시스템소프트웨어

HW2 Bomblab

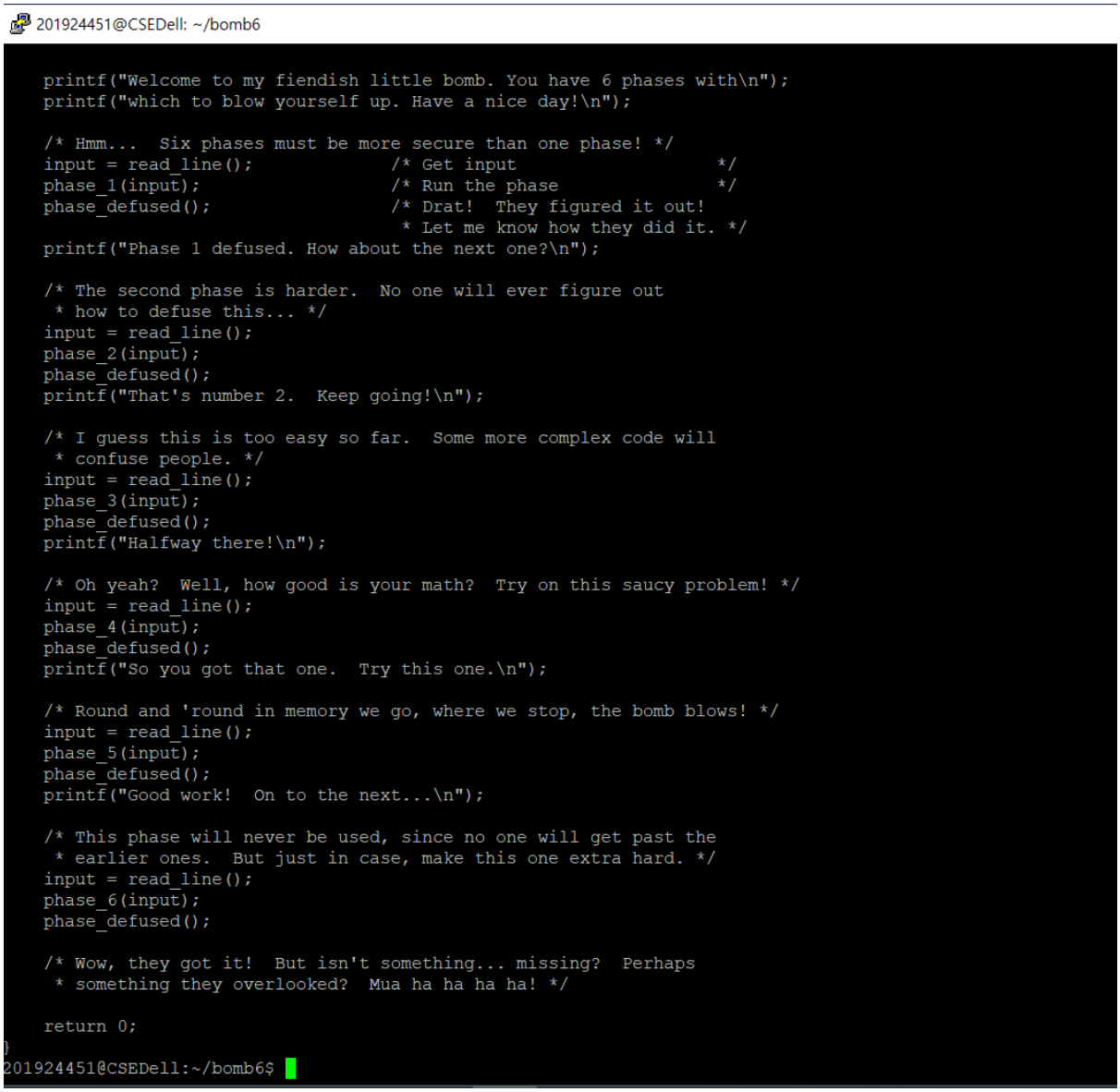
REPORT

분반	059
소속	
학번	
이름	김태훈

1. Main 함수

201924451@CSEDell: ~/bomb6

```
1167:       83 ff 02                cmp    $0x2,%edi
116a:       0f 85 21 01 00 00       jne    1291 <main+0x137>
1170:       48 8b 7e 08             mov    0x8(%rsi),%rdi
1174:       48 8d 35 4d 1e 00 00    lea    0x1e4d(%rip),%rsi        # 2fc8 <array.3417+0x488>
117b:       e8 30 fe ff ff          callq  fb0 <fopen@plt>
1180:       48 89 05 29 35 20 00    mov    %rax,0x203529(%rip)        # 2046b0 <infile>
1187:       48 85 c0                test   %rax,%rax
118a:       0f 84 df 00 00 00       je     126f <main+0x115>
1190:       e8 95 06 00 00          callq  182a <initialize_bomb>
1195:       48 8d 3d 8c 18 00 00    lea    0x188c(%rip),%rdi        # 2a28 <_IO_stdin_used+0x88>
119c:       e8 2f fd ff ff          callq  ed0 <puts@plt>
11a1:       48 8d 3d c0 18 00 00    lea    0x18c0(%rip),%rdi        # 2a68 <_IO_stdin_used+0xc8>
11a8:       e8 23 fd ff ff          callq  ed0 <puts@plt>
11ad:       e8 92 09 00 00          callq  1b44 <read_line>
11b2:       48 89 c7                mov    %rax,%rdi
11b5:       e8 fa 00 00 00          callq  12b4 <phase_1>
11ba:       e8 c9 0a 00 00          callq  1c88 <phase_defused>
11bf:       48 8d 3d d2 18 00 00    lea    0x18d2(%rip),%rdi        # 2a98 <_IO_stdin_used+0xf8>
11c6:       e8 05 fd ff ff          callq  ed0 <puts@plt>
11cb:       e8 74 09 00 00          callq  1b44 <read_line>
11d0:       48 89 c7                mov    %rax,%rdi
11d3:       e8 fc 00 00 00          callq  12d4 <phase_2>
11d8:       e8 ab 0a 00 00          callq  1c88 <phase_defused>
11dd:       48 8d 3d f7 17 00 00    lea    0x17f7(%rip),%rdi        # 29db <_IO_stdin_used+0x3b>
11e4:       e8 e7 fc ff ff          callq  ed0 <puts@plt>
11e9:       e8 56 09 00 00          callq  1b44 <read_line>
11ee:       48 89 c7                mov    %rax,%rdi
11f1:       e8 4d 01 00 00          callq  1343 <phase_3>
11f6:       e8 8d 0a 00 00          callq  1c88 <phase_defused>
11fb:       48 8d 3d f7 17 00 00    lea    0x17f7(%rip),%rdi        # 29f9 <_IO_stdin_used+0x59>
1202:       e8 c9 fc ff ff          callq  ed0 <puts@plt>
1207:       e8 38 09 00 00          callq  1b44 <read_line>
120c:       48 89 c7                mov    %rax,%rdi
120f:       e8 4f 02 00 00          callq  1463 <phase_4>
1214:       e8 6f 0a 00 00          callq  1c88 <phase_defused>
1219:       48 8d 3d a8 18 00 00    lea    0x18a8(%rip),%rdi        # 2ac8 <_IO_stdin_used+0x128>
1220:       e8 ab fc ff ff          callq  ed0 <puts@plt>
1225:       e8 1a 09 00 00          callq  1b44 <read_line>
122a:       48 89 c7                mov    %rax,%rdi
122d:       e8 a6 02 00 00          callq  14d8 <phase_5>
1232:       e8 51 0a 00 00          callq  1c88 <phase_defused>
1237:       48 8d 3d ca 17 00 00    lea    0x17ca(%rip),%rdi        # 2a08 <_IO_stdin_used+0x68>
123e:       e8 8d fc ff ff          callq  ed0 <puts@plt>
1243:       e8 fc 08 00 00          callq  1b44 <read_line>
1248:       48 89 c7                mov    %rax,%rdi
124b:       e8 1b 03 00 00          callq  156b <phase_6>
1250:       e8 33 0a 00 00          callq  1c88 <phase_defused>
1255:       b8 00 00 00 00          mov    $0x0,%eax
125a:       5b                      pop    %rbx
```



objdump와 bomb.c를 통해 문제는 phase1부터 phase6까지 있음을 알 수 있다.

즉

callq 1b44 <read_line>

mov %rax %rdi

callq <phase_X>

callq <phase_defused>

을 통해 read_line을 호출하고, 반환값을 함수의 첫번째 인자로 한 다음 phase_X와 phase_defused를 호출하는 것을 알 수 있다. 즉 phase_1~phase_6에는 입력한 문자열이 첫번째 인자로 들어간다.

2. phase_1

```
00000000000012b4 <phase_1>:
    12b4:       48 83 ec 08             sub    $0x8,%rsp
    12b8:       48 8d 35 2d 18 00 00    lea    0x182d(%rip),%rsi        # 2aec <_IO_stdin_used+0x14c>
    12bf:       e8 ff 04 00 00          callq  17c3 <strings_not_equal>
    12c4:       85 c0                   test   %eax,%eax
    12c6:       75 05                   jne    12cd <phase_1+0x19>
    12c8:       48 83 c4 08             add    $0x8,%rsp
    12cc:       c3                      retq
    12cd:       e8 f5 07 00 00          callq  1ac7 <explode_bomb>
    12d2:       eb f4                   jmp    12c8 <phase_1+0x14>
```

비교할 문자열을 %rsi로 옮긴 후(lea 0x182d(%rip), %rsi)

(%rdi에는 입력한 문자열이 들어가있다.)

