

1. Phase 1

```
((gdb) disas phase_1
Dump of assembler code for function phase_1:
    0x000000000000000012c4 <+0>:    sub    $0x8,%rsp
    0x000000000000000012c8 <+4>:    lea    0x1901(%ip),%rsi      # 0x2bd0
    0x000000000000000012cf <+11>:   callq  0x187f <strings_not_equal>
    0x000000000000000012d4 <+16>:   test   %eax,%eax
    0x000000000000000012d6 <+18>:   jne    0x12dd <phase_1+25>
    0x000000000000000012d8 <+20>:   add    $0x8,%rsp
    0x000000000000000012dc <+24>:   retq
    0x000000000000000012dd <+25>:   callq  0x1b83 <explode_bomb>
    0x000000000000000012e2 <+30>:   jmp    0x12d8 <phase_1+20>
End of assembler dump.)
```

phase_1 함수를 보면 strings_not_equal 함수 호출을 통해 문자열 비교가 이루어짐을 추측할 수 있고, test, jne를 보아 함수의 return value가 0이 아닐 경우 explode임을 알 수 있다.

```
((gdb) print (char*) 0x2bd0
$9 = 0x2bd0 "You can Russia from land here in Alaska."
```

strings_not_equal 함수를 호출하기 전에 lea로 rsi에 할당되는 값(0x2bd0)의 내용을 조회했더니 답인 "You can Russia from land here in Alaska."을 알 수 있었다.

2. Phase 2

```
0x000000000000000012e6 <+2>:    sub    $0x28,%rsp
0x000000000000000012ea <+6>:    mov    %fs:0x28,%rax
0x000000000000000012f3 <+15>:   mov    %rax,0x18(%rsp)
0x000000000000000012f8 <+20>:   xor    %eax,%eax
0x000000000000000012fa <+22>:   mov    %rsp,%rsi
0x000000000000000012fd <+25>:   callq  0x1bbf <read_six_numbers>
0x00000000000000001302 <+30>:   cmpl  $0x0,(%rsp)
0x00000000000000001306 <+34>:   js    0x1312 <phase_2+46>
0x00000000000000001308 <+36>:   mov    $0x1,%ebx
0x0000000000000000130d <+41>:   mov    %rsp,%rbp
0x00000000000000001310 <+44>:   jmp    0x1323 <phase_2+63>
0x00000000000000001312 <+46>:   callq  0x1b83 <explode_bomb>
```

먼저 스택 포인터인 rsp를 rsi로 옮기고, read_six_numbers를 실행한 뒤 rsp 위치의 값이 0보다 작으면 explode_bomb으로 jump 하는 것으로 보아 read_six_numbers는 2번째 arg로 배열의 주소를 받아 입력 받은 6개의 숫자를 채우는 함수로 추측하였고 read_six_numbers 함수를 disass한 결과 sscanf와 sscanf 실행 전 lea 명령으로 sscanf의 arg에 rsi에서 4의 배수로 더해진 값을 넣는 것을 보고 확신하게 되었다. js를 통해 첫번째 숫자는 0 이상이기만 하면 됨을 알아내었다.

```
0x000000000000000012fd <+25>:   callq  0x1bbf <read_six_numbers>
...
0x00000000000000001308 <+36>:   mov    $0x1,%ebx          # int ebx = 1
0x0000000000000000130d <+41>:   mov    %rsp,%rbp          # int* rbp = arr;
0x00000000000000001310 <+44>:   jmp    0x1323 <phase_2+63>  # goto phase_2+63;
...
0x00000000000000001319 <+53>:   add    $0x1,%rbx          # rbx++;
0x0000000000000000131d <+57>:   cmp    $0x6,%rbx
0x00000000000000001321 <+61>:   je    0x1336 <phase_2+82>  # if(rbx == 6) goto phase_2+86;
...
0x00000000000000001323 <+63>:   mov    %ebx,%eax          # int eax = ebx;
0x00000000000000001325 <+65>:   add    -0x4(%rbp,%rbx,4),%eax  # eax += arr + rbx * 4 - 4;
0x00000000000000001329 <+69>:   cmp    %eax,0x0(%rbp,%rbx,4)  # if(eax == arr + rbx * 4) goto phase_2 + 53;
0x0000000000000000132d <+73>:   je    0x1319 <phase_2+53>  # ( if(arr[rbx - 1] + ebx == arr[rbx]) )
0x0000000000000000132f <+75>:   callq  0x1b83 <explode_bomb> # else explode;
0x00000000000000001334 <+80>:   jmp    0x1319 <phase_2+53>  # goto phase_2 + 53;
0x00000000000000001336 <+82>:   mov    0x18(%rsp),%rax      #
```

