Lab 3

20200437 김채현

본 Lab의 목표는 폭탄이 터지지 않게 6개의 phase를 통과시키는 input을 찾는 것이다.

main

```
   0x00000000000014a8 <+95>:     callq  0x15a7 <phase_1>
   0x00000000000014ad <+100>:    callq  0x1dae <phase_defused>
   0x00000000000014b2 <+105>:    lea    0x1c3f(%rip),%rdi        # 0x30f8
   0x00000000000014b9 <+112>:    callq  0x1200
   0x00000000000014be <+117>:    callq  0x1c66 <read_line>
   0x00000000000014c3 <+122>:    mov    %rax,%rdi
   0x00000000000014c6 <+125>:    callq  0x15cb <phase_2>
   0x00000000000014cb <+130>:    callq  0x1dae <phase_defused>
   0x00000000000014d0 <+135>:    lea    0x1b66(%rip),%rdi        # 0x303d
   0x00000000000014d7 <+142>:    callq  0x1200
   0x00000000000014dc <+147>:    callq  0x1c66 <read_line>
   0x00000000000014e1 <+152>:    mov    %rax,%rdi
   0x00000000000014e4 <+155>:    callq  0x163d <phase_3>
   0x00000000000014e9 <+160>:    callq  0x1dae <phase_defused>
   0x00000000000014ee <+165>:    lea    0x1b66(%rip),%rdi        # 0x305b
   0x00000000000014f5 <+172>:    callq  0x1200
   0x00000000000014fa <+177>:    callq  0x1c66 <read_line>
   0x00000000000014ff <+182>:    mov    %rax,%rdi
   0x0000000000001502 <+185>:    callq  0x1764 <phase_4>
   0x0000000000001507 <+190>:    callq  0x1dae <phase_defused>
   0x000000000000150c <+195>:    lea    0x1c15(%rip),%rdi        # 0x3128
   0x0000000000001513 <+202>:    callq  0x1200
   0x0000000000001518 <+207>:    callq  0x1c66 <read_line>
   0x000000000000151d <+212>:    mov    %rax,%rdi
   0x0000000000001520 <+215>:    callq  0x17dd <phase_5>
   0x0000000000001525 <+220>:    callq  0x1dae <phase_defused>
   0x000000000000152a <+225>:    lea    0x1b39(%rip),%rdi        # 0x306a
   0x0000000000001531 <+232>:    callq  0x1200
   0x0000000000001536 <+237>:    callq  0x1c66 <read_line>
   0x000000000000153b <+242>:    mov    %rax,%rdi
   0x000000000000153e <+245>:    callq  0x186b <phase_6>
   0x0000000000001543 <+250>:    callq  0x1dae <phase_defused>
```

gdb를 통해 main함수를 disassemble 해주었다. phase_1부터 phase_6까지 각각의 함수를 호출하는 것을 볼 수 있다.

---

phase_1

gdb를 통해 phase_1을 disassemble 해주었다.

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x00000000000015a7 <+0>:     repz nop %edx
   0x00000000000015ab <+4>:     sub    $0x8,%rsp
   0x00000000000015af <+8>:     lea    0x1b96(%rip),%rsi        # 0x314c
   0x00000000000015b6 <+15>:    callq  0x1ae1 <strings_not_equal>
   0x00000000000015bb <+20>:    test   %eax,%eax
   0x00000000000015bd <+22>:    jne    0x15c4 <phase_1+29>
   0x00000000000015bf <+24>:    add    $0x8,%rsp
   0x00000000000015c3 <+28>:    retq
   0x00000000000015c4 <+29>:    callq  0x1bf5 <explode_bomb>
   0x00000000000015c9 <+34>:    jmp    0x15bf <phase_1+24>
```

strings_not_equal 호출 전 0x1b96(%rip) 주소를 %rsi에 저장한다. 이 때 x/s 0x314c를 해주어 "Wow! Brazil is big."이라는 문자열이 들어있음을 확인하였다. strings_not_equal 함수를 호출한 후 +20에서 함수의 리턴값이 들어갈 %eax를 test해주고 결과가 같지 않으면 phase_1+29로 점프해 explode_bomb이 실행된다. explode_bomb이 실행되면 폭탄이 터진다.

strings_not_equal을 disassemble하여 해석해보면 입력한 값을 %rdi에 저장해준다. strings_not_equal에 breakpoint를 걸어 r을 해주고 임의로 hi라는 string을 입력하였다. %rdi에 있는 값과 앞에서 했지만 %rsi에 있는 값을 확인해준다.

```
Breakpoint 1, 0x0000555555555ae1 in strings_not_equal ()
(gdb) x/s $rdi
0x5555555596a0 <input_strings>:  "hi"
(gdb) x/s $rsi
0x55555555714c:     "Wow! Brazil is big."
```

해석하면 strings_not_equal은 %rdi에 있는 input string과 %rsi에 있는 string을 비교하여 다르면 1, 같으면 0을 반환함을 알 수 있다. 그렇기에 test를 했을 때 %eax에 대해 bitwise and를 수행하면 zero flag에 0&0, 즉 0을 저장하고 jne이 실행되어 폭탄이 터지는 것이다. 폭탄이 터지지 않기 위해서는 input string을 %rsi에 있는 string과 같게 하면 된다. 따라서 정답은 "Wow! Brazil is big."이다.

---

phase_2

gdb를 통해 phase_2을 disassemble 해주었다.

