시스템 프로그래밍_201811259_배수빈

[BOMBLAB 풀이]

##첫번째문제 풀이 방법##

정답 : qjxlftndjqtek.

1. <strings_not_equal> 함수

```
0x000000000000172b <+0>:     push   %r12
0x000000000000172d <+2>:     push   %rbp
0x000000000000172e <+3>:     push   %rbx
0x000000000000172f <+4>:     mov    %rdi,%rbx
0x0000000000001732 <+7>:     mov    %rsi,%rbp
0x0000000000001735 <+10>:    callq  0x170e <string_length>
0x000000000000173a <+15>:    mov    %eax,%r12d
0x000000000000173d <+18>:    mov    %rbp,%rdi
0x0000000000001740 <+21>:    callq  0x170e <string_length>
0x0000000000001745 <+26>:    mov    $0x1,%edx
0x000000000000174a <+31>:    cmp    %eax,%r12d
0x000000000000174d <+34>:    je     0x1756 <strings_not_equal+43>
0x000000000000174f <+36>:    mov    %edx,%eax
0x0000000000001751 <+38>:    pop    %rbx
0x0000000000001752 <+39>:    pop    %rbp
0x0000000000001753 <+40>:    pop    %r12
0x0000000000001755 <+42>:    retq
0x0000000000001756 <+43>:    movzbl (%rbx),%eax
0x0000000000001759 <+46>:    test   %al,%al
0x000000000000175b <+48>:    je     0x1784 <strings_not_equal+89>
0x000000000000175d <+50>:    cmp    0x0(%rbp),%al
0x0000000000001760 <+53>:    jne    0x178b <strings_not_equal+96>
0x0000000000001762 <+55>:    add    $0x1,%rbx
0x0000000000001766 <+59>:    add    $0x1,%rbp
0x000000000000176a <+63>:    movzbl (%rbx),%eax
0x000000000000176d <+66>:    test   %al,%al
0x000000000000176f <+68>:    je     0x177d <strings_not_equal+82>
0x0000000000001771 <+70>:    cmp    %al,0x0(%rbp)
0x0000000000001774 <+73>:    je     0x1762 <strings_not_equal+55>
0x0000000000001776 <+75>:    mov    $0x1,%edx
0x000000000000177b <+80>:    jmp    0x174f <strings_not_equal+36>
0x000000000000177d <+82>:    mov    $0x0,%edx
0x0000000000001782 <+87>:    jmp    0x174f <strings_not_equal+36>
0x0000000000001784 <+89>:    mov    $0x0,%edx
0x0000000000001789 <+94>:    jmp    0x174f <strings_not_equal+36>
0x000000000000178b <+96>:    mov    $0x1,%edx
0x0000000000001790 <+101>:   jmp    0x174f <strings_not_equal+36>
```

입력하는 문자열과 저장되어있는 문자열을 비교하는 것이기 때문에 이 함수에서 정담을 알아 낼 수 있을 것이라고 생각해서 가장 처음 이 함수를 분석했습니다.

우선 이 함수에서는 %r12, %rbp %rbx의 값들을 잃지 않기 위해서 stack에 push해둡니다. 그 이후 %rdi에 저장되어있는 값을 %rbx에 저장시키고 rsi에 저장되어있는 값를 %rbp에 저장해 둡니다. 여기서 %rdi와 %rsi 에 어떤 값들이 매개변수로 이용되는지 알기 위해서 break를 다음줄(0x0000000000001735 <+10>)에 걸어두고 (x/s $rdi)를 통해 rdi와 rsi에 저장되는지 확인한 결과 $rdi에는 내가 첫 번째 폭탄에 대해 입력한 문자열, %rsi에는 'qjxlftndjqtek'이 저장되어있었습니다.

언제부터 %rsi에 'qjxlftndjqtek'가 저장되어있었는지 궁금해서 이 함수를 호출한 phase_1함수를 분석해보았습니다.

