9>:     callq  0x40132e <strings_not_equal>
   0x0000000000400efe <+14>:    test   %eax,%eax
   0x0000000000400f00 <+16>:    je     0x400f07 <phase_1+23>
   0x0000000000400f02 <+18>:    callq  0x401594 <explode_bomb>
   0x0000000000400f07 <+23>:    add    $0x8,%rsp
   0x0000000000400f0b <+27>:    retq
End of assembler dump.
```

strings_not_equal 함수를 보아 어떠한 문자열을 입력 받고 이를 비교하여 같아야 explode_bomb 의 호출을 피할 수 있는 것으로 보인다.

<+4>에서 esi 에 어떠한 메모리 값이 저장되고 strings_not_equal 은 이를 인자로 받는 것으로 보인다. 따라서 esi 에 저장되는 메모리 값을 조사해본다.

```
(gdb) x/s 0x4024e0
0x4024e0:       "There are rumors on the internets."
```

There are rumors on the internets.라는 문자열을 얻을 수 있다. 이를 입력해보면

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
There are rumors on the internets.
Phase 1 defused. How about the next one?
```

phase_1 이 해체된다.

Phase_2

phase_2 를 disassemble 해본다.

```
(gdb) disas 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  0x4015ca <read_six_numbers>
   0x0000000000400f1a <+14>:    cmpl   $0x1,(%rsp)
   0x0000000000400f1e <+18>:    je     0x400f40 <phase_2+52>
   0x0000000000400f20 <+20>:    callq  0x401594 <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  0x401594 <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
End of assembler dump.
```

<+9>의 read_six_numbers 함수를 보아 6 개의 숫자를 입력 받는 것으로 보인다. 바로 다음 줄인 <+14>에서 rsp 에 저장된 값을 1 과 비교하는데 rsp 에 무슨 정보가 저장되어 있는지 조사해본다.

```
(gdb) info register
rax            0x6      6
rbx            0x0      0
rcx            0x7fffffffe410   140737488348176
rdx            0x0      0
rsi            0x0      0
rdi            0x7fffffffde00   140737488346624
rbp            0x0      0x0
rsp            0x7fffffffe420   0x7fffffffe420
r8             0x7ffff7dd5060   140737351864416
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x400cd0 4197584
r13            0x7fffffffe540   140737488348480
r14            0x0      0
r15            0x0      0
rip            0x400f1a 0x400f1a <phase_2+14>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

```
(gdb) x/12wx 0x7fffffffe420
0x7fffffffe420: 0x00000001      0x00000002      0x00000003      0x00000004
0x7fffffffe430: 0x00000005      0x00000006      0x004014c7      0x00000000
0x7fffffffe440: 0x00000000      0x00000000      0x00000000      0x00000000
```

임의의 6 개 숫자 1 2 3 4 5 6 을 입력했을 때 이들이 차례대로 저장되어 있는 것을 확인할 수 있다.

```
   0x0000000000400f40 <+52>:    lea    0x4(%rsp),%rbx
   0x0000000000400f45 <+57>:    lea    0x18(%rsp),%rbp
   0x0000000000400f4a <+62>:    jmp    0x400f27 <phase_2+27>
```

이후 <+52>로 jump 하게 되는데 rbx 에 rsp+4 의 주소가 저장되고 rbp 에는 rsp+24 의 주소가 저장된다.

```
   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  0x401594 <explode_bomb>
   0x0000000000400f35 <+41>:    add    $0x4,%rbx
   0x0000000000400f39 <+45>:    cmp    %rbp,%rbx
   0x0000000000400f3c <+48>:    jne    0x400f27 <phase_2+27>
```

<+27>로 이동하면 loop 에 진입하게 되는데 rbx(rsp+n 의 주소)의 4 바이트 전의 값을 eax 에 저장한 후, eax 에 이를 *2 한 값을 저장한다. 이를 rbx 의 값과 비교한 후 같지 않으면 explode_bomb 을 호출한다.

같으면 <+41>로 이동하여 rbx 포인터를 4 바이트 전진시키고 rbx 가 rbp 즉, rsp+24 에 도달할 때까지 loop 를 반복한다. loop 는 총 24/4=6 번 반복될 것이다.

해당 코드를 분석해보자면 rsp 에는 입력된 6 개의 숫자가 저장된다. 첫번째 숫자는 1 이어야 하는데 이에 2 를 계속해서 곱해주면서 나머지 5 개의 입력 숫자와 비교를 한다. 결국 2 의 등비수열이 입력되어야 explode_bomb 의 호출을 피할 수 있다. 이에 따라 1 2 4 8 16 32 를 입력한다.

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

Breakpoint 7, 0x0000000000400f0c in phase_2 ()
(gdb) next
Single stepping until exit from function phase_2,
which has no line number information.
main (argc=<optimized out>, argv=<optimized out>) at bomb.c:83
83          phase_defused();
```

phase_2 가 해체된다.

Phase 3

phase_3 코드를 disassemble 해본다.

```
Dump of assembler code for function phase_3:
=> 0x0000000000400f53 <+0>:     sub    $0x18,%rsp
   0x0000000000400f57 <+4>:     lea    0x8(%rsp),%rcx
   0x0000000000400f5c <+9>:     lea    0xc(%rsp),%rdx
   0x0000000000400f61 <+14>:    mov    $0x4027dd,%esi
   0x0000000000400f66 <+19>:    mov    $0x0,%eax
   0x0000000000400f6b <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x0000000000400f70 <+29>:    cmp    $0x1,%eax
   0x0000000000400f73 <+32>:    jg     0x400f7a <phase_3+39>
   0x0000000000400f75 <+34>:    callq  0x401594 <explode_bomb>
   0x0000000000400f7a <+39>:    cmpl   $0x7,0xc(%rsp)
   0x0000000000400f7f <+44>:    ja     0x400fe7 <phase_3+148>
   0x0000000000400f81 <+46>:    mov    0xc(%rsp),%eax
   0x0000000000400f85 <+50>:    jmpq   *0x402540(,%rax,8)
   0x0000000000400f8c <+57>:    mov    $0x0,%eax
   0x0000000000400f91 <+62>:    jmp    0x400f98 <phase_3+69>
   0x0000000000400f93 <+64>:    mov    $0x398,%eax
   0x0000000000400f98 <+69>:    sub    $0x25b,%eax
   0x0000000000400f9d <+74>:    jmp    0x400fa4 <phase_3+81>
   0x0000000000400f9f <+76>:    mov    $0x0,%eax
   0x0000000000400fa4 <+81>:    add    $0x2a1,%eax
   0x0000000000400fa9 <+86>:    jmp    0x400fb0 <phase_3+93>
   0x0000000000400fab <+88>:    mov    $0x0,%eax
   0x0000000000400fb0 <+93>:    sub    $0x2ad,%eax
   0x0000000000400fb5 <+98>:    jmp    0x400fbc <phase_3+105>
   0x0000000000400fb7 <+100>:   mov    $0x0,%eax
   0x0000000000400fbc <+105>:   add    $0x2ad,%eax
   0x0000000000400fc1 <+110>:   jmp    0x400fc8 <phase_3+117>
   0x0000000000400fc3 <+112>:   mov    $0x0,%eax
   0x0000000000400fc8 <+117>:   sub    $0x2ad,%eax
   0x0000000000400fcd <+122>:   jmp    0x400fd4 <phase_3+129>
   0x0000000000400fcf <+124>:   mov    $0x0,%eax
   0x0000000000400fd4 <+129>:   add    $0x2ad,%eax
   0x0000000000400fd9 <+134>:   jmp    0x400fe0 <phase_3+141>
   0x0000000000400fdb <+136>:   mov    $0x0,%eax
   0x0000000000400fe0 <+141>:   sub    $0x2ad,%eax
   0x0000000000400fe5 <+146>:   jmp    0x400ff1 <phase_3+158>
   0x0000000000400fe7 <+148>:   callq  0x401594 <explode_bomb>
   0x0000000000400fec <+153>:   mov    $0x0,%eax
   0x0000000000400ff1 <+158>:   cmpl   $0x5,0xc(%rsp)
   0x0000000000400ff6 <+163>:   jg     0x400ffe <phase_3+171>
   0x0000000000400ff8 <+165>:   cmp    0x8(%rsp),%eax
   0x0000000000400ffc <+169>:   je     0x401003 <phase_3+176>
   0x0000000000400ffe <+171>:   callq  0x401594 <explode_bomb>
   0x0000000000401003 <+176>:   add    $0x18,%rsp
   0x0000000000401007 <+180>:   retq
```

<+24> sscan 함수 호출 이후 eax가 1보다 커야 explode_bomb의 호출을 피할 수 있다.

```
(gdb) x/s 0x4027dd
0x4027dd:       "%d %d"
```

입력 형식을 보니 역시 두 개의 인자 즉 1보다 많은 인자를 입력 받는 것을 알 수 있다.

```
(gdb) r text.txt
Starting program: /home/std/hyeony312/datalab/BombLab/bomb82/bomb text.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
3 90
```