<strings_not_equal>을 호출하고 그 반환값이 1이면

jne <phase_1+0x19>를 통해 callq <explode_bomb>로 분기하고 explode_bomb를 호출한다.

만약 반환값이 0이면 explode_bomb를 호출하지 않고 종료한다.

아마 strings_not_equal은 두 문자열이 같으면 0, 다르면 1을 반환하는 것 같다.

3. strings_not_equal

gdb를 실행하여 strings_not_equal에 breakpoint를 건다.

```
201924451@CSEDell:~/bomb6$ gdb bomb
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...done.
(gdb) b strings_not_equal
Breakpoint 1 at 0x17c3
(gdb)
```

그리고 프로그램을 실행하여 임의의 문자열을 입력해 breakpoint까지 간다.

```
(gdb) b strings_not_equal
Breakpoint 1 at 0x17c3
(gdb) r
Starting program: /home/sys059/201924451/bomb6/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
test

Breakpoint 1, 0x00005555555557c3 in strings_not_equal ()
(gdb) disas strings_not_equal
Dump of assembler code for function strings_not_equal:
=> 0x00005555555557c3 <+0>:     push   %r12
   0x00005555555557c5 <+2>:     push   %rbp
   0x00005555555557c6 <+3>:     push   %rbx
   0x00005555555557c7 <+4>:     mov    %rdi,%rbx
   0x00005555555557ca <+7>:     mov    %rsi,%rbp
   0x00005555555557cd <+10>:    callq  0x5555555557a6 <string_length>
   0x00005555555557d2 <+15>:    mov    %eax,%r12d
   0x00005555555557d5 <+18>:    mov    %rbp,%rdi
   0x00005555555557d8 <+21>:    callq  0x5555555557a6 <string_length>
```

strings_not_equal에서 %rdi에는 입력한 문자열이, %rsi에는 비교할 문자열이 들어있다.

x/s %rsi 를 통해 %rsi의 값을 보면

```
(gdb) x/s $rsi
0x555555556aec: "Public speaking is very easy."
(gdb)
```

따라서 phase_1의 답은 Public speaking is very easy. 임을 알 수 있다.

```
(gdb) r
Starting program: /home/sys059/201924451/bomb6/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Public speaking is very easy.
Phase 1 defused. How about the next one?
```

4. phase_2 and read_six_numbers

```
(gdb) disas phase_2
Dump of assembler code for function phase_2:
   0x00005555555552d4 <+0>:     push   %rbp
   0x00005555555552d5 <+1>:     push   %rbx
   0x00005555555552d6 <+2>:     sub    $0x28,%rsp
   0x00005555555552da <+6>:     mov    %fs:0x28,%rax
   0x00005555555552e3 <+15>:    mov    %rax,0x18(%rsp)
   0x00005555555552e8 <+20>:    xor    %eax,%eax
   0x00005555555552ea <+22>:    mov    %rsp,%rsi
   0x00005555555552ed <+25>:    callq  0x555555555b03 <read_six_numbers>
   0x00005555555552f2 <+30>:    cmpl   $0x0,(%rsp)
   0x00005555555552f6 <+34>:    jne    0x5555555552ff <phase_2+43>
   0x00005555555552f8 <+36>:    cmpl   $0x1,0x4(%rsp)
   0x00005555555552fd <+41>:    je     0x555555555304 <phase_2+48>
   0x00005555555552ff <+43>:    callq  0x555555555ac7 <explode_bomb>
   0x0000555555555304 <+48>:    mov    %rsp,%rbx
   0x0000555555555307 <+51>:    lea    0x10(%rbx),%rbp
   0x000055555555530b <+55>:    jmp    0x555555555316 <phase_2+66>
   0x000055555555530d <+57>:    add    $0x4,%rbx
   0x0000555555555311 <+61>:    cmp    %rbp,%rbx
   0x0000555555555314 <+64>:    je     0x555555555327 <phase_2+83>
   0x0000555555555316 <+66>:    mov    0x4(%rbx),%eax
   0x0000555555555319 <+69>:    add    (%rbx),%eax
   0x000055555555531b <+71>:    cmp    %eax,0x8(%rbx)
   0x000055555555531e <+74>:    je     0x55555555530d <phase_2+57>
   0x0000555555555320 <+76>:    callq  0x555555555ac7 <explode_bomb>
   0x0000555555555325 <+81>:    jmp    0x55555555530d <phase_2+57>
   0x0000555555555327 <+83>:    mov    0x18(%rsp),%rax
   0x000055555555532c <+88>:    xor    %fs:0x28,%rax
   0x0000555555555335 <+97>:    jne    0x55555555533e <phase_2+106>
   0x0000555555555337 <+99>:    add    $0x28,%rsp
   0x000055555555533b <+103>:   pop    %rbx
   0x000055555555533c <+104>:   pop    %rbp
   0x000055555555533d <+105>:   retq
   0x000055555555533e <+106>:   callq  0x555555554ef0 <__stack_chk_fail@plt>
End of assembler dump.
(gdb)
```

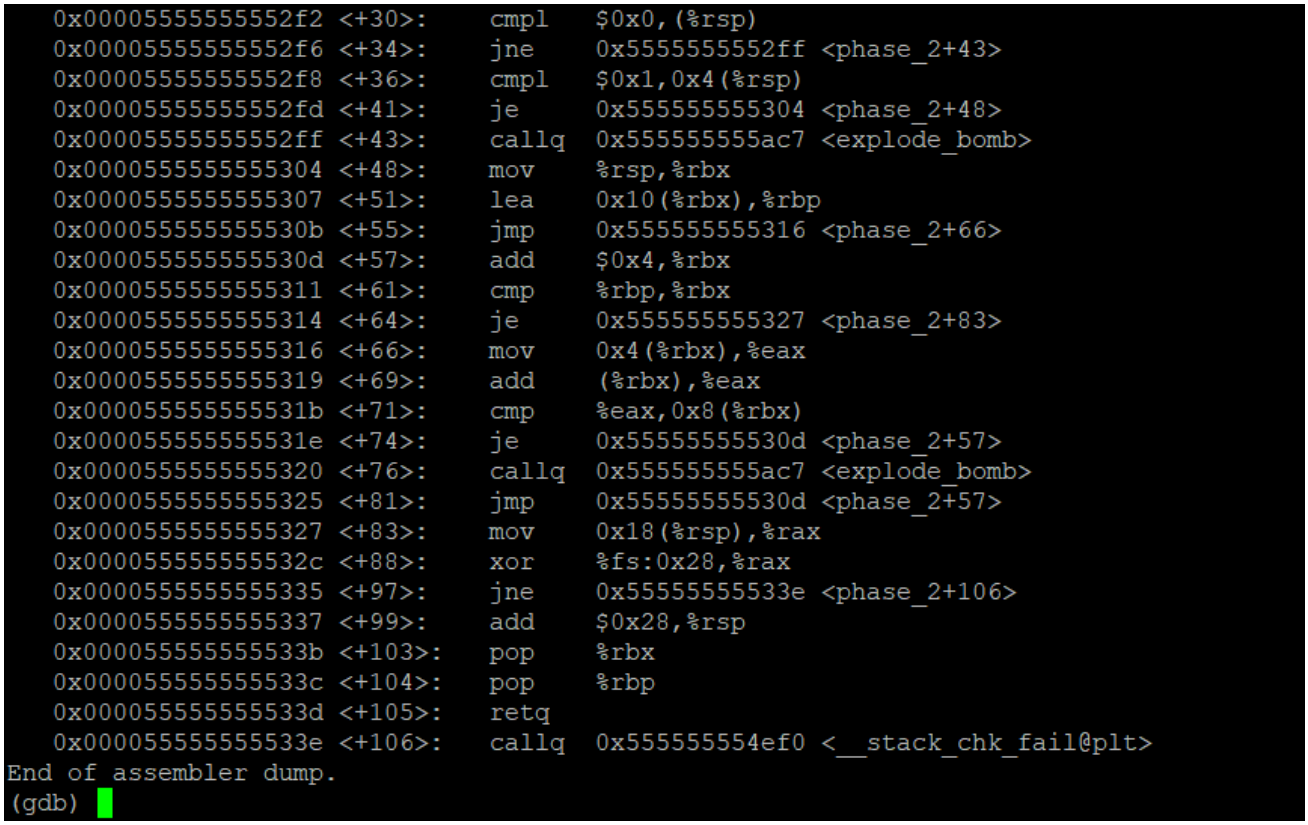
```
(gdb) disas read_six_numbers
Dump of assembler code for function read_six_numbers:
   0x0000555555555b03 <+0>:     sub    $0x8,%rsp
   0x0000555555555b07 <+4>:     mov    %rsi,%rdx
   0x0000555555555b0a <+7>:     lea    0x4(%rsi),%rcx
   0x0000555555555b0e <+11>:    lea    0x14(%rsi),%rax
   0x0000555555555b12 <+15>:    push   %rax
   0x0000555555555b13 <+16>:    lea    0x10(%rsi),%rax
   0x0000555555555b17 <+20>:    push   %rax
   0x0000555555555b18 <+21>:    lea    0xc(%rsi),%r9
   0x0000555555555b1c <+25>:    lea    0x8(%rsi),%r8
   0x0000555555555b20 <+29>:    lea    0x12c2(%rip),%rsi        # 0x555555556de9
   0x0000555555555b27 <+36>:    mov    $0x0,%eax
   0x0000555555555b2c <+41>:    callq  0x555555554f90 <__isoc99_sscanf@plt>
   0x0000555555555b31 <+46>:    add    $0x10,%rsp
   0x0000555555555b35 <+50>:    cmp    $0x5,%eax
   0x0000555555555b38 <+53>:    jle    0x555555555b3f <read_six_numbers+60>
   0x0000555555555b3a <+55>:    add    $0x8,%rsp
   0x0000555555555b3e <+59>:    retq
   0x0000555555555b3f <+60>:    callq  0x555555555ac7 <explode_bomb>
End of assembler dump.
(gdb)
```