2. <phase_1> 함수

```
0x00005555555552c4 <+0>:     sub    $0x8,%rsp
0x00005555555552c8 <+4>:     lea    0xd1d(%rip),%rsi        # 0x555555555fec
0x00005555555552cf <+11>:    callq  0x55555555572b <strings_not_equal>
0x00005555555552d4 <+16>:    test   %eax,%eax
0x00005555555552d6 <+18>:    jne    0x5555555552dd <phase_1+25>
0x00005555555552d8 <+20>:    add    $0x8,%rsp
0x00005555555552dc <+24>:    retq
0x00005555555552dd <+25>:    callq  0x555555555c02 <explode_bomb>
0x00005555555552e2 <+30>:    jmp    0x5555555552d8 <phase_1+20>
```

phase_1함수의 어셈블리 코드에서 가장처음으로 하는 작업은 %rsp레지스터에서 8을 빼서 스택의 크기를 늘려주는 것입니다. 다음의 코드는 0xdld(%rip)를 lea연산하여 %rsi레지스터에 저장하는 것인데 breakpoint를 이 코드에 건 후 (break *phase_1+4) rsi레지스터의 값을 확인하기 위해 x/s $rsi를 하면 0x1: <error: Cannot access memory at address 0x1> 가 나오게 됩니다. 즉 lea연산을 하기 전에는 rsi 레지스터에 1이라는 주소가 저장되어있었기 때문입니다.

이후 %rsi레지스터에 lea연산의 결과값이 저장됩니다. lea연산을 통해 저장되는 결과값이 정답인지 확인하기 위해서 <strings_not_equal>함수를 call하기 전에 break를 걸어서 rsi레지스터에 저장된 값을 보니 'qjxlftndjqtek.' 가 나왔습니다.

##두번째문제 풀이 방법##

정답 : 1 6 11 16 21 26 / 2 7 12 17 22 27 등

```
mp of assembler code for function phase_2:
 0x00005555555552e4 <+0>:     push   %rbp
 0x00005555555552e5 <+1>:     push   %rbx
 0x00005555555552e6 <+2>:     sub    $0x28,%rsp
 0x00005555555552ea <+6>:     mov    %fs:0x28,%rax
 0x00005555555552f3 <+15>:    mov    %rax,0x18(%rsp)
 0x00005555555552f8 <+20>:    xor    %eax,%eax
 0x00005555555552fa <+22>:    mov    %rsp,%rbp
 0x00005555555552fd <+25>:    mov    %rbp,%rsi
 0x0000555555555300 <+28>:    callq  0x555555555c28 <read_six_numbers>
 0x0000555555555305 <+33>:    mov    %rbp,%rbx
 0x0000555555555308 <+36>:    add    $0x14,%rbp
 0x000055555555530c <+40>:    jmp    0x555555555317 <phase_2+51>
 0x000055555555530e <+42>:    add    $0x4,%rbx
 0x0000555555555312 <+46>:    cmp    %rbp,%rbx
 0x0000555555555315 <+49>:    je     0x555555555328 <phase_2+68>
 0x0000555555555317 <+51>:    mov    (%rbx),%eax
 0x0000555555555319 <+53>:    add    $0x5,%eax
 0x000055555555531c <+56>:    cmp    %eax,0x4(%rbx)
 0x000055555555531f <+59>:    je     0x55555555530e <phase_2+42>
 0x0000555555555321 <+61>:    callq  0x555555555c02 <explode_bomb>
 0x0000555555555326 <+66>:    jmp    0x55555555530e <phase_2+42>
 0x0000555555555328 <+68>:    mov    0x18(%rsp),%rax
 0x000055555555532d <+73>:    xor    %fs:0x28,%rax
 0x0000555555555336 <+82>:    jne    0x55555555533f <phase_2+91>
 0x0000555555555338 <+84>:    add    $0x28,%rsp
 0x000055555555533c <+88>:    pop    %rbx
 0x000055555555533d <+89>:    pop    %rbp
 0x000055555555533e <+90>:    retq
 0x000055555555533f <+91>:    callq  0x555555554eb0 <__stack_chk_fail@plt>
```

이 함수에서 read_six_numbers 이후에는

+33 : %rbp에 들어있는 내가 입력한 첫 번째 숫자의 주소를 %rbx에 넣어둔다.