임의의 두 개의 수 3, 90을 넣은 후 레지스터 값을 조사해본다.

```
(gdb) x/6u $rsp
0x7fffffffe430: 4294960440      32767   90      3
0x7fffffffe440: 4294960432      32767
```

앞선 코드에 따라 rsp+8과 rsp+12에 각각 90과 3이 들어가 있는 것을 확인할 수 있다. 즉, rsp+8과 rsp+12에는 각각 두번째 입력 숫자, 첫번째 입력 숫자가 차례대로 저장된다.

```
   0x0000000000400f6b <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x0000000000400f70 <+29>:    cmp    $0x1,%eax
   0x0000000000400f73 <+32>:    jg     0x400f7a <phase_3+39>
   0x0000000000400f75 <+34>:    callq  0x401594 <explode_bomb>
   0x0000000000400f7a <+39>:    cmpl   $0x7,0xc(%rsp)
   0x0000000000400f7f <+44>:    ja     0x400fe7 <phase_3+148>
   0x0000000000400f81 <+46>:    mov    0xc(%rsp),%eax
   0x0000000000400f85 <+50>:    jmpq   *0x402540(,%rax,8)
```

<+39>에서 rsp+12(첫번째 입력 숫자)가 7 이하이면 jump를 건너뛰어 그 다음 jump로 이동한다. 7을 초과하게 되면 explode_bomb이 호출된다.

<+50>에서 0x402540으로부터 첫번째 입력한 값*8 byte만큼 떨어진 주소로 이동한다.

```
(gdb) x/1wx 0x402540
0x402540:       0x00400f93
```

<+50>에서 0x402540 + rax*8로 이동하므로,

0x402540 + 3*8 = 0x402558을 조사한다.

```
(gdb) x/1wx 0x402558
0x402558:       0x00400fab
```

```
(gdb) nexti
0x0000000000400fab in phase_3 ()
```

조사해서 얻은 주소로 jump하는 것을 확인할 수 있다.

```
0x0000000000400fab <+88>:    mov    $0x0,%eax
0x0000000000400fb0 <+93>:    sub    $0x2ad,%eax
0x0000000000400fb5 <+98>:    jmp    0x400fbc <phase_3+105>
0x0000000000400fb7 <+100>:   mov    $0x0,%eax
0x0000000000400fbc <+105>:   add    $0x2ad,%eax
0x0000000000400fc1 <+110>:   jmp    0x400fc8 <phase_3+117>
0x0000000000400fc3 <+112>:   mov    $0x0,%eax
0x0000000000400fc8 <+117>:   sub    $0x2ad,%eax
0x0000000000400fcd <+122>:   jmp    0x400fd4 <phase_3+129>
0x0000000000400fcf <+124>:   mov    $0x0,%eax
0x0000000000400fd4 <+129>:   add    $0x2ad,%eax
0x0000000000400fd9 <+134>:   jmp    0x400fe0 <phase_3+141>
0x0000000000400fdb <+136>:   mov    $0x0,%eax
0x0000000000400fe0 <+141>:   sub    $0x2ad,%eax
0x0000000000400fe5 <+146>:   jmp    0x400ff1 <phase_3+158>
0x0000000000400fe7 <+148>:   callq  0x401594 <explode_bomb>
0x0000000000400fec <+153>:   mov    $0x0,%eax
0x0000000000400ff1 <+158>:   cmpl   $0x5,0xc(%rsp)
0x0000000000400ff6 <+163>:   jg     0x400ffe <phase_3+171>
0x0000000000400ff8 <+165>:   cmp    0x8(%rsp),%eax
0x0000000000400ffc <+169>:   je     0x401003 <phase_3+176>
0x0000000000400ffe <+171>:   callq  0x401594 <explode_bomb>
0x0000000000401003 <+176>:   add    $0x18,%rsp
0x0000000000401007 <+180>:   retq
```

이후 코드의 흐름을 살펴보면

ex) eax=0-675=-675

- ➔ <+105>로 이동: eax=eax+675 = 0
- ➔ <+117>로 이동: eax=0-675=-675
- ➔ ...
- ➔ <+158>로 이동 >> eax는 입력값에 따라 -675 or 0 or 675의 값을 가질 수 있다.

eax는 675라는 값이 더해지거나 빼지는 업데이트로 입력한 숫자에 따라 변환된 값을 가진 채 <+158>에 진입한다. <+158>에서는 rsp+12의 값과 5를 비교하는데 첫번째 입력값이 5보다 작아야 함을 알 수 있다.

첫번째 숫자를 4로 한 임의의 숫자를 입력하였을 때 레지스터 값을 조사해본다. <+158>에 중단점을 설정했을 때

```
Breakpoint 5, 0x0000000000400ff1 in phase_3 ()
(gdb) info register
rax            0x0      0
```

eax에는 0이라는 값이 저장되어 있는 것을 볼 수 있다.

이후 <+165>에서 rsp+8과 eax를 비교하는데, 이 둘의 값이 같아야 explode_bomb의 호출을 피할 수 있다. 따라서 첫번째 수가 4일 때, eax는 0이므로 두번째 입력값도 0이어야 한다.

4 0을 입력해보면

```
That's number 2.  Keep going!
4 0
```

```
Single stepping until exit from function phase_3,
which has no line number information.
main (argc=<optimized out>, argv=<optimized out>) at bomb.c:90
90          phase_defused();
```

phase_3가 해체된다.

Phase_4

```
(gdb) disas
Dump of assembler code for function phase_4:
=> 0x000000000040103b <+0>:     sub    $0x18,%rsp
   0x000000000040103f <+4>:     lea    0x8(%rsp),%rcx
   0x0000000000401044 <+9>:     lea    0xc(%rsp),%rdx
   0x0000000000401049 <+14>:    mov    $0x4027dd,%esi
   0x000000000040104e <+19>:    mov    $0x0,%eax
   0x0000000000401053 <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x0000000000401058 <+29>:    cmp    $0x2,%eax
   0x000000000040105b <+32>:    jne    0x401064 <phase_4+41>
   0x000000000040105d <+34>:    cmpl   $0xe,0xc(%rsp)
   0x0000000000401062 <+39>:    jbe    0x401069 <phase_4+46>
   0x0000000000401064 <+41>:    callq  0x401594 <explode_bomb>
   0x0000000000401069 <+46>:    mov    $0xe,%edx
   0x000000000040106e <+51>:    mov    $0x0,%esi
   0x0000000000401073 <+56>:    mov    0xc(%rsp),%edi
   0x0000000000401077 <+60>:    callq  0x401008 <func4>
   0x000000000040107c <+65>:    cmp    $0x7,%eax
   0x000000000040107f <+68>:    jne    0x401088 <phase_4+77>
   0x0000000000401081 <+70>:    cmpl   $0x7,0x8(%rsp)
   0x0000000000401086 <+75>:    je     0x40108d <phase_4+82>
   0x0000000000401088 <+77>:    callq  0x401594 <explode_bomb>
   0x000000000040108d <+82>:    add    $0x18,%rsp
   0x0000000000401091 <+86>:    retq
End of assembler dump.
```

input의 형식을 조사한다.

```
(gdb) x/s 0x4027dd
0x4027dd:       "%d %d"
```

코드를 살펴보면 <+29> 2개의 인자를 받아야 explode_bomb로 향하는 jump를 피할 수 있다.

이후 explode_bomb이 호출되는 세 가지 분기점이 있는데

```
 0x000000000040105d <+34>:    cmpl   $0xe,0xc(%rsp)
 0x0000000000401062 <+39>:    jbe    0x401069 <phase_4+46>
 0x0000000000401064 <+41>:    callq  0x401594 <explode_bomb>
```

```
0x0000000000401077 <+60>:    callq  0x401008 <func4>
0x000000000040107c <+65>:    cmp    $0x7,%eax
0x000000000040107f <+68>:    jne    0x401088 <phase_4+77>
0x0000000000401081 <+70>:    cmpl   $0x7,0x8(%rsp)
0x0000000000401086 <+75>:    je     0x40108d <phase_4+82>
0x0000000000401088 <+77>:    callq  0x401594 <explode_bomb>
```

첫번째는 rsp+12에 저장된 값이 14보다 작은 경우이고, 두번째는 fun4의 결과 eax가 7이 아닐 때,

마지막은 rsp+8에 저장된 값이 7이 아닐 때이다.