read_six_numbers에서 sscanf를 호출한 뒤, 그 반환값이 5보다 같거나 작으면 폭탄이 터지는 것을 알 수 있다. 그리고 sscanf의 2번째 인자가 입력의 포멧을 결정하므로 read_six_numbers를 실행하면서 %rsi의 값을 보면

```
Breakpoint 2, 0x0000555555555b03 in read_six_numbers ()
(gdb) x/s $rsi
0x7fffffffe460: "\300\206uUUU"
(gdb) ni
0x0000555555555b07 in read_six_numbers ()
(gdb) x/s $rsi
0x7fffffffe460: "\300\206uUUU"
(gdb) ni
0x0000555555555b0a in read_six_numbers ()
(gdb) x/s $rsi
0x7fffffffe460: "\300\206uUUU"
(gdb) ni
0x0000555555555b0e in read_six_numbers ()
(gdb) x/s $rsi
0x7fffffffe460: "\300\206uUUU"
(gdb) ni
0x0000555555555b12 in read_six_numbers ()
(gdb) x/s $rsi
0x7fffffffe460: "\300\206uUUU"
(gdb) ni
0x0000555555555b13 in read_six_numbers ()
(gdb) x/s $rsi
0x7fffffffe460: "\300\206uUUU"
(gdb) ni
0x0000555555555b17 in read_six_numbers ()
(gdb) x/s $rsi
0x7fffffffe460: "\300\206uUUU"
(gdb) ni
0x0000555555555b18 in read_six_numbers ()
(gdb) x/s $rsi
0x7fffffffe460: "\300\206uUUU"
(gdb) ni
0x0000555555555b1c in read_six_numbers ()
(gdb) x/s $rsi
0x7fffffffe460: "\300\206uUUU"
(gdb) ni
0x0000555555555b20 in read_six_numbers ()
(gdb) x/s $rsi
0x7fffffffe460: "\300\206uUUU"
(gdb) ni
0x0000555555555b27 in read_six_numbers ()
(gdb) x/s $rsi
0x555555556de9: "%d %d %d %d %d %d"
(gdb)
```

총 6개의 정수를 입력받는다는 것을 알 수 있다.

다시 phase_2로 돌아와서 read_six_numbers를 호출한 이후의 코드를 보면



cmpl $0x0 (%rsp)

jne ... <phase_2+43>

을 통해 입력받은 첫번째 숫자(일반적으로 scanf에 주소값을 넘겨주므로, stack에 저장된다.)가 0이 아니면 폭탄이 터지는 것을 알 수 있다.

또한

cmpl $0x1 0x4(%rsp)

je ... <phase_2+48>

을 통해 입력받은 2번째 숫자가 1이 아니면 폭탄이 터지는 것을 알 수 있다.

mov %rsp %rbx에서 이제 rsp가 가리키는 값은 rbx가 가리키는 값이다. 즉 rbx에는 첫번째 값의 주소가 있다.

그리고 rbp에는 rbx에 0x10을 더한 값이 저장된다.

그리고 +66과 +69에서 %eax에는 1번째 숫자와 2번째 숫자 합이 저장되고, 이것을 3번째 숫자와 비교하여 같지 않으면 폭탄이 터진다.

같으면 +57로 분기하여 rbx에 0x4를 더하고 다시 위의 과정을 반복하므로 답은 다음과 같다

0 1 1 2 3 5

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/sys059/201924451/bomb6/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Public speaking is very easy.
Phase 1 defused. How about the next one?
0 1 1 2 3 5

Breakpoint 2, 0x0000555555555b03 in read_six_numbers ()
(gdb) c
Continuing.
That's number 2.  Keep going!
```

5. phase_3

```
Dump of assembler code for function phase_3:
   0x0000000000001343 <+0>:     sub    $0x18,%rsp
   0x0000000000001347 <+4>:     mov    %fs:0x28,%rax
   0x0000000000001350 <+13>:    mov    %rax,0x8(%rsp)
   0x0000000000001355 <+18>:    xor    %eax,%eax
   0x0000000000001357 <+20>:    lea    0x4(%rsp),%rcx
   0x000000000000135c <+25>:    mov    %rsp,%rdx
   0x000000000000135f <+28>:    lea    0x1a8f(%rip),%rsi        # 0x2df5
   0x0000000000001366 <+35>:    callq  0xf90 <__isoc99_sscanf@plt>
   0x000000000000136b <+40>:    cmp    $0x1,%eax
   0x000000000000136e <+43>:    jle    0x138d <phase_3+74>
   0x0000000000001370 <+45>:    cmpl   $0x7,(%rsp)
   0x0000000000001374 <+49>:    ja     0x1413 <phase_3+208>
   0x000000000000137a <+55>:    mov    (%rsp),%eax
   0x000000000000137d <+58>:    lea    0x179c(%rip),%rdx        # 0x2b20
   0x0000000000001384 <+65>:    movslq (%rdx,%rax,4),%rax
   0x0000000000001388 <+69>:    add    %rdx,%rax
   0x000000000000138b <+72>:    jmpq   *%rax
   0x000000000000138d <+74>:    callq  0x1ac7 <explode_bomb>
   0x0000000000001392 <+79>:    jmp    0x1370 <phase_3+45>
   0x0000000000001394 <+81>:    mov    $0x1c1,%eax
   0x0000000000001399 <+86>:    jmp    0x13a0 <phase_3+93>
   0x000000000000139b <+88>:    mov    $0x0,%eax
```

```
0x00000000000013a0 <+93>:    sub    $0x3da,%eax
0x00000000000013a5 <+98>:    add    $0x3af,%eax
0x00000000000013aa <+103>:   sub    $0x33f,%eax
0x00000000000013af <+108>:   add    $0x33f,%eax
0x00000000000013b4 <+113>:   sub    $0x33f,%eax
0x00000000000013b9 <+118>:   add    $0x33f,%eax
0x00000000000013be <+123>:   sub    $0x33f,%eax
0x00000000000013c3 <+128>:   cmpl   $0x5,(%rsp)
0x00000000000013c7 <+132>:   jg     0x13cf <phase_3+140>
0x00000000000013c9 <+134>:   cmp    %eax,0x4(%rsp)
0x00000000000013cd <+138>:   je     0x13d4 <phase_3+145>
0x00000000000013cf <+140>:   callq  0x1ac7 <explode_bomb>
0x00000000000013d4 <+145>:   mov    0x8(%rsp),%rax
0x00000000000013d9 <+150>:   xor    %fs:0x28,%rax
0x00000000000013e2 <+159>:   jne    0x141f <phase_3+220>
0x00000000000013e4 <+161>:   add    $0x18,%rsp
0x00000000000013e8 <+165>:   retq
0x00000000000013e9 <+166>:   mov    $0x0,%eax
0x00000000000013ee <+171>:   jmp    0x13a5 <phase_3+98>
0x00000000000013f0 <+173>:   mov    $0x0,%eax
0x00000000000013f5 <+178>:   jmp    0x13aa <phase_3+103>
0x00000000000013f7 <+180>:   mov    $0x0,%eax
0x00000000000013fc <+185>:   jmp    0x13af <phase_3+108>
0x00000000000013fe <+187>:   mov    $0x0,%eax
0x0000000000001403 <+192>:   jmp    0x13b4 <phase_3+113>
0x0000000000001405 <+194>:   mov    $0x0,%eax
0x000000000000140a <+199>:   jmp    0x13b9 <phase_3+118>
0x000000000000140c <+201>:   mov    $0x0,%eax
0x0000000000001411 <+206>:   jmp    0x13be <phase_3+123>
0x0000000000001413 <+208>:   callq  0x1ac7 <explode_bomb>
0x0000000000001418 <+213>:   mov    $0x0,%eax
0x000000000000141d <+218>:   jmp    0x13c3 <phase_3+128>
0x000000000000141f <+220>:   callq  0xef0 <__stack_chk_fail@plt>
```

sscanf를 호출하므로, phase_3를 한줄씩 실행하면서 %rsi 값을 확인하였다.