이후 숫자들은 어셈블리를 c로 해석하면서 rbx로 1부터 5까지 5번 도는 반복문임을 알았고, 통과 조건은 arr[rbx - 1] + ebx == arr[rbx] 임을 알아냈다. 즉, 1,2,3,4,5씩 커지면 되는 것이다. 0,1,3,6,10,15를 입력했더니 통과되었다.

3. Phase 3

```

0x0000000000000001366 <+20>:    lea    0xf(%rsp),%rcx
0x000000000000000136b <+25>:    lea    0x10(%rsp),%rdx
0x0000000000000001370 <+30>:    lea    0x14(%rsp),%r8
0x0000000000000001375 <+35>:    lea    0x18aa(%rip),%rsi      # 0x2c26
0x000000000000000137c <+42>:    callq 0xfa0 <__isoc99_sscanf@plt>

```

```

(gdb) print (char*) 0x2c26
$8 = 0x2c26 "%d %c %d"

```

먼저 phase3 내부 sscanf의 입력을 알아보고 싶어 rsi에 들어가는 0x2c26의 내용을 print하였고, "%d %c %d" 형식으로 입력 받음을 알 수 있었다. sscanf 전의 lea 명령어들을 통해 rsp+0xf에 char, rsp+0x10에 첫번째 int, rsp+0x14에 마지막 int를 입력 받음을 알 수 있다.

```

0x0000000000000001386 <+52>: cmpl $0x7,0x10(%rsp)
0x000000000000000138b <+57>: ja 0x1496 <phase_3+324> # if(7 > n1) explode_bomb;
0x0000000000000001391 <+63>: mov 0x10(%rsp),%eax      # eax = n1;
0x0000000000000001395 <+67>: lea 0x18a4(%rip),%rdx      # rdx = 0x2c40
0x000000000000000139c <+74>: movslq (%rdx,%rax,4),%rax      # rax = *(0x2c40 + 4 * n1); 점프테이블
0x00000000000000013a0 <+78>: add %rdx,%rax      # rax += 0x2c40;
0x00000000000000013a3 <+81>: jmpq *%rax      # goto rax

```

첫번째 입력(n1) 값에 따라 jump table을 통해 jump(switch-case)함을 알 수 있다.

```

(gdb) x/wx 0x2c40  (gdb) x/wx 0x2c44  (gdb) x/wx 0x2c48
jump table을 확인해보면, 0x2c40: 0xfffffe76c  0x2c44: 0xfffffe78e  0x2c48: 0xfffffe7b0

```

n1	pos	value	goto (value+0x2c40)
0	0x2c40	0xfffffe76c	0x13ac
1	0x2c44	0xfffffe78e	0x13ce
2	0x2c48	0xfffffe7b0	0x13f0

```

0x00000000000000013ac <+90>: mov $0x78,%eax
0x00000000000000013b1 <+95>: cmpl $0x254,0x14(%rsp)
0x00000000000000013b9 <+103>: je 0x14a0 <phase_3+334>
0x00000000000000013bf <+109>: callq 0x1b83 <explode_bomb>

```

13ac부터 확인해보면, 0x254(596)과 마지막 입력 값이 같아야 explode를 피할 수 있음(je 0x14a0)을 알 수 있다.

```

0x00000000000000014a0 <+334>: cmp %al,0xf(%rsp)
0x00000000000000014a4 <+338>: je 0x14ab <phase_3+345>
0x00000000000000014a6 <+340>: callq 0x1b83 <explode_bomb>

```

14a0을 보면, al과 입력받은 문자가 같아야 explode를 피할 수 있다. 13ac에서 0x78(120)을 eax에 mov했으므로 x를 입력해야 됨을 알 수 있다. 나머지 jump table은 무시하고 0x596을 입력했더니 통과되었다.