rsp+12와 rsp+8이 각각 rdx, rcx에 로드되므로 임의의 두 숫자를 입력한 후 이들에 어떤 값들이 저장되는지 조사해본다.

```
Halfway there!
5 9
```
임의의 숫자 5 9를 입력한다.

```
rcx            0x7fffffffe448   140737488348232
rdx            0x7fffffffe44c   140737488348236
```

```
(gdb) x/d 0x7fffffffe448
0x7fffffffe448: 9
(gdb) x/d 0x7fffffffe44c
0x7fffffffe44c: 5
```

rcx=rsp+8 에는 9, 두번째 입력 숫자가 저장되어 있고, rdx=rsp+12 에는 첫번째 입력 숫자인 5가 저장되어 있다. 따라서 위 explode_bomb을 호출하는 분기점의 조건을 고려할 때, 첫번째 숫자는 14보다 작아야 하고, 두번째 숫자는 7이어야 한다는 것을 알 수 있다. 마지막으로 고려 해야할 분기점은 func4 호출 이후의 것인데 이를 알아보기 위해 func4을 disassemble 해본다.

```
(gdb) disas func4
Dump of assembler code for function func4:
   0x0000000000401008 <+0>:     push   %rbx
   0x0000000000401009 <+1>:     mov    %edx,%eax
   0x000000000040100b <+3>:     sub    %esi,%eax
   0x000000000040100d <+5>:     mov    %eax,%ebx
   0x000000000040100f <+7>:     shr    $0x1f,%ebx
   0x0000000000401012 <+10>:    add    %ebx,%eax
   0x0000000000401014 <+12>:    sar    %eax
   0x0000000000401016 <+14>:    lea    (%rax,%rsi,1),%ebx
   0x0000000000401019 <+17>:    cmp    %edi,%ebx
   0x000000000040101b <+19>:    jle    0x401029 <func4+33>
   0x000000000040101d <+21>:    lea    -0x1(%rbx),%edx
   0x0000000000401020 <+24>:    callq  0x401008 <func4>
   0x0000000000401025 <+29>:    add    %ebx,%eax
   0x0000000000401027 <+31>:    jmp    0x401039 <func4+49>
   0x0000000000401029 <+33>:    mov    %ebx,%eax
   0x000000000040102b <+35>:    cmp    %edi,%ebx
   0x000000000040102d <+37>:    jge    0x401039 <func4+49>
   0x000000000040102f <+39>:    lea    0x1(%rbx),%esi
   0x0000000000401032 <+42>:    callq  0x401008 <func4>
   0x0000000000401037 <+47>:    add    %ebx,%eax
   0x0000000000401039 <+49>:    pop    %rbx
   0x000000000040103a <+50>:    retq
End of assembler dump.
```

edx=14, esi=0, edi=두번째 입력 숫자 를 인자로 받는 재귀함수임을 알 수 있다.

```
   0x0000000000401008 <+0>:     push   %rbx
   0x0000000000401009 <+1>:     mov    %edx,%eax
   0x000000000040100b <+3>:     sub    %esi,%eax
   0x000000000040100d <+5>:     mov    %eax,%ebx
   0x000000000040100f <+7>:     shr    $0x1f,%ebx
   0x0000000000401012 <+10>:    add    %ebx,%eax
   0x0000000000401014 <+12>:    sar    %eax
```

에서 eax에는 ((14-0)+0)/2 = 7의 값을 갖게 된다는 사실을 파악할 수 있다.

```
   0x0000000000401016 <+14>:    lea    (%rax,%rsi,1),%ebx
   0x0000000000401019 <+17>:    cmp    %edi,%ebx
   0x000000000040101b <+19>:    jle    0x401029 <func4+33>
   0x000000000040101d <+21>:    lea    -0x1(%rbx),%edx
   0x0000000000401020 <+24>:    callq  0x401008 <func4>
```

에서 ebx = 7+0*1 = 7이며 <+17>에서 edi와 ebx를 비교한다. 즉 두번째 입력 숫자가 7보다 크거나 같으면 분기되어 재귀함수 호출을 피한다.

```
   0x0000000000401029 <+33>:    mov    %ebx,%eax
   0x000000000040102b <+35>:    cmp    %edi,%ebx
   0x000000000040102d <+37>:    jge    0x401039 <func4+49>
   0x000000000040102f <+39>:    lea    0x1(%rbx),%esi
   0x0000000000401032 <+42>:    callq  0x401008 <func4>
   0x0000000000401037 <+47>:    add    %ebx,%eax
   0x0000000000401039 <+49>:    pop    %rbx
   0x000000000040103a <+50>:    retq
```

에서는 eax=ebx=7인 상태에서 edi와 ebx를 비교하게 되는데, 두번째 입력 숫자가 7보다 작거나 같으면 재귀함수 호출을 피할 수 있다.

가장 간단한 형태의 함수 작동을 테스트하기 위해서 어떠한 재귀 호출이 일어나지 않는 경우를 생각해보면 edi가 7이어야 한다. 이를 넣어보면 결국 eax는 7의 값을 가진 채 마무리되는데, 앞서 살펴본 phase_4 코드에서 func4 이후 eax가 7이어야 explode_bomb을 피할 수 있다. 따라서 올바른 입력 값을 찾았음을 알 수 있다. 따라서 7 7을 입력하면 폭탄이 해체된다.

```
Halfway there!
7 7

Breakpoint 2, 0x000000000040103b in phase_4 ()
(gdb) next
Single stepping until exit from function phase_4,
which has no line number information.
main (argc=<optimized out>, argv=<optimized out>) at bomb.c:96
96          phase_defused();
```

phase_4가 해체되었다.

Phase_5

```
(gdb) disas
Dump of assembler code for function phase_5:
=> 0x0000000000401092 <+0>:     push   %rbx
   0x0000000000401093 <+1>:     sub    $0x10,%rsp
   0x0000000000401097 <+5>:     mov    %rdi,%rbx
   0x000000000040109a <+8>:     callq  0x401311 <string_length>
   0x000000000040109f <+13>:    cmp    $0x6,%eax
   0x00000000004010a2 <+16>:    je     0x4010e3 <phase_5+81>
   0x00000000004010a4 <+18>:    callq  0x401594 <explode_bomb>
   0x00000000004010a9 <+23>:    jmp    0x4010e3 <phase_5+81>
   0x00000000004010ab <+25>:    movzbl (%rbx,%rax,1),%edx
   0x00000000004010af <+29>:    and    $0xf,%edx
   0x00000000004010b2 <+32>:    movzbl 0x402580(%rdx),%edx
   0x00000000004010b9 <+39>:    mov    %dl,(%rsp,%rax,1)
   0x00000000004010bc <+42>:    add    $0x1,%rax
   0x00000000004010c0 <+46>:    cmp    $0x6,%rax
   0x00000000004010c4 <+50>:    jne    0x4010ab <phase_5+25>
   0x00000000004010c6 <+52>:    movb   $0x0,0x6(%rsp)
   0x00000000004010cb <+57>:    mov    $0x40252e,%esi
   0x00000000004010d0 <+62>:    mov    %rsp,%rdi
   0x00000000004010d3 <+65>:    callq  0x40132e <strings_not_equal>
   0x00000000004010d8 <+70>:    test   %eax,%eax
   0x00000000004010da <+72>:    je     0x4010ea <phase_5+88>
   0x00000000004010dc <+74>:    callq  0x401594 <explode_bomb>
   0x00000000004010e1 <+79>:    jmp    0x4010ea <phase_5+88>
   0x00000000004010e3 <+81>:    mov    $0x0,%eax
   0x00000000004010e8 <+86>:    jmp    0x4010ab <phase_5+25>
   0x00000000004010ea <+88>:    add    $0x10,%rsp
   0x00000000004010ee <+92>:    pop    %rbx
   0x00000000004010ef <+93>:    nop
   0x00000000004010f0 <+94>:    retq
End of assembler dump.
```

Phase_5의 코드다.

<+8>을 살펴보면 string_length라는 함수가 보이는데, string을 입력 받고 그 길이를 추출하는 함수임을 추측할 수 있다. 이를 확인하기 위해 임의의 문자열을 입력해본다.

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

Breakpoint 2, 0x0000000000401092 in phase_5 ()
```

```
(gdb) info register
rax            0xb      11
```

abcdefghijk를 입력했을 때 문자열의 길이인 11이 eax에 저장된다.

<+13>에서 eax가 6과 비교되고 같지 않으면 explode_bomb를 만나게 된다. 따라서 길이가 6인 문자열을 입력해야 한다는 사실을 알 수 있다.

이를 통과하면 <+81>로 이동하여 eax의 값은 0이 되고 <+25>로 돌아온다.

```
   0x00000000004010ab <+25>:    movzbl (%rbx,%rax,1),%edx
   0x00000000004010af <+29>:    and    $0xf,%edx
   0x00000000004010b2 <+32>:    movzbl 0x402580(%rdx),%edx
   0x00000000004010b9 <+39>:    mov    %dl,(%rsp,%rax,1)
   0x00000000004010bc <+42>:    add    $0x1,%rax
   0x00000000004010c0 <+46>:    cmp    $0x6,%rax
   0x00000000004010c4 <+50>:    jne    0x4010ab <phase_5+25>
```

이후 loop에 진입하게 되는데 <+25>의 역할을 이해하기 위해서 rbx의 정체를 파악할 필요가 있다. 코드 초반을 보면

```
Dump of assembler code for function phase_5:
=> 0x0000000000401092 <+0>:     push   %rbx
   0x0000000000401093 <+1>:     sub    $0x10,%rsp
   0x0000000000401097 <+5>:     mov    %rdi,%rbx
   0x000000000040109a <+8>:     callq  0x401311 <string_length>
```

에서 rdi가 rbx에 저장되는 사실을 알 수 있고, rdi는 string_length의 인자로 전달된다. rdi와 rbx에 공통적으로 저장되어 있는 메모리 값을 조사해본다.