```
(gdb) disas phase_2
Dump of assembler code for function phase_2:
   0x00005555555555cb <+0>:     repz nop %edx
   0x00005555555555cf <+4>:     push   %rbp
   0x00005555555555d0 <+5>:     push   %rbx
   0x00005555555555d1 <+6>:     sub    $0x28,%rsp
   0x00005555555555d5 <+10>:    mov    %fs:0x28,%rax
   0x00005555555555de <+19>:    mov    %rax,0x18(%rsp)
   0x00005555555555e3 <+24>:    xor    %eax,%eax
   0x00005555555555e5 <+26>:    mov    %rsp,%rsi
   0x00005555555555e8 <+29>:    callq  0x555555555c21 <read_six_numbers>
   0x00005555555555ed <+34>:    cmpl   $0x0,(%rsp)
   0x00005555555555f1 <+38>:    js     0x5555555555fd <phase_2+50>
   0x00005555555555f3 <+40>:    mov    %rsp,%rbp
   0x00005555555555f6 <+43>:    mov    $0x1,%ebx
   0x00005555555555fb <+48>:    jmp    0x555555555615 <phase_2+74>
   0x00005555555555fd <+50>:    callq  0x555555555bf5 <explode_bomb>
   0x0000555555555602 <+55>:    jmp    0x5555555555f3 <phase_2+40>
   0x0000555555555604 <+57>:    callq  0x555555555bf5 <explode_bomb>
   0x0000555555555609 <+62>:    add    $0x1,%ebx
   0x000055555555560c <+65>:    add    $0x4,%rbp
   0x0000555555555610 <+69>:    cmp    $0x6,%ebx
   0x0000555555555613 <+72>:    je     0x555555555621 <phase_2+86>
   0x0000555555555615 <+74>:    mov    %ebx,%eax
   0x0000555555555617 <+76>:    add    0x0(%rbp),%eax
   0x000055555555561a <+79>:    cmp    %eax,0x4(%rbp)
   0x000055555555561d <+82>:    je     0x555555555609 <phase_2+62>
   0x000055555555561f <+84>:    jmp    0x555555555604 <phase_2+57>
   0x0000555555555621 <+86>:    mov    0x18(%rsp),%rax
   0x0000555555555626 <+91>:    xor    %fs:0x28,%rax
   0x000055555555562f <+100>:   jne    0x555555555638 <phase_2+109>
   0x0000555555555631 <+102>:   add    $0x28,%rsp
   0x0000555555555635 <+106>:   pop    %rbx
   0x0000555555555636 <+107>:   pop    %rbp
   0x0000555555555637 <+108>:   retq
   0x0000555555555638 <+109>:   callq  0x555555555220
```

read_six_numbers 함수를 호출하는 것을 보고 6개의 input을 받는다는 것을 예상할 수 있다. 실제로 read_six_numbers 함수를 disassemble 해보았다.

```
(gdb) disas read_six_numbers
Dump of assembler code for function read_six_numbers:
   0x0000555555555c21 <+0>:     repz nop %edx
   0x0000555555555c25 <+4>:     sub    $0x8,%rsp
   0x0000555555555c29 <+8>:     mov    %rsi,%rdx
```

```
   0x0000555555555c2c <+11>:     lea    0x4(%rsi),%rcx
   0x0000555555555c30 <+15>:     lea    0x14(%rsi),%rax
   0x0000555555555c34 <+19>:     push   %rax
   0x0000555555555c35 <+20>:     lea    0x10(%rsi),%rax
   0x0000555555555c39 <+24>:     push   %rax
   0x0000555555555c3a <+25>:     lea    0xc(%rsi),%r9
   0x0000555555555c3e <+29>:     lea    0x8(%rsi),%r8
   0x0000555555555c42 <+33>:     lea    0x1682(%rip),%rsi #0x5555555572cb
   0x0000555555555c49 <+40>:     mov    $0x0,%eax
   0x0000555555555c4e <+45>:     callq  0x5555555552c0
   0x0000555555555c53 <+50>:     add    $0x10,%rsp
   0x0000555555555c57 <+54>:     cmp    $0x5,%eax
   0x0000555555555c5a <+57>:     jle 0x555555555c61 <read_six_numbers+64>
   0x0000555555555c5c <+59>:     add    $0x8,%rsp
   0x0000555555555c60 <+63>:     retq
   0x0000555555555c61 <+64>:     callq  0x555555555bf5 <explode_bomb>
```

+57에 jle을 하여 폭탄이 터지지 않게 하기 위해서는 %eax 값이 5 작거나 같으면 안되므로 6 이상이 되어야한다. 즉, 입력한 숫자의 개수가 6개보다 작을 경우 폭탄이 터진다.

```
(gdb) x/s 0x5555555572cb
0x5555555572cb:     "%d %d %d %d %d %d"
```

6개의 정수를 입력해야 한다는 것을 확실히 알 수 있다.

break를 read_six_numbers 이후에 걸고 임의로 1 2 3 4 5 6의 입력값을 주어 어디에 저장되는지 보았다. %rsp부터 %rsp+20까지 주소에 차례로 저장되어 있음을 볼 수 있었다.

```
(gdb) x/c $rsp
0x7fffffffe4d0:     1 '\001'
(gdb) x/c $rsp+4
0x7fffffffe4d4:     2 '\002'
(gdb) x/c $rsp+8
0x7fffffffe4d8:     3 '\003'
(gdb) x/c $rsp+12
0x7fffffffe4dc:     4 '\004'
(gdb) x/c $rsp+16
0x7fffffffe4e0:     5 '\005'
(gdb) x/c $rsp+20
0x7fffffffe4e4:     6 '\006'
```

다시 phase_2로 돌아가 해석해보면 +34에서 %rsp의 값이 음수이면 explode_bomb으로 점프하여 폭탄이 터진다. 따라서 우선 첫 번째 숫자는 0 이상이어야 한다. +40부터 해석을 해보면 %rsp를 %rbp로 옮긴 후 0x1을 %ebx에 옮기고 +74로 점프하여 %ebx를 %eax로 옮긴다. %eax 값에 %rbp, 즉 %rsp의 값을 더해 %eax에 저장한다. 이 때 %eax값과 %rbp+4의 값이 같지 않으면 +57로 점프하여 폭탄이 터지게 된다. %rbp+4 즉, 다음 입력값인 %rsp+4의 값과 %eax의 값이 같

아야한다. 그럼 +62로 점프하게 되는데 +62에서는 %ebx값을 1만큼, %rbp값을 4만큼 증가시킨 후 %ebx 값이 6이 아니면 다시 +74부터 실행한다. 이를 간략하게 나타내면 다음과 같다.