4. Phase 4

sscanf 직전의 rsi 값을 확인하니 입력이 2개의 정수(%d %d)임을 알 수 있었다.

```

0x0000000000000001518 <+20>: lea 0x4(%rsp),%rcx
0x000000000000000151d <+25>: mov %rsp,%rdx
0x0000000000000001520 <+28>: lea 0x1996(%rip),%rsi      # 0x2ebd
0x0000000000000001527 <+35>: callq 0xfa0 <__isoc99_sscanf@plt>

```

rsp에 첫번째 정수(n1), rsp+4에 두번째 정수(n2)를 입력받고 있다.

```

1531 <+45>: cmpl $0xe,(%rsp)
1535 <+49>: jbe 0x153c <phase_4+56>
1537 <+51>: callq 0x1b83 <explode_bomb> を 통해 첫번째 정수는 0xE(14) 이하여야 함을 알 수 있다.

```

```

0x000000000000000153c <+56>: mov $0xe,%edx      # edx = 0x14
0x0000000000000001541 <+61>: mov $0x0,%esi      # esi = 0
0x0000000000000001546 <+66>: mov (%rsp),%edi      # edi = n1
0x0000000000000001549 <+69>: callq 0x14c5 <func4>      # func4(n1, 0, 0xE)
0x000000000000000154e <+74>: cmp $0x4,%eax
0x0000000000000001551 <+77>: jne 0x155a <phase_4+86>      # if(func4 결과 != 4) explode;
0x0000000000000001553 <+79>: cmpl $0x4,0x4(%rsp)
0x0000000000000001558 <+84>: je 0x155f <phase_4+91>      # if(n2 == 4) jump 155f;
0x000000000000000155a <+86>: callq 0x1b83 <explode_bomb>      # explode
0x000000000000000155f <+91>: mov 0x8(%rsp),%rax

```

이후 코드를 분석해보면 func4(n1, 0, 14)의 결과가 4이고, n2 값이 4여야 통과됨을 찾을 수 있다.

func4를 분석해보았더니 이분탐색 느낌의 재귀함수가 나왔다.

```

# edi = n1, esi = 0, edx = 14
0x00000000000000014c9 <+4>: mov %edx,%eax      # eax = 14;
0x00000000000000014cb <+6>: sub %esi,%eax      # eax -= 0;
0x00000000000000014cd <+8>: mov %eax,%ecx      # ecx = eax; (14)
0x00000000000000014cf <+10>: shr $0x1f,%ecx      # ecx = 0;
0x00000000000000014d2 <+13>: add %eax,%ecx      # ecx += 14; (14)
0x00000000000000014d4 <+15>: sar %ecx      # ecx /= 2; (7)
0x00000000000000014d6 <+17>: add %esi,%ecx      # ecx += 0; (7)
0x00000000000000014d8 <+19>: cmp %edi,%ecx
0x00000000000000014da <+21>: jg 0x14ea <func4+37>      # if(7 > n1) jump 14ea;
0x00000000000000014dc <+23>: mov $0x0,%eax      # eax = 0;

```

```

0x000000000000000014e1 <+28>: cmp    %edi,%ecx
0x000000000000000014e3 <+30>: jl     0x14f6 <func4+49> # if(7 < n1) jump 14f6;
0x000000000000000014e5 <+32>: add    $0x8,%rsp
0x000000000000000014e9 <+36>: retq
0x000000000000000014ea <+37>: lea    -0x1(%rcx),%edx
0x000000000000000014ed <+40>: callq 0x14c5 <func4>
0x000000000000000014f2 <+45>: add    %eax,%eax
0x000000000000000014f4 <+47>: jmp    0x14e5 <func4+32>
0x000000000000000014f6 <+49>: lea    0x1(%rcx),%esi
0x000000000000000014f9 <+52>: callq 0x14c5 <func4>
0x000000000000000014fe <+57>: lea    0x1(%rax,%rax,1),%eax
0x00000000000000001502 <+61>: jmp    0x14e5 <func4+32>

```

n1에 월 넣어야 4가 return 될지 바로 알기 어려워서 c로 해당 함수를 똑같이 구현해서 1부터 14까지 넣어보았고, 2를 넣었을 때 4가 return됨을 알 수 있었다. 따라서 2~4를 입력했더니 통과되었다.