```
(gdb) info register
rax            0xb      11
rbx            0x604900 6310144
rcx            0xb      11
rdx            0x60490b 6310155
rsi            0x604900 6310144
rdi            0x604900 6310144
```

```
(gdb) x/s 0x604900
0x604900 <input_strings+320>:   "abcdefghijk"
```

입력해둔 임의의 문자열이 저장되어 있다. 따라서 rbx는 사용자가 입력하는 문자열을 저장하는 역할을 한다고 할 수 있다.

0x00000000004010ab <+25>: movzbl (%rbx,%rax,1),%edx

0x00000000004010af <+29>: and $0xf,%edx

은 edx에 rbx + rax*1을 저장하고 edx의 하위 4비트만을 남겨놓겠다는 의미임을 알 수 있다.

그리고 다음 코드를 살펴보면

0x00000000004010b2 <+32>: movzbl 0x402580(%rdx),%edx

0x00000000004010b9 <+39>: mov %dl,(%rsp,%rax,1)

0x402580(%rdx)의 값을 edx에 다시 저장하겠다고 되어있다. 무슨 의미인지 파악하기 위해 0x402580의 메모리를 조사해본다.

```
(gdb) x/s 0x402580
0x402580 <array.3161>:  "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
```

"maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"이라는 문자열이 저장된 것을 확인할 수 있다.

이 문자열에서 rdx번째의 문자를 추출하여 저장하겠다는 의미인데, rdx는 하위 4비트만이 유효하므로 0~15 사이의 값만을 가질 수 있다. 따라서 문자열 앞 16개의 문자(maduiersnfotvbyl) 중 하나가 추출된다.

0x00000000004010b9 <+39>: mov %dl,(%rsp,%rax,1)

에서 추출한 문자를 rsp에 저장하고

```
0x00000000004010bc <+42>:    add    $0x1,%rax
0x00000000004010c0 <+46>:    cmp    $0x6,%rax
0x00000000004010c4 <+50>:    jne    0x4010ab <phase_5+25>
```

에서 rax에 1을 더하고 값이 6이 될 때까지 위 과정을 반복한다. Loop가 끝나면 사용자가 입력한 문자열에 따라 maduiersnfotvbyl에서 선택된 6개의 문자가 다시 저장될 것이다.

```
0x00000000004010c6 <+52>:    movb   $0x0,0x6(%rsp)
0x00000000004010cb <+57>:    mov    $0x40252e,%esi
0x00000000004010d0 <+62>:    mov    %rsp,%rdi
0x00000000004010d3 <+65>:    callq  0x40132e <strings_not_equal>
0x00000000004010d8 <+70>:    test   %eax,%eax
0x00000000004010da <+72>:    je     0x4010ea <phase_5+88>
0x00000000004010dc <+74>:    callq  0x401594 <explode_bomb>
0x00000000004010e1 <+79>:    jmp    0x4010ea <phase_5+88>
0x00000000004010e3 <+81>:    mov    $0x0,%eax
0x00000000004010e8 <+86>:    jmp    0x4010ab <phase_5+25>
0x00000000004010ea <+88>:    add    $0x10,%rsp
0x00000000004010ee <+92>:    pop    %rbx
0x00000000004010ef <+93>:    nop
0x00000000004010f0 <+94>:    retq
```

이후 0x40252e를 esi에, rsp를 rdi에 각각 저장한 후 strings_not_equal을 호출한다. 이후 text의 결과에 의해 explode_bomb의 호출 여부가 결정되는데 eax가 0이어야 함을 알 수 있다.

```
(gdb) x/s 0x40252e
0x40252e:       "sabres"
```

0x40252e의 메모리를 조사해보면 위 사진처럼 sabres라는 문자열이 저장되어 있다. rsi는 rsp의 값을 받으므로 앞선 loop에서 추출된 6개의 문자열일 것이다.

strings_not_equal 함수는 함수 이름에 따라 인수로 들어오는 두 문자열이 같은지를 판단하는 함수로 추측된다. 그럼 rsp에 저장된 문자열이 sabres가 되게 하려면 어떤 문자열을 입력해야 할지 고민해본다.

다시 앞으로 돌아가서 maduiersnfotvbyl에 각각 순서를 표시해본다.

m	a	d	u	i	e	r	s	n	f	o	t	v	b	y	l
0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

sabres 각각의 문자에 해당하는 순서를 나열하면 7 1 d 6 5 7이 된다. 따라서 입력한 문자열의 문자의 하위 4비트가 순서대로 7 1 d 6 5 7이어야 함을 알 수 있다. 아스키코드 표에서 문자의 16진법 표기를 알아보았을 때

G: 0x47, A: 0x41, M: 0x4D, F: 0x46, E: 0x45, G: 0x47

두번째 자리가 각각 7 1 d 6 5 7인 문자를 찾아볼 수 있었다. 따라서 찾은 문자열 GAMFEG를 입력하면

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

Breakpoint 2, 0x0000000000401092 in phase_5 ()
(gdb) next
Single stepping until exit from function phase_5,
which has no line number information.
main (argc=<optimized out>, argv=<optimized out>) at bomb.c:102
102         phase_defused();
(gdb)
```

폭탄이 해체됨을 알 수 있다.

Phase_6

```
Dump of assembler code for function phase_6:
=> 0x00000000004010f1 <+0>:     push   %r14
   0x00000000004010f3 <+2>:     push   %r13
   0x00000000004010f5 <+4>:     push   %r12
   0x00000000004010f7 <+6>:     push   %rbp
   0x00000000004010f8 <+7>:     push   %rbx
   0x00000000004010f9 <+8>:     sub    $0x50,%rsp
   0x00000000004010fd <+12>:    lea    0x30(%rsp),%r13
   0x0000000000401102 <+17>:    mov    %r13,%rsi
   0x0000000000401105 <+20>:    callq  0x4015ca <read_six_numbers>
   0x000000000040110a <+25>:    mov    %r13,%r14
   0x000000000040110d <+28>:    mov    $0x0,%r12d
   0x0000000000401113 <+34>:    mov    %r13,%rbp
   0x0000000000401116 <+37>:    mov    0x0(%r13),%eax
   0x000000000040111a <+41>:    sub    $0x1,%eax
   0x000000000040111d <+44>:    cmp    $0x5,%eax
   0x0000000000401120 <+47>:    jbe    0x401127 <phase_6+54>
   0x0000000000401122 <+49>:    callq  0x401594 <explode_bomb>
   0x0000000000401127 <+54>:    add    $0x1,%r12d
   0x000000000040112b <+58>:    cmp    $0x6,%r12d
   0x000000000040112f <+62>:    je     0x401153 <phase_6+98>
   0x0000000000401131 <+64>:    mov    %r12d,%ebx
   0x0000000000401134 <+67>:    movslq %ebx,%rax
   0x0000000000401137 <+70>:    mov    0x30(%rsp,%rax,4),%eax
   0x000000000040113b <+74>:    cmp    %eax,0x0(%rbp)
   0x000000000040113e <+77>:    jne    0x401145 <phase_6+84>
   0x0000000000401140 <+79>:    callq  0x401594 <explode_bomb>
   0x0000000000401145 <+84>:    add    $0x1,%ebx
   0x0000000000401148 <+87>:    cmp    $0x5,%ebx
   0x000000000040114b <+90>:    jle    0x401134 <phase_6+67>
   0x000000000040114d <+92>:    add    $0x4,%r13
   0x0000000000401151 <+96>:    jmp    0x401113 <phase_6+34>
   0x0000000000401153 <+98>:    lea    0x48(%rsp),%rsi
   0x0000000000401158 <+103>:   mov    %r14,%rax
   0x000000000040115b <+106>:   mov    $0x7,%ecx
   0x0000000000401160 <+111>:   mov    %ecx,%edx
   0x0000000000401162 <+113>:   sub    (%rax),%edx
   0x0000000000401164 <+115>:   mov    %edx,(%rax)
   0x0000000000401166 <+117>:   add    $0x4,%rax
   0x000000000040116a <+121>:   cmp    %rsi,%rax
   0x000000000040116d <+124>:   jne    0x401160 <phase_6+111>
   0x000000000040116f <+126>:   mov    $0x0,%esi
   0x0000000000401174 <+131>:   jmp    0x401196 <phase_6+165>
   0x0000000000401176 <+133>:   mov    0x8(%rdx),%rdx
   0x000000000040117a <+137>:   add    $0x1,%eax
   0x000000000040117d <+140>:   cmp    %ecx,%eax
   0x000000000040117f <+142>:   jne    0x401176 <phase_6+133>
---Type <return> to continue, or q <return> to quit---
```

```
---Type <return> to continue, or q <return> to quit---
   0x0000000000401181 <+144>:   jmp    0x401188 <phase_6+151>
   0x0000000000401183 <+146>:   mov    $0x6042f0,%edx
   0x0000000000401188 <+151>:   mov    %rdx,(%rsp,%rsi,2)
   0x000000000040118c <+155>:   add    $0x4,%rsi
   0x0000000000401190 <+159>:   cmp    $0x18,%rsi
   0x0000000000401194 <+163>:   je     0x4011ab <phase_6+186>
   0x0000000000401196 <+165>:   mov    0x30(%rsp,%rsi,1),%ecx
   0x000000000040119a <+169>:   cmp    $0x1,%ecx
   0x000000000040119d <+172>:   jle    0x401183 <phase_6+146>
   0x000000000040119f <+174>:   mov    $0x1,%eax
   0x00000000004011a4 <+179>:   mov    $0x6042f0,%edx
   0x00000000004011a9 <+184>:   jmp    0x401176 <phase_6+133>
   0x00000000004011ab <+186>:   mov    (%rsp),%rbx
   0x00000000004011af <+190>:   lea    0x8(%rsp),%rax
   0x00000000004011b4 <+195>:   lea    0x30(%rsp),%rsi
   0x00000000004011b9 <+200>:   mov    %rbx,%rcx
   0x00000000004011bc <+203>:   mov    (%rax),%rdx
   0x00000000004011bf <+206>:   mov    %rdx,0x8(%rcx)
   0x00000000004011c3 <+210>:   add    $0x8,%rax
   0x00000000004011c7 <+214>:   cmp    %rsi,%rax
   0x00000000004011ca <+217>:   je     0x4011d1 <phase_6+224>
   0x00000000004011cc <+219>:   mov    %rdx,%rcx
   0x00000000004011cf <+222>:   jmp    0x4011bc <phase_6+203>
   0x00000000004011d1 <+224>:   movq   $0x0,0x8(%rdx)
   0x00000000004011d9 <+232>:   mov    $0x5,%ebp
   0x00000000004011de <+237>:   mov    0x8(%rbx),%rax
   0x00000000004011e2 <+241>:   mov    (%rax),%eax
   0x00000000004011e4 <+243>:   cmp    %eax,(%rbx)
   0x00000000004011e6 <+245>:   jge    0x4011ed <phase_6+252>
   0x00000000004011e8 <+247>:   callq  0x401594 <explode_bomb>
   0x00000000004011ed <+252>:   mov    0x8(%rbx),%rbx
   0x00000000004011f1 <+256>:   sub    $0x1,%ebp
   0x00000000004011f4 <+259>:   jne    0x4011de <phase_6+237>
   0x00000000004011f6 <+261>:   add    $0x50,%rsp
   0x00000000004011fa <+265>:   pop    %rbx
   0x00000000004011fb <+266>:   pop    %rbp
   0x00000000004011fc <+267>:   pop    %r12
   0x00000000004011fe <+269>:   pop    %r13
   0x0000000000401200 <+271>:   pop    %r14
   0x0000000000401202 <+273>:   retq
End of assembler dump.
(gdb)
```