+36 : %rbp 즉, 내가 입력한 첫 번째 숫자가 들어있는 주소에 20을 더한 주소를 rbp에 넣어둔다. (rbp에는 내가 입력한 여섯 번째 숫자가 들어있는 주소가 저장됨)

+51 : 이후 rbx에 들어있는 값 즉, 내가 입력한 첫 번재 숫자를 eax에 넣어둔다.

+53 : eax의 값 즉 첫 번째 숫자에 5를 더한 값을 eax에 저장한다.

+56 : rbx+4 의 주소에 있는 값 즉 두 번째로 입력한 값을 eax의 값(첫번째값 +5) 과 비교한다. 같지 않다면 폭탄은 터지고, 같다면 같은 방식으로 5번째 값까지 비교하고

+46 : 6번째 값이 들어있는 주소와 rbp즉 내가 첫 번째로 입력한숫자+20의 값이 저장되어있는 주소를 비교해서 같다면 문제는 풀린다!

즉 2번 문제는 n번째에 입력하는 수가 n-1번째에 입력하는 수보다 5커야하는 것이다. 따라서 답은 1 6 11 16 21 26 혹은 3 8 13 18 23 28의 방식으로 5씩 커지는 6개의 숫자이다.

##세번째문제 풀이 방법##

정답 : 0 198 / 1 587 / 2 341 / 3 330 / 4 253 / 5 125 / 6 755

세 번째 문제는 내가 입력한 것을 이 phase_3에서 필요한 인자로 만드는 <__isoc99_sscanf>에 들어가기전에 rsi를 확인하면 "%d %d"가 저장되어있다. 이는 재가 입력한 것이 정수 두 개를 사용할 것 임을 의미 하기 때문에 두 개의 정수를 입력해야함을 알 수 있다.

sscanf함수를 빠져 나온 이후에 rax에는 sscanf에서 변환시킨 값의 개수 (1개의 정수를 입력했으면 1, 2개의 정수를 입력했으면 2, 3개이상의 정수를 입력하더라도 2개의 정수만 변환되기 때문에 2) 가 저장된다. rsp에는 내가 입력한 첫 번째 정수가 저장되고, rsp+4에는 내가 입력한 두 번째 값이 저장된다.

+40 : rax값이 1보다 작거나 같으면 boom (즉 입력하는 값이 2개여야한다.)
+45 : rsp값이 7보다 크면 boom
(즉 내가 입력하는 첫 번째 정수는 7보다 작거나 같아야한다.)

```
0x0000555555555377 <+51>:  mov      (%rsp),%eax
0x000055555555537a <+54>:  lea      0xc7f(%rip),%rdx        # 0x555555556000
0x0000555555555381 <+61>:  movslq  (%rdx,%rax,4),%rax
0x0000555555555385 <+65>:  add      %rdx,%rax
0x0000555555555388 <+68>:  jmpq     *%rax
```

+51 : rsp 즉 첫 번째로 입력하는 숫자를 rax에 넣어준다.
+54 : rdx에 0x555555556000를 저장한다.
+61 : rdx +rax*4에 해당하는 주소를 rax에 저장한다. 이 주소는 rax에 따라 달라짐을 알 수 있다.
+65 : rax에 rdx를 더한다.
+68 : rax에 저장되어 있는 주소로 이동한다.

+51에서 +65까지의 연산들은 결국 rax에 앞으로 점프해야할 곳의 주소를 넣어주기 위해서 한 연산들이다. +61줄에서 4와 곱하는 수가 rax 즉 내가 입력한 첫 번째 정수이기 때문에 내가 입력하는 숫자가 작을수록 주소의 숫자가 작은 곳으로 이동할 것이고, 클수록 이동해야하는 곳의 주소가 큰 숫자 일 것이다.

첫 번째 정수로 1을 입력했을 때 +77로 이동해서 rax 에 0x24b 즉 587을 넣는다.
+77 : mov $0x24b,%eax
같은 방식으로 첫 번째 정수에 2를 넣었을 때 341, 3을 넣었을 때 330, 4를 넣었을 때 253, 5를 넣었을 때 125, 6을 넣었을 때 755가 저장이 된다.
첫 번째 정수로 0을 입력했을 때는 +131번째 줄로 이동해서 rax에 198을 저장한다.
이렇게 rax의 경우에 따라 값을 넣은 뒤에 143번줄로 이동한다.
+143 : cmp %eax,0x4(%rsp)
rax 와 rsp+4 즉 내가 입력한 두 번째 숫자를 비교하고 같으면 풀리고, 다르면 boom!