```
%ebx = 1일 때, %rsp     + 1 = 4(%rsp)
%ebx = 2일 때, 4(%rsp)  + 2 = 8(%rsp)
%ebx = 3일 때, 8(%rsp)  + 3 = 12(%rsp)
%ebx = 4일 때, 12(%rsp) + 4 = 16(%rsp)
%ebx = 5일 때, 16(%rsp) + 5 = 20(%rsp)
```

이와 같이 %ebx 값이 6이 되었을 때 +86으로 점프하고 이후 코드를 수행한 후 return한다. 따라서 정답은 첫 번째 값은 0 이상이고 이전 값에 대해 1, 2, 3, 4, 5만큼 증가하는 수열을 가진 6개의 숫자이다. "0 1 3 6 10 15"도 가능하고, "1 2 4 7 11 16"도 가능하고, "6 7 9 12 16 21" 등등 위의 조건을 만족하는 모든 6개의 숫자가 다 가능하다. 이 중 "1 2 4 7 11 16"을 solution으로 내놓으려 한다.

---

phase_3

gdb를 통해 phase_3을 disassemble 해주었다.

```
(gdb) disas phase_3
Dump of assembler code for function phase_3:
   0x000055555555563d <+0>:     repz nop %edx
   0x0000555555555641 <+4>:     sub    $0x18,%rsp
   0x0000555555555645 <+8>:     mov    %fs:0x28,%rax
   0x000055555555564e <+17>:    mov    %rax,0x8(%rsp)
   0x0000555555555653 <+22>:    xor    %eax,%eax
   0x0000555555555655 <+24>:    lea    0x4(%rsp),%rcx
   0x000055555555565a <+29>:    mov    %rsp,%rdx
   0x000055555555565d <+32>:    lea   0x1c73(%rip),%rsi  #0x5555555572d7
   0x0000555555555664 <+39>:    callq  0x5555555552c0
   0x0000555555555669 <+44>:    cmp    $0x1,%eax
   0x000055555555566c <+47>:    jle    0x55555555568c <phase_3+79>
   0x000055555555566e <+49>:    cmpl   $0x7,(%rsp)
   0x0000555555555672 <+53>:    ja     0x555555555712 <phase_3+213>
   0x0000555555555678 <+59>:    mov    (%rsp),%eax
   0x000055555555567b <+62>:    lea    0x1aee(%rip),%rdx #0x555555557170
   0x0000555555555682 <+69>:    movslq (%rdx,%rax,4),%rax
   0x0000555555555686 <+73>:    add    %rdx,%rax
   0x0000555555555689 <+76>:    ds
   0x000055555555568a <+77>:    jmpq   *%rax
   0x000055555555568c <+79>:    callq  0x555555555bf5 <explode_bomb>
```

+79까지만 적어보았다. phase_3은 함수 내에서 따로 호출하는 함수가 없기 때문에 입력 형태를 알기 어렵다. 어떤 형태의 입력값이 들어가야 하는지 알기 위해 +32 위치의 src 주소를 x/s 명령어로 확인해보았다.

```
(gdb) x/s 0x5555555572d7
0x5555555572d7:     "%d %d"
```

2개의 정수를 입력하면 된다는 것을 유추할 수 있다.

+44에서 %eax값과 0x1을 비교해서 1보다 작거나 같으면 +79로 점프하여 explode_bomb이 수행되어 폭탄이 터진다. 또한 이후 +49에서는 %rsp의 값과 0x7을 비교하여 7보다 크면 +213으로 점프하여 폭탄이 터진다. %eax와 %rsp값을 확인하기 위해 break를 걸고 임의로 111 222를 입력해보았다.

```
Breakpoint 1, 0x0000555555555669 in phase_3 ()
(gdb) x/d $rsp
0x7fffffffe4f0:     111
(gdb) x/d $rsp+4
0x7fffffffe4f4:     222
(gdb) x/d $eax
0x2:                Cannot access memory at address 0x2
(gdb) i r
rax            0x2   2
rbx            0x0   0
rcx            0x20  32
```

%rsp에는 첫 번째 입력값이, %rsp+4에는 두 번째 입력값이 있었고, %eax는 메모리에 접근할 수 없었다. i r 명령어를 이용하여 eax값에는 2가 있기에 폭탄이 터지지 않고 넘어갈 수 있음을 확인하였다.

이를 통해 첫 번째 입력값은 7 이하가 되어야함을 알 수 있다.

입력값을 1 222로 바꾸고 다시 phase_3을 실행하였다.

```
After phase+3+76
   0x000055555555568a <+77>:     jmpq   *%rax
   0x000055555555568c <+79>:     callq  0x555555555bf5 <explode_bomb>
   0x0000555555555691 <+84>:     jmp    0x55555555566e <phase_3+49>
   0x0000555555555693 <+86>:     mov    $0x20a,%eax
   0x0000555555555698 <+91>:     sub    $0xd5,%eax
   0x000055555555569d <+96>:     add    $0x353,%eax
   0x00005555555556a2 <+101>:    sub    $0x95,%eax
   0x00005555555556a7 <+106>:    add    $0x95,%eax
   0x00005555555556ac <+111>:    sub    $0x95,%eax
   0x00005555555556b1 <+116>:    add    $0x95,%eax
   0x00005555555556b6 <+121>:    sub    $0x95,%eax
   0x00005555555556bb <+126>:    cmpl   $0x5,(%rsp)
   0x00005555555556bf <+130>:    jg     0x5555555556c7 <phase_3+138>
```