phase_6의 코드다. read_six_numbers라는 함수가 있는 것으로 보아 6개의 수를 입력 받는 것으로 보인다. 임의의 6개의 수를 입력해본다.

```
Good work!  On to the next...
2 4 1 3 5 6

Breakpoint 2, 0x00000000004010f1 in phase_6 ()
```

```
Dump of assembler code for function phase_6:
=> 0x00000000004010f1 <+0>:     push   %r14
   0x00000000004010f3 <+2>:     push   %r13
   0x00000000004010f5 <+4>:     push   %r12
   0x00000000004010f7 <+6>:     push   %rbp
   0x00000000004010f8 <+7>:     push   %rbx
   0x00000000004010f9 <+8>:     sub    $0x50,%rsp
   0x00000000004010fd <+12>:    lea    0x30(%rsp),%r13
   0x0000000000401102 <+17>:    mov    %r13,%rsi
   0x0000000000401105 <+20>:    callq  0x4015ca <read_six_numbers>
   0x000000000040110a <+25>:    mov    %r13,%r14
   0x000000000040110d <+28>:    mov    $0x0,%r12d
   0x0000000000401113 <+34>:    mov    %r13,%rbp
   0x0000000000401116 <+37>:    mov    0x0(%r13),%eax
   0x000000000040111a <+41>:    sub    $0x1,%eax
   0x000000000040111d <+44>:    cmp    $0x5,%eax
   0x0000000000401120 <+47>:    jbe    0x401127 <phase_6+54>
   0x0000000000401122 <+49>:    callq  0x401594 <explode_bomb>
```

<+44>에서 eax를 5와 비교한 후 eax가 5보다 크면 explode_bomb가 호출된다는 것을 확인할 수 있다. eax는 <+37>에서 r13의 값의 전달받고, 이에 1을 뺀 값이다. eax의 정체를 파악하기 위해 read_six_numbers 이후 r13을 조사해본다.

```
(gdb) nexti
0x000000000040110a in phase_6 ()
(gdb) info register
rax            0x6      6
rbx            0x7fffffffe538   140737488348472
rcx            0x7fffffffe3c0   140737488348096
rdx            0x0      0
rsi            0x0      0
rdi            0x7fffffffddb0   140737488346544
rbp            0x0      0x0
rsp            0x7fffffffe3d0   0x7fffffffe3d0
r8             0x7ffff7dd5060   140737351864416
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x400cd0 4197584
r13            0x7fffffffe400   140737488348160
r14            0x0      0
r15            0x0      0
rip            0x40110a 0x40110a <phase_6+25>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) x/6d 0x7fffffffe400
0x7fffffffe400: 2       4       1       3
0x7fffffffe410: 5       6
```

처음 입력한 6개의 숫자가 차례대로 저장되어 있다.

즉 eax에는 입력한 첫 번째 숫자가 저장되고, 이에서 1을 뺀 값이 5를 넘어가면 안된다. 즉, 첫 번째 숫자는 6 이하여야 한다.

```
   0x0000000000401113 <+34>:    mov    %r13,%rbp
   0x0000000000401116 <+37>:    mov    0x0(%r13),%eax
   0x000000000040111a <+41>:    sub    $0x1,%eax
   0x000000000040111d <+44>:    cmp    $0x5,%eax
   0x0000000000401120 <+47>:    jbe    0x401127 <phase_6+54>
   0x0000000000401122 <+49>:    callq  0x401594 <explode_bomb>
   0x0000000000401127 <+54>:    add    $0x1,%r12d
   0x000000000040112b <+58>:    cmp    $0x6,%r12d
   0x000000000040112f <+62>:    je     0x401153 <phase_6+98>
   0x0000000000401131 <+64>:    mov    %r12d,%ebx
   0x0000000000401134 <+67>:    movslq %ebx,%rax
   0x0000000000401137 <+70>:    mov    0x30(%rsp,%rax,4),%eax
   0x000000000040113b <+74>:    cmp    %eax,0x0(%rbp)
   0x000000000040113e <+77>:    jne    0x401145 <phase_6+84>
   0x0000000000401140 <+79>:    callq  0x401594 <explode_bomb>
   0x0000000000401145 <+84>:    add    $0x1,%ebx
   0x0000000000401148 <+87>:    cmp    $0x5,%ebx
   0x000000000040114b <+90>:    jle    0x401134 <phase_6+67>
   0x000000000040114d <+92>:    add    $0x4,%r13
   0x0000000000401151 <+96>:    jmp    0x401113 <phase_6+34>
```

하지만 여기서 끝나는 것이 아니라 loop를 돌게 되어있다. <+34>에서 <+96>까지 총 6번 반복을 하게 되어있다. r12d는 0부터 1씩 추가되다가 6이 되면 je문을 통해 loop 밖을 빠져나올 수 있다. 앞선 코드에서 r13에는 rsp+48인 주소값 즉, 입력된 숫자의 주소가 차례대로 저장되어 있다. 이 주소에 4byte씩 더해가며 6개의 수에 대한 검사를 진행한다. 따라서 입력한 6개의 수가 모두 6 이하의 값이어야 통과할 수 있다.

```
0x0000000000401153 <+98>:    lea    0x48(%rsp),%rsi
0x0000000000401158 <+103>:   mov    %r14,%rax
0x000000000040115b <+106>:   mov    $0x7,%ecx
0x0000000000401160 <+111>:   mov    %ecx,%edx
0x0000000000401162 <+113>:   sub    (%rax),%edx
0x0000000000401164 <+115>:   mov    %edx,(%rax)
0x0000000000401166 <+117>:   add    $0x4,%rax
0x000000000040116a <+121>:   cmp    %rsi,%rax
0x000000000040116d <+124>:   jne    0x401160 <phase_6+111>
0x000000000040116f <+126>:   mov    $0x0,%esi
0x0000000000401174 <+131>:   jmp    0x401196 <phase_6+165>
```

<+98>로 이동하면 새로운 loop에 진입하게 되는데 loop를 탈출하는 조건은 rsi와 rax가 같을 때이다. rsi는 <+98>에서 볼 수 있듯이 입력한 정수의 마지막 주소를 저장한 상태이고, rax는 r14의 값을 받았는데 r14는 앞선 코드에서 입력한 주소의 첫번째 주소를 가지고 있음을 알 수 있다. <+117>에서 rax에 4를 더해주는데 이를 통해 입력한 6개의 수를 하나씩 검사하며 rax가 마지막 주소를 가리킬 때 loop를 탈출하게 된다는 것을 알 수 있다.