##네번째문제 풀이 방법##

답 : 23

- <phase_4> (func4호출 이전)

```
Dump of assembler code for function phase_4:
   0x0000555555555427 <+0>:     sub    $0x18,%rsp
   0x000055555555542b <+4>:     mov    %fs:0x28,%rax
   0x0000555555555434 <+13>:    mov    %rax,0x8(%rsp)
   0x0000555555555439 <+18>:    xor    %eax,%eax
   0x000055555555543b <+20>:    lea    0x4(%rsp),%rdx
   0x0000555555555440 <+25>:    lea    0xeb8(%rip),%rsi        # 0x5555555562ff
   0x0000555555555447 <+32>:    callq  0x555555554f80 <__isoc99_sscanf@plt>
   0x000055555555544c <+37>:    cmp    $0x1,%eax
   0x000055555555544f <+40>:    jne    0x555555555458 <phase_4+49>
   0x0000555555555451 <+42>:    cmpl   $0x0,0x4(%rsp)
   0x0000555555555456 <+47>:    jg     0x55555555545d <phase_4+54>
   0x0000555555555458 <+49>:    callq  0x555555555c02 <explode_bomb>
   0x000055555555545d <+54>:    mov    0x4(%rsp),%edi
   0x0000555555555461 <+58>:    callq  0x5555555553f8 <func4>
   0x0000555555555466 <+63>:    cmp    $0xb520,%eax
   0x000055555555546b <+68>:    je     0x555555555472 <phase_4+75>
   0x000055555555546d <+70>:    callq  0x555555555c02 <explode_bomb>
   0x0000555555555472 <+75>:    mov    0x8(%rsp),%rax
   0x0000555555555477 <+80>:    xor    %fs:0x28,%rax
   0x0000555555555480 <+89>:    jne    0x555555555487 <phase_4+96>
   0x0000555555555482 <+91>:    add    $0x18,%rsp
   0x0000555555555486 <+95>:    retq
   0x0000555555555487 <+96>:    callq  0x555555554eb0 <__stack_chk_fail@plt>
```

+32 : sscanf함수를 호출하기 전 rsi레지스터를 확인해보면 "%d"가 저장되어있다.
phase_3에서 말한바와 같이 입력하는 숫자가 하나임을 알 수 있다.

+37 : eax의 값과 1을 비교함으로써 입력한 정수가 하나가 보다 작으면 터짐을 알 수 있다.

+42 : rsp+4에는 내가 입력한 수가 저장되어있는 메모리의 주소가 저장되어있다.
즉 내가 입력한 수가 0보다 커야한다.

+54 : 내가 입력한 정수를 edi에 넣어둔다. (func4함수에 대한 첫 번째 인자로 넘어감)

+58 : <func4> 호출

- <func4>함수

```
Dump of assembler code for function func4:
   0x00000000000013f8 <+0>:     mov    $0x1,%eax
   0x00000000000013fd <+5>:     cmp    $0x1,%edi
   0x0000000000001400 <+8>:     jg     0x1404 <func4+12>
   0x0000000000001402 <+10>:    repz retq
   0x0000000000001404 <+12>:    push   %rbp
   0x0000000000001405 <+13>:    push   %rbx
   0x0000000000001406 <+14>:    sub    $0x8,%rsp
   0x000000000000140a <+18>:    mov    %edi,%ebx
   0x000000000000140c <+20>:    lea    -0x1(%rdi),%edi
   0x000000000000140f <+23>:    callq  0x13f8 <func4>
   0x0000000000001414 <+28>:    mov    %eax,%ebp
   0x0000000000001416 <+30>:    lea    -0x2(%rbx),%edi
   0x0000000000001419 <+33>:    callq  0x13f8 <func4>
   0x000000000000141e <+38>:    add    %ebp,%eax
   0x0000000000001420 <+40>:    add    $0x8,%rsp
   0x0000000000001424 <+44>:    pop    %rbx
   0x0000000000001425 <+45>:    pop    %rbp
   0x0000000000001426 <+46>:    retq
```