```
   0x00005555555556c1 <+132>:    cmp    %eax,0x4(%rsp)
   0x00005555555556c5 <+136>:    je     0x5555555556cc <phase_3+143>
   0x00005555555556c7 <+138>:    callq  0x555555555bf5 <explode_bomb>
   0x00005555555556cc <+143>:    mov    0x8(%rsp),%rax
   0x00005555555556d1 <+148>:    xor    %fs:0x28,%rax
   0x00005555555556da <+157>:    jne    0x55555555571e <phase_3+225>
   0x00005555555556dc <+159>:    add    $0x18,%rsp
   0x00005555555556e0 <+163>:    retq
   0x00005555555556e1 <+164>:    mov    $0x0,%eax
   0x00005555555556e6 <+169>:    jmp    0x555555555698 <phase_3+91>
   0x00005555555556e8 <+171>:    mov    $0x0,%eax
   0x00005555555556ed <+176>:    jmp    0x55555555569d <phase_3+96>
   0x00005555555556ef <+178>:    mov    $0x0,%eax
   0x00005555555556f4 <+183>:    jmp    0x5555555556a2 <phase_3+101>
   0x00005555555556f6 <+185>:    mov    $0x0,%eax
   0x00005555555556fb <+190>:    jmp    0x5555555556a7 <phase_3+106>
   0x00005555555556fd <+192>:    mov    $0x0,%eax
   0x0000555555555702 <+197>:    jmp    0x5555555556ac <phase_3+111>
   0x0000555555555704 <+199>:    mov    $0x0,%eax
   0x0000555555555709 <+204>:    jmp    0x5555555556b1 <phase_3+116>
   0x000055555555570b <+206>:    mov    $0x0,%eax
   0x0000555555555710 <+211>:    jmp    0x5555555556b6 <phase_3+121>
   0x0000555555555712 <+213>:    callq  0x555555555bf5 <explode_bomb>
   0x0000555555555717 <+218>:    mov    $0x0,%eax
   0x000055555555571c <+223>:    jmp    0x5555555556bb <phase_3+126>
   0x000055555555571e <+225>:    callq  0x555555555220
```

+77에서 +79로 넘어가 폭탄이 터지는지 확인하기 위해서 +77에서의 %rax값을 확인해야한다.

```
Breakpoint 3, 0x000055555555568a in phase_3 ()
(gdb) x/x $rax
0x5555555556e1 <phase_3+164>:    0xb8
```

%rax는 0x5555555556e1을 가지고 있었고 +164로 점프하게 만드는 문장이었다.

+164에서는 %eax에 0을 저장한 후 +91로 점프하여 -0xd5+0x353-0x95+0x95-0x95+0x95-0x95=-213+851-149+149-149+149-149=+489를 eax에 더해준다. 이후 +126에서 %rsp값이 5보다 크면 +138로 가서 폭탄이 터진다. 따라서 %rsp, 즉 첫 번째 입력값은 5 이하가 되어야한다. 이후 +132에서 %rsp+4 값이 %eax값 489와 같아야 폭탄을 피해 +143으로 점프하여 return할 수 있다. 따라서 두 번째 입력값은 489가 되어야한다.

```
After phase_3+163
   0x00005555555556e1 <+164>:    mov    $0x0,%eax
   0x00005555555556e6 <+169>:    jmp    0x555555555698 <phase_3+91>
```

```
   0x00005555555556e8 <+171>:    mov    $0x0,%eax
   0x00005555555556ed <+176>:    jmp    0x55555555569d <phase_3+96>
   0x00005555555556ef <+178>:    mov    $0x0,%eax
   0x00005555555556f4 <+183>:    jmp    0x5555555556a2 <phase_3+101>
   0x00005555555556f6 <+185>:    mov    $0x0,%eax
   0x00005555555556fb <+190>:    jmp    0x5555555556a7 <phase_3+106>
   0x00005555555556fd <+192>:    mov    $0x0,%eax
   0x0000555555555702 <+197>:    jmp    0x5555555556ac <phase_3+111>
   0x0000555555555704 <+199>:    mov    $0x0,%eax
   0x0000555555555709 <+204>:    jmp    0x5555555556b1 <phase_3+116>
   0x000055555555570b <+206>:    mov    $0x0,%eax
   0x0000555555555710 <+211>:    jmp    0x5555555556b6 <phase_3+121>
```

이 부분에서 phase+96, +101, +106, +111, +116, +121의 의미가 궁금해졌다. 첫 번째 입력값에 따라 +77에서의 %rax값이 달라질 것이라고 예상했고 직접 첫 번째 입력값에 2, 3, 4, 5를 입력하여 값을 확인하였다. 실제로 %rax값이 첫 번째 입력값이 2일 때는 +171로, 3일 때는 +178로, 4일 때는 +185로, 5일 때는 +192의 주소값을 가짐을 확인하였다. 6일 때는 +199로, 7일 때는 +206으로 점프하게 했지만 이후 첫 번째 입력값이 6 이상이면 폭탄이 터지므로 의미는 없다. 따라서 첫 번째 입력값이 2일 때는 eax에 0을 넣고 +101로 점프하여 -0x353-0x95+0x95-0x95+0x95-0x95=851-149+149-149+149-149=+702를 eax에 더해준다. 따라서 두 번째 값은 702가 되어야한다. 이 과정을 표로 나타내면 다음과 같다.

첫 번째 입력값	계산과정	두 번째 입력값
1	0-0xd5+0x353-0x95+0x95-0x95+0x95-0x95	489
2	0+0x353-0x95+0x95-0x95+0x95-0x95	702
3	0-0x95+0x95-0x95+0x95-0x95	-149
4	0+0x95-0x95+0x95-0x95	0
5	0-0x95+0x95-0x95	-149

따라서 답은 5개가 될 수 있다. 이 중 "1 489"를 solution으로 내놓으려 한다.

---

phase_4

gdb를 통해 phase_4를 disassemble 해주었다.