Loop 도중에는

```
0x000000000040115b <+106>:   mov    $0x7,%ecx
0x0000000000401160 <+111>:   mov    %ecx,%edx
0x0000000000401162 <+113>:   sub    (%rax),%edx
0x0000000000401164 <+115>:   mov    %edx,(%rax)
```

rax = 7 - rax에 저장된 값 이 저장된다는 사실을 알 수 있다. 따라서 1 2 3 4 5 6을 입력했다면 6 5 4 3 2 1로 변환되어 저장될 것이다.

```
0x0000000000401176 <+133>:   mov    0x8(%rdx),%rdx
0x000000000040117a <+137>:   add    $0x1,%eax
0x000000000040117d <+140>:   cmp    %ecx,%eax
0x000000000040117f <+142>:   jne    0x401176 <phase_6+133>
Type <return> to continue, or q <return> to quit---
0x0000000000401181 <+144>:   jmp    0x401188 <phase_6+151>
0x0000000000401183 <+146>:   mov    $0x6042f0,%edx
0x0000000000401188 <+151>:   mov    %rdx,(%rsp,%rsi,2)
0x000000000040118c <+155>:   add    $0x4,%rsi
0x0000000000401190 <+159>:   cmp    $0x18,%rsi
0x0000000000401194 <+163>:   je     0x4011ab <phase_6+186>
0x0000000000401196 <+165>:   mov    0x30(%rsp,%rsi,1),%ecx
0x000000000040119a <+169>:   cmp    $0x1,%ecx
0x000000000040119d <+172>:   jle    0x401183 <phase_6+146>
0x000000000040119f <+174>:   mov    $0x1,%eax
0x00000000004011a4 <+179>:   mov    $0x6042f0,%edx
0x00000000004011a9 <+184>:   jmp    0x401176 <phase_6+133>
0x00000000004011ab <+186>:   mov    (%rsp),%rbx
0x00000000004011af <+190>:   lea    0x8(%rsp),%rax
```

새로운 loop에 진입하게 된다. 탈출 조건은 rsi가 24가 될 때이다. 일단 loop에 처음 진입하면 <+165>로 이동하게 된다. rsi가 0이므로 ecx에는 변환된 첫번째 숫자가 저장된다. <+172>에서 첫번째 숫자가 1보다 커 jump를 하지 않는다고 가정하면 edx에는 0x6042f0이라는 이상한 값이 저장되게 된다.

```
(gdb) x/d 0x006042f0
0x6042f0 <node1>:       892
```

이를 조사해보면 node1이라는 node와 안의 값 892를 발견할 수 있다. 이후 <+133>으로 이동하면 작은 loop에 진입하게 되는데,

```
0x0000000000401176 <+133>:   mov    0x8(%rdx),%rdx
0x000000000040117a <+137>:   add    $0x1,%eax
0x000000000040117d <+140>:   cmp    %ecx,%eax
0x000000000040117f <+142>:   jne    0x401176 <phase_6+133>
```

rdx에 ecx 즉, 변환된 첫번째 숫자만큼 8을 곱한 값만큼 떨어진 주소를 입력하라는 의미이다.

rdx = rdx + 8*ecx

```
0x0000000000401188 <+151>:   mov    %rdx,(%rsp,%rsi,2)
0x000000000040118c <+155>:   add    $0x4,%rsi
0x0000000000401190 <+159>:   cmp    $0x18,%rsi
0x0000000000401194 <+163>:   je     0x4011ab <phase_6+186>
```

이후 저장한 rdx의 주소값을 rsp+2*rsi에 저장한다. <+155>에서 rsi는 4씩 증가하며 24가 되면 loop는 종료된다. 따라서 <+151>은 rsp에 입력한 숫자에 따라 다른 메모리 값을 8바이트의 공간

을 두고 차례대로 저장한다는 의미이다. loop가 종료된 시점에서 rsp에 어떤 값이 저장되어 있는지 확인해본다.

```
(gdb) x/12wx 0x7fffffffe3d0
0x7fffffffe3d0: 0x00604340      0x00000000      0x00604330      0x00000000
0x7fffffffe3e0: 0x00604320      0x00000000      0x00604310      0x00000000
0x7fffffffe3f0: 0x00604300      0x00000000      0x006042f0      0x00000000
(gdb) x/d 0x00604340
0x604340 <node6>:       625
(gdb) x/d 0x00604330
0x604330 <node5>:       195
(gdb) x/d 0x00604320
0x604320 <node4>:       582
(gdb) x/d 0x00604310
0x604310 <node3>:       615
(gdb) x/d 0x00604300
0x604300 <node2>:       738
(gdb) x/d 0x006042f0
0x6042f0 <node1>:       892
```

1 2 3 4 5 6을 입력했을 때, rsp에 저장된 주소를 조사한 값이다. 저장된 메모리 값을 순서대로 조사해보니 node1부터 6까지 총 6개 node의 존재를 파악할 수 있었다.

위 loop의 코드를 분석한 결과를 생각해보면 eax에 저장되어 있는 숫자를 n이라고 했을 때, rsp에는 node1에서 n*8만큼 떨어진 값을 차례대로 저장하게 되고 그 결과가 바로 위 사진이라는 뜻이다. 즉, node 6개가 순서대로 8바이트씩 떨어져 위치해 있었고 사용자가 입력한 숫자에 따라 node가 rsp에 저장된다.

```
0x00000000004011ab <+186>:    mov    (%rsp),%rbx
0x00000000004011af <+190>:    lea    0x8(%rsp),%rax
0x00000000004011b4 <+195>:    lea    0x30(%rsp),%rsi
0x00000000004011b9 <+200>:    mov    %rbx,%rcx
0x00000000004011bc <+203>:    mov    (%rax),%rdx
0x00000000004011bf <+206>:    mov    %rdx,0x8(%rcx)
0x00000000004011c3 <+210>:    add    $0x8,%rax
0x00000000004011c7 <+214>:    cmp    %rsi,%rax
0x00000000004011ca <+217>:    je     0x4011d1 <phase_6+224>
0x00000000004011cc <+219>:    mov    %rdx,%rcx
0x00000000004011cf <+222>:    jmp    0x4011bc <phase_6+203>
0x00000000004011d1 <+224>:    movq   $0x0,0x8(%rdx)
0x00000000004011d9 <+232>:    mov    $0x5,%ebp
0x00000000004011de <+237>:    mov    0x8(%rbx),%rax
0x00000000004011e2 <+241>:    mov    (%rax),%eax
0x00000000004011e4 <+243>:    cmp    %eax,(%rbx)
0x00000000004011e6 <+245>:    jge    0x4011ed <phase_6+252>
0x00000000004011e8 <+247>:    callq  0x401594 <explode_bomb>
0x00000000004011ed <+252>:    mov    0x8(%rbx),%rbx
0x00000000004011f1 <+256>:    sub    $0x1,%ebp
0x00000000004011f4 <+259>:    jne    0x4011de <phase_6+237>
0x00000000004011f6 <+261>:    add    $0x50,%rsp
0x00000000004011fa <+265>:    pop    %rbx
0x00000000004011fb <+266>:    pop    %rbp
0x00000000004011fc <+267>:    pop    %r12
0x00000000004011fe <+269>:    pop    %r13
0x0000000000401200 <+271>:    pop    %r14
0x0000000000401202 <+273>:    retq
```

<+186>에서 rbx에 rsp가 가리키는 주소의 값을 저장한다. 이후 몇몇 데이터를 저장하는 과정을 거친 후 마지막 loop에 진입하게 된다.

```
0x00000000004011d1 <+224>:    movq   $0x0,0x8(%rdx)
0x00000000004011d9 <+232>:    mov    $0x5,%ebp
0x00000000004011de <+237>:    mov    0x8(%rbx),%rax
0x00000000004011e2 <+241>:    mov    (%rax),%eax
0x00000000004011e4 <+243>:    cmp    %eax,(%rbx)
0x00000000004011e6 <+245>:    jge    0x4011ed <phase_6+252>
0x00000000004011e8 <+247>:    callq  0x401594 <explode_bomb>
0x00000000004011ed <+252>:    mov    0x8(%rbx),%rbx
0x00000000004011f1 <+256>:    sub    $0x1,%ebp
0x00000000004011f4 <+259>:    jne    0x4011de <phase_6+237>
0x00000000004011f6 <+261>:    add    $0x50,%rsp
0x00000000004011fa <+265>:    pop    %rbx
0x00000000004011fb <+266>:    pop    %rbp
0x00000000004011fc <+267>:    pop    %r12
0x00000000004011fe <+269>:    pop    %r13
0x0000000000401200 <+271>:    pop    %r14
0x0000000000401202 <+273>:    retq
```

<+237>에서 rax에는 rbx+8의 값 즉, rax에는 rbx가 가리키는 값의 다음 값을 저장한다.