+0 : eax에 1을 넣어준다.
+5 : edi즉 내가 입력한 정수를 1과 비교하여 1보다 크면 func4+12로, 아니면 리턴한다.
// 입력하는 값이 0이하이면 리턴값이 무조건 1
+18 : edi즉 내가 입력한 정수를 ebx에 저장해둔다.
+20 : 내가 입력한 수에서 1을 뺀 수(ebx-1)를 인자로하여 func4를 실행(+23)한다.
+28 : +23에서 실행된 func4에 대한 리턴값이 eax에 저장되어있다.
이 eax값을 ebp에 저장한다.
+30 : 내가 입력한 수에서 2를 뺀 수(ebx-2)를 인자로하여 func4를 실행(+33)한다.
+38 : +33에서 실행된 func4에 대한 리턴값이 eax에 저장되어있다.
eax값에 ebp의 값을 더해 저장한다. (ebp : (내가입력한수-1)에대한 func4리턴값)
즉 이 함수는 리턴값이 1일 때 까지 재귀가 실행되는 것이다.
func4(n) = func4(n-1) + func4(n-2) 의 식으로 표현되는 함수 즉 피보나치 함수이다.

- <phase_4> (func4리턴 이후)
+63 : rax 즉 func4의 리턴값과 0xb520을 비교하여 같으면 리턴, 다르면 boom!

답내는 과정 :
0xb520은 46368 이다. 입력하는 정수가 23일 때 func4의 값이 46368이므로 답은 23이다.

##다섯번째문제 풀이 방법##
답 : aabced /112345

```
Dump of assembler code for function phase_5:
   0x000000000000148c <+0>:     push   %rbx
   0x000000000000148d <+1>:     mov    %rdi,%rbx
   0x0000000000001490 <+4>:     callq  0x170e <string_length>
   0x0000000000001495 <+9>:     cmp    $0x6,%eax
   0x0000000000001498 <+12>:    jne    0x14cb <phase_5+63>
   0x000000000000149a <+14>:    mov    %rbx,%rax
   0x000000000000149d <+17>:    lea    0x6(%rbx),%rdi
   0x00000000000014a1 <+21>:    mov    $0x0,%ecx
   0x00000000000014a6 <+26>:    lea    0xb73(%rip),%rsi        # 0x2020 <array.3409>
   0x00000000000014ad <+33>:    movzbl (%rax),%edx
   0x00000000000014b0 <+36>:    and    $0xf,%edx
   0x00000000000014b3 <+39>:    add    (%rsi,%rdx,4),%ecx
   0x00000000000014b6 <+42>:    add    $0x1,%rax
   0x00000000000014ba <+46>:    cmp    %rdi,%rax
   0x00000000000014bd <+49>:    jne    0x14ad <phase_5+33>
   0x00000000000014bf <+51>:    cmp    $0x37,%ecx
   0x00000000000014c2 <+54>:    je     0x14c9 <phase_5+61>
   0x00000000000014c4 <+56>:    callq  0x1c02 <explode_bomb>
   0x00000000000014c9 <+61>:    pop    %rbx
   0x00000000000014ca <+62>:    retq
   0x00000000000014cb <+63>:    callq  0x1c02 <explode_bomb>
   0x00000000000014d0 <+68>:    jmp    0x149a <phase_5+14>
```

+1 : rdi에 들어있는 값(내가 입력한 문자열이 저장되어있는 주소)을 rbx에 저장한다.
+4 : <string_length>에서 rax를 통해 리턴하는 값은 내가 입력한 문자열의 문자 개수이다.
+9 : eax(입력한 문자열의 문자갯수) 가 6이 아니면 boom! 맞으면 다음줄로 넘어간다.

즉 문자6개로 이루어진 문자열을 입력해야한다.

+12 : rbx(내가 입력한 문자열이 들어있는 첫번째주소)을 rax에 넣는다.

+17 : rbx+6 의 주소(문자열이 abcdef로 이어져있다면 f가 들어있는 주소보다 1큰 주소)를 rdi에 저장한다.

+21 : ecx에 0을 저장한다.