```
(gdb) disas phase_4
Dump of assembler code for function phase_4:
   0x0000555555555764 <+0>:      repz nop %edx
   0x0000555555555768 <+4>:      sub    $0x18,%rsp
   0x000055555555576c <+8>:      mov    %fs:0x28,%rax
```

```
   0x0000555555555775 <+17>:    mov    %rax,0x8(%rsp)
   0x000055555555577a <+22>:    xor    %eax,%eax
   0x000055555555577c <+24>:    lea    0x4(%rsp),%rcx
   0x0000555555555781 <+29>:    mov    %rsp,%rdx
   0x0000555555555784 <+32>:    lea    0x1b4c(%rip),%rsi #0x5555555572d7
   0x000055555555578b <+39>:    callq  0x5555555552c0
   0x0000555555555790 <+44>:    cmp    $0x2,%eax
   0x0000555555555793 <+47>:    jne    0x55555555579b <phase_4+55>
   0x0000555555555795 <+49>:    cmpl   $0xe,(%rsp)
   0x0000555555555799 <+53>:    jbe    0x5555555557a0 <phase_4+60>
   0x000055555555579b <+55>:    callq  0x555555555bf5 <explode_bomb>
```

phase_3에서 한 방법과 같이 어떤 형태의 입력값이 들어가야 하는지 알아보기 위해 +32 위치의 src 주소를 x/s 명령어로 확인해보았다.

```
(gdb) x/s 0x5555555572d7
0x5555555572d7:     "%d %d"
```

2개의 정수를 입력하면 된다는 것을 유추할 수 있다.

break를 걸고 임의로 7 9를 입력하여 입력값이 어디에 저장되는지 알아보았다.

```
(gdb) x/d $rsp
0x7fffffffe4f0:     7
(gdb) x/d $rsp+4
0x7fffffffe4f4:     9
```

%rsp에 첫 번째 입력값이, %rsp+4에 두 번째 입력값이 저장됨을 알 수 있다.

+44에서 %eax값이 2가 아니면 +55로 점프하여 explode_bomb을 호출한다. i r 명령어를 통해 %eax값을 검사해보면 2가 들어있고 덕분에 +49로 넘어갈 수 있다. +49에서는 %rsp와 0xe를 비교하여 작거나 같아야 폭탄을 피해 +60으로 넘어갈 수 있다. 따라서 첫 번째 입력값은 14 이하이어야 한다.

```
After phase_4+55
   0x00005555555557a0 <+60>:    mov    $0xe,%edx
   0x00005555555557a5 <+65>:    mov    $0x0,%esi
   0x00005555555557aa <+70>:    mov    (%rsp),%edi
   0x00005555555557ad <+73>:    callq  0x555555555723 <func4>
   0x00005555555557b2 <+78>:    cmp    $0x7,%eax
   0x00005555555557b5 <+81>:    jne    0x5555555557be <phase_4+90>
   0x00005555555557b7 <+83>:    cmpl   $0x7,0x4(%rsp)
   0x00005555555557bc <+88>:    je     0x5555555557c3 <phase_4+95>
   0x00005555555557be <+90>:    callq  0x555555555bf5 <explode_bomb>
```

%rdx에 14를, %rsi에 0을, %edi에 %rsp 즉 첫 번째 입력값을 저장한 후 func4를 호출한다. func4의 return값이 7이 아니면 +90으로 점프하여 explode_bomb이 호출되게 된다. 따라서 func4의

return값은 7이 되어야한다. +83에서 %rsp+4, 즉 두 번째 입력값과 7을 비교하여 같아야 폭탄을 피해 +95로 넘어갈 수 있다. 따라서 두 번째 입력값은 7이다.

func4를 disassemble 해보면 다음과 같다.

```
(gdb) disas func4
Dump of assembler code for function func4:
   0x0000555555555723 <+0>:     repz nop %edx
   0x0000555555555727 <+4>:     sub    $0x8,%rsp
   0x000055555555572b <+8>:     mov    %edx,%eax
   0x000055555555572d <+10>:    sub    %esi,%eax
   0x000055555555572f <+12>:    mov    %eax,%ecx
   0x0000555555555731 <+14>:    shr    $0x1f,%ecx
   0x0000555555555734 <+17>:    add    %eax,%ecx
   0x0000555555555736 <+19>:    sar    %ecx
   0x0000555555555738 <+21>:    add    %esi,%ecx
   0x000055555555573a <+23>:    cmp    %edi,%ecx
   0x000055555555573c <+25>:    jg     0x55555555574a <func4+39>
   0x000055555555573e <+27>:    mov    $0x0,%eax
   0x0000555555555743 <+32>:    jl     0x555555555756 <func4+51>
   0x0000555555555745 <+34>:    add    $0x8,%rsp
   0x0000555555555749 <+38>:    retq
   0x000055555555574a <+39>:    lea    -0x1(%rcx),%edx
   0x000055555555574d <+42>:    callq  0x555555555723 <func4>
   0x0000555555555752 <+47>:    add    %eax,%eax
   0x0000555555555754 <+49>:    jmp    0x555555555745 <func4+34>
   0x0000555555555756 <+51>:    lea    0x1(%rcx),%esi
   0x0000555555555759 <+54>:    callq  0x555555555723 <func4>
   0x000055555555575e <+59>:    lea    0x1(%rax,%rax,1),%eax
   0x0000555555555762 <+63>:    jmp    0x555555555745 <func4+34>
```

+8부터 해석해보면 %eax에 %edx를 저장해준 다음 %esi를 뺀다. 계산하면 %eax값은 14가 된다. %ecx에 %eax를 옮기고 %ecx값을 31만큼 right shift해준다. 14>>31를 하면 0이 된다. %ecx에 %eax를 더해주면 14가 된다. %ecx를 1만큼 right shift해주면 %ecx/2와 같으므로 7이 된다. %edi, 즉 첫 번째 입력값과 %ecx값 7을 비교한다.

첫 번째 입력값이 7보다 미만이면 +39로 점프한다. +39에서는 %edx에 %rcx-1을 저장하고 다시 func4를 호출한다. 호출이 끝났을 때 실행되는 다음 문장은 +47이다. %eax에 2*%eax값을 저장하고 +34로 가서 return하게 된다. return값은 7이 되어야 하는데 그럼 +47에서 recusion이 끝난 후 %eax값은 3.5가 되어야한다. %eax에는 정수만 저장되므로 3.5가 저장될 수 없고 첫 번째 입력값은 7 이상이란 것을 알 수 있다.

첫 번째 입력값은 7이면 +27에서 %rax에 0을 저장하고 return한다. func4의 return값은 7이 되어

야하므로 첫 번째 입력값은 7이 되면 안된다.

첫 번째 입력값이 7 초과 14이하이면 %eax에 0을 저장하고 +51로 점프한다. +51에서는 %esi에 %rcx+1을 저장하고 다시 func4를 호출한다. +54에서 recusion이 끝난 후 %eax에는 2*%rax+1이 저장되고 +34로 가서 return된다. %eax가 7이 되려면 7=2*3+1, 3=2*1+1, 1=2*0+1이므로 recursion이 3번 호출된다는 것을 알 수 있다. 첫 번째 +23에 도착했을 때 %ecx값은 7, 두 번째에는 11, 세 번째에는 13, 네 번째에는 14이다. 즉, recursion이 3 번째 호출되었을 때 +23에서 %ecx값은 14이다. 이 때 +23에서 %edi값과 %ecx값이 동일해서 함수가 return해야 한다. %edi값이 14이어야하므로 첫 번째 입력값이 14이어야 함을 알 수 있다.

따라서 정답은 "14 7"이다.

---

phase_5

gdb를 통해 phase_5를 disassemble 해주었다.