```
End of assembler dump.
(gdb) b phase_3
Breakpoint 1 at 0x1343
(gdb) r
Starting program: /home/sys059/201924451/bomb6/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Public speaking is very easy.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2.  Keep going!
test

Breakpoint 1, 0x0000555555555343 in phase_3 ()
(gdb) x/s $rsi
0x3:    <error: Cannot access memory at address 0x3>
(gdb) ni
0x0000555555555347 in phase_3 ()
(gdb) x/s $rsi
0x3:    <error: Cannot access memory at address 0x3>
(gdb) ni
0x0000555555555350 in phase_3 ()
(gdb) x/s $rsi
0x3:    <error: Cannot access memory at address 0x3>
(gdb) ni
0x0000555555555355 in phase_3 ()
(gdb) x/s $rsi
0x3:    <error: Cannot access memory at address 0x3>
(gdb) ni
0x0000555555555357 in phase_3 ()
(gdb) x/s $rsi
0x3:    <error: Cannot access memory at address 0x3>
(gdb) ni
0x000055555555535c in phase_3 ()
(gdb) x/s $rsi
0x3:    <error: Cannot access memory at address 0x3>
(gdb) ni
0x000055555555535f in phase_3 ()
(gdb) x/s $rsi
0x3:    <error: Cannot access memory at address 0x3>
(gdb) ni
0x0000555555555366 in phase_3 ()
(gdb) x/s $rsi
0x555555556df5: "%d %d"
(gdb)
```

```
   0x000055555555535c <+25>:    mov    %rsp,%rdx
   0x000055555555535f <+28>:    lea    0x1a8f(%rip),%rsi        # 0x555555556df5
=> 0x0000555555555366 <+35>:    callq  0x555555554f90 <__isoc99_sscanf@plt>
   0x000055555555536b <+40>:    cmp    $0x1,%eax
   0x000055555555536e <+43>:    jle    0x55555555538d <phase_3+74>
   0x0000555555555370 <+45>:    cmpl   $0x7,(%rsp)
```

따라서 두개의 정수 값을 받는 것을 알 수 있다.

또한 <+40> <+43>에서 sscanf의 반환값과 0x1을 비교하여 같거나 작으면 폭탄이 터지도록 되어있다.

그리고 <+45>와 <+49>에서 첫번째 입력값이 7보다 크면 폭탄이 터지도록 되어있다. (각각 +74와 +208로 분기하는데, 이는 explode_bomb를 실행한다.)

<+88>부터 <+134>까지 내용을 보면 eax를 0으로 초기화하고 많은 sub와 add를 한 후 두번째 입력값과 비교하여 같으면 explode_bomb를 건너뛰는 것을 알 수 있다. 또한 <+128> 과 <+132>에서 첫번째 값이 5보다 크면 폭탄이 터지도록 되어있다.

```
0x000055555555539b <+88>:    mov    $0x0,%eax
0x00005555555553a0 <+93>:    sub    $0x3da,%eax
0x00005555555553a5 <+98>:    add    $0x3af,%eax
0x00005555555553aa <+103>:   sub    $0x33f,%eax
0x00005555555553af <+108>:   add    $0x33f,%eax
0x00005555555553b4 <+113>:   sub    $0x33f,%eax
0x00005555555553b9 <+118>:   add    $0x33f,%eax
0x00005555555553be <+123>:   sub    $0x33f,%eax
0x00005555555553c3 <+128>:   cmpl   $0x5,(%rsp)
0x00005555555553c7 <+132>:   jg     0x5555555553cf <phase_3+140>
0x00005555555553c9 <+134>:   cmp    %eax,0x4(%rsp)
0x00005555555553cd <+138>:   je     0x5555555553d4 <phase_3+145>
0x00005555555553cf <+140>:   callq  0x555555555ac7 <explode_bomb>
```

0x0 – 0x3da + 0x3af – 0x33f + 0x33f – 0x33f + 0x33f – 0x33f = 0x0-0x3da+0x3af-0x33f = -0x36a = -874

```
(gdb) ni
0x000055555555535c in phase_3 ()
(gdb) ni
0x000055555555535f in phase_3 ()
(gdb) ni
0x0000555555555366 in phase_3 ()
(gdb) ni
0x000055555555536b in phase_3 ()
(gdb) ni
0x000055555555536e in phase_3 ()
(gdb) ni
0x0000555555555370 in phase_3 ()
(gdb) ni
0x0000555555555374 in phase_3 ()
(gdb) ni
0x000055555555537a in phase_3 ()
(gdb) ni
0x000055555555537d in phase_3 ()
(gdb) x/s $rdx
0x7fffffffe484: "\005"
(gdb) ni
0x0000555555555384 in phase_3 ()
(gdb) ni
0x0000555555555388 in phase_3 ()
(gdb) ni
0x000055555555538b in phase_3 ()
(gdb) ni
0x000055555555539b in phase_3 ()
(gdb) ni
0x00005555555553a0 in phase_3 ()
(gdb) ni
0x00005555555553a5 in phase_3 ()
(gdb) ni
0x00005555555553aa in phase_3 ()
(gdb) ni
0x00005555555553af in phase_3 ()
(gdb) ni
0x00005555555553b4 in phase_3 ()
(gdb) ni
0x00005555555553b9 in phase_3 ()
(gdb) ni
0x00005555555553be in phase_3 ()
(gdb) ni
0x00005555555553c3 in phase_3 ()
(gdb) x/s $eax
0xffffffffffffc96:     <error: Cannot access memory at address 0xffffffffffffc96>
(gdb) x/s $rax
0xfffffc96:     <error: Cannot access memory at address 0xfffffc96>
(gdb)
```

phase_3에 breakpoint를 걸어서 ni를 계속실행하면서 %rax 레지스터의 값을 보면 rax에 0xfffffc96이 저장되어있는 것을 알 수 있다. (1을 넣었을 때)

```
   0x00055555555553e8 <+165>:   retq
   0x00055555555553e9 <+166>:   mov    $0x0,%eax
   0x00055555555553ee <+171>:   jmp    0x5555555553a5 <phase_3+98>
   0x00055555555553f0 <+173>:   mov    $0x0,%eax
   0x00055555555553f5 <+178>:   jmp    0x5555555553aa <phase_3+103>
   0x00055555555553f7 <+180>:   mov    $0x0,%eax
   0x00055555555553fc <+185>:   jmp    0x5555555553af <phase_3+108>
   0x00055555555553fe <+187>:   mov    $0x0,%eax
   0x0005555555555403 <+192>:   jmp    0x5555555553b4 <phase_3+113>
   0x0005555555555405 <+194>:   mov    $0x0,%eax
---Type <return> to continue, or q <return> to quit---
   0x00055555555540a <+199>:   jmp    0x5555555553b9 <phase_3+118>
   0x00055555555540c <+201>:   mov    $0x0,%eax
   0x0005555555555411 <+206>:   jmp    0x5555555553be <phase_3+123>
   0x0005555555555413 <+208>:   callq  0x555555555ac7 <explode_bomb>
   0x0005555555555418 <+213>:   mov    $0x0,%eax
   0x00055555555541d <+218>:   jmp    0x5555555553c3 <phase_3+128>
   0x00055555555541f <+220>:   callq  0x555555554ef0 <__stack_chk_fail@plt>
End of assembler dump.
(gdb)
```

retq 이후 부분을 보면 eax에 0을 넣고 sub와 add하는 중간 부분으로 분기하는 것을 반복하고 있다.

```
   0x000055555555537a <+55>:    mov    (%rsp),%eax
   0x000055555555537d <+58>:    lea    0x179c(%rip),%rdx        # 0x555555556b20
   0x0005555555555384 <+65>:    movslq (%rdx,%rax,4),%rax
   0x0005555555555388 <+69>:    add    %rdx,%rax
   0x000055555555538b <+72>:    jmpq   *%rax
```

이는 jmpq *%rax와 관련있어보인다.

만약에 2를 넣고 %rax를 보면

```
(gdb) ni
0x0000555555555370 in phase_3 ()
(gdb) ni
0x0000555555555374 in phase_3 ()
(gdb) ni
0x000055555555537a in phase_3 ()
(gdb) ni
0x000055555555537d in phase_3 ()
(gdb) ni
0x0000555555555384 in phase_3 ()
(gdb) ni
0x0000555555555388 in phase_3 ()
(gdb) ni
0x000055555555538b in phase_3 ()
(gdb) ni
0x00005555555553e9 in phase_3 ()
(gdb) ni
0x00005555555553ee in phase_3 ()
(gdb) ni
0x00005555555553a5 in phase_3 ()
(gdb) ni
0x00005555555553aa in phase_3 ()
(gdb) ni
0x00005555555553af in phase_3 ()
(gdb) ni
0x00005555555553b4 in phase_3 ()
(gdb) ni
0x00005555555553b9 in phase_3 ()
(gdb) ni
0x00005555555553be in phase_3 ()
(gdb) ni
0x00005555555553c3 in phase_3 ()
(gdb) x/s $eax
0x70:   <error: Cannot access memory at address 0x70>
(gdb) x/s $rax
0x70:   <error: Cannot access memory at address 0x70>
(gdb)
```

%rax에 있는 값이 다른 것을 알 수 있다. (38b에서 3e9으로 점프한다)

따라서 답은

2 112

이다(0x70 = 112)

4. phase_4

```
(gdb) disassemble phase_4
Dump of assembler code for function phase_4:
   0x0000555555555463 <+0>:     sub    $0x18,%rsp
   0x0000555555555467 <+4>:     mov    %fs:0x28,%rax
   0x0000555555555470 <+13>:    mov    %rax,0x8(%rsp)
   0x0000555555555475 <+18>:    xor    %eax,%eax
   0x0000555555555477 <+20>:    lea    0x4(%rsp),%rcx
   0x000055555555547c <+25>:    mov    %rsp,%rdx
   0x000055555555547f <+28>:    lea    0x196f(%rip),%rsi        # 0x555555556df5
   0x0000555555555486 <+35>:    callq  0x555555554f90 <__isoc99_sscanf@plt>
   0x000055555555548b <+40>:    cmp    $0x2,%eax
   0x000055555555548e <+43>:    jne    0x555555555496 <phase_4+51>
   0x0000555555555490 <+45>:    cmpl   $0xe,(%rsp)
   0x0000555555555494 <+49>:    jbe    0x55555555549b <phase_4+56>
   0x0000555555555496 <+51>:    callq  0x555555555ac7 <explode_bomb>
   0x000055555555549b <+56>:    mov    $0xe,%edx
   0x00005555555554a0 <+61>:    mov    $0x0,%esi
   0x00005555555554a5 <+66>:    mov    (%rsp),%edi
   0x00005555555554a8 <+69>:    callq  0x555555555424 <func4>
   0x00005555555554ad <+74>:    cmp    $0x2,%eax
   0x00005555555554b0 <+77>:    jne    0x5555555554b9 <phase_4+86>
   0x00005555555554b2 <+79>:    cmpl   $0x2,0x4(%rsp)
   0x00005555555554b7 <+84>:    je     0x5555555554be <phase_4+91>
   0x00005555555554b9 <+86>:    callq  0x555555555ac7 <explode_bomb>
   0x00005555555554be <+91>:    mov    0x8(%rsp),%rax
   0x00005555555554c3 <+96>:    xor    %fs:0x28,%rax
   0x00005555555554cc <+105>:   jne    0x5555555554d3 <phase_4+112>
   0x00005555555554ce <+107>:   add    $0x18,%rsp
   0x00005555555554d2 <+111>:   retq
   0x00005555555554d3 <+112>:   callq  0x555555554ef0 <__stack_chk_fail@plt>
End of assembler dump.
(gdb)
```

이번에도 %rsi의 값을 보면서 sscanf에 어떤 인자를 넘겨주는지 보자.

