먼저 disas main 명령어를 이용하여 각 phase를 나타내는 함수들의 이름이 phase_1, phase_2, … , phase_6임을 확인했다.

Phase_1

```
(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  0x1b2f <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  0x1c43 <explode_bomb>
   0x00000000000015c9 <+34>:    jmp    0x15bf <phase_1+24>
End of assembler dump.
(gdb) x/s 0x314c
0x314c: "Wow! Brazil is big."
(gdb) r
Starting program: /home/std/greenidea/CSED211/Lab3/bomb314/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
```

disassemble을 하자 위와 같이 나왔다. 먼저 주목한 부분은 <+8>, <+15>였다. <+8>에서 rsi 레지스터에 어떤 content가 들어가고, 그다음 strings_not_equal이라는 함수가 호출된다. 그런데 rsi 레지스터는 보통 함수의 첫번째 인자에 대응되는 레지스터이다. 또, 함수의 이름을 봤을 때 이 함수는 스트링이 같은지를 비교하는 함수인 것 같다고 생각했다. 그렇다면 rsi 레지스터에 들어가는 content는 string이 아닐까라는 생각으로 x/s 0x314c를 수행하여 content를 확인했다. 그리고 strings_not_equal의 리턴값은 rax 레지스터에 담길 것이다. 그런데 <+20>, <+22>를 보면 방금 찾았던 문자열과 같은 문자열이 아니라면 <+29>로 점프시키는 것이 아닐까하는 생각이 들었다.

그래서 실행시켜 Wow! Brazil is big.을 입력했더니 phase 1이 defused되었다. strings_not_equal을 굳이 disas하지 않고 간단하게 문제가 풀려버렸다.

Phase_2

```
(gdb) disas phase_2
Dump of assembler code for function phase_2:
   0x00000000000015cb <+0>:     repz nop %edx
   0x00000000000015cf <+4>:     push   %rbp
   0x00000000000015d0 <+5>:     push   %rbx
   0x00000000000015d1 <+6>:     sub    $0x28,%rsp
   0x00000000000015d5 <+10>:    mov    %fs:0x28,%rax
   0x00000000000015de <+19>:    mov    %rax,0x18(%rsp)
   0x00000000000015e3 <+24>:    xor    %eax,%eax
   0x00000000000015e5 <+26>:    mov    %rsp,%rsi
   0x00000000000015e8 <+29>:    callq  0x1c6f <read_six_numbers>
   0x00000000000015ed <+34>:    cmpl   $0x0,(%rsp)
   0x00000000000015f1 <+38>:    js     0x15fd <phase_2+50>
   0x00000000000015f3 <+40>:    mov    %rsp,%rbp
   0x00000000000015f6 <+43>:    mov    $0x1,%ebx
   0x00000000000015fb <+48>:    jmp    0x1615 <phase_2+74>
   0x00000000000015fd <+50>:    callq  0x1c43 <explode_bomb>
   0x0000000000001602 <+55>:    jmp    0x15f3 <phase_2+40>
   0x0000000000001604 <+57>:    callq  0x1c43 <explode_bomb>
   0x0000000000001609 <+62>:    add    $0x1,%ebx
   0x000000000000160c <+65>:    add    $0x4,%rbp
   0x0000000000001610 <+69>:    cmp    $0x6,%ebx
   0x0000000000001613 <+72>:    je     0x1621 <phase_2+86>
   0x0000000000001615 <+74>:    mov    %ebx,%eax
   0x0000000000001617 <+76>:    add    0x0(%rbp),%eax
   0x000000000000161a <+79>:    cmp    %eax,0x4(%rbp)
   0x000000000000161d <+82>:    je     0x1609 <phase_2+62>
   0x000000000000161f <+84>:    jmp    0x1604 <phase_2+57>
   0x0000000000001621 <+86>:    mov    0x18(%rsp),%rax
   0x0000000000001626 <+91>:    xor    %fs:0x28,%rax
   0x000000000000162f <+100>:   jne    0x1638 <phase_2+109>
   0x0000000000001631 <+102>:   add    $0x28,%rsp
   0x0000000000001635 <+106>:   pop    %rbx
   0x0000000000001636 <+107>:   pop    %rbp
   0x0000000000001637 <+108>:   retq
   0x0000000000001638 <+109>:   callq  0x1220
End of assembler dump.
```

phase_2를 disassemble했다. 그리고 눈에 들어온 것은 <+29>에서 호출되는 read_six_numbers였다. 그래서 이 함수도 disassemble했다.

```
(gdb) disas read_six_numbers
Dump of assembler code for function read_six_numbers:
   0x0000000000001c6f <+0>:     repz nop %edx
   0x0000000000001c73 <+4>:     sub    $0x8,%rsp
   0x0000000000001c77 <+8>:     mov    %rsi,%rdx
   0x0000000000001c7a <+11>:    lea    0x4(%rsi),%rcx
   0x0000000000001c7e <+15>:    lea    0x14(%rsi),%rax
   0x0000000000001c82 <+19>:    push   %rax
   0x0000000000001c83 <+20>:    lea    0x10(%rsi),%rax
   0x0000000000001c87 <+24>:    push   %rax
   0x0000000000001c88 <+25>:    lea    0xc(%rsi),%r9
   0x0000000000001c8c <+29>:    lea    0x8(%rsi),%r8
   0x0000000000001c90 <+33>:    lea    0x1674(%rip),%rsi        # 0x330b
   0x0000000000001c97 <+40>:    mov    $0x0,%eax
   0x0000000000001c9c <+45>:    callq  0x12c0
   0x0000000000001ca1 <+50>:    add    $0x10,%rsp
   0x0000000000001ca5 <+54>:    cmp    $0x5,%eax
   0x0000000000001ca8 <+57>:    jle    0x1caf <read_six_numbers+64>
   0x0000000000001caa <+59>:    add    $0x8,%rsp
   0x0000000000001cae <+63>:    retq
   0x0000000000001caf <+64>:    callq  0x1c43 <explode_bomb>
End of assembler dump.
(gdb) x/s 0x330b
0x330b: "%d %d %d %d %d %d"
```

그리고 x/s 0x330b를 하여 0x330b의 메모리 content를 확인했다. %d가 6개, 즉 정수 6개가 입력으로 들어온다는 것을 캐치할 수 있었다. 그리고 다시 phase_2로 돌아가서 read_six_numbers후에 무엇이 진행되는지를 확인했다. <+34>, <+38>에서 %rsp에 저장된 값이 음수이면 <+50>으로 점

프하여 폭탄이 터진다는 것을 알 수 있었다. 그래서 %rsp에 들어있는 값이 무엇인지 알아보고자 다음과 같이 callq후에 breakpoint를 만들어줬다.

```
(gdb) b *phase_2+34
Breakpoint 2 at 0x5555555555ed
```

그다음 실행을 시키고 임의의 수 여섯개를 입력해보았다.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/std/greenidea/CSED211/Lab3/bomb314/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 4 8 9 14 2
```

그다음 rsp의 content를 10진수로 확인해보았다.

```
(gdb) x/d $rsp
0x7fffffffe450: 1
(gdb) x/6d $rsp
0x7fffffffe450: 1       4       8       9
0x7fffffffe460: 14      2
```

위와 같이 1이 나왔다. 그래서 1이 첫 입력이었으니 rsp부터 6개의 입력이 저장된 것이 아닐까? 라고 생각하고 4바이트씩 6번, 즉 x/6d커맨드로 rsp부터 6개의 content를 확인해보았다. 그랬더니 입력한 수들이 순서대로 나옴을 확인할 수 있었다.

일단 지금까지 아는 사실은 첫 입력이 음수가 되면 안된다는 것이다. 즉, 첫 입력은 0 또는 양의 정수여야 한다.