```
(gdb) disas phase_5
Dump of assembler code for function phase_5:
   0x00005555555557dd <+0>:     repz nop %edx
   0x00005555555557e1 <+4>:     push   %rbx
   0x00005555555557e2 <+5>:     sub    $0x10,%rsp
   0x00005555555557e6 <+9>:     mov    %rdi,%rbx
   0x00005555555557e9 <+12>:    mov    %fs:0x28,%rax
   0x00005555555557f2 <+21>:    mov    %rax,0x8(%rsp)
   0x00005555555557f7 <+26>:    xor    %eax,%eax
   0x00005555555557f9 <+28>:    callq  0x555555555ac0 <string_length>
   0x00005555555557fe <+33>:    cmp    $0x6,%eax
   0x0000555555555801 <+36>:    jne    0x555555555858 <phase_5+123>
```

phase_5에서는 string_length라는 함수를 호출해주는데 이는 input string의 길이를 %rax에 저장하여 return하는 함수이다. +33에서 %eax가 6이 아니면 +123으로 가게 되는데 +123에서는 explode_bomb을 호출한다. 따라서 폭탄이 터지지 않기 위해서는 입력이 6글자의 string이 되어야 함을 알 수 있다. 또한 string_length를 disassemble 해보면 입력 string의 주소가 %rdi에 저장됨을 알 수 있다.

```
After phase_5+36
   0x0000555555555803 <+38>:    mov    $0x0,%eax
   0x0000555555555808 <+43>:    lea    0x1981(%rip),%rcx
   0x000055555555580f <+50>:    movzbl (%rbx,%rax,1),%edx
   0x0000555555555813 <+54>:    and    $0xf,%edx
   0x0000555555555816 <+57>:    movzbl (%rcx,%rdx,1),%edx
   0x000055555555581a <+61>:    mov    %dl,0x1(%rsp,%rax,1)
```

```
   0x000055555555581e <+65>:    add    $0x1,%rax
   0x0000555555555822 <+69>:    cmp    $0x6,%rax
   0x0000555555555826 <+73>:    jne    0x55555555580f <phase_5+50>
   0x0000555555555828 <+75>:    movb   $0x0,0x7(%rsp)
   0x000055555555582d <+80>:    lea    0x1(%rsp),%rdi
   0x0000555555555832 <+85>:    lea    0x1927(%rip),%rsi
   0x0000555555555839 <+92>:    callq 0x555555555ae1 <strings_not_equal>
   0x000055555555583e <+97>:    test   %eax,%eax
   0x0000555555555840 <+99>:    jne    0x55555555585f <phase_5+130>
```

phase_5에서는 phase_1과 같이 strings_not_equal이란 함수가 호출된다. strings_not_equal은 입력 string인 %rdi와 %rsi 메모리의 string 값이 같으면 1을 return하는 함수이다. 만약 같지 않으면 test에서 걸리게 되고 jne를 통해 +130으로 점프하여 폭탄이 터지게 된다. +85에서 strings_not_equal이 호출되기 전 %rsi의 값이 업데이트 되는데 x/s 명령어를 이용하여 주소 0x3160의 string을 읽어보면 "oilers"임을 알 수 있다. 또한 +80에서 %rdi 값이 %rsp값으로 업데이트 되는데 strings_not_equal 함수를 만족시키기 위해서는 %rsp 값이 "oilers"가 되어야 한다.

+38에서 %eax에 0을 저장한 후, +65에서 %rax에 1을 더한 후 %rax값이 6이 아닌 경우 +50으로 돌아간다. 이 loop를 6번 반복한 후 +75에서 %rsp+7의 값에 0을 넣는다. 이는 %rsp+7에 NULL을 넣는 것이라고 볼 수 있다. loop 안의 +50에서 %edx에 %rbx+%rax+1을 저장하는데 이는 입력 string의 %rax+1 번째 문자이다. +54에서 0xf와의 bitwise and 연산을 수행하여 마지막 4bit만 남겨 %edx에 저장한다.

```
(gdb) x/s 0x3190
0x3190 <array.3471>: "maduiersnfotvbylWow! You've defused the secret stage!"
(gdb) x/s 0x3160
0x3160:              "oilers"
```

주소 0x3190에 위치한 string은 maduiersnfotvbyl로 시작한다. %edx에 %rcx+%rdx+1의 값을 저장한다. 즉, %edx 값은 maduiersnfotvbyl의 %rdx번째 문자가 된다. 그 후 %dl (%rdx의 마지막 byte)를 %rsp+%rax+1, 즉 string의 %rax+1 번째 위치에 메모리를 저장한다.

"oilers"는 maduiersnfotvbyl에서 11번째, 5번째, 16번째, 6번째, 7번째, 8번째 문자로 이루어진 string이다. 마지막 4 bit를 mask하면 각각 10, 4, 15, 5, 6, 7이 되어야한다. 이를 아스키코드에서 문자로 나타내면 16k+10, 16k+4, 16k+15, 16k+5, 16k+6, 16k+7 꼴로 만들어야 한다. k에 6을 대입하면 아스키코드 값으로 소문자가 나온다. 따라서 k=6 인 경우 jdoefg가 된다.

10진	k=6	문자
16k+10	106	j
16k+4	100	d
16k+15	111	o

16k+5	101	e
16k+6	102	f
16k+7	103	g

따라서 답은 "jdoefg"이다.

---

phase_6

gdb를 통해 phase_6을 disassemble 해주었다.