+26 : rsi에 0x2020의 주소(배열의 시작주소)를 저장한다.

+33 : rax에 저장되어있는 주소가 갖고 있는 값 (입력한 첫 번째 문자)을 edx에 넣어둔다. (edx에는 숫자를 입력할 경우 원래 입력한 값+0x30의 값, 문자(알파벳)를 입력했을 경우 알파벳의 인덱스 +0x60값이 더해져서 저장됨)

+36 : 여기서 edx(0011 nnnn) 과 0xf(0000 1111)을 and연산하면 nnnn만 남게되고 여기서 +33줄에서 생긴 +0x30의 값이 사라지고 내가 입력한 첫 번째 문자 (or 알파벳의 인덱스)만 남는다. => edx에 첫 번째로 입력한 숫자가 저장된다.

+39 : rsi(배열의 시작주소) + 4*rdx(입력한숫자) 의 주소가 갖는 값을 ecx에 저장한다.
// ex : 내가 입력한 값이 3이라면 (시작주소 +12)에 저장된값을 저장하는것

+42 : rax(내가 입력한 문자열이 들어있는 첫번째주소)에 1을 더한다.
다음문자로 넘어가는 것이다. (이전에 첫 번째 글자에 대해서 했다면 두번째문자로)

+46 : 앞서 진행한 인덱스+1에 해당하는 번째의 문자가 저장되어있는 주소(rax)와 rdi값을 비교하여 같지 않다면 다시 +33으로 돌아가 반복문을 돈다.

// 즉 +33에서 +46까지는 반복문 : 문자열의 처음부터 끝까지 해당연산을 하는 것!

+51 : 반복문을 돌면서 배열에서 배열의 인덱스 (인덱스 = 입력한 값)에 저장되어있는 값이 6번에 걸쳐 저장된상태의 rcx와 0x37 을 비교한다.
같으면 리턴, 다르면 boom!

답내는 과정 :

내 프로그램에는 배열이 0x2020에 저장되어있었다. (+26줄에 주석처리 되어있다.)
따라서 해당 주소부터 값들을 확인해보았다.

```
(gdb) x/18x 0x2020
0x2020 <array.3409>:    0x00000002      0x0000000a      0x00000006      0x00000001
0x2030 <array.3409+16>: 0x0000000c      0x00000010      0x00000009      0x00000003
0x2040 <array.3409+32>: 0x00000004      0x00000007      0x0000000e      0x00000005
0x2050 <array.3409+48>: 0x0000000b      0x00000008      0x0000000f      0x0000000d
0x2060: 0x21776f57      0x756f5920
```

배열은 종 16개로 이루어져있었으며 각각 4바이트로 이루어져있었다.

2 10 6 1 / 12 16 9 3 / 4 7 14 5 / 11 8 15 13

총 여섯 개의 숫자의 합이 0x37 즉 55가 되어야 한다.

10+ 10+ 6+ 1+ 12 +16 = 55 이므로 해당하는 숫자가 위치하는 배열의 인덱스6개가 답이다.

즉 1 1 2 3 4 5 가된다. aabcdf도 가능하다. 각 숫자의 자리가 바뀌어 511234도 가능하다.

##여섯번째문제 풀이 방법##

답 : 6 3 1 2 5 4

+26 : rsp와 r12가 같은 주소를 갖게 된다.
+32 : <read_six_numbers> 호출
함수 내부에서는 입력하는 정수의 개수가 6개 이하이면 Boom!
+37 : 함수가 리턴되고 나면
rax에는 내가입력한 값들의 개수가 입력되어있고,
r12 (rsp와 같은주소가짐) 에는 내가 입력한 첫 번째 숫자가 들어있는 주소가
저장되어있다. 4바이트 간격으로 이후의 숫자들도 저장되어있다.
+37 : r13d에 0을 대입, +82로 jmp //r13d에는 내가 입력하는 값들의 인덱스가 저장됨