5. Phase 5

```

...
0x0000000000000000157e <+5>: mov    %rdi,%rbx          # rbx = input_str_ptr
...
0x00000000000000001591 <+24>: callq 0x1862 <string_length> # eax = string_length(input_str_ptr)
0x00000000000000001596 <+29>: cmp    $0x6,%eax
0x00000000000000001599 <+32>: jne    0x15f0 <phase_5+119> # if(eax != 6) jump 15f0; (explode)
0x0000000000000000159b <+34>: mov    $0x0,%eax
0x000000000000000015a0 <+39>: lea    0x16b9(%rip),%rcx
0x000000000000000015a7 <+46>: movzbl (%rbx,%rax,1),%edx
0x000000000000000015ab <+50>: and    $0xf,%edx
0x000000000000000015ae <+53>: movzbl (%rcx,%rdx,1),%edx
0x000000000000000015b2 <+57>: mov    %dl,0x1(%rsp,%rax,1)
0x000000000000000015b6 <+61>: add    $0x1,%rax
0x000000000000000015ba <+65>: cmp    $0x6,%rax
0x000000000000000015be <+69>: jne    0x15a7 <phase_5+46> # if(rax != 6) jump 15a7; (0~5 6번 반복)
0x000000000000000015c0 <+71>: movb   $0x0,0x7(%rsp)      # *(rsp + 7) = 0; ((rsp + 1)[6] = 0)
0x000000000000000015c5 <+76>: lea    0x1(%rsp),%rdi
0x000000000000000015ca <+81>: lea    0x165e(%rip),%rsi
0x000000000000000015d1 <+88>: callq 0x187f <strings_not_equal> # eax = strings_not_equal(rsp + 1, 0x2c2f)
0x000000000000000015d6 <+93>: test   %eax,%eax
0x000000000000000015d8 <+95>: jne    0x15f7 <phase_5+126> # if(eax != 0) jump 0x15f7; (explode)
0x000000000000000015da <+97>: mov    0x8(%rsp),%rax

```

phase_5 함수를 분석했더니 입력 문장의 길이가 6이어야하고, 입력된 문자열들의 글자를 하나씩 뽑아 하위 4비트만 남긴 뒤 0x2c60에 위치하는 문자열에서 그에 해당하는 번째의 문자를 가져와 (0x2c60[input_str_ptr[rax] & 0xf]) 배열에 저장한 뒤, string_not_equal이라는 함수를 통해 0x2c2f에 있는 devils라는 문자열과 비교하고 있음을 알 수 있었다. 0x2c60에 위치한 문자열은 "maduiersnfotvbylSo you think..."인데, 0x2c60[input_str_ptr[rax] & 0xf]이기 때문에 16글자 이상 가져올 수 없다. devils를 만드려면 0x2 0x5 0xC 0x4 0xF 0x7번째 문자들을 가져와야 한다. 하위 4비트만 일치하면 되므로 아스키 코드표를 보고 해당하는 BELDOG를 입력했더니 통과되었다.

6. Phase 6

```

0x0000000000000000161f <+28>: mov    %rsp,%r13          # r13 = rsp;
0x00000000000000001622 <+31>: mov    %r13,%rsi
0x00000000000000001625 <+34>: callq 0x1bbf <read_six_numbers> # read_six_numbers(input_str_ptr, r13)
0x0000000000000000162a <+39>: mov    %r13,%r12          # r12 = rsp
0x0000000000000000162d <+42>: mov    $0x0,%r14d
0x00000000000000001633 <+48>: jmp    0x165a <phase_6+87> # # jump 165a;