```
Breakpoint 2, 0x0000555555555463 in phase_4 ()
(gdb) ni
0x0000555555555467 in phase_4 ()
(gdb) ni
0x0000555555555470 in phase_4 ()
(gdb) ni
0x0000555555555475 in phase_4 ()
(gdb) ni
0x0000555555555477 in phase_4 ()
(gdb) ni
0x000055555555547c in phase_4 ()
(gdb) ni
0x000055555555547f in phase_4 ()
(gdb) x/s $rsi
0x4:    <error: Cannot access memory at address 0x4>
(gdb) ni
0x0000555555555486 in phase_4 ()
(gdb) x/s $rsi
0x555555556df5: "%d %d"
(gdb)
```

이번에도 2개의 정수를 입력받는다는 것을 알 수 있다.

```
   0x000000000000148b <+40>:    cmp    $0x2,%eax
   0x000000000000148e <+43>:    jne    0x1496 <phase_4+51>
```

<+40>과 <+43>을 통해 sscanf의 반환값이 2가 아니면 explode_bomb 호출하는 명령으로 분기하는 것을 알 수 있다.

```
   0x000000000000148e <+43>:    jne    0x1496 <phase_4+51>
   0x0000000000001490 <+45>:    cmpl   $0xe,(%rsp)
   0x0000000000001494 <+49>:    jbe    0x149b <phase_4+56>
   0x0000000000001496 <+51>:    callq  0x1ac7 <explode_bomb>
   0x000000000000149b <+56>:    mov    $0xe,%edx
```

또한 <+45>와 <+49>를 통해 첫번째 숫자 값이 0xe보다 작거나 같아야 explode_bomb를 건너뜀을 알 수 있다. 즉 15보다 작아야한다.

```
   0x000000000000149b <+56>:    mov    $0xe,%edx
   0x00000000000014a0 <+61>:    mov    $0x0,%esi
   0x00000000000014a5 <+66>:    mov    (%rsp),%edi
   0x00000000000014a8 <+69>:    callq  0x1424 <func4>
   0x00000000000014ad <+74>:    cmp    $0x2,%eax
   0x00000000000014b0 <+77>:    jne    0x14b9 <phase_4+86>
   0x00000000000014b2 <+79>:    cmpl   $0x2,0x4(%rsp)
   0x00000000000014b7 <+84>:    je     0x14be <phase_4+91>
   0x00000000000014b9 <+86>:    callq  0x1ac7 <explode_bomb>
   0x00000000000014be <+91>:    mov    0x8(%rsp),%rax
   0x00000000000014c3 <+96>:    xor    %fs:0x28,%rax
   0x00000000000014cc <+105>:   jne    0x14d3 <phase_4+112>
   0x00000000000014ce <+107>:   add    $0x18,%rsp
   0x00000000000014d2 <+111>:   retq
   0x00000000000014d3 <+112>:   callq  0xef0 <__stack_chk_fail@plt>
End of assembler dump.
(gdb)
```

이후에 %edx에 15를, %esi에 0을, %edi에 첫번째 숫자를 넣고 func4를 호출

(즉 func4(첫번째 숫자,0,15))하고 그 반환값이 2가 아니면 explode_bomb로 분기한다.

그리고 두번째 숫자와 2를 비교하여 같으면 explode_bomb를 건너뛴다. 즉 두번째 숫자는 2다.

이제 func4를 살펴보면

```
(gdb) disassemble func4
Dump of assembler code for function func4:
   0x0000000000001424 <+0>:     sub    $0x8,%rsp
   0x0000000000001428 <+4>:     mov    %edx,%eax
   0x000000000000142a <+6>:     sub    %esi,%eax
   0x000000000000142c <+8>:     mov    %eax,%ecx
   0x000000000000142e <+10>:    shr    $0x1f,%ecx
   0x0000000000001431 <+13>:    add    %eax,%ecx
   0x0000000000001433 <+15>:    sar    %ecx
   0x0000000000001435 <+17>:    add    %esi,%ecx
   0x0000000000001437 <+19>:    cmp    %edi,%ecx
   0x0000000000001439 <+21>:    jg     0x1449 <func4+37>
   0x000000000000143b <+23>:    mov    $0x0,%eax
   0x0000000000001440 <+28>:    cmp    %edi,%ecx
   0x0000000000001442 <+30>:    jl     0x1455 <func4+49>
   0x0000000000001444 <+32>:    add    $0x8,%rsp
   0x0000000000001448 <+36>:    retq
   0x0000000000001449 <+37>:    lea    -0x1(%rcx),%edx
   0x000000000000144c <+40>:    callq  0x1424 <func4>
   0x0000000000001451 <+45>:    add    %eax,%eax
   0x0000000000001453 <+47>:    jmp    0x1444 <func4+32>
   0x0000000000001455 <+49>:    lea    0x1(%rcx),%esi
   0x0000000000001458 <+52>:    callq  0x1424 <func4>
   0x000000000000145d <+57>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000001461 <+61>:    jmp    0x1444 <func4+32>
End of assembler dump.
(gdb)
```

재귀함수임을 알 수 있다.

이 함수를 분석하면

```
   0x0000000000001428 <+4>:     mov    %edx,%eax
   0x000000000000142a <+6>:     sub    %esi,%eax
   0x000000000000142c <+8>:     mov    %eax,%ecx
   0x000000000000142e <+10>:    shr    $0x1f,%ecx
   0x0000000000001431 <+13>:    add    %eax,%ecx
   0x0000000000001433 <+15>:    sar    %ecx
   0x0000000000001435 <+17>:    add    %esi,%ecx
   0x0000000000001437 <+19>:    cmp    %edi,%ecx
   0x0000000000001439 <+21>:    jg     0x1449 <func4+37>
```

%rax = 3번째 인자값 - 2번째 인자값 (+4~+8)

%ecx = ((%rax >>(logical) 31) + %rax)>>1(arithmetic) + 2번째 인자값

이것을 첫번째 인자값과 비교하여 크면 <func4+37>로 분기한다.

func4(첫번째 숫자, 0, 15)인 경우

%rax = 15-0=15

%ecx = 15>>1 + 0 = 15/2 (반내림) = 7

즉 첫번째 인자값이 7보다 작으면 <func4+37>로 분기한다.

```
   0x000000000000143b <+23>:    mov    $0x0,%eax
   0x0000000000001440 <+28>:    cmp    %edi,%ecx
   0x0000000000001442 <+30>:    jl     0x1455 <func4+49>
   0x0000000000001444 <+32>:    add    $0x8,%rsp
   0x0000000000001448 <+36>:    retq
```

그 후 %rax에 0을 넣고 %ecx와 %edi를 비교하여 %ecx가 작으면 <func4+49>로 분기한다.

func4(첫번째 숫자,0,15)인 경우 첫번째 인자값이 7보다 크면 <func4+49>로 분기한다.

즉 정리하면 입력받은 첫번째 숫자가 7보다 작으면 <func4+37>로 분기하고, 7보다 크면 <func4+49>로 분기하고, 입력받은 첫번째 숫자가 7이면 0을 리턴한다.