---

③
+52 : ebx즉 인덱스가 1늘어나고 이 값이 5보다 크면 +78줄로 이동,

②
+60 (+60부터 +인덱스 즉, r13d, ebx가 1~5일때만 실행된다.)
: rax에 ebx의 값을 저장한다.
+63 : rsp+rax*4의 주소가 갖는 값(내가 입력한 다음숫자 n+1번째) 을 rax에 저장한다.
+66 : rax(n+1번째)와 rbp(내가 입력한 숫자 n번째)를 비교해서 같으면 boom!
+76 : 다르면 +52로 점프
//내가 입력한 값이 이전에 입력한 값과 다른 수 여야함

①
+78 : r12 +4 즉 내가 입력한 다음값으로 넘어간다.
+82 : r12 (내가 입력한 값이 저장되어있는 주소)를 rbp에 저장한다.
+85 : r12에 저장된 주소에 있는 값(내가 입력한 값)을 eax에 저장한다.
+89 : eax에서 1을 빼고 5와 비교한다. eax에서 1뺀 값이 5보다 큰 경우 boom!
+97 : eax에서 1뺀 값이 5보다 작거나 같은 경우
r13d(인덱스)에 1을 더해서
r13d가 6보다 작을 때까지 r13d가 6과 같아지면 +160으로 점프한다.
+107 : r13d를 ebx에 저장한 채로 +60으로 점프한다.

// 이 반복문에서는 총 여섯 개의 값의 1번째에서부터 6번째 까지 이 값이 6보다 크면 boom 하도록 한다. 여기서 얻어낸 조건은 내가 입력하는 6개의 숫자들은 모두 6보다 작거나 같아야함을 알 수 있다.

// ①②③의 거대한 반복문에서 얻어낼 수 있는 조건은 결국
'입력하는 여섯 개의 숫자는 서로 달라야하고 6보다 크면 안된다.' 이다.

위의 반복문을 이해하기가 힘들어서 정리해보았다.

int형의 배열의 이름을 arr라고할 때

```
for(int I = 0 ; i<6 ; Ii+){
        int num = arr[i];
        if(arr[i]-1 >5) { explode_bomb();}          //모든 숫자는 6보다 작거나 같아야함
        for( int j = i+1 ; j<6 ; j++){
                if(arr[j] == num) { explode_bomb();}
                        //입력하는 숫자는 모두 서로 달라야함
                        (여기서는 다음숫자와만 비교했지만 반복문을 돌며 모든 수를 비교)
        }
}
```

위의 반복문을 실행한 이후 내가 입력한 숫자의 배열이 조건에 해당한다면 이 부분으로 넘어 올 것이다.

우선 이 부분에서 주석으로 node1의 주소가 나와 있기 때문에 출력해서 노드가 어떤 값을 가지고 있는지 확인 해보았다.

```
(gdb) x/24x 0x203620
0x203620 <node1>:       0x0000026b      0x00000001      0x00203630      0x00000000
0x203630 <node2>:       0x00000215      0x00000002      0x00203640      0x00000000
0x203640 <node3>:       0x000002ed      0x00000003      0x00203650      0x00000000
0x203650 <node4>:       0x00000179      0x00000004      0x00203660      0x00000000
0x203660 <node5>:       0x00000191      0x00000005      0x00203120      0x00000000
0x203670:       0x00000000      0x00000000      0x00000000      0x00000000
```

4바이트씩 저장하고 있는 것이 무엇인지 확인해 보니 첫 번째 4바이트에는 숫자를 갖고있고, 두 번째 4바이트에는 인덱스, 세 번째 4바이트에는 다음노드의 주소를 저장하고 있었다. 즉 node는 하나의 구조체임을 알 수 있었다. 그리고 node5의 세 번째 4바이트에 저장되어있는 주소도 다음 노드의 주소를 가지고 있을 것이기 때문에 저 0x00203120도 출력해보았다.

```
(gdb) x/24x 0x00203120
0x203120 <node6>:       0x000003a7      0x00000006      0x00000000      0x00000000
```

node6도 같은 형태의 구조체였고 세 번째 4바이트에 0이 저장되어있는 것을 보니 이 노드가 마지막 노드임을 알수 있었다.

+160에서 시작하고

+138 : rsp+rsi*4 (rsp는 내가 입력한 수를 저장하는 주소. rsi로 인덱스 알수 있음)
의 주소에 저장된 값 (해당 인덱스에 맞는 내가 입력한 숫자)을 ecx에 저장한다.