```
0x00005555555555f3 <+40>:    mov    %rsp,%rbp
0x00005555555555f6 <+43>:    mov    $0x1,%ebx
0x00005555555555fb <+48>:    jmp    0x555555555615 <phase_2+74>
0x00005555555555fd <+50>:    callq  0x555555555c43 <explode_bomb>
0x0000555555555602 <+55>:    jmp    0x5555555555f3 <phase_2+40>
0x0000555555555604 <+57>:    callq  0x555555555c43 <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
```

첫 입력이 음수가 아니라면 rbp에 rsp값이 저장된다. 그리고 rbx에 1이 저장된다. 그리고 <+74>로 이동한다. rax에 rbx가 저장되므로 rax = 1인 상황이다. 그리고 rax에 rbp+0이 더해진다. rbp+0

에는 rsp가 들어있으므로 rax = 1 + rsp가 된다. 그리고 rax와 rbp+4를 cmp로 비교한다. 그다음 je, 즉 두개가 같다면 <+62>로 이동하고 다르다면 <+57>로 이동해서 폭탄이 터지게 된다. 따라서 1+rsp와 rbp+4가 같아야한다. 1+rsp는 첫번째 입력+1을 의미하고, rbp+4는 두번째 입력을 의미하므로(rsp부터 4바이트씩 저장되어있으므로) 두번째 입력이 첫번째입력+1이 되어야한다는 조건을 얻을 수 있다.

이 조건도 만족하여 <+62>로 이동했다고 하자. 그럼 rbx에 1이 더해지고(현재 rbx=2), rbp에 4가 더해진다.(현재rbp=rsp+4) 그리고 6과 rbx를 비교한다. 그리고 6과 rbx는 다르므로 <+74>에 진입하여 방금전에 했던 과정이 그대로 반복된다. 단, 달라진 점은 rbx가 아까보다 1이 커진 2라는 것과, rbp에 4가 더해져 rbp는 첫번째 인풋의 다음 인풋인 두번째 인풋을 가리킨다는 것이다. 즉, 이번에는 세번째 입력이 두번째입력+2가 되어야한다는 조건이 나오게 된다.

그리고 다시 <+62>로 이동하여 rbx에 1이, rbp에 4가 더해질것이다. 그리고 6과 rbx를 비교할 것이다. 즉, 첫번째 인풋부터 마지막 인풋까지 순서대로 1, 2, 3, 4, 5가 더해지고 있다는 것을 알 수 있다. 첫번째 숫자를 한개로 좁힐 수 있는 조건은 따로 없어서 0이상의 숫자인 1을 시작으로 1, 2, 3, 4, 5가 더해지는 인풋을 입력해보았다. 페이즈 2를 통과했음을 확인할 수 있었다.

```
(gdb) r
Starting program: /home/std/greenidea/CSED211/Lab3/bomb314/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
```

Phase_3

```
(gdb) disas phase_3
Dump of assembler code for function phase_3:
   0x000055555555563d <+0>:     repz nop %edx
   0x0000555555555641 <+4>:     sub    $0x28,%rsp
   0x0000555555555645 <+8>:     mov    %fs:0x28,%rax
   0x000055555555564e <+17>:    mov    %rax,0x18(%rsp)
   0x0000555555555653 <+22>:    xor    %eax,%eax
   0x0000555555555655 <+24>:    lea    0xf(%rsp),%rcx
   0x000055555555565a <+29>:    lea    0x10(%rsp),%rdx
   0x000055555555565f <+34>:    lea    0x14(%rsp),%r8
   0x0000555555555664 <+39>:    lea    0x1af5(%rip),%rsi        # 0x555555557160
   0x000055555555566b <+46>:    callq  0x5555555552c0
```

먼저 <+39>를 보고 x/s를 이용하여 content를 확인했다.

```
(gdb) x/s 0x555555557160
0x555555557160: "%d %c %d"
```

인풋이 정수 문자 정수 형태임을 알 수 있었다.

```
0x0000555555555670 <+51>:    cmp    $0x2,%eax
0x0000555555555673 <+54>:    jle    0x555555555695 <phase_3+88>
0x0000555555555675 <+56>:    cmpl   $0x7,0x10(%rsp)
0x000055555555567a <+61>:    ja     0x55555555578d <phase_3+336>
0x0000555555555680 <+67>:    mov    0x10(%rsp),%eax
0x0000555555555684 <+71>:    lea    0x1af5(%rip),%rdx        # 0x555555557180
0x000055555555568b <+78>:    movslq (%rdx,%rax,4),%rax
0x000055555555568f <+82>:    add    %rdx,%rax
0x0000555555555692 <+85>:    ds
0x0000555555555693 <+86>:    jmpq   *%rax
```

<+51>, <+54>에서 rax가 2이하이면 터진다는 것을 알 수 있었다. 그래서 rax에 들어가는 값이 뭔지 확인하려고 테스트를 해보았다.

```
That's number 2.  Keep going!
3 d 5

Breakpoint 3, 0x0000555555555670 in phase_3 ()
(gdb) i r rax
rax            0x3      3
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/std/greenidea/CSED211/Lab3/bomb314/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
14 r 7

Breakpoint 3, 0x0000555555555670 in phase_3 ()
(gdb) i r rax
rax            0x3      3
```

먼저 인풋을 3 d 5로 넣었는데 rax에 저장된 값이 3이 나와서, rax에 첫 입력이 저장되는건지 체크하기 위해 다른 입력을 넣어봤는데 여전히 3이 나와서 그건 아니라는 결론을 얻었다. rax가 2 이하가 되어 터지는 조건이 무엇인지는 모르겠지만 일단 넘어가기로 했다.

그다음은 체크할 부분은 <+56>, <+61>이다. rsp+16이 7보다 크면 <+336>으로 점프하여 터진다. 따라서 0x10은 10진수로 16이므로 rsp+16은 7이하여야 한다. 그래서 <+56>에 breakpoint를 만들어 rsp+16에 어떤 것이 저장되어 있는지 확인해보았다.

```
(gdb) b *phase_3+56
Breakpoint 4 at 0x555555555675
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/std/greenidea/CSED211/Lab3/bomb314/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
23 z 6
```

```
(gdb) x/d $rsp+16
0x7fffffffe470: 23
(gdb)
```

인풋으로 23 z 6을 주었는데, rsp+16에도 23이 저장되어 있다. 즉, 첫번째 인풋이 rsp+16에 저장

된다는 것을 확인할 수 있었다. 따라서 조건을 하나 얻었다. 첫번째 인풋은 7이하의 정수이다.

이 조건을 만족했다고 가정하고 <+67>부터 다시 읽어보면, rax에 어떤 값이 저장됨을 확인할 수 있다. 그리고 <+86>에서 rax가 가리키는 주소로 점프한다.