<+243>에서 둘을 비교한 후 eax가 더 커야 explode_bomb가 호출되는 것을 피할 수 있다. 이렇게 모든 값에 대해 검사를 진행하는 것으로 보아 앞의 값이 뒤의 값보다 큰 내림차순의 값을 가져야 통과할 수 있는 것으로 보인다.

```
0x604340 <node6>:       625
(gdb) x/d 0x00604330
0x604330 <node5>:       195
(gdb) x/d 0x00604320
0x604320 <node4>:       582
(gdb) x/d 0x00604310
0x604310 <node3>:       615
(gdb) x/d 0x00604300
0x604300 <node2>:       738
(gdb) x/d 0x006042f0
0x6042f0 <node1>:       892
```

따라서 node의 값이 큰 순서대로 1 2 6 3 4 5 node가 저장되어 있어야 한다. 하지만 이는 7에서 입력한 숫자를 뺀 결과이므로 입력은 6 5 1 4 3 2를 해줘야 한다.

```
Good work!  On to the next...
6 5 1 4 3 2

Breakpoint 6, 0x00000000004011ab in phase_6 ()
(gdb) next
Single stepping until exit from function phase_6,
which has no line number information.

Breakpoint 7, 0x00000000004011d1 in phase_6 ()
(gdb) next
Single stepping until exit from function phase_6,
which has no line number information.
main (argc=<optimized out>, argv=<optimized out>) at bomb.c:109
109         phase_defused();
```

phase_6가 해제된 것을 볼 수 있다.

secrete_phase

```
(gdb) disas phase_defused
Dump of assembler code for function phase_defused:
   0x0000000000401732 <+0>:     sub    $0x68,%rsp
   0x0000000000401736 <+4>:     mov    $0x1,%edi
   0x000000000040173b <+9>:     callq  0x4014d0 <send_msg>
   0x0000000000401740 <+14>:    cmpl   $0x6,0x203055(%rip)        # 0x60479c <num_input_strings>
   0x0000000000401747 <+21>:    jne    0x4017b6 <phase_defused+132>
   0x0000000000401749 <+23>:    lea    0x10(%rsp),%r8
   0x000000000040174e <+28>:    lea    0x8(%rsp),%rcx
   0x0000000000401753 <+33>:    lea    0xc(%rsp),%rdx
   0x0000000000401758 <+38>:    mov    $0x402827,%esi
   0x000000000040175d <+43>:    mov    $0x6048b0,%edi
   0x0000000000401762 <+48>:    mov    $0x0,%eax
   0x0000000000401767 <+53>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x000000000040176c <+58>:    cmp    $0x3,%eax
   0x000000000040176f <+61>:    jne    0x4017a2 <phase_defused+112>
   0x0000000000401771 <+63>:    mov    $0x402830,%esi
   0x0000000000401776 <+68>:    lea    0x10(%rsp),%rdi
   0x000000000040177b <+73>:    callq  0x40132e <strings_not_equal>
   0x0000000000401780 <+78>:    test   %eax,%eax
   0x0000000000401782 <+80>:    jne    0x4017a2 <phase_defused+112>
   0x0000000000401784 <+82>:    mov    $0x402688,%edi
   0x0000000000401789 <+87>:    callq  0x400b40 <puts@plt>
   0x000000000040178e <+92>:    mov    $0x4026b0,%edi
   0x0000000000401793 <+97>:    callq  0x400b40 <puts@plt>
   0x0000000000401798 <+102>:   mov    $0x0,%eax
   0x000000000040179d <+107>:   callq  0x401241 <secret_phase>
   0x00000000004017a2 <+112>:   mov    $0x4026e8,%edi
   0x00000000004017a7 <+117>:   callq  0x400b40 <puts@plt>
   0x00000000004017ac <+122>:   mov    $0x402718,%edi
   0x00000000004017b1 <+127>:   callq  0x400b40 <puts@plt>
   0x00000000004017b6 <+132>:   add    $0x68,%rsp
   0x00000000004017ba <+136>:   retq
End of assembler dump.
```

phase_defude를 disassemble 해보면 위와 같은 코드를 얻을 수 있다. <+107>에서 secret_phase가 존재한다는 것을 알 수 있다. 코드를 분석해 보았을 때 secret_phase를 호출하기 위해서는 <+61>, <+80>의 jump문에서 분기를 하면 안된다. <+61> 바로 위에 보이는 sscan 함수가 익숙하다.

```
Dump of assembler code for function phase_4:
   0x000000000040103b <+0>:     sub    $0x18,%rsp
   0x000000000040103f <+4>:     lea    0x8(%rsp),%rcx
   0x0000000000401044 <+9>:     lea    0xc(%rsp),%rdx
   0x0000000000401049 <+14>:    mov    $0x4027dd,%esi
   0x000000000040104e <+19>:    mov    $0x0,%eax
   0x0000000000401053 <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x0000000000401058 <+29>:    cmp    $0x2,%eax
   0x000000000040105b <+32>:    jne    0x401064 <phase_4+41>
   0x000000000040105d <+34>:    cmpl   $0xe,0xc(%rsp)
```

phase_4에서 똑같은 함수를 마주친 적이 있다. 주소도 같은 것을 보아 같은 함수일 것이다. 하지만 phase_4에서는 인자를 두개 입력 받은 반면 secret_phase의 <+58>을 보면 eax를 3과 비교하는 것을 볼 수 있다. 즉, 인자를 하나 더 받아야 한다는 것이고 <+78>에서는 eax가 0이어야 하므로 <+73>에서 인자와 특정 문자열이 같아야 한다는 것을 알 수 있다. strings_not_equal에 인자

로 들어가는 esi에 저장되는 값을 조사해본다.

```
(gdb) x/s 0x402830
0x402830:       "DrEvil"
```

DrEvil이라는 문자열을 얻을 수 있었다. 따라서 secret_phase에 진입하려면 phase_4의 솔루션에 DrEvil을 추가하여 입력해야 함을 알 수 있다.

```
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

문자열을 추가하니 secret_phase에 진입할 수 있었다.

secret_phase를 disassemble 해본다.

```
Dump of assembler code for function secret_phase:
   0x0000000000401241 <+0>:     push   %rbx
   0x0000000000401242 <+1>:     callq  0x40160c <read_line>
   0x0000000000401247 <+6>:     mov    $0xa,%edx
   0x000000000040124c <+11>:    mov    $0x0,%esi
   0x0000000000401251 <+16>:    mov    %rax,%rdi
   0x0000000000401254 <+19>:    callq  0x400c00 <strtol@plt>
   0x0000000000401259 <+24>:    mov    %rax,%rbx
   0x000000000040125c <+27>:    lea    -0x1(%rax),%eax
   0x000000000040125f <+30>:    cmp    $0x3e8,%eax
   0x0000000000401264 <+35>:    jbe    0x40126b <secret_phase+42>
   0x0000000000401266 <+37>:    callq  0x401594 <explode_bomb>
   0x000000000040126b <+42>:    mov    %ebx,%esi
   0x000000000040126d <+44>:    mov    $0x604110,%edi
   0x0000000000401272 <+49>:    callq  0x401203 <fun7>
   0x0000000000401277 <+54>:    cmp    $0x6,%eax
   0x000000000040127a <+57>:    je     0x401281 <secret_phase+64>
   0x000000000040127c <+59>:    callq  0x401594 <explode_bomb>
   0x0000000000401281 <+64>:    mov    $0x402508,%edi
   0x0000000000401286 <+69>:    callq  0x400b40 <puts@plt>
   0x000000000040128b <+74>:    callq  0x401732 <phase_defused>
   0x0000000000401290 <+79>:    pop    %rbx
   0x0000000000401291 <+80>:    retq
End of assembler dump.
```

explode_bomb을 호출하게 되는 분기점이 2개 보인다. <+27>과 <+30>에서 eax가 1001 이하여야 한다는 것을 파악할 수 있고, <+54>에서는 fun7의 결과로 eax가 6이어야 한다는 것을 알 수 있다.

fun7의 기능을 알아보기 위해 함수의 코드를 살펴본다.