+141 : rax에는 1저장

+146 : rdx에 <node1>이 저장되어있는 주소를 저장한다.

+153 : ecx즉 내가 입력한 값이 1보다 크면 +112로 점프

+112 : rdx+8 의 메모리에 접근하여 값을 rdx에 저장하는데 여기서 rdx에 저장하는 값은
다음노드의 주소이다.

노드의 번호가 내가 입력한 숫자와 같을 때까지 넘어가기 위해서이다.
+116 : **eax에 1을 더한다.** (인덱스 키우기)
+119 : ecx(내가 입력한 수) 와 eax(인덱스)와 비교해서 다르면 다시 **+112로 가서** 반복.
+123 : 같으면 그때 rdx에 저장되있는 주소 (ecx가 2였다면 node2가 저장되어있는 주소)
를 rsp+0x20+rsi*8에 저장한다.

//+112 ~ +123의 반복문은 내가 입력한 값과 같은 숫자를 갖는 노드의 주소를 찾는 반복문

+128 : **rsi에 1을 더하고**
+136 : rsi가 6과 같을 때까지 **+138로 돌아가서** 해당 반복문을 반복한다.

//+138 ~ +136의 반복분은 (+112 ~ +123의 반복문)에서 찾아낸 노드의 주소를 저장할 스택의 주소를 한칸 씩(8바이트) 키워주는 것

이렇게 스택에 노드의 주소들을 차례로 저장한 뒤에 +167로 점프한다.

+167 : 첫 번째 노드의 주소를 rbx에 저장해둔다.

+167부터 +217까지는 스택에 내가 저장해둔 노드 순서대로
첫 번째 노드의 주소 +8에 두 번째 노드의 주소를 저장해두고
두 번째 노드의 주소 +8에 세 번째 노드의 주소를 저장해두고
세 번째 노드의 주소 +8에 네 번째 노드의 주소를 저장해두고
네 번째 노드의 주소 +8에 다섯 번째 노드의 주소를 저장해두고
다섯 번째 노드의 주소 +8에 여섯 번째 노드의 주소를 저장해두고
여섯 번째 노드의 주소 +8에 0을 저장해둔다.

이렇게 저장해둔 뒤 연산을 시작한다.
+241 : rbx+8 (첫번째 노드의 주소+8에 저장되어 있는 값 즉 두 번째 노드의 주소)
를 rax에 넣고
+245 : rax (두번째 노드의 주소에 저장되어있는 값)을 eax에 옮긴다.
+247 : eax(두번째 노드의 값)와 rbx의 값(첫번째 노드의 값)을 비교한다.
두 번째 노드의 값이 더 크면 boom!
첫 번째 노드의 값이 더 크면 rbx(첫번째 노드의 주소)에
rbx+8(두번째 노드의 주소)를 저장한다.
//다음 노드의 주소를 rbx에 저장하여 연산을 반복한다.
이 연산은 즉 다음 노드에 저장되어 있는 값이 해당 노드에 저장되어 있는 값보다 작거나 같아야 됨을 의미한다.

따라서 노드의 저장되어 있는 값들이 내림차순 되어야하고 그 값들이 저장되어있는 노드의

번호에 해당하는 수열이 답이 되는 것이다.

답내는 과정 :

```
(gdb) x/24x 0x203620
0x203620 <node1>:       0x0000026b      0x00000001      0x00203630      0x00000000
0x203630 <node2>:       0x00000215      0x00000002      0x00203640      0x00000000
0x203640 <node3>:       0x000002ed      0x00000003      0x00203650      0x00000000
0x203650 <node4>:       0x00000179      0x00000004      0x00203660      0x00000000
0x203660 <node5>:       0x00000191      0x00000005      0x00203120      0x00000000
0x203670:       0x00000000      0x00000000      0x00000000      0x00000000
```

```
(gdb) x/24x 0x00203120
0x203120 <node6>:       0x000003a7      0x00000006      0x00000000      0x00000000
```

값들을 내림차순으로 정리해보면
3a7, 2ed, 26d, 215, 191, 179 이다.
해당 값들의 노드번호는 6, 3, 1, 2, 5, 4이므로
6 3 1 2 5 4가 정답인 것이다.