```
   0x0000000000001444 <+32>:    add    $0x8,%rsp
   0x0000000000001448 <+36>:    retq
   0x0000000000001449 <+37>:    lea    -0x1(%rcx),%edx
   0x000000000000144c <+40>:    callq  0x1424 <func4>
   0x0000000000001451 <+45>:    add    %eax,%eax
   0x0000000000001453 <+47>:    jmp    0x1444 <func4+32>
   0x0000000000001455 <+49>:    lea    0x1(%rcx),%esi
   0x0000000000001458 <+52>:    callq  0x1424 <func4>
   0x000000000000145d <+57>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000001461 <+61>:    jmp    0x1444 <func4+32>
End of assembler dump.
(gdb)
```

retq 이후를 분석해보면

*참고 : %rax = 3번째 인자값 - 2번째 인자값 (+4~+8)

%ecx = ((%rax >>(logical) 31) + %rax)>>1(arithmetic) + 2번째 인자값

<+37>~<+47> : 3번째 인자를 %rcx에 1을 뺀 값으로 하여 func4를 호출하고 eax * 2를 하여 리턴한다,

(func4(첫번째 인자, 두번째 인자, %ecx - 1)) #if 첫번째 인자 <%ecx

<+49>~<+62> : 2번째 인자를 %rcx에서 1을 더한 값으로 하여 func4를 호출하고 eax*2 + 1하여 리턴한다.

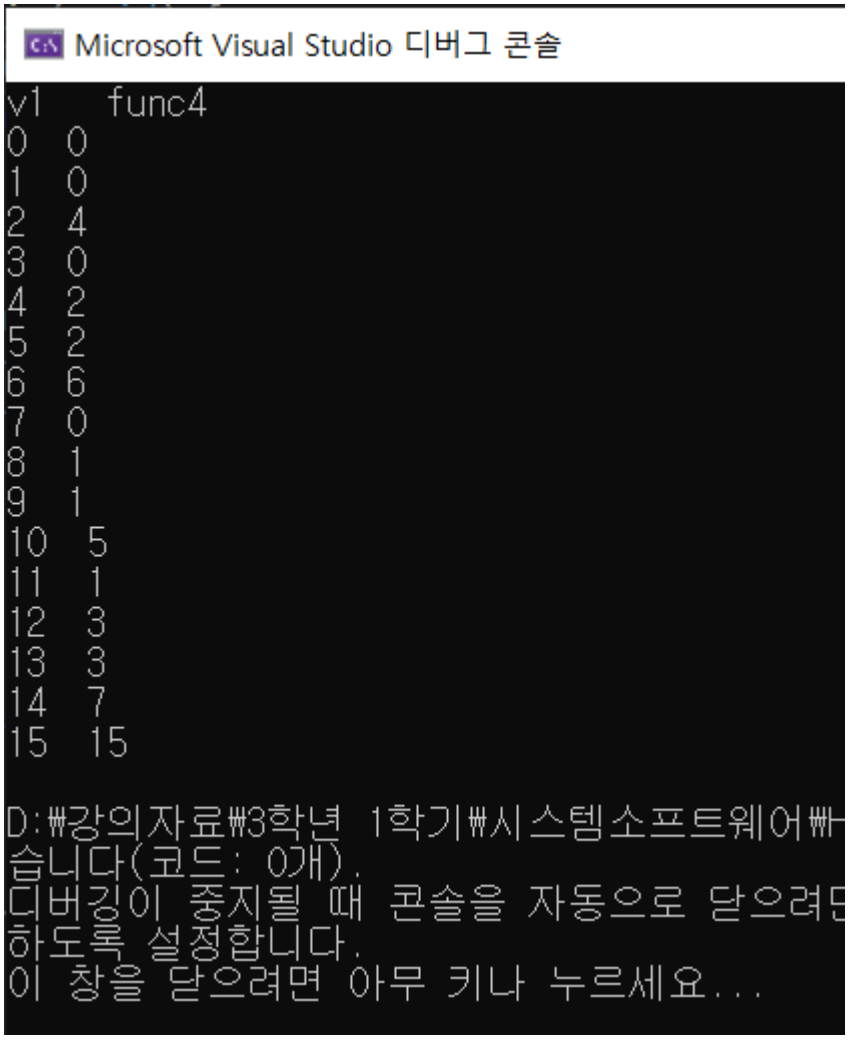
(func4(첫번째 인자,%ecx + 1, 세번째 인자)) #if 첫번째 인자 >%ecx

위 재귀함수를 c++언어로 구현하면

```cpp
#include <iostream>
using namespace std;
int func4(int v1, int v2, int v3);
int main(void) {
    cout << "v1" << "   func4" << endl;
    for (int i = 0; i <= 15; ++i) {
        cout << i << "  " << func4(i, 0, 15) << endl;
    }
}
int func4(int v1, int v2, int v3) {
    int rax = v3 - v2;
    int ecx = (((int)((unsigned int)rax >> 31) + rax) >> 1) + v2;

    if (v1 < ecx) {
        int ret = func4(v1, v2, ecx - 1);
        return ret + ret;
    }
    else if (v1 > ecx) {
        int ret = func4(v1, ecx + 1, v3);
        return ret + ret + 1;
    }
    return 0;
}
```

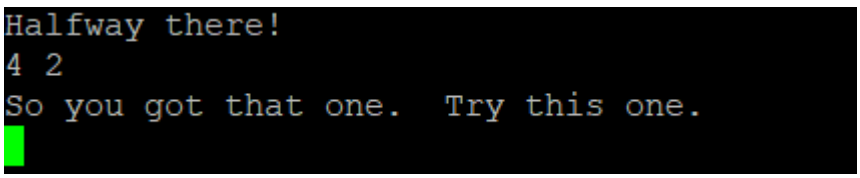
이고 메인 함수를 실행하여 첫번째 숫자에 따른 func4의 반환값을 보면



따라서 첫번째 숫자의 답은 4나 5임을 알 수 있다.

따라서 정답은

4 2 이다.



6. phase_5

```
Dump of assembler code for function phase_5:
   0x00005555555554d8 <+0>:     sub    $0x18,%rsp
   0x00005555555554dc <+4>:     mov    %fs:0x28,%rax
   0x00005555555554e5 <+13>:    mov    %rax,0x8(%rsp)
   0x00005555555554ea <+18>:    xor    %eax,%eax
   0x00005555555554ec <+20>:    lea    0x4(%rsp),%rcx
   0x00005555555554f1 <+25>:    mov    %rsp,%rdx
   0x00005555555554f4 <+28>:    lea    0x18fa(%rip),%rsi        # 0x555555556df5
   0x00005555555554fb <+35>:    callq  0x555555554f90 <__isoc99_sscanf@plt>
   0x0000555555555500 <+40>:    cmp    $0x1,%eax
   0x0000555555555503 <+43>:    jle    0x55555555555f <phase_5+135>
   0x0000555555555505 <+45>:    mov    (%rsp),%eax
   0x0000555555555508 <+48>:    and    $0xf,%eax
   0x000055555555550b <+51>:    mov    %eax,(%rsp)
   0x000055555555550e <+54>:    cmp    $0xf,%eax
   0x0000555555555511 <+57>:    je     0x555555555545 <phase_5+109>
   0x0000555555555513 <+59>:    mov    $0x0,%ecx
   0x0000555555555518 <+64>:    mov    $0x0,%edx
   0x000055555555551d <+69>:    lea    0x161c(%rip),%rsi        # 0x555555556b40 <array.3417>
   0x0000555555555524 <+76>:    add    $0x1,%edx
   0x0000555555555527 <+79>:    cltq
   0x0000555555555529 <+81>:    mov    (%rsi,%rax,4),%eax
   0x000055555555552c <+84>:    add    %eax,%ecx
   0x000055555555552e <+86>:    cmp    $0xf,%eax
   0x0000555555555531 <+89>:    jne    0x555555555524 <phase_5+76>
   0x0000555555555533 <+91>:    movl   $0xf,(%rsp)
   0x000055555555553a <+98>:    cmp    $0xf,%edx
   0x000055555555553d <+101>:   jne    0x555555555545 <phase_5+109>
   0x000055555555553f <+103>:   cmp    %ecx,0x4(%rsp)
   0x0000555555555543 <+107>:   je     0x55555555554a <phase_5+114>
   0x0000555555555545 <+109>:   callq  0x555555555ac7 <explode_bomb>
   0x000055555555554a <+114>:   mov    0x8(%rsp),%rax
   0x000055555555554f <+119>:   xor    %fs:0x28,%rax
   0x0000555555555558 <+128>:   jne    0x555555555566 <phase_5+142>
   0x000055555555555a <+130>:   add    $0x18,%rsp
   0x000055555555555e <+134>:   retq
   0x000055555555555f <+135>:   callq  0x555555555ac7 <explode_bomb>
   0x0000555555555564 <+140>:   jmp    0x555555555505 <phase_5+45>
   0x0000555555555566 <+142>:   callq  0x555555554ef0 <__stack_chk_fail@plt>
End of assembler dump.
(gdb)
```

이번에도 %rsi의 값을 sscanf를 호출하기 직전까지 실행한 후 보면

```
Breakpoint 2, 0x00005555555554d8 in phase_5 ()
(gdb) ni
0x00005555555554dc in phase_5 ()
(gdb) ni
0x00005555555554e5 in phase_5 ()
(gdb) ni
0x00005555555554ea in phase_5 ()
(gdb) ni
0x00005555555554ec in phase_5 ()
(gdb) ni
0x00005555555554f1 in phase_5 ()
(gdb) ni
0x00005555555554f4 in phase_5 ()
(gdb) x/s $rsi
0x5:    <error: Cannot access memory at address 0x5>
(gdb) ni
0x00005555555554fb in phase_5 ()
(gdb) x/s $rsi
0x555555556df5: "%d %d"
(gdb)
```

이번에도 두개의 정수를 입력받는 것을 알 수 있다.

```
0x0000555555555500 <+40>:    cmp    $0x1,%eax
0x0000555555555503 <+43>:    jle    0x55555555555f <phase_5+135>
```

만약 입력받은 정수의 개수가 1보다 같거나 작으면 explode_bomb로 분기하는 것을 알 수 있다.