```
(gdb) disas fun7
Dump of assembler code for function fun7:
   0x0000000000401203 <+0>:     sub    $0x8,%rsp
   0x0000000000401207 <+4>:     test   %rdi,%rdi
   0x000000000040120a <+7>:     je     0x401237 <fun7+52>
   0x000000000040120c <+9>:     mov    (%rdi),%edx
   0x000000000040120e <+11>:    cmp    %esi,%edx
   0x0000000000401210 <+13>:    jle    0x40121f <fun7+28>
   0x0000000000401212 <+15>:    mov    0x8(%rdi),%rdi
   0x0000000000401216 <+19>:    callq  0x401203 <fun7>
   0x000000000040121b <+24>:    add    %eax,%eax
   0x000000000040121d <+26>:    jmp    0x40123c <fun7+57>
   0x000000000040121f <+28>:    mov    $0x0,%eax
   0x0000000000401224 <+33>:    cmp    %esi,%edx
   0x0000000000401226 <+35>:    je     0x40123c <fun7+57>
   0x0000000000401228 <+37>:    mov    0x10(%rdi),%rdi
   0x000000000040122c <+41>:    callq  0x401203 <fun7>
   0x0000000000401231 <+46>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401235 <+50>:    jmp    0x40123c <fun7+57>
   0x0000000000401237 <+52>:    mov    $0xffffffff,%eax
   0x000000000040123c <+57>:    add    $0x8,%rsp
   0x0000000000401240 <+61>:    retq
End of assembler dump.
```

코드를 훑어보면 rdi를 인자로 받는 일종의 재귀함수임을 알 수 있다.

```
   0x000000000040126d <+44>:    mov    $0x604110,%edi
   0x0000000000401272 <+49>:    callq  0x401203 <fun7>
```

에서 fun7이 호출되기 전 edi에 특정 값이 저장되는 것을 확인할 수 있다. 이를 조사해보면

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

36이라는 값이 저장되어 있음을 알 수 있다. fun7에서 rdi의 주소를 계속해서 업데이트 해주는 것으로 보아 일련의 정보가 연속적으로 저장되어 있을 것이라고 추측한다. 따라서 이후의 범위에 대한 조사를 추가적으로 진행해본다.

```
(gdb) x/20d 0x604110
0x604110 <n1>:  36      0       6308144 0
0x604120 <n1+16>:       6308176 0       0       0
0x604130 <n21>: 8       0       6308272 0
0x604140 <n21+16>:      6308208 0       0       0
0x604150 <n22>: 50      0       6308240 0
```

16바이트 단위로 유의미해 보이는 값이 저장되어 있는 것을 볼 수 있다. 라벨링도 되어 있는 것으로 보아 후속 정보가 저장되어 있을 것으로 범위를 계속 넓혀가며 탐색해본다.

```
(gdb) x/200d 0x604110
0x604110 <n1>:  36      0       6308144 0
0x604120 <n1+16>:       6308176 0       0       0
0x604130 <n21>: 8       0       6308272 0
0x604140 <n21+16>:      6308208 0       0       0
0x604150 <n22>: 50      0       6308240 0
0x604160 <n22+16>:      6308304 0       0       0
0x604170 <n32>: 22      0       6308496 0
0x604180 <n32+16>:      6308432 0       0       0
0x604190 <n33>: 45      0       6308336 0
0x6041a0 <n33+16>:      6308528 0       0       0
0x6041b0 <n31>: 6       0       6308368 0
0x6041c0 <n31+16>:      6308464 0       0       0
0x6041d0 <n34>: 107     0       6308400 0
0x6041e0 <n34+16>:      6308560 0       0       0
0x6041f0 <n45>: 40      0       0       0
0x604200 <n45+16>:      0       0       0       0
0x604210 <n41>: 1       0       0       0
0x604220 <n41+16>:      0       0       0       0
0x604230 <n47>: 99      0       0       0
0x604240 <n47+16>:      0       0       0       0
0x604250 <n44>: 35      0       0       0
0x604260 <n44+16>:      0       0       0       0
0x604270 <n42>: 7       0       0       0
0x604280 <n42+16>:      0       0       0       0
0x604290 <n43>: 20      0       0       0
0x6042a0 <n43+16>:      0       0       0       0
0x6042b0 <n46>: 47      0       0       0
0x6042c0 <n46+16>:      0       0       0       0
0x6042d0 <n48>: 1001    0       0       0
0x6042e0 <n48+16>:      0       0       0       0
0x6042f0 <node1>:       892     1       6308608 0
```

<n48+16>까지 일정한 형식의 라벨링이 되어있는 일련의 정보를 얻을 수 있었다.

다시 fun7으로 돌아와 또 하나의 인수 esi에 대해 탐색해본다. secret_phase에서 fun7이 호출되기 직전 esi의 값을 확인해본다.

```
But finding it and solving it are quite different...
35

Breakpoint 2, 0x000000000040126d in secret_phase ()
(gdb) info register
rax            0x22     34
rbx            0x23     35
rcx            0x0      0
rdx            0xa      10
rsi            0x23     35
rdi            0x1999999999999999       1844674407370955161
rbp            0x0      0x0
rsp            0x7fffffffe3d0   0x7fffffffe3d0
r8             0x7ffff7dd5060   140737351864416
r9             0x6049a2 6310306
r10            0x23     35
r11            0x0      0
r12            0x400cd0 4197584
r13            0x7fffffffe530   140737488348464
r14            0x0      0
r15            0x0      0
rip            0x40126d 0x40126d <secret_phase+44>
eflags         0x297    [ CF PF AF SF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

임의로 입력해두었던 35가 저장되어 있는 것을 확인할 수 있다.

```
(gdb) disas fun7
Dump of assembler code for function fun7:
   0x0000000000401203 <+0>:     sub    $0x8,%rsp
   0x0000000000401207 <+4>:     test   %rdi,%rdi
   0x000000000040120a <+7>:     je     0x401237 <fun7+52>
   0x000000000040120c <+9>:     mov    (%rdi),%edx
   0x000000000040120e <+11>:    cmp    %esi,%edx
   0x0000000000401210 <+13>:    jle    0x40121f <fun7+28>
   0x0000000000401212 <+15>:    mov    0x8(%rdi),%rdi
   0x0000000000401216 <+19>:    callq  0x401203 <fun7>
   0x000000000040121b <+24>:    add    %eax,%eax
   0x000000000040121d <+26>:    jmp    0x40123c <fun7+57>
   0x000000000040121f <+28>:    mov    $0x0,%eax
   0x0000000000401224 <+33>:    cmp    %esi,%edx
   0x0000000000401226 <+35>:    je     0x40123c <fun7+57>
   0x0000000000401228 <+37>:    mov    0x10(%rdi),%rdi
   0x000000000040122c <+41>:    callq  0x401203 <fun7>
   0x0000000000401231 <+46>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401235 <+50>:    jmp    0x40123c <fun7+57>
   0x0000000000401237 <+52>:    mov    $0xffffffff,%eax
   0x000000000040123c <+57>:    add    $0x8,%rsp
   0x0000000000401240 <+61>:    retq
End of assembler dump.
```

fun7에서 재귀 호출을 하는 경우가 두 가지 있다.

```
   0x000000000040120e <+11>:    cmp    %esi,%edx
   0x0000000000401210 <+13>:    jle    0x40121f <fun7+28>
   0x0000000000401212 <+15>:    mov    0x8(%rdi),%rdi
   0x0000000000401216 <+19>:    callq  0x401203 <fun7>
   0x000000000040121b <+24>:    add    %eax,%eax
   0x000000000040121d <+26>:    jmp    0x40123c <fun7+57>
```

```
   0x000000000040121f <+28>:    mov    $0x0,%eax
   0x0000000000401224 <+33>:    cmp    %esi,%edx
   0x0000000000401226 <+35>:    je     0x40123c <fun7+57>
   0x0000000000401228 <+37>:    mov    0x10(%rdi),%rdi
   0x000000000040122c <+41>:    callq  0x401203 <fun7>
   0x0000000000401231 <+46>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401235 <+50>:    jmp    0x40123c <fun7+57>
```

아까 확인했듯이 edi에는 16바이트를 간격으로 일련의 수가 저장되어 있다. 일련의 수와 입력한 수를 계속해서 비교해 나가는 것으로 추측할 수 있다.

eax의 값이 위 두 경우에 각각 eax*2, eax*2+1의 값으로 업데이트 되는 것을 확인할 수 있다. eax의 값이 결국 6이 되는 것이 목표이므로 (((eax*2+1)*2+1)*2)의 과정을 거쳐야 한다.

eax의 업데이트 방식과 n1-n48까지의 라벨링 방식을 보았을 때 트리구조와 관련이 깊어 보인다.

앞서 얻은 일련의 수를 트리로 구성해보면

36

8 50

6 22 45 107

1 7 20 35 40 47 99 1001

의 모양을 가지게 된다.

코드를 분석해 보았을 때 left node로 이동하면 *2, 반대는 *2+1 연산이 되는 것으로 보여진다. 따라서 left > right > right 연산을 해야 최종적으로 6을 얻을 수 있다. 35를 입력하면 된다.

```
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
```