```
 0x0000555555555693 <+86>:    jmpq   *%rax
```

그래서 rax가 가리키는 주소를 알아보기 위해 <+86>에 breakpoint를 만들었다.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/std/greenidea/CSED211/Lab3/bomb314/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
3 d 9

Breakpoint 1, 0x0000555555555693 in phase_3 ()
(gdb) i r rax
rax            0x555555555702   93824992237314
```

그래서 확인해보니 jmpq *%rax의 결과 <+197> 지점으로 이동함을 확인할 수 있었다.

```
0x0000555555555702 <+197>:   mov    $0x61,%eax
0x0000555555555707 <+202>:   cmpl   $0x225,0x14(%rsp)
0x000055555555570f <+210>:   je     0x555555555797 <phase_3+346>
-Type <return> to continue, or q <return> to quit---
0x0000555555555715 <+216>:   callq  0x555555555c43 <explode_bomb>
```

rax에 0x61(십진수로 97)이 저장되고 rsp+0x14(십진수로 rsp+20)와 0x225(십진수로 549)가 같으면 <+346>으로 점프하고 다르면 터지는 것을 확인할 수 있다. 느낌상 rsp+16에 첫번째 인풋이 저장되었었고, 549와 같아야 된다는 것을 볼때 세번째 인풋이 549여야 된다는 조건을 줄 것 같기는 하다. 그래도 확실하지 않으니 일단 <+202>에 breakpoint를 만들어보았다.

```
(gdb) b *phase_3+202
Breakpoint 2 at 0x555555555707
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/std/greenidea/CSED211/Lab3/bomb314/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
3 h 892

Breakpoint 2, 0x0000555555555707 in phase_3 ()
(gdb) x/d $rsp+20
0x7fffffffe474: 892
```

확인해본 결과 예상대로 rsp+20에는 세번째 인풋이 들어있었다. 결국 세번째 인풋은 549가 되어

야한다는 것이다. 이제 세번째 인풋이 549라고 가정하면 점프가 수행되어 <+346>에 도착할테니 여기를 살펴보자.

```
0x0000555555555797 <+346>:   cmp    %al,0xf(%rsp)
0x000055555555579b <+350>:   jne    0x5555555557b2 <phase_3+373>
0x000055555555579d <+352>:   mov    0x18(%rsp),%rax
0x00005555555557a2 <+357>:   xor    %fs:0x28,%rax
0x00005555555557ab <+366>:   jne    0x5555555557b9 <phase_3+380>
0x00005555555557ad <+368>:   add    $0x28,%rsp
0x00005555555557b1 <+372>:   retq
0x00005555555557b2 <+373>:   callq  0x555555555c43 <explode_bomb>
0x00005555555557b7 <+378>:   jmp    0x55555555579d <phase_3+352>
0x00005555555557b9 <+380>:   callq  0x555555555220
```

rsp+15가 rax와 다르면 <+373>으로 이동해서 터진다는 것을 알 수 있다. rsp+15는 문자였던 두번째 인풋일 가능성이 매우 높아보인다. 그런데 어차피 rax에 들어있는 값을 확인해야하니 <+346>에 breakpoint를 만들어두고 rsp+15와 rax를 확인해보자.

```
That's number 2.  Keep going!
3 A 549

Breakpoint 3, 0x0000555555555797 in phase_3 ()
(gdb) i r rax
rax            0x61     97
```

먼저 rax에는 0x61, 즉 아스키코드표에 따라 소문자 a가 들어있음을 확인할 수 있다.

```
(gdb) x/bx $rsp+15
0x7fffffffe46f: 0x41
```

그리고 rsp+15에는 두번째 인풋으로 주었던 A에 대응되는 16진수인 0x41이 들어있음을 확인할 수 있다. 따라서 rsp+15는 두번째 인풋을 의미하는 것이 맞고, 두번째 인풋은 a가 되어야 함을 알 수 있다.

첫번째 인풋을 처음에 문제가 없었던 3으로두고 ,두번째 인풋을 a, 세번째 인풋을 549로 하여 phase_3를 통과할 수 있었다.

```
(gdb) r
Starting program: /home/std/greenidea/CSED211/Lab3/bomb314/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
3 a 549
Halfway there!
```

Phase_4

먼저 <+32>의 content를 확인했다.

```
0x000055555555557f4 <+0>:     repz nop %edx
0x000055555555557f8 <+4>:     sub    $0x18,%rsp
0x000055555555557fc <+8>:     mov    %fs:0x28,%rax
0x0000555555555805 <+17>:     mov    %rax,0x8(%rsp)
0x000055555555580a <+22>:     xor    %eax,%eax
0x000055555555580c <+24>:     lea    0x4(%rsp),%rcx
0x0000555555555811 <+29>:     mov    %rsp,%rdx
0x0000555555555814 <+32>:     lea    0x1afc(%rip),%rsi        # 0x555555557317
```

그 결과 두개의 정수를 입력으로 받음을 확인할 수 있었다.

```
(gdb) x/s 0x555555557317
0x555555557317: "%d %d"
```

그리고 <+44>, <+47>에서 rax가 2가 아니면 <+55>로 이동하여 터지는 것을 확인할 수 있다.

```
0x0000555555555814 <+32>:    lea    0x1afc(%rip),%rsi        # 0x555555557317
0x000055555555581b <+39>:    callq  0x5555555552c0
0x0000555555555820 <+44>:    cmp    $0x2,%eax
0x0000555555555823 <+47>:    jne    0x55555555582b <phase_4+55>
```

따라서 <+44>에 breakpoint를 설정한 후 rax에 들어가는 값이 무엇인지 확인해보았다.

```
Halfway there!
7 9

Breakpoint 4, 0x0000555555555820 in phase_4 ()
(gdb) i r rax
rax            0x2      2
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/std/greenidea/CSED211/Lab3/bomb314/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
3 a 549
Halfway there!
5 8

Breakpoint 4, 0x0000555555555820 in phase_4 ()
(gdb) i r rax
rax            0x2      2
```

7 9 그리고 5 8을 인풋으로 2번 확인해본결과 2번 모두 rax에는 2가 들어있다. 생각해보니 <+39>에서 호출된것은 scanf였을거고 이것의 리턴 벨류는 인풋의 개수이니 이 부분은 인풋으로 2개의 정수가 잘 입력되었는지를 확인하고 있다는 것을 알 수 있었다.

```
0x0000555555555825 <+49>:    cmpl   $0xe,(%rsp)
0x0000555555555829 <+53>:    jbe    0x555555555830 <phase_4+60>
0x000055555555582b <+55>:    callq  0x555555555c43 <explode_bomb>
```

그래서 그다음으로 넘어가서 <+49>, <+53>을 확인해보았다. 여기서는 rsp가 14보다 작거나 같으면 <+60>으로 점프하고 그렇지 않으면 터지는 것을 확인할 수 있었다. 그래서 또 <+49>에

breakpoint를 설정하고 rsp에 저장된 값을 확인해보았다.

```
Halfway there!
7 5

Breakpoint 5, 0x0000555555555825 in phase_4 ()
(gdb) i r rsp
rsp            0x7fffffffe470   0x7fffffffe470
(gdb) x/d $rsp
0x7fffffffe470: 7
```

```
Halfway there!
43 5

Breakpoint 5, 0x0000555555555825 in phase_4 ()
(gdb) x/d $rsp
0x7fffffffe470: 43
```

rsp에는 첫번째 인풋이 들어간다는 것을 확인할 수 있었다. 따라서 첫번째 인풋은 14이하의 정수라는 조건을 얻었다. 그래서 이 조건을 만족한다고 가정하고 <+60>으로 이동하자.

```
0x0000555555555830 <+60>:    mov    $0xe,%edx
0x0000555555555835 <+65>:    mov    $0x0,%esi
0x000055555555583a <+70>:    mov    (%rsp),%edi
0x000055555555583d <+73>:    callq  0x5555555557be <func4>
0x0000555555555842 <+78>:    cmp    $0x15,%eax
0x0000555555555845 <+81>:    jne    0x55555555584e <phase_4+90>
```