```

위 부분을 통해 r13(rsp)에 배열로 입력 받고 있음을 알 수 있었다.

```

0x0000000000000000163c <+57>: add    $0x1,%ebx
0x0000000000000000163f <+60>: cmp    $0x5,%ebx
0x00000000000000001642 <+63>: jg     0x1656 <phase_6+83>
0x00000000000000001644 <+65>: movslq %ebx,%rax
0x00000000000000001647 <+68>: mov    (%rsp,%rax,4),%eax
0x0000000000000000164a <+71>: cmp    %eax,0x0(%rbp)
0x0000000000000000164d <+74>: jne    0x163c <phase_6+57>

0x0000000000000000164f <+76>: callq 0x1b83 <explode_bomb>
0x00000000000000001654 <+81>: jmp    0x163c <phase_6+57>

0x00000000000000001656 <+83>: add    $0x4,%r13
                                         # # r13++;

0x0000000000000000165a <+87>: mov    %r13,%rbp
0x0000000000000000165d <+90>: mov    0x0(%r13),%eax
0x00000000000000001661 <+94>: sub    $0x1,%eax
0x00000000000000001664 <+97>: cmp    $0x5,%eax
0x00000000000000001667 <+100>: ja    0x1635 <phase_6+50>
0x00000000000000001669 <+102>: add    $0x1,%r14d
0x0000000000000000166d <+106>: cmp    $0x6,%r14d
0x00000000000000001671 <+110>: je    0x1678 <phase_6+117>
0x00000000000000001673 <+112>: mov    %r14d,%ebx
0x00000000000000001676 <+115>: jmp    0x1644 <phase_6+65>

```

차례대로 해석하다 보니 input 배열에서 6 이상의 값이 있을 경우 explode가 됨을 알 수 있었고, rbp와 r13을 증가시키면서 서로 비교하는 부분을 확인 할 수 있었다. 즉, 입력 받은 값들 중 중복이 있으면 explode 하는 2중 반복문임을 확인할 수 있었다. 이를 통해 입력에 중복된 값이 있으면 안되는 것을 알게 되었다.

```

0x00000000000000001678 <+117>: lea    0x18(%r12),%rcx      # rcx = r12 + 6;
0x0000000000000000167d <+122>: mov    $0x7,%edx      # edx = 7
0x00000000000000001682 <+127>: mov    %edx,%eax      # eax = 7
0x00000000000000001684 <+129>: sub    (%r12),%eax      # eax -= r12[0]
0x00000000000000001688 <+133>: mov    %eax,(%r12)      # r12[0] = eax
0x0000000000000000168c <+137>: add    $0x4,%r12      # r12 += 1;
0x00000000000000001690 <+141>: cmp    %r12,%rcx
0x00000000000000001693 <+144>: jne    0x1682 <phase_6+127>  # if(r12 != rcx) jump 1682 (반복)

```

계속 해석해 나가다 보니 하나의 반복문이 더 존재했는데, input array의 주소를 저장해 두었던 r12를 증가시키면서 array의 6개 원소 값에 각각 7-(기준 값) 한 것을 저장하고 있음을 알 수 있었다.

```

0x00000000000000001695 <+146>: mov    $0x0,%esi      # esi = 0
0x0000000000000000169a <+151>: jmp    0x16b6 <phase_6+179>  # jump 16b6; (jump-to-middle?)

0x0000000000000000169c <+153>: mov    0x8(%rdx),%rdx      # # # rdx = *(rdx + 8); (*0x204238, 248, 258, 268, 278, 110, 000)(Linked List)
0x000000000000000016a0 <+157>: add    $0x1,%eax      # # # eax++;
0x000000000000000016a3 <+160>: cmp    %ecx,%eax
0x000000000000000016a5 <+162>: jne    0x169c <phase_6+153>  # # # if(eax != ecx) jump 169c; (반복)

0x000000000000000016a7 <+164>: mov    %rdx,0x20(%rsp,%rsi,8)  # # (rsp + 32)[rsi] = rdx;
0x000000000000000016a8 <+165>: add    $0x1,%rsi      # # rsi++;
0x000000000000000016b0 <+173>: cmp    $0x6,%rsi
0x000000000000000016b4 <+177>: je     0x16cc <phase_6+201>  # # if(rsi == 6) jump 16cc (break);

0x000000000000000016b6 <+179>: mov    (%rsp,%rsi,4),%ecx      # # ecx = rsp[rsi]
0x000000000000000016b9 <+182>: mov    $0x1,%eax      # # eax = 1
0x000000000000000016be <+187>: lea    0x202b6b(%rip),%rdx  # # rdx = 0x204230 # 0x204230 <node1>
0x000000000000000016c5 <+194>: cmp    $0x1,%ecx
0x000000000000000016c8 <+197>: jg    0x169c <phase_6+153>  # # if(ecx > 1) jump 169c;
0x000000000000000016ca <+199>: jmp    0x16a7 <phase_6+164>  # # else jump 16a7;