```
0x0000555555555505 <+45>:    mov    (%rsp),%eax
0x0000555555555508 <+48>:    and    $0xf,%eax
0x000055555555550b <+51>:    mov    %eax,(%rsp)
0x000055555555550e <+54>:    cmp    $0xf,%eax
0x0000555555555511 <+57>:    je     0x555555555545 <phase_5+109>
0x0000555555555513 <+59>:    mov    $0x0,%ecx
0x0000555555555518 <+64>:    mov    $0x0,%edx
0x000055555555551d <+69>:    lea    0x161c(%rip),%rsi        # 0x555555556b40 <array.3417>
0x0000555555555524 <+76>:    add    $0x1,%edx
0x0000555555555527 <+79>:    cltq
```

그 이후를 보면 입력받은 첫번째 숫자를 %eax에 옮기고, eax와 0xf를 and하여 eax에 저장한다. 그리고 eax와 0xf를 비교하여 같으면 explode_bomb로 분기한다. 즉 입력한 첫번째 숫자는 16진수로 바꾸었을 때 0x.....f가되면 안된다.(즉 어떤 것을 입력하던지, 실제로 계산에 사용하는 숫자는 0~15범위이다.)

그리고 %ecx = 0, %edx = 0이 되며 %rsi값은 gdb를 통해 보면

```
(gdb) x/64d $rsi
0x555555556b40 <array.3417>:    10      0       0       0       2       0       0       0
0x555555556b48 <array.3417+8>:  14      0       0       0       7       0       0       0
0x555555556b50 <array.3417+16>: 8       0       0       0       12      0       0       0
0x555555556b58 <array.3417+24>: 15      0       0       0       11      0       0       0
0x555555556b60 <array.3417+32>: 0       0       0       0       4       0       0       0
0x555555556b68 <array.3417+40>: 1       0       0       0       13      0       0       0
0x555555556b70 <array.3417+48>: 3       0       0       0       9       0       0       0
0x555555556b78 <array.3417+56>: 6       0       0       0       5       0       0       0
```

배열임을 알 수 있다.

```
0x0000555555555529 <+81>:    mov    (%rsi,%rax,4),%eax
0x000055555555552c <+84>:    add    %eax,%ecx
0x000055555555552e <+86>:    cmp    $0xf,%eax
0x0000555555555531 <+89>:    jne    0x555555555524 <phase_5+76>
0x0000555555555533 <+91>:    movl   $0xf,(%rsp)
0x000055555555553a <+98>:    cmp    $0xf,%edx
0x000055555555553d <+101>:   jne    0x555555555545 <phase_5+109>
0x000055555555553f <+103>:   cmp    %ecx,0x4(%rsp)
0x0000555555555543 <+107>:   je     0x55555555554a <phase_5+114>
0x0000555555555545 <+109>:   callq  0x555555555ac7 <explode_bomb>
```

그 이후를 보면 <+76> 부터 <+89>까지 보면 %edx에 1씩 더하고 rax에 %rsi + 4*rax를 대입 즉 다음 배열을 대입한 다음 ecx에는 eax를 더하고 이 과정을 eax가 0xf이 아닐 때 동안 반복한다.

그 후 %edx와 0xf를 비교하여 같지 않으면 explode_bomb로 분기하고, 2번째 입력한 숫자와 %ecx를 비교하여 같으면 explode_bomb를 건너뛴다.

즉 이것을 c++ 코드로 구현하면

```
bool phase_5(int v1, int v2) {
    int rax = v1;
    rax = rax & 0xf;
    if (rax == 0xf) return false; //explode_bomb

    int ecx = 0;
    int edx = 0;
    int rsi[16] = {10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5};
    while (rax != 0xf) {
        edx++;
        rax = rsi[rax];
        ecx += rax;

    }
    if (edx != 0xf) return false;//explode_bomb
    cout << ecx;
    if (ecx == v2) return true;
    return false;//explode_bomb

}
```

와 비슷하고

```
int main(void) {
    cout << " v1 " << "ecx(answer of v2)" << endl;
    for(int i=0;i<=15;++i)
    {

        cout << i << " ";
        phase_5(i, 0);
        cout << endl;
    }


}
```

이렇게 메인함수를 구현하여 실행하면

```
Microsoft Visual Studio 디버그 콘솔
 v1 ecx(answer of v2)
0
1
2
3
4
5 115
6
7
8
9
10
11
12
13
14
15
```

따라서 답은 5 115이다.

7. phase_6

```
(gdb) disassemble phase_6
Dump of assembler code for function phase_6:
   0x000055555555556b <+0>:     push   %r13
   0x000055555555556d <+2>:     push   %r12
   0x000055555555556f <+4>:     push   %rbp
   0x0000555555555570 <+5>:     push   %rbx
   0x0000555555555571 <+6>:     sub    $0x68,%rsp
   0x0000555555555575 <+10>:    mov    %fs:0x28,%rax
   0x000055555555557e <+19>:    mov    %rax,0x58(%rsp)
   0x0000555555555583 <+24>:    xor    %eax,%eax
   0x0000555555555585 <+26>:    mov    %rsp,%r12
   0x0000555555555588 <+29>:    mov    %r12,%rsi
   0x000055555555558b <+32>:    callq  0x555555555b03 <read_six_numbers>
```

6개의 숫자를 받는다는 것을 알 수 있다. 그리고 mov %rsp %r12를 통해 나중에 read_six_numbers를 호출하고 나면 %r12 레지스터에 첫번째 숫자가 들어간다는 것을 알 수 있다.

```
   0x0000555555555590 <+37>:    mov    $0x0,%r13d
   0x0000555555555596 <+43>:    jmp    0x5555555555bd <phase_6+82>
```

6개의 숫자를 받은 다음 %r13d에 0을 넣고<phase_6+82>로 점프한다. 확인해보면

```
   0x00005555555555bd <+82>:    mov    %r12,%rbp
   0x00005555555555c0 <+85>:    mov    (%r12),%eax
   0x00005555555555c4 <+89>:    sub    $0x1,%eax
   0x00005555555555c7 <+92>:    cmp    $0x5,%eax
   0x00005555555555ca <+95>:    ja     0x555555555598 <phase_6+45>
```

첫번째 숫자의 주소를 %rbp로 옮기고, 첫번째 숫자를 %rax로 옮기고, %rax에 1을 뺀 다음, 5와 비교하여 크면 explode_bomb로 분기한다. 즉 첫번째 숫자는 7보다 작아야한다.

```
   0x00005555555555cc <+97>:    add    $0x1,%r13d
   0x00005555555555d0 <+101>:   cmp    $0x6,%r13d
   0x00005555555555d4 <+105>:   je     0x55555555560b <phase_6+160>
```

그 후 %r13d에 1을 더하고, 6과 비교하여 같으면 <phase_6+160>으로 분기한다.

(1)6과 같아서 분기했을 때

```
   0x000055555555560b <+160>:   mov    $0x0,%esi
   0x0000555555555610 <+165>:   jmp    0x5555555555f5 <phase_6+138>
```

%rsi에 0을 넣고 <phase_6+138>로 분기한다.

```
   0x00005555555555f5 <+138>:   mov    (%rsp,%rsi,4),%ecx
   0x00005555555555f8 <+141>:   mov    $0x1,%eax
   0x00005555555555fd <+146>:   lea    0x202c2c(%rip),%rdx        # 0x555555758230 <node1>
   0x0000555555555604 <+153>:   cmp    $0x1,%ecx
---Type <return> to continue, or q <return> to quit---
   0x0000555555555607 <+156>:   jg     0x5555555555db <phase_6+112>
   0x0000555555555609 <+158>:   jmp    0x5555555555e6 <phase_6+123>
```

%ecx에 입력받은 %rsi번째 값을 넣고, %rax에 1을 넣은다음, 1과 ecx를 비교하여 ecx가 크면 112로, 아니면 123으로 분기한다.

```
(gdb) x/32w 0x204230
0x204230 <node1>:       687     1       2114112 0
0x204240 <node2>:       298     2       2114128 0
0x204250 <node3>:       922     3       2114144 0
0x204260 <node4>:       943     4       2114160 0
0x204270 <node5>:       339     5       2113808 0
0x204280 <host_table>:  11855   0       11863   0
0x204290 <host_table+16>:       11889   0       11915   0
0x2042a0 <host_table+32>:       11940   0       0       0
(gdb)
```

```
0000000000204110 <node6>:
  204110:       47 00 00                rex.RXB add %r8b,(%r8)
  204113:       00 06                   add    %al,(%rsi)
        ...
```

```
(gdb) x/32w 0x204110
0x204110 <node6>:       71      6       0       0
0x204120 <user_password>:       1802789753      1900374600      812216371       1244685431
0x204130 <user_password+16>:    1382238794      0       959524914       892613682
0x204140 <userid+8>:    49      6       0       0
0x204150 <n1>:  36      0       2113904 0
0x204160 <n1+16>:       2113936 0       0       0
0x204170 <n21>: 8       0       2114032 0
0x204180 <n21+16>:      2113968 0       0       0
```

node에는 다음과 같은 값이 있다

(1-1).112로 분기했을 때 (ecx>1)

```
   0x00005555555555db <+112>:   mov    0x8(%rdx),%rdx
   0x00005555555555df <+116>:   add    $0x1,%eax
   0x00005555555555e2 <+119>:   cmp    %ecx,%eax
   0x00005555555555e4 <+121>:   jne    0x5555555555db <phase_6+112>
```

rdx가 다음 값(node)을 가리키도록 하고, eax에 1을 더한 후, ecx와 eax가

같지 않을 때 까지 반복한다.