%rdx에 14를 저장한다. 그리고 rsi에 0을 저장한다. rdi에 rsp, 즉 첫번째 인풋을 저장한다. 그리고 함수 func4를 호출한다. 그리고 함수의 리턴벨류인 rax가 21이 아니면 <+90>으로 점프하여 터진다. 따라서 함수의 리턴 벨류가 21이 되어야한다. 함수가 어떻게 돌아가는지 알기위해 func4를 disassemble해보자.

```
(gdb) disas func4
Dump of assembler code for function func4:
   0x00005555555557be <+0>:     repz nop %edx
   0x00005555555557c2 <+4>:     push   %rbx
   0x00005555555557c3 <+5>:     mov    %edx,%eax
   0x00005555555557c5 <+7>:     sub    %esi,%eax
   0x00005555555557c7 <+9>:     mov    %eax,%ebx
   0x00005555555557c9 <+11>:    shr    $0x1f,%ebx
   0x00005555555557cc <+14>:    add    %eax,%ebx
   0x00005555555557ce <+16>:    sar    %ebx
   0x00005555555557d0 <+18>:    add    %esi,%ebx
   0x00005555555557d2 <+20>:    cmp    %edi,%ebx
   0x00005555555557d4 <+22>:    jg     0x5555555557dc <func4+30>
   0x00005555555557d6 <+24>:    jl     0x5555555557e8 <func4+42>
   0x00005555555557d8 <+26>:    mov    %ebx,%eax
   0x00005555555557da <+28>:    pop    %rbx
   0x00005555555557db <+29>:    retq
   0x00005555555557dc <+30>:    lea    -0x1(%rbx),%edx
   0x00005555555557df <+33>:    callq  0x5555555557be <func4>
   0x00005555555557e4 <+38>:    add    %eax,%ebx
   0x00005555555557e6 <+40>:    jmp    0x5555555557d8 <func4+26>
   0x00005555555557e8 <+42>:    lea    0x1(%rbx),%esi
   0x00005555555557eb <+45>:    callq  0x5555555557be <func4>
   0x00005555555557f0 <+50>:    add    %eax,%ebx
---Type <return> to continue, or q <return> to quit---
   0x00005555555557f2 <+52>:    jmp    0x5555555557d8 <func4+26>
End of assembler dump.
```

<+5>, <+7>에서 rax = rdx – rsi가된다. 그리고 <+9>, <+11>, <+14>, <+16> <+18>에 의해 rbx = (rax>>31) + (rax>>1) + rsi = 0 + rax/2 + rsi가 된다. rax = rdx-rsi = 14-0 = 14이고 rsi = 0이므로 rbx = 7이다. 그리고 <+20>에서 rbx=7과 첫번째 인풋인 rdi를 비교하여 rbx가 크면 <+30>으로, 작으면 <+42>로 점프하고, 같은 경우 rax = rbx를 수행하고 함수를 종료하게 된다. 일단 지금 상황에서 같으면 7을 리턴하고 함수가 끝나서 리턴벨류가 21이 아니므로 터지게 된다. 따라서 <+30> 또는 <+42>로의 점프가 이루어져야한다는 것을 알 수 있다.

<+30>부터 살펴보자. <+30>, <+33>에 breakpoint를 만들어주고 <+30>으로 점프시키기 위해 rbx=7보다 작은 값인 5를 첫번째 인풋으로 넣었다. rbx에 7, rdx에 14가 있는 것을 확인하고, <+30>에서 lea명령이 실행된후 rbx-1인 6이 rdx에 들어가는 것을 확인했다.

```
Halfway there!
5 21

Breakpoint 5, 0x00005555555557dc in func4 ()
(gdb) i r rbx
rbx            0x7      7
(gdb) i r rdx
rdx            0xe      14
(gdb) ni

Breakpoint 6, 0x00005555555557df in func4 ()
(gdb) i r rdx
rdx            0x6      6
(gdb)
```

그리고 func4가 다시 호출된다. 이말은 rdx값이 달라진 상태로 지금까지의 과정이 반복된다는 것이다. 그리고 이 func4의 리턴값인 rax를 rbx에 더해 리턴하여 함수가 끝난다. 일단 이건 이상태로 두고 <+42>를 먼저 살펴보자. rsi = rbx+1을하고 func4를 다시 호출한다. 이말은 rsi값이 달라진 상태로 지금까지의 과정이 반복된다는 것이다. 그리고 이 func4의 리턴값인 rax를 rbx에 더해 리턴하여 함수가 끝난다. 지금까지 분석한 내용을 바탕으로 func4를 C코드로 바꾸어 보았다.

```
int func4(int rdi, int rsi, int rdx)

{

        int rax = rdx – rsi;

        int rbx = (rax>>31) +rax/2 + rsi;

        if(rbx > rdi) return func4(rdi, rsi, rbx-1) + rbx;

        else if(rbx < rdi) return func4(rdi, rbx+1, rdx) + rbx;

        else return rbx;

}
```

초기 rsi = 0, rdx = 14이므로 rbx == rdi라서 바로 리턴되는 경우는 리턴벨류가 7이 되어 터지게 되므로 인풋은 7이 아닌 수가 되어야 한다. 또, 리턴벨류가 21이 되어야 하고, rbx = 7이므로 재귀적으로 도는 func4의 리턴벨류는 14가 되어야함을 알 수 있다. 이제 이 재귀함수의 규칙을 찾으면 되는데 굳이 규칙을 직접 찾을 필요가 없다. 간단하게 저 함수에서 rdi를 signed int의 범위에 해당되는 모든 정수 i에 대해 for문을 돌려 func4의 리턴값이 21이 되는 i를 찾아주는 프로그램을 짜서 확인해보면 되기 때문이다. 그 결과 func4의 리턴벨류를 21로 만들어주는 rdi는 6임을 확인할 수 있었다. 즉, 첫번째 인풋은 6이 되어야 한다.

첫번째 인풋이 6이라면 <+81>에서 점프가 일어나지 않을 것이므로 <+83>을 보자. rsp+4가 21이 아니면 <+88>에서 점프가 일어나지 않아서 터지게 된다.

```
0x000055555555583d <+73>:    callq  0x5555555557be <func4>
0x0000555555555842 <+78>:    cmp    $0x15,%eax
0x0000555555555845 <+81>:    jne    0x55555555584e <phase_4+90>
0x0000555555555847 <+83>:    cmpl   $0x15,0x4(%rsp)
0x000055555555584c <+88>:    je     0x555555555853 <phase_4+95>
0x000055555555584e <+90>:    callq  0x555555555c43 <explode_bomb>
0x0000555555555853 <+95>:    mov    0x8(%rsp),%rax
0x0000555555555858 <+100>:   xor    %fs:0x28,%rax
0x0000555555555861 <+109>:   jne    0x555555555868 <phase_4+116>
0x0000555555555863 <+111>:   add    $0x18,%rsp
0x0000555555555867 <+115>:   retq
0x0000555555555868 <+116>:   callq  0x555555555220
```

따라서 rsp+4에 들어있는 것이 무엇이지 확인해보면 된다. <+83>에 breakpoint를 걸어두고 확인해보았다.

```
Halfway there!
6 73

Breakpoint 1, 0x0000555555555847 in phase_4 ()
(gdb) x/d $rsp+4
0x7fffffffe474: 73
```