```

이후 2종 반복문을 확인할 수 있었는데, ppt에 phase 6이 linked list/structs라는 힌트를 보고 처음 16be에 lea를 통해 rdx에 0x204230을 넣는 부분에 gdb에서 node1이라는 주석을 달았길래 struct에 관련된 부분일거라 추측했고, 0x204230을 읽어보니 0x204230 <node1>: 0x00000148 라는 값이 나와 struct라고 가정하고 계속 진행했다.

내부 2번째 반복문에서 0x204230+8 에 접근하고, 0x204230+8 에 저장된 주소에 접근하는 방식임을 알았고, linked list일거라 생각되어 0x204230+8, *(0x204230+8), *(0x204230+8)+8...에 접근해봤더니 6개짜리 linked list임을 알 수 있었다.

node	+0	+8	node	+0	+8
1	0x148(328)	0x204240	4	0x2ba(698)	0x204270
2	0x16a(362)	0x204250	5	0x327(807)	0x204110
3	0x358(856)	0x204260	6	0x2d4(724)	0x0

```

(lldb) x 0x204238
0x204238 <node1+8>: 0x00204240
(lldb) x 0x204248
0x204248 <node2+8>: 0x00204250
(lldb) x 0x204258
0x204258 <node3+8>: 0x00204260
(lldb) x 0x204268
0x204268 <node4+8>: 0x00204270
(lldb) x 0x204278
0x204278 <node5+8>: 0x00204110
(lldb) x 0x204118
0x204118 <node6+8>: 0x00000000

```

따라서 이 부분의 코드는 이전에 (7-입력 값) 했던 배열에서 수들을 차례로 가져와 (7-입력 값) 번째의 node 구조체의 주소를 새로운 배열(rsp+32)에 저장하고 있음을 알 수 있다.

```

0x000000000000000016cc <+201>: mov    0x20(%rsp),%rbx      # rbx = (rsp + 32)[0]; (&node(7-a1))
0x000000000000000016d1 <+206>: mov    0x28(%rsp),%rax      # rax = (rsp + 32)[1]; (&node(7-a2))
0x000000000000000016d6 <+211>: mov    %rax,0x8(%rbx)      # (rsp + 32)[0] + 8 = (rsp + 32)[1]; (node(7-a1)[1]=&node(7-a2))
0x000000000000000016da <+215>: mov    0x30(%rsp),%rdx      # rdx = (rsp + 32)[2]; (&node(7-a3))
0x000000000000000016df <+220>: mov    %rdx,0x8(%rax)      # (rsp + 32)[1] + 8 = (rsp + 32)[1][0]; (node(7-a2)[1]=&node(7-a3))
0x000000000000000016e3 <+224>: mov    0x38(%rsp),%rax
0x000000000000000016e8 <+229>: mov    %rax,0x8(%rdx)      # ... Linked List의 주소들을 업데이트
0x000000000000000016ec <+233>: mov    0x40(%rsp),%rdx
0x000000000000000016f1 <+238>: mov    %rdx,0x8(%rax)
0x000000000000000016f5 <+242>: mov    0x48(%rsp),%rax
0x000000000000000016fa <+247>: mov    %rax,0x8(%rdx)
0x000000000000000016fe <+251>: movq   $0x0,0x8(%rax)
0x00000000000000001703 <+259>: mov    $0x5,%ebp      # ebp = 5;
0x0000000000000000170b <+264>: jmp    0x1716 <phase_6+275>  # jump 1716;