```
   0x00005555555555e6 <+123>:   mov    %rdx,0x20(%rsp,%rsi,8)
   0x00005555555555eb <+128>:   add    $0x1,%rsi
   0x00005555555555ef <+132>:   cmp    $0x6,%rsi
   0x00005555555555f3 <+136>:   je     0x555555555612 <phase_6+167>
   0x00005555555555f5 <+138>:   mov    (%rsp,%rsi,4),%ecx
   0x00005555555555f8 <+141>:   mov    $0x1,%eax
   0x00005555555555fd <+146>:   lea    0x202c2c(%rip),%rdx        # 0x555555758230 <node1>
   0x0000555555555604 <+153>:   cmp    $0x1,%ecx
   0x0000555555555607 <+156>:   jg     0x5555555555db <phase_6+112>
---Type <return> to continue, or q <return> to quit---
   0x0000555555555609 <+158>:   jmp    0x5555555555e6 <phase_6+123>
```

%rsp + 8*%rsi + 0x20에 rdx를 저장한다.(배열로 생각하면 double rsp[%rsi + 4])

rsi에 1을 더한 후 rsi가 6이면 167로 분기하고 그렇지 않으면 %rsi값을 0으로 초기화하지 않고 (1)과정을 반복한다.

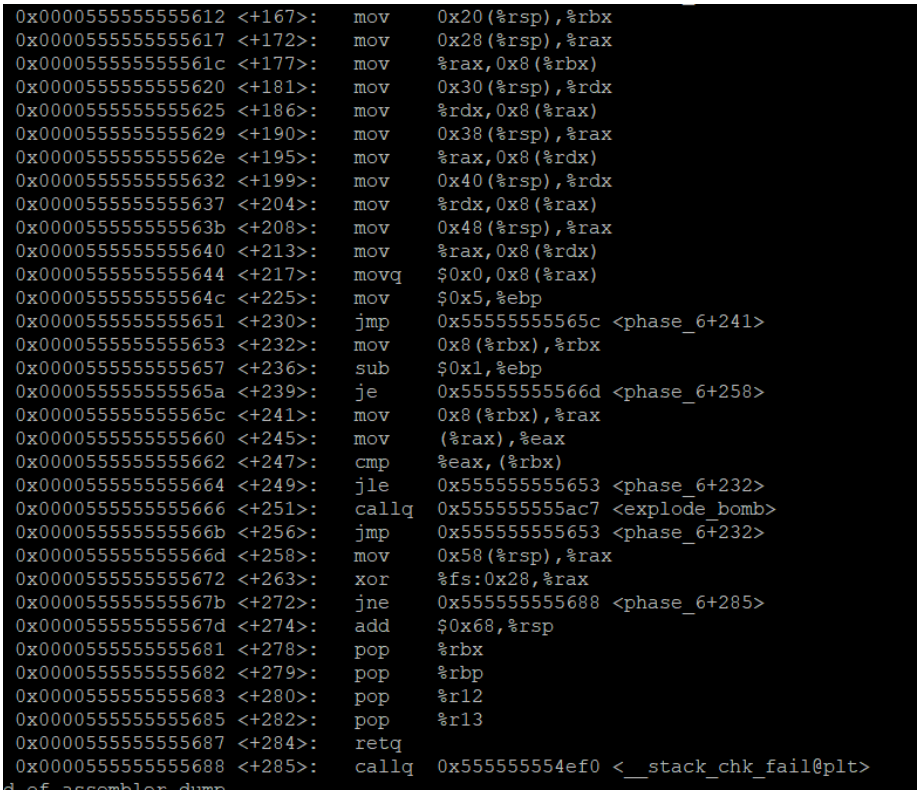
(1-2).123으로 분기했을 때(ecx<1)

```
   0x00005555555555e6 <+123>:   mov    %rdx,0x20(%rsp,%rsi,8)
   0x00005555555555eb <+128>:   add    $0x1,%rsi
   0x00005555555555ef <+132>:   cmp    $0x6,%rsi
   0x00005555555555f3 <+136>:   je     0x555555555612 <phase_6+167>
   0x00005555555555f5 <+138>:   mov    (%rsp,%rsi,4),%ecx
   0x00005555555555f8 <+141>:   mov    $0x1,%eax
   0x00005555555555fd <+146>:   lea    0x202c2c(%rip),%rdx        # 0x555555758230 <node1>
   0x0000555555555604 <+153>:   cmp    $0x1,%ecx
   0x0000555555555607 <+156>:   jg     0x5555555555db <phase_6+112>
---Type <return> to continue, or q <return> to quit---
   0x0000555555555609 <+158>:   jmp    0x5555555555e6 <phase_6+123>
```

%rsp + 8*%rsi + 0x20에 rdx를 저장한다.(배열로 생각하면 double rsp[%rsi + 4])

rsi에 1을 더한 후 rsi가 6이면 167로 분기하고 그렇지 않으면 %rsi값을 0으로 초기화하지 않고 (1)과정을 반복한다.

(1-(1,2)-1).167로 분기



rbx에 rsp[4]를 옮기고 rax에 rsp[5]를 옮긴다. 그리고 rsp[5] (==rbx[1])

rax를 옮긴다.

rdx에 rsp[6]의 주소를 옮기고 rdx 주소를 rsp[6](==rax[1])로 옮긴다.

그후 rax에 rsp[7]를 주소를 옮기고 rsp[7](==rdx[1])에 rax 주소를 옮긴다.

그리고 rdx에 rsp[8]를 주소를 옮기고 rsp[8](==rax[1])에 rdx 주소를 옮긴다.

 rax에 rsp[9] 주소를 옮긴다.

rax[1](==rsp[10])에 0을 옮기고 %rbp에 5를 옮기고 <+241>로 점프한

다.

<+241>로 분기한 후에는

rax에 rbx[1]를 넣고, rax에 가리키는 값을 eax에 넣는다. 즉

eax = rbx[1]이다. 그리고 eax와 (%rbx)와 값을 비교하여 (%rbx)가 더 작거나 같아야 폭탄이 터지지 않음을 알 수 있다.

만약 작거나 같다면 <+232>로 분기한다.

<+232>로 분기한 후에는 rbx가 가리키는 곳을 한 칸 옮기고 ebp에서 1을 뺀 후 <+241>이후 과정을 반복하고, 이를 ebp가 0이 아닐 때 동안 반복한다.

(2) 6과 같지 않아서 분기하지 않았을 때

```
0x000055555555556d6 <+107>:   mov    %r13d,%ebx
0x000055555555556d9 <+110>:   jmp    0x5555555555a7 <phase_6+60>
```

ebx에 %r13d를 옮기고 <+60>으로 분기한다.

```
  0x000055555555555a7 <+60>:    movslq %ebx,%rax
  0x000055555555555aa <+63>:    mov    (%rsp,%rax,4),%eax
  0x000055555555555ad <+66>:    cmp    %eax,0x0(%rbp)
  0x000055555555555b0 <+69>:    jne    0x55555555559f <phase_6+52>
  0x000055555555555b2 <+71>:    callq  0x555555555ac7 <explode_bomb>
  0x000055555555555b7 <+76>:    jmp    0x55555555559f <phase_6+52>
```

rax에 rbx를 옮기고 rax에 입력 받은 숫자 중 rax번째를 옮긴다. 그리고 rbp와

rax를 비교하여 같지 않으면 <+52>로 분기한다.

```
  0x000000000000159f <+52>:    add    $0x1,%ebx
  0x00000000000015a2 <+55>:    cmp    $0x5,%ebx
  0x00000000000015a5 <+58>:    jg     0x15b9 <phase_6+78>
  0x00000000000015a7 <+60>:    movslq %ebx,%rax
  0x00000000000015aa <+63>:    mov    (%rsp,%rax,4),%eax
  0x00000000000015ad <+66>:    cmp    %eax,0x0(%rbp)
  0x00000000000015b0 <+69>:    jne    0x159f <phase_6+52>
  0x00000000000015b2 <+71>:    callq  0x1ac7 <explode_bomb>
  0x00000000000015b7 <+76>:    jmp    0x159f <phase_6+52>
```

<+52>로 분기한 이후에는 rbx에 1을 더하고 rbx가 5보다 크면 <+78>로 분기하고 아니면 (2)를 반복한다.

```
  0x00000000000015b9 <+78>:    add    $0x4,%r12
  0x00000000000015bd <+82>:    mov    %r12,%rbp
  0x00000000000015c0 <+85>:    mov    (%r12),%eax
  0x00000000000015c4 <+89>:    sub    $0x1,%eax
  0x00000000000015c7 <+92>:    cmp    $0x5,%eax
  0x00000000000015ca <+95>:    ja     0x1598 <phase_6+45>
  0x00000000000015cc <+97>:    add    $0x1,%r13d
  0x00000000000015d0 <+101>:   cmp    $0x6,%r13d
  0x00000000000015d4 <+105>:   je     0x160b <phase_6+160>
```

<+78>로 분기한 이후에는 r12가 입력한 숫자 중 다음 숫자를 가리키도록 하고, rbp에 r12 주소를 옮긴 다음, rax에는 r12가 가리키는 실제 값을 옮긴다. 이후 phase_6 중 일부과정을 제외하고 다시 반복한다.

즉 정리하면, 입력한 값에 해당되는 노드가 stack에 차곡차곡 쌓이는 것을 알 수 있다.(1->node1, 2->node2 ...)

그리고 예를 들어 (%rsp)와 8(%rsp)중에서 (%rsp)가 더 작아야하므로 오름차순으로 쌓여야 한다.

그러므로 답은

6 2 5 1 3 4

이다.

8. 최종 답

```
(gdb) r
Starting program: /home/sys059/201924451/bomb6/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Public speaking is very easy.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2.  Keep going!
2 112
Halfway there!
4 2
So you got that one.  Try this one.
5 115
Good work!  On to the next...
6 2 5 1 3 4
Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
[Inferior 1 (process 1566) exited normally]
(gdb)
```