예상은 했지만 역시 그 정체는 두번째 인풋이었다. 즉, 두번째 인풋이 21이어야한다는 결론을 얻을 수 있다. 따라서 phase4의 답은 6 21이 됨을 알 수 있다.

```
(gdb) r
Starting program: /home/std/greenidea/CSED211/Lab3/bomb314/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
3 a 549
Halfway there!
6 21
So you got that one.  Try this one.
```

Phase_5

```
(gdb) disas phase_5
Dump of assembler code for function phase_5:
   0x000055555555586d <+0>:     repz nop %edx
   0x0000555555555871 <+4>:     push   %rbx
   0x0000555555555872 <+5>:     mov    %rdi,%rbx
   0x0000555555555875 <+8>:     callq  0x555555555b0e <string_length>
   0x000055555555587a <+13>:    cmp    $0x6,%eax
   0x000055555555587d <+16>:    jne    0x5555555558ab <phase_5+62>
   0x000055555555587f <+18>:    mov    %rbx,%rax
   0x0000555555555882 <+21>:    lea    0x6(%rbx),%rdi
   0x0000555555555886 <+25>:    mov    $0x0,%ecx
   0x000055555555588b <+30>:    lea    0x190e(%rip),%rsi        # 0x5555555571a0 <array.3472>
   0x0000555555555892 <+37>:    movzbl (%rax),%edx
   0x0000555555555895 <+40>:    and    $0xf,%edx
   0x0000555555555898 <+43>:    add    (%rsi,%rdx,4),%ecx
   0x000055555555589b <+46>:    add    $0x1,%rax
   0x000055555555589f <+50>:    cmp    %rdi,%rax
   0x00005555555558a2 <+53>:    jne    0x5555555555892 <phase_5+37>
   0x00005555555558a4 <+55>:    cmp    $0x40,%ecx
   0x00005555555558a7 <+58>:    jne    0x5555555558b2 <phase_5+69>
   0x00005555555558a9 <+60>:    pop    %rbx
   0x00005555555558aa <+61>:    retq
   0x00005555555558ab <+62>:    callq  0x555555555c43 <explode_bomb>
   0x00005555555558b0 <+67>:    jmp    0x55555555587f <phase_5+18>
   0x00005555555558b2 <+69>:    callq  0x555555555c43 <explode_bomb>
   0x00005555555558b7 <+74>:    jmp    0x5555555558a9 <phase_5+60>
```

처음 주목한 부분은 <+8>, <+13>이었다. string_length라는 함수의 리턴벨류가 6이 아니면 <+16>에서 점프가 일어나서 터지는 것을 볼 수 있었다. 나는 이것을 통해 입력으로 문자열이 들어오고 문자열의 길이가 6이 아니면 터진다고 추측하였다. 그래서 <+18>에 breakpoint를 만들어두고 길이가 6인 문자열과 길이가 6이 아닌 문자열을 입력해보았다.

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

Breakpoint 2, 0x000055555555587f in phase_5 ()
```

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

BOOM!!!
The bomb has blown up.
[Inferior 1 (process 13366) exited with code 010]
```

예상대로 문자열의 길이가 6이 되어야 한다는 조건을 얻을 수 있었다.

```
0x000055555555587f <+18>:    mov    %rbx,%rax
0x0000555555555882 <+21>:    lea    0x6(%rbx),%rdi
0x0000555555555886 <+25>:    mov    $0x0,%ecx
0x000055555555588b <+30>:    lea    0x190e(%rip),%rsi        # 0x5555555571a0 <array.3472>
0x0000555555555892 <+37>:    movzbl (%rax),%edx
0x0000555555555895 <+40>:    and    $0xf,%edx
0x0000555555555898 <+43>:    add    (%rsi,%rdx,4),%ecx
0x000055555555589b <+46>:    add    $0x1,%rax
```

문자열의 길이가 6이라고 가정하고 <+18>부터 살펴보자. 일단 rbx에 들어있는 것이 무엇인지 확인하기 위해 <+21>에 breakpoint를 만들었다. 그리고 qwerty라는 문자열을 입력으로 준 결과 rbx부터 문자열을 이루는 문자들이 1바이트 단위로 들어있음을 확인할 수 있었다. 그리고 rbx+문

자열길이인 rbx+6에는 '₩0'이 들어있음도 확인할 수 있었다.

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

Breakpoint 3, 0x0000555555555882 in phase_5 ()
(gdb) i r rbx
rbx            0x5555555597e0   93824992253920
(gdb) x/c $rbx
0x5555555597e0 <input_strings+320>:     113 'q'
(gdb) x/c $rbx+1
0x5555555597e1 <input_strings+321>:     119 'w'
(gdb) x/c $rbx+2
0x5555555597e2 <input_strings+322>:     114 'r'
(gdb) x/c $rbx+3
0x5555555597e3 <input_strings+323>:     101 'e'
(gdb) x/c $rbx+4
0x5555555597e4 <input_strings+324>:     116 't'
(gdb) x/c $rbx+5
0x5555555597e5 <input_strings+325>:     121 'y'
(gdb) x/c $rbx+6
0x5555555597e6 <input_strings+326>:     0 '\000'
```

그러면 결국 <+18>에서 rax = rbx가 의미하는 것은 rax에 'q'를 저장하는 것이다. 그리고 <+21>에서 rdi에 '₩0'이 저장된다. 그리고 rcx = 0이 수행되고 <+30>에 도착한다.

```
(gdb) ni
0x0000555555555892 in phase_5 ()
(gdb) i r rsi
rsi                0x5555555571a0   93824992244128
```

ni 커맨드를 이용하여 <+30>의 lea 커맨드가 수행된 후 rsi에 들어있는 값이 무엇인지 확인해보았다. array의 주소임을 확인할 수 있었다. 그래서 일단 array에 들어있는 값들이 뭔지 확인하고자 했다. 그런데 array의 사이즈를 몰라서 일단 적당히 32바이트정도를 확인해보았다.

```
(gdb) x/32d 0x5555555571a0
0x5555555571a0 <array.3472>:    2       0       0       0       10      0       0       0
0x5555555571a8 <array.3472+8>:  6       0       0       0       1       0       0       0
0x5555555571b0 <array.3472+16>: 12      0       0       0       16      0       0       0
0x5555555571b8 <array.3472+24>: 9       0       0       0       3       0       0       0
```

이런 숫자들이 들어있다는 것을 확인할 수 있었다. 일단 지금으로서는 array와 관련된 정보를 더 얻을 수 있는게 없기때문에 계속 진행하기로 했다.