```

반복문 실행 이후 코드를 해석해보니 (rsp+32) 배열에 넣은 node의 순서대로 node들의 2번째 값(다음 node의 주소)을 업데이트 하고 있음을 알 수 있었다.

```

0x0000000000000000170d <+266>: mov    0x8(%rbx),%rbx      # # rbx = 다음 노드;
0x00000000000000001711 <+270>: sub    $0x1,%ebp      # # ebp -= 1;
0x00000000000000001714 <+273>: je     0x1727 <phase_6+292>  # # if(epb == 0) break;
0x00000000000000001716 <+275>: mov    0x8(%rbx),%rax      # # rax = rbx[1]; (rax=node(7-a1)[1])
0x0000000000000000171a <+279>: mov    (%rax),%eax      # # eax = rax; (eax=node(7-a2)[1])
0x0000000000000000171c <+281>: cmp    %eax,%rbx
0x0000000000000000171e <+283>: jge    0x170d <phase_6+266>  # # if(*rbx >= eax) jump 170d; (반복) (node(7-a1)[0]>=node(7-a2)[0])
0x00000000000000001720 <+285>: callq  0x1b83 <explode_bomb>  # # else explode;

```

이후 반복문이 하나 더 존재했는데, node들을 순차적으로 확인하면서 앞의 node가 가지고 있는 값(+0)이 뒤의 노드가 가지고 있는 값보다 커야 explode가 되지 않으므로 아까 확인한 값들을 사용하면, node3, 5, 6, 4, 2, 1 순이어야 함을 알 수 있다. 아까 7-(입력 값) 하는 코드가 있었으므로 답은 4 2 1 3 5 6이다.

7. Phase Secret

먼저, secret_phase 함수를 실행할 방법을 찾기 위해 main에서 실행되는 함수들을 모두 disass해서 secret_phase에 대한 내용이 있는 함수를 보았더니 phase_defused 함수에 있는 것을 찾을 수 있었다.

```
=> 0x000055555555d62 <+30>:    cmpl  $0x6,0x202943(%rip)      # 0x5555557586
ac <num_input_strings>
0x000055555555d69 <+37>:    je     0x5555555555d84 <phase_defused+64>
```

이 부분을 통해 일단 0x5555557586의 값이 6이 되어야 함을 알 수 있었고, 이 값이 의미하는 바를 알기 위해 phase_defuse+30에 bp를 걸었다. phase1을 성공했을 때는 1, 2를 성공했을 때는 2 이런 식으로 값이 되는 것을 보아 일단 폭탄 6개를 다 해체 한 후에 실행되는 부분임을 알 수 있었다.

```
0x00005555555555d84 <+64>:    lea   0xc(%rsp),%rcx
0x00005555555555d89 <+69>:    lea   0x8(%rsp),%rdx
0x00005555555555d8e <+74>:    lea   0x10(%rsp),%r8
0x00005555555555d93 <+79>:    lea   0x116d(%rip),%rsi      # 0x555555556f07
0x00005555555555d9a <+86>:    lea   0x202a0f(%rip),%rdi      # 0x5555557587
b0 <input_strings+240>
0x00005555555555d91 <+93>:    mov   $0x0,%eax
0x00005555555555d96 <+98>:    callq 0x5555555554fa0 <_isoc99_sscanf@plt>
```

이후 sscanf를 호출하고 있었는데, input(rdi)를 어디서 가져오는지 보기 위해서

```
((gdb) print (char*) 0x5555557587b0 <input_strings+240> "2 4"
$6 = 0x5555557587b0 <input_strings+240> "2 4"
명령을 실행했고, phase 4때의 입력 값이랑 동일함을 확인 할 수 있었다. $9 = 0x555555556f07 "%d %d %s" 명령을 통해 phase 4 입력 값 뒤에 문자열을 하나 더 확인하고 있음을 보았고, 그 값은 r8인 (rsp+0x10)에 저장됨을 알 수 있다.
```

```
0x00005555555555dca <+134>:   lea   0x10(%rsp),%rdi
0x00005555555555dcf <+139>:   lea   0x113a(%rip),%rsi      # 0x555555556f10
0x00005555555555dd6 <+146>:   callq 0x55555555587f <strings_not_equal>
0x00005555555555ddb <+151>:   test  %eax,%eax
0x00005555555555ddd <+153>:   jne   0x5555555555db0 <phase_defused+108>
```