```
(gdb) disas phase_6
Dump of assembler code for function phase_6:
   0x000055555555586b <+0>:     repz nop %edx
   0x000055555555586f <+4>:     push   %r14
   0x0000555555555871 <+6>:     push   %r13
   0x0000555555555873 <+8>:     push   %r12
   0x0000555555555875 <+10>:    push   %rbp
   0x0000555555555876 <+11>:    push   %rbx
   0x0000555555555877 <+12>:    sub    $0x60,%rsp
   0x000055555555587b <+16>:    mov    %fs:0x28,%rax
   0x0000555555555884 <+25>:    mov    %rax,0x58(%rsp)
   0x0000555555555889 <+30>:    xor    %eax,%eax
   0x000055555555588b <+32>:    mov    %rsp,%r13
   0x000055555555588e <+35>:    mov    %r13,%rsi
   0x0000555555555891 <+38>:    callq  0x555555555c21 <read_six_numbers>
```

phase_6에서도 read_six_numbers를 호출한다. 따라서 입력값은 6개의 정수이고 값들은 %rsp부터 %rsp+20까지 주소에 차례로 저장되어 있음을 알 수 있다.

read_six_numbers 후에 break를 걸고 임의로 1 2 3 4 5 6을 입력하여 실행시켰다.

```
After phase_6+118
   0x0000555555555896 <+43>:    mov    $0x1,%r14d
   0x000055555555589c <+49>:    mov    %rsp,%r12
   0x000055555555589f <+52>:    jmp    0x5555555558c9 <phase_6+94>
   0x00005555555558a1 <+54>:    callq  0x555555555bf5 <explode_bomb>
   0x00005555555558a6 <+59>:    jmp    0x5555555558d8 <phase_6+109>
   0x00005555555558a8 <+61>:    add    $0x1,%rbx
   0x00005555555558ac <+65>:    cmp    $0x5,%ebx
   0x00005555555558af <+68>:    jg     0x5555555558c1 <phase_6+86>
   0x00005555555558b1 <+70>:    mov    (%r12,%rbx,4),%eax
   0x00005555555558b5 <+74>:    cmp    %eax,0x0(%rbp)
   0x00005555555558b8 <+77>:    jne    0x5555555558a8 <phase_6+61>
   0x00005555555558ba <+79>:    callq  0x555555555bf5 <explode_bomb>
```

```
   0x00005555555558bf <+84>:    jmp    0x5555555558a8 <phase_6+61>
   0x00005555555558c1 <+86>:    add    $0x1,%r14
   0x00005555555558c5 <+90>:    add    $0x4,%r13
   0x00005555555558c9 <+94>:    mov    %r13,%rbp
   0x00005555555558cc <+97>:    mov    0x0(%r13),%eax
   0x00005555555558d0 <+101>:   sub    $0x1,%eax
   0x00005555555558d3 <+104>:   cmp    $0x5,%eax
   0x00005555555558d6 <+107>:   ja     0x5555555558a1 <phase_6+54>
   0x00005555555558d8 <+109>:   cmp    $0x5,%r14d
   0x00005555555558dc <+113>:   jg     0x5555555558e3 <phase_6+120>
   0x00005555555558de <+115>:   mov    %r14,%rbx
   0x00005555555558e1 <+118>:   jmp    0x5555555558b1 <phase_6+70>
```

+43에서 %r14에 1을, %r12에 %rsp를 저장한 후 +94로 점프한다. ni 명령어를 이용해 한 줄씩 디버깅하고 i r 명령어를 이용해 register 안의 값을 확인한 결과 %r13에는 %rsp와 같은 값이 들어있다. 결국 +97에서 %eax에는 %rsp 값이 들어있고 %eax에서 1을 뺀 것이 5보다 크면 점프하여 explode_bomb을 호출한다. 즉, %rsp 값인 첫 번째 값이 6보다 작아야 한다는 뜻이다.

이후 %rbx값에 %r14값을 넣고 +70으로 다시 점프한다. +70에서는 %eax에 %r12+4*%rbx, 즉 %rsp+4의 값인 두 번째 값을 %eax에 넣는다. %eax값과 %rbp값을 비교하여 같다면 폭탄이 터진다. 이는 두 번째 입력값과 첫 번째 입력값이 달라야 한다는 뜻이다. 이 후 +61로 점프한다.

+61에서는 %ebx에서 1을 뺀 것이 5보다 작으면, 즉 %ebx가 6보다 작으면 +86으로 다시 점프한다. +86에서는 %r14에 1을 더하고, %r13에 4를 더해서 위의 과정을 반복한다. 여기서 %ebx와 %r14는 현재 몇 번째 입력값인지를 나타내는 인덱스이고, %r13은 그 입력값을 가리키는 주소이다.

따라서 이 반복문은 첫 번째 입력값부터 여섯 번째 입력값까지 1과 6 사이의 값인지, 같은 입력값은 있는지를 비교하는 일을 수행한다.