```
0x0000555555555892 <+37>:    movzbl (%rax),%edx
0x0000555555555895 <+40>:    and    $0xf,%edx
0x0000555555555898 <+43>:    add    (%rsi,%rdx,4),%ecx
0x000055555555589b <+46>:    add    $0x1,%rax
```

<+37>, <+40>, <+43>, <+46>에 의해 rdx = rax & 0xf가 수행된다. 이것은 rax에서 하위 4개의 비트만을 그대로 두고 나머지 비트는 전부 0으로 만들어 rdx에 저장한다는 뜻이다. 따라서 rdx는 0이상 15이하의 수가 된다. 그리고 rcx = rcx + rsi + rdx*4 = rcx + rsi[rdx]와 rax += 1이 수행되어 rax는 다음 문자를 가리키게 된다.(지금 상황에서는 두번째 문자를 가리키게됨)

```
0x000055555555589f <+50>:    cmp    %rdi,%rax
0x00005555555558a2 <+53>:    jne    0x555555555892 <phase_5+37>
0x00005555555558a4 <+55>:    cmp    $0x40,%ecx
0x00005555555558a7 <+58>:    jne    0x5555555558b2 <phase_5+69>
0x00005555555558a9 <+60>:    pop    %rbx
0x00005555555558aa <+61>:    retq
0x00005555555558ab <+62>:    callq  0x555555555c43 <explode_bomb>
0x00005555555558b0 <+67>:    jmp    0x55555555587f <phase_5+18>
0x00005555555558b2 <+69>:    callq  0x555555555c43 <explode_bomb>
0x00005555555558b7 <+74>:    jmp    0x5555555558a9 <phase_5+60>
```

그다음 <+50>, <+53>에서 rax와 rdi가 다르면 37로 점프하는 것을 확인할 수 있다. 결국 rax가 가리키는 것이 '₩0'이 아니라면 <+37>로 돌아가 위 과정이 다시 반복되는 것이다. 그리고 만약 같다면 <+55>로 진행되므로, rcx가 64가 아니라면 <+69>로 이동하여 터지게 된다. 그리고 rcx가 64라면 리턴하고 페이즈5는 종료됨을 알 수 있다. 즉, 문자열을 이루는 6개의 문자에 대해 순서대로 보면서 모든 문자를 다봤을때 rcx에 저장된 값이 64가 되어야 터지지 않는다. 초기에 rcx에는 0이 들어있기 때문에 결국 rsi가 가리키는 array에 들어있는 수들 중에서 중복을 포함하여 6개를 고르고 그 6개의 합이 64가 되는 케이스를 찾으면 된다. 그리고 0 <= rdx <= 15이고 rcx += rsi[rdx]가 수행된다는 점에서 배열의 인덱스가 0~15라고 추측할 수 있다. 따라서 배열의 시작 주소로부터 64바이트를 확인함으로써 array table을 볼 수 있다. array table은 다음과 같다.

```
(gdb) x/64d 0x5555555571a0
0x5555555571a0 <array.3472>:    2       0       0       0       10      0       0       0
0x5555555571a8 <array.3472+8>:  6       0       0       0       1       0       0       0
0x5555555571b0 <array.3472+16>: 12      0       0       0       16      0       0       0
0x5555555571b8 <array.3472+24>: 9       0       0       0       3       0       0       0
0x5555555571c0 <array.3472+32>: 4       0       0       0       7       0       0       0
0x5555555571c8 <array.3472+40>: 14      0       0       0       5       0       0       0
0x5555555571d0 <array.3472+48>: 11      0       0       0       8       0       0       0
0x5555555571d8 <array.3472+56>: 15      0       0       0       13      0       0       0
```

여기서 64를 만들기 위해서는 array[14]=15를 4번, array[0]=2을 2번 선택하면 된다. 즉, rdx의 값이 0 또는 14가 되어야한다. rdx = rax & 0xf로 정의되므로 rdx = 0을 위해서는 rax의 아스키코드가 임의의 한자리 자연수 n에 대하여 0xn0이면 된다. 그리고 rdx = 14를 위해서는 rax의 아스키코드가 임의의 한자리 자연수 m에 대하여 0xme이면 된다. 그래서 이 조건을 만족하는 문자를 아스키코드표에서 자유롭게 고르면 되는데, 나는 0x30, 0x6e를 선택했다.

0x30은 문자 '0', 0x6e는 문자 'n'에 대응되고, 지금까지 찾은 조건들을 보면 문자들의 등장 순서에는 제약이 없기 때문에 '0'을 2번 'n'을 4번 사용하는 문자열이면 전부 패스가능할 것이다. 그래서 00nnnn을 입력하고 phase5를 패스했다.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/std/greenidea/CSED211/Lab3/bomb314/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
3 a 549
Halfway there!
6 21
So you got that one.  Try this one.
00nnnn
Good work!  On to the next...
```

Phase_6

```
(gdb) disas phase_6
Dump of assembler code for function phase_6:
   0x00005555555558b9 <+0>:     repz nop %edx
   0x00005555555558bd <+4>:     push   %r14
   0x00005555555558bf <+6>:     push   %r13
   0x00005555555558c1 <+8>:     push   %r12
   0x00005555555558c3 <+10>:    push   %rbp
   0x00005555555558c4 <+11>:    push   %rbx
   0x00005555555558c5 <+12>:    sub    $0x60,%rsp
   0x00005555555558c9 <+16>:    mov    %fs:0x28,%rax
   0x00005555555558d2 <+25>:    mov    %rax,0x58(%rsp)
   0x00005555555558d7 <+30>:    xor    %eax,%eax
   0x00005555555558d9 <+32>:    mov    %rsp,%r13
   0x00005555555558dc <+35>:    mov    %r13,%rsi
   0x00005555555558df <+38>:    callq  0x555555555c6f <read_six_numbers>
```

먼저 disassemble을 했을때 read_six_numbers를 호출하는 부분에서 이번 페이즈의 인풋은 6개의 정수라는 것을 체크했다.

```
0x00005555555558df <+38>:    callq  0x555555555c6f <read_six_numbers>
0x00005555555558e4 <+43>:    mov    $0x1,%r14d
0x00005555555558ea <+49>:    mov    %rsp,%r12
0x00005555555558ed <+52>:    jmp    0x555555555917 <phase_6+94>
0x00005555555558ef <+54>:    callq  0x555555555c43 <explode_bomb>
0x00005555555558f4 <+59>:    jmp    0x555555555926 <phase_6+109>
0x00005555555558f6 <+61>:    add    $0x1,%rbx
0x00005555555558fa <+65>:    cmp    $0x5,%ebx
0x00005555555558fd <+68>:    jg     0x55555555590f <phase_6+86>
0x00005555555558ff <+70>:    mov    (%r12,%rbx,4),%eax
0x0000555555555903 <+74>:    cmp    %eax,0x0(%rbp)
0x0000555555555906 <+77>:    jne    0x5555555558f6 <phase_6+61>
0x0000555555555908 <+79>:    callq  0x555555555c43 <explode_bomb>
0x000055555555590d <+84>:    jmp    0x5555555558f6 <phase_6+61>
0x000055555555590f <+86>:    add    $0x1,%r14
0x0000555555555913 <+90>:    add    $0x4,%r13
0x0000555555555917 <+94>:    mov    %r13,%rbp
0x000055555555591a <+97>:    mov    0x0(%r13),%eax
0x000055555555591e <+101>:   sub    $0x1,%eax
0x0000555555555921 <+104>:   cmp    $0x5,%eax
0x0000555555555924 <+107>:   ja     0x5555555558ef <phase_6+54>
```

그리고 <+94>부터 보면 rbp = r13, rax = r13, rax -= 1이 수행되고 rax가 5보다 크면 <+54>로 점프하여 터지는 것을 확인할 수 있다. 또, ja는 unsigned로 해석하여 수를 비교하므로 따라서 rax가 1이상 6이하의 정수가 되어야함을 알 수 있다. 그럼 r13이 무엇인지를 알아야 한다. 그래서 일단 <+94>에 breakpoint를 걸어두고 1 2 3 4 5 6과 2 3 4 5 6 7을 입력해보았다.

```
r13            0x7fffffffe400    140737488348160
```

```
(gdb) x/d 0x7fffffffe400
0x7fffffffe400: 1
```

```
r13            0x7fffffffe400    140737488348160
```

```
(gdb) x/d 0x7fffffffe400
0x7fffffffe400: 2
```

그리고 그 결과 r13에는 첫번째 인풋이 저장되는 메모리의 주소가 저장됨을 확인할 수 있었다.