이 부분에서 (rsp+0x10)의 문자열과 \$10 = 0x555555556f10 "DrEvil" 를 비교하고 있음을 알았고, secret_phase의 실행 조건은 phase_4의 입력 뒤에 DrEvil을 덧붙이는 것임을 알 수 있었다.

```
0x00005555555555789 <+1>:    callq 0x5555555555c00 <read_line>
0x0000555555555578e <+6>:    mov   $0xa,%edx
0x00005555555555793 <+11>:   mov   $0x0,%esi
0x00005555555555798 <+16>:   mov   %rax,%rdi
0x0000555555555579b <+19>:   callq 0x5555555554f80 <strtol@plt>
0x000055555555557a0 <+24>:   mov   %rax,%rbx
0x000055555555557a3 <+27>:   lea   -0x1(%rax),%eax
0x000055555555557a6 <+30>:   cmp   $0x3e8,%eax
0x000055555555557ab <+35>:   ja    0x55555555557d8 <secret_phase+80>
```

이제 secret_phase를 disass 해보았는데, 한 줄을 읽고 strtol을 통해 10진수 숫자 1개를 얻음을 알 수 있었고, 그 값에서 1을 뺀게 0xE8(1000)보다 클 경우 explode임을 알 수 있었다.

```
0x000055555555557ad <+37>:   mov   %ebx,%esi
0x000055555555557af <+39>:   lea   0x20299a(%rip),%rdi      # 0x5555557581
50 <n1>
0x000055555555557b6 <+46>:   callq 0x555555555749 <fun7>
0x000055555555557bb <+51>:   cmp   $0x7,%eax
0x000055555555557be <+54>:   je    0x55555555557c5 <secret_phase+61>
0x000055555555557c0 <+56>:   callq 0x5555555555b83 <explode_bomb>
```

이후 부분에서 fun7(0x555555758150, input_num)을 호출하는 부분이 있었고, 리턴 값이 7이 아니면 explode임을 확인 할 수 있었다. n1의 값은 0x555555758150 <n1>: 0x00000024(36)임을 확인했으므로 fun7을 disass해보았다.

```
0x0000555555555574c <+3>:   je    0x555555555782 <fun7+57> # if(!rdi) return;
0x0000555555555574d <+5>:   sub   $0x8,%rsp
0x00005555555555752 <+9>:   mov   (%rdi),%edx      # edx = *rdi; *arg1
0x00005555555555754 <+11>:  cmp   %esi,%edx
0x00005555555555756 <+13>:  jg    0x555555555766 <fun7+29> # if(edx > esi) jump +29; (arg1 > input_num)
0x00005555555555758 <+15>:  mov   $0x0,%eax      # eax = 0
0x0000555555555575d <+20>:  cmp   %esi,%edx
0x0000555555555575f <+22>:  jne   0x5555555555773 <fun7+42> # if(edx != esi) jump +42; (arg1 != input_num)
0x00005555555555761 <+24>:  add   $0x8,%rsp
0x00005555555555765 <+28>:  retq 
0x00005555555555766 <+29>:  mov   0x8(%rdi),%rdi      # rdi = *(rdi+8)
0x0000555555555576a <+33>:  callq 0x555555555749 <fun7>      # eax = fun7(*(rdi+8), input_num)
0x0000555555555576f <+38>:  add   %eax,%eax      # eax *= 2;
0x00005555555555771 <+40>:  jmp   0x555555555761 <fun7+24> # jump +24; (return)
0x00005555555555773 <+42>:  mov   0x10(%rdi),%rdi      # rdi = *(rdi+16);
0x00005555555555774 <+46>:  callq 0x555555555749 <fun7>      # eax = 2 * fun7(*(rdi+16), input_num) + 1
0x0000555555555577c <+51>:  lea   0x1(%rax,%rax,1),%eax      # eax = 2 * rx + 1;
0x00005555555555780 <+55>:  jmp   0x5555555555761 <fun7+24> # jump +24; (return)
```

fun7을 해석해보니 arg1이 입력 값보다 크면 return 2 * fun7(*(rdi+8), input_num)을, 다르면(즉, 작으면) return 2 * fun7(*(rdi+16), input_num) + 1을 실행, 같으면 return 0임을 확인할 수 있었다. 최종 return값이 7이 되려면 2*fun7(..)+1이 3번 실행되면 되므로 (2*(2*(2*0+1)+1)+1 = 7) 입력 값은 *(2*(2*(2*0+1)+1)+16)의 값과 같으면 된다.

0x5555557580f0 <n48>: 1001 해당 값은 1001이므로 답은 1001이다.