```
After phase_6+118
   0x00005555555558e3 <+120>:   mov    $0x0,%esi
   0x00005555555558e8 <+125>:   mov    (%rsp,%rsi,4),%ecx
   0x00005555555558eb <+128>:   mov    $0x1,%eax
   0x00005555555558f0 <+133>:   lea    0x3919(%rip),%rdx
   0x00005555555558f7 <+140>:   cmp    $0x1,%ecx
   0x00005555555558fa <+143>:   jle    0x555555555907 <phase_6+156>
   0x00005555555558fc <+145>:   mov    0x8(%rdx),%rdx
   0x0000555555555900 <+149>:   add    $0x1,%eax
   0x0000555555555903 <+152>:   cmp    %ecx,%eax
   0x0000555555555905 <+154>:   jne    0x5555555558fc <phase_6+145>
   0x0000555555555907 <+156>:   mov    %rdx,0x20(%rsp,%rsi,8)
   0x000055555555590c <+161>:   add    $0x1,%rsi
   0x0000555555555910 <+165>:   cmp    $0x6,%rsi
   0x0000555555555914 <+169>:   jne    0x5555555558e8 <phase_6+125>
```

%esi에 0을 저장하고 %ecx에는 %rsp+4*%rsi를 저장한다. 여기에서 %rsi는 6개의 입력값들에 순서대로 접근하기 위한 인덱스이다. 따라서 %ecx에는 두 번째 입력값이 담긴다. %ecx값은 당연히 1보다 크므로 +156으로 점프한다. +156에서 %rdx 값에는 <node1>의 값이 있다.

이 후 메모리 값에도 node값이 있을 것이라고 추측하고, 실제로 그 값을 확인하였다.

```
(gdb) x/d 0x555555559210
0x555555559210 <node1>:    500
(gdb) x/d 0x555555559210+16
0x555555559220 <node2>:    98
(gdb) x/d 0x555555559210+32
0x555555559230 <node3>:    199
(gdb) x/d 0x555555559210+48
0x555555559240 <node4>:    83
(gdb) x/d 0x555555559210+64
0x555555559250 <node5>:    579
(gdb) x/d 0x555555559110
0x555555559110 <node6>:    888
```

<node1>부터 <node5>까지 16byte 간격으로 찾을 수 있었으며, 특이하게도 <node6>은 <node1>의 주소 100 이전에 위치해 있었다.

+145에서 %rdx+8의 위치에 다음 node의 주소가 저장되어 있을 것이라고 추측하고, 실제로 그 값을 확인하였다.

```
(gdb) x/x 0x555555559210+8
0x555555559218 <node1+8>:  0x55559220
(gdb) x/x 0x555555559210+24
0x555555559228 <node2+8>:  0x55559230
(gdb) x/x 0x555555559210+40
0x555555559238 <node3+8>:  0x55559240
(gdb) x/x 0x555555559210+56
0x555555559248 <node4+8>:  0x55559250
(gdb) x/x 0x555555559210+72
0x555555559258 <node5+8>:  0x55559110
(gdb) x/x 0x555555559110+8
0x555555559118 <node6+8>:  0x00000000
```

따라서 이 node들은 LinkedList라고 추측할 수 있으며 주소를 보면 <node1>-<node2>-...-<node6>까지 순서대로 연결되어 있다.

```
After phase_6+169
     0x000055555555591b <+176>:   mov    0x28(%rsp),%rax
     0x0000555555555920 <+181>:   mov    %rax,0x8(%rbx)
     0x0000555555555924 <+185>:   mov    0x30(%rsp),%rdx
     0x0000555555555929 <+190>:   mov    %rdx,0x8(%rax)
     0x000055555555592d <+194>:   mov    0x38(%rsp),%rax
```

```
0x0000555555555932 <+199>:   mov    %rax,0x8(%rdx)
0x0000555555555936 <+203>:   mov    0x40(%rsp),%rdx
0x000055555555593b <+208>:   mov    %rdx,0x8(%rax)
0x000055555555593f <+212>:   mov    0x48(%rsp),%rax
0x0000555555555944 <+217>:   mov    %rax,0x8(%rdx)
0x0000555555555948 <+221>:   movq   $0x0,0x8(%rax)
0x0000555555555950 <+229>:   mov    $0x5,%ebp
0x0000555555555955 <+234>:   jmp    0x555555555960 <phase_6+245>
0x0000555555555957 <+236>:   mov    0x8(%rbx),%rbx
0x000055555555595b <+240>:   sub    $0x1,%ebp
0x000055555555595e <+243>:   je     0x555555555971 <phase_6+262>
0x0000555555555960 <+245>:   mov    0x8(%rbx),%rax
0x0000555555555964 <+249>:   mov    (%rax),%eax
0x0000555555555966 <+251>:   cmp    %eax,(%rbx)
0x0000555555555968 <+253>:   jge    0x555555555957 <phase_6+236>
0x000055555555596a <+255>:   callq  0x555555555bf5 <explode_bomb>
0x000055555555596f <+260>:   jmp    0x555555555957 <phase_6+236>
0x0000555555555971 <+262>:   mov    0x58(%rsp),%rax
 0x0000555555555976 <+267>:  xor    %fs:0x28,%rax
 0x000055555555597f <+276>:  jne    0x55555555598e <phase_6+291>
 0x0000555555555981 <+278>:  add    $0x60,%rsp
 0x0000555555555985 <+282>:  pop    %rbx
 0x0000555555555986 <+283>:  pop    %rbp
 0x0000555555555987 <+284>:  pop    %r12
 0x0000555555555989 <+286>:  pop    %r13
 0x000055555555598b <+288>:  pop    %r14
 0x000055555555598d <+290>:  retq
 0x000055555555598e <+291>:  callq  0x555555555220
```

위 코드는 저장된 node를 순서대로 보며, node 안의 값이 내림차순으로 정렬되어 있는지를 검사하는 코드이다. +251에서 %eax에는 (%rbx) node의 인덱스 다음으로 입력한 정수를 인덱스로 갖는 node 안의 값이 저장되어 있다. (%rbx) 노드의 값보다 %eax 값이 작아야 폭탄이 터지지 않는다. 즉, 이전 node 안의 값보다 다음 node 안의 값이 작아야 폭탄이 터지지 않는다. 따라서 입력값은 각 node 안의 값들이 내림차순으로 정렬되도록 하는 인덱스 순서이다.

Node index	값
1	500
2	98
3	199
4	83
5	579
6	888

888>579>500>199>98>83이므로 따라서 답은 6 5 1 3 2 4 이다.

따라서 solution을 정리하면 다음과 같다.

```
"Wow! Brazil is big."
"1 2 4 7 11 16"
"1 489"
"14 7"
"jdoefg"
"6 5 1 3 2 4"
```