그다음은 r14d가 5보다 크면 <+120>으로 점프가 이루어짐을 확인할 수 있다. 그렇지 않다면 rbx = r14를 수행하고 <+70>으로 점프가 이루어진다.

```
0x0000555555555926 <+109>:   cmp    $0x5,%r14d
0x000055555555592a <+113>:   jg     0x555555555931 <phase_6+120>
0x000055555555592c <+115>:   mov    %r14,%rbx
0x000055555555592f <+118>:   jmp    0x5555555558ff <phase_6+70>
0x0000555555555931 <+120>:   mov    $0x0,%esi
0x0000555555555936 <+125>:   mov    (%rsp,%rsi,4),%ecx
0x0000555555555939 <+128>:   mov    $0x1,%eax
0x000055555555593e <+133>:   lea    0x38cb(%rip),%rdx        # 0x555555559210 <node1>
0x0000555555555945 <+140>:   cmp    $0x1,%ecx
0x0000555555555948 <+143>:   jle    0x555555555955 <phase_6+156>
0x000055555555594a <+145>:   mov    0x8(%rdx),%rdx
0x000055555555594e <+149>:   add    $0x1,%eax
0x0000555555555951 <+152>:   cmp    %ecx,%eax
0x0000555555555953 <+154>:   jne    0x55555555594a <phase_6+145>
```

그렇다면 r14에는 무엇이 저장되어 있는가? 2 3 4 5 6 7이라는 인풋을 주고, <+106>에 breakpoint를 걸어 이때 r14의 정보를 확인해보면 1이 나오는 것을 알 수 있다.

```
(gdb) i r r14
r14            0x1         1
```

일단 인풋의 값과는 관련이 없어보이긴 하는데 더 얻을 수 있는 정보가 없으니 계속 진행해보자. r14가 1이므로 <+120>으로의 점프는 이루어지지 않을 것이다. <+115>에서 rbx = r14가 되므로 rbx =1이 되고, <+70>으로 점프가 일어난다.

<+70>에서는 rax = r12+4*rbx가 수행되고 rbp와 rax가 다르면 <+61>로 점프가 일어나고 같으면 터지는 것을 알 수 있다. 그리고 r12에는 r13처럼 첫번째 인풋이 저장된 메모리의 주소가 들어있다.

```
(gdb) i r r12
r12            0x7fffffffe400   140737488348160
```

그리고 r12+4*rbx에는 두번째 인풋이 저장되어있음도 확인할 수 있다.

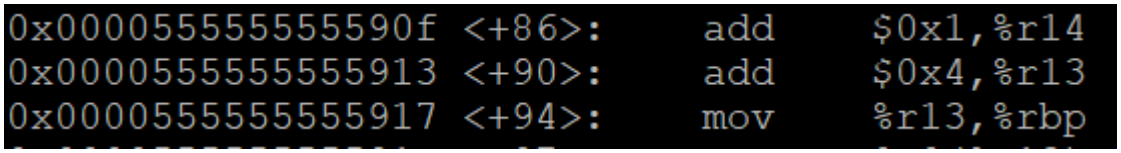
```
(gdb) x/d $r12+$rbx+$rbx+$rbx+$rbx
0x7fffffffe404: 3
```

그리고 아까전에 rbp = r13을 했었기 때문에 rbp에는 첫번째 인풋의 주소가 들어있음을 알 수 있다. 따라서 (rbp)는 첫번째 인풋, rax는 두번째 인풋이므로 같지 않고, <+61>로 점프가 일어나게 된다.

```
0x00005555555558f6 <+61>:    add    $0x1,%rbx
0x00005555555558fa <+65>:    cmp    $0x5,%ebx
0x00005555555558fd <+68>:    jg     0x55555555590f <phase_6+86>
```

여기서는 rbx++을 수행한 후 rbx가 5보다 크면 <+86>으로 점프하고 그렇지 않으면 <+70>으로 점프하여 위에서 설명한 내용을 반복한다. 여기 진입하는 시점에 rbx = 1인 상태이므로 5번 반복

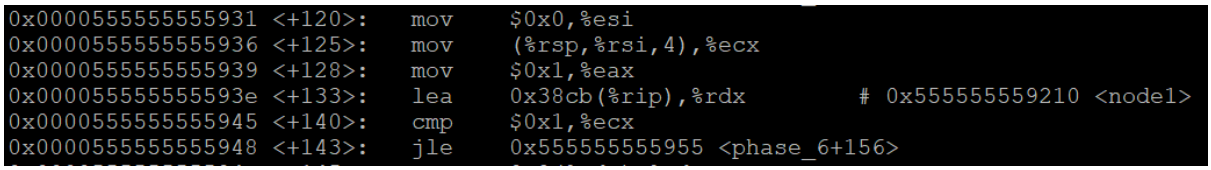
함을 알 수 있다. 즉, 2번째 인풋부터 6번째 인풋까지가 첫번째 인풋과 같지 않은지 확인하면서 같으면 터지게 만든다는 것을 알 수 있다. 이 조건을 만족한다고 가정하자. 그러면 <+68>에서 <+86>으로 점프가 일어날 것이다.



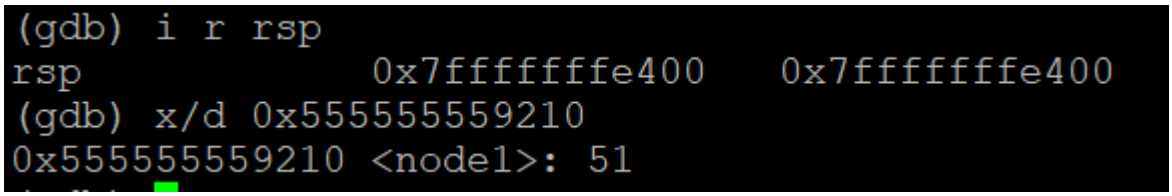

그리고 r14++을 수행해주고 r13에 4를 더해준다. 이것은 r13이 두번째 인풋을 가리키도록 바꾸겠다는 것이다. 그리고 위에서 설명한 <+94>부터의 과정이 반복된다. 이로써 알 수 있는 것은 두번째 인풋을 기준으로 3번째 인풋부터 6번째 인풋까지가 2번째 인풋과 같은지를 또 체크하고 세번째 인풋을 기준으로 또 체크하고를 계속한다는 것이다.

결국 6개의 인풋에 중복이 있으면 안되고, 6개의 인풋은 모두 1이상 6이하의 정수여야한다는 조건을 얻을 수 있다.

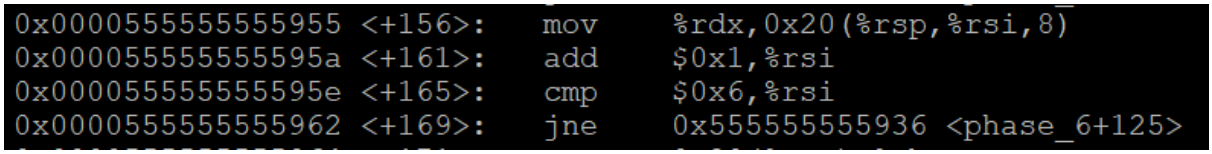
이 중복검사 과정이 끝나고 <+120>으로 점프가 이루어지면 어떻게 되는지 살펴보자.




rsi = 0이 실행되고 rcx = rsp + rsi*4가 수행된다. 그리고 rax = 1이 수행되고 rdx에 node1이 저장된다. 그리고 rcx가 1보다 작거나 같으면 <+156>으로 점프가 발생한다.




rsp와 node1의 정보를 확인해본 결과 rsp에는 첫번째 인풋의 주소가 node1에는 51라는 숫자가 들어있었다. 따라서 rcx = rsp[rsi]를 의미함을 알 수 있었다. 그리고 현재 인풋으로 1 2 3 4 5 6을 넣은 상태이기 때문에 rcx = 1이 되어 <+156>으로 점프가 발생한다.




<+156>부터는 rsp+rsi*8+0x20에 rdx가 저장되고, rsi++이 수행된 뒤 rsi가 6이 아니면 <+125>로 점프하게 된다. <+125>로 점프가 이루어지면 rcx가 두번째 인풋을 가리키게 되고, 따라서 <+143>에서 점프가 일어나지 않고 <+145>로 진행된다.

```
0x000055555555594a <+145>:    mov    0x8(%rdx),%rdx
0x000055555555594e <+149>:    add    $0x1,%eax
0x0000555555555951 <+152>:    cmp    %ecx,%eax
0x0000555555555953 <+154>:    jne    0x55555555594a <phase_6+145>
```

<+145>에서 rdx = rdx+8, rax++이 수행되고 rax와 rcx가 다르면 <+145>로 점프가 일어나 이 과정을 반복한다. 즉, rax와 rcx가 같아질때까지 이 과정을 반복하는 것이다. 다시 말해, rax를 rcx가 가리키는 인풋과 같게 만들어준다. 그리고 현재 rdx에는 node1이 저장되어 있는데 rdx+8이 나와서 node1부터 연속적으로 저장되어 있는건지 궁금해서 x/24w를 통해 확인해보았다.

```
(gdb) x/24w 0x555555559210
0x555555559210 <node1>: 51      1       1431671328      21845
0x555555559220 <node2>: 473     2       1431671344      21845
0x555555559230 <node3>: 343     3       1431671360      21845
0x555555559240 <node4>: 694     4       1431671376      21845
0x555555559250 <node5>: 625     5       1431671056      21845
0x555555559260 <host_table>:    1431663473      21845   1431663499      21845
```

그 결과 총 5개의 노드가 존재하고 host_table이라는 것이 존재함을 확인할 수 있었고, rdx+8로 인해 rdx는 현재 2번 노드가 되었음도 알 수 있었다. 그리고 <+156>부터 위에서 설명했던 과정이 다시 수행됨을 알 수 있다. 결국 6개의 각 인풋에 대해 node input의 값을 대응시켜주는 것이다. 예를 들어, 인풋이 2 3 1 4 5 6이라면 rsp+0x20부터 8바이트 단위로 node2, node3, … , node5, host_table이 저장된다.

```
0x0000555555555964 <+171>:    mov    0x20(%rsp),%rbx
0x0000555555555969 <+176>:    mov    0x28(%rsp),%rax
0x000055555555596e <+181>:    mov    %rax,0x8(%rbx)
0x0000555555555972 <+185>:    mov    0x30(%rsp),%rdx
0x0000555555555977 <+190>:    mov    %rdx,0x8(%rax)
0x000055555555597b <+194>:    mov    0x38(%rsp),%rax
0x0000555555555980 <+199>:    mov    %rax,0x8(%rdx)
0x0000555555555984 <+203>:    mov    0x40(%rsp),%rdx
0x0000555555555989 <+208>:    mov    %rdx,0x8(%rax)
0x000055555555598d <+212>:    mov    0x48(%rsp),%rax
0x0000555555555992 <+217>:    mov    %rax,0x8(%rdx)
0x0000555555555996 <+221>:    movq   $0x0,0x8(%rax)
0x000055555555599e <+229>:    mov    $0x5,%ebp
0x00005555555559a3 <+234>:    jmp    0x5555555559ae <phase_6+245>
```

이 과정이 끝나면 <+171>에 진입하게 된다. <+171>에서 rbx = rsp+0x20이 수행되는데 이것은 첫번째 인풋의 노드에 해당된다. <+176>부터 <+217>까지의 과정은 다음과 같다. rbx가 가리키는 주소가 16진수로 20이라고 할때 28, 30 38, 40, 48에 저장된 값을 rax, rdx 레지스터를 이용하여 그대로 자기 자리에 옮겨주는 것이다. 즉 의미가 없는 명령들이다. <+221>에서는 50에 해당하는 위치에 0을 저장하고, <+229>에서는 rbp에 5를 저장한다. 그리고 <+245>로 점프한다.

```
0x00005555555559a5 <+236>:   mov    0x8(%rbx),%rbx
0x00005555555559a9 <+240>:   sub    $0x1,%ebp
0x00005555555559ac <+243>:   je     0x5555555559bf <phase_6+262>
0x00005555555559ae <+245>:   mov    0x8(%rbx),%rax
0x00005555555559b2 <+249>:   mov    (%rax),%eax
0x00005555555559b4 <+251>:   cmp    %eax,(%rbx)
0x00005555555559b6 <+253>:   jle    0x5555555559a5 <phase_6+236>
0x00005555555559b8 <+255>:   callq  0x555555555c43 <explode_bomb>
0x00005555555559bd <+260>:   jmp    0x5555555559a5 <phase_6+236>
0x00005555555559bf <+262>:   mov    0x58(%rsp),%rax
```

<+245>, <+249>에서는 rax에 두번째 인풋에 대응되는 노드가 저장된다. 그리고 <+251>, <+253>에서 rbx와 rax를 비교하여 rbx가 가리키는 노드의 값이 rbx 다음 노드의 값인 rax의 값 이하이면 <+236>으로 점프하고 그렇지 않으면 터진다. 그리고 <+236>부터는 rbx를 다음 인풋이 가리키는 노드로 바꾸고 rbp -= 1을 수행한다. 그래서 rbp가 0이되면 <+262>로 점프한다. 즉, n번째 인풋에 대응되는 노드가 n+1번째 인풋에 대응되는 노드보다 작은지 1<=n<=5인 n에 대하여 확인한다는 것이다. 즉, 패스를 위해서는 인풋에 대응하는 노드가 오름차순이 되게하면 된다. 그런데 현재 6번째 인풋에 대응하는 노드6이 무엇인지 모르는 상황이다. 그래서 <+251>에 breakpoint를 만들고, 6 5 4 3 2 1이라는 인풋을 입력하여 아래와 같이 node6가 무엇인지 알아냈다.

```
(gdb) i r rbx
rbx            0x555555559110   93824992252176
(gdb) x/w $rbx
0x555555559110 <node6>: 616
```

정리를 해보자면,

node1: 51  node2: 473  node3: 343  node4: 694  node5: 625  node6: 616

이므로 오름차순이 되게하는 input은 1 3 2 6 5 4이다. 이것으로 모든 페이즈를 패스하였다.

```
(gdb) r
Starting program: /home/std/greenidea/CSED211/Lab3/bomb314/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
3 a 549
Halfway there!
6 21
So you got that one.  Try this one.
00nnnn
Good work!  On to the next...
1 3 2 6 5 4
Congratulations! You've defused the bomb!
[Inferior 1 (process 12522) exited normally]
```

```
[greenidea@programming2 bomb314]$ ./bomb solution.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
Congratulations! You've defused the bomb!
[greenidea@programming2 bomb314]$
```