Bomb Lab Report (Bomb44)

2016314364 박수헌

- Detail description about code & screenshot

Phase_1

Phase_1에 break point를 지정하고 phase_1 함수를 disas 해보았다.

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x00000000000013d4 <+0>:     sub    $0x8,%rsp
   0x00000000000013d8 <+4>:     lea    0x1931(%rip),%rsi        # 0x2d10
   0x00000000000013df <+11>:    callq  0x19de <strings_not_equal>
   0x00000000000013e4 <+16>:    test   %eax,%eax
   0x00000000000013e6 <+18>:    jne    0x13ed <phase_1+25>
   0x00000000000013e8 <+20>:    add    $0x8,%rsp
   0x00000000000013ec <+24>:    retq
   0x00000000000013ed <+25>:    callq  0x1c89 <explode_bomb>
   0x00000000000013f2 <+30>:    jmp    0x13e8 <phase_1+20>
End of assembler dump.
(gdb)
```

strings_not_equal이라는 함수를 통해, 나온 리턴값은 %eax에 저장되므로, +16에서 %eax가 0이면, +25로 jump해 explode하는 것을 볼 수 있다. 따라서 strings_not_equal 함수의 리턴값이 1이어야한다는 것을 알 수 있다.

```
(gdb) disas strings_not_equal
Dump of assembler code for function strings_not_equal:
   0x00000000000019de <+0>:     push   %r12
   0x00000000000019e0 <+2>:     push   %rbp
   0x00000000000019e1 <+3>:     push   %rbx
   0x00000000000019e2 <+4>:     mov    %rdi,%rbx
   0x00000000000019e5 <+7>:     mov    %rsi,%rbp
   0x00000000000019e8 <+10>:    callq  0x19c1 <string_length>
   0x00000000000019ed <+15>:    mov    %eax,%r12d
   0x00000000000019f0 <+18>:    mov    %rbp,%rdi
   0x00000000000019f3 <+21>:    callq  0x19c1 <string_length>
   0x00000000000019f8 <+26>:    mov    $0x1,%edx
   0x00000000000019fd <+31>:    cmp    %eax,%r12d
   0x0000000000001a00 <+34>:    je     0x1a09 <strings_not_equal+43>
   0x0000000000001a02 <+36>:    mov    %edx,%eax
   0x0000000000001a04 <+38>:    pop    %rbx
   0x0000000000001a05 <+39>:    pop    %rbp
   0x0000000000001a06 <+40>:    pop    %r12
   0x0000000000001a08 <+42>:    retq
   0x0000000000001a09 <+43>:    movzbl (%rbx),%eax
   0x0000000000001a0c <+46>:    test   %al,%al
   0x0000000000001a0e <+48>:    je     0x1a37 <strings_not_equal+89>
   0x0000000000001a10 <+50>:    cmp    0x0(%rbp),%al
   0x0000000000001a13 <+53>:    jne    0x1a3e <strings_not_equal+96>
   0x0000000000001a15 <+55>:    add    $0x1,%rbx
   0x0000000000001a19 <+59>:    add    $0x1,%rbp
   0x0000000000001a1d <+63>:    movzbl (%rbx),%eax
   0x0000000000001a20 <+66>:    test   %al,%al
   0x0000000000001a22 <+68>:    je     0x1a30 <strings_not_equal+82>
   0x0000000000001a24 <+70>:    cmp    %al,0x0(%rbp)
   0x0000000000001a27 <+73>:    je     0x1a15 <strings_not_equal+55>
   0x0000000000001a29 <+75>:    mov    $0x1,%edx
   0x0000000000001a2e <+80>:    jmp    0x1a02 <strings_not_equal+36>
   0x0000000000001a30 <+82>:    mov    $0x0,%edx
   0x0000000000001a35 <+87>:    jmp    0x1a02 <strings_not_equal+36>
   0x0000000000001a37 <+89>:    mov    $0x0,%edx
   0x0000000000001a3c <+94>:    jmp    0x1a02 <strings_not_equal+36>
   0x0000000000001a3e <+96>:    mov    $0x1,%edx
   0x0000000000001a43 <+101>:   jmp    0x1a02 <strings_not_equal+36>
End of assembler dump.
(gdb)
```

이 때, %rdi에는 입력한 문자열, %rsi에는 phase_1의 +4줄의 $0x2cd10 주소의 값이 들어있다.

```
(gdb) x/s 0x2d10
0x2d10: "The grass is always greener on the other side of the fence."
(gdb)
```

즉. 입력한 %rdi 값과, %rsi 값이 같으면 strings_not_equal 함수는 1을 리턴하는 것을 알 수 있었다.

그래서 phase_1 답은 "The grass is always greener on the other side of the fence."라는 것을 알아냈다.

Phase_2

phase_2에 break point를 건 후, hi라는 임의의 답을 입력하고 disas했다.

```
(gdb) disas
Dump of assembler code for function phase_2:
=> 0x00005555555553f4 <+0>:     push   %rbp
   0x00005555555553f5 <+1>:     push   %rbx
   0x00005555555553f6 <+2>:     sub    $0x28,%rsp
   0x00005555555553fa <+6>:     mov    %fs:0x28,%rax
   0x0000555555555403 <+15>:    mov    %rax,0x18(%rsp)
   0x0000555555555408 <+20>:    xor    %eax,%eax
   0x000055555555540a <+22>:    mov    %rsp,%rsi
   0x000055555555540d <+25>:    callq  0x555555555cc5 <read_six_numbers>
   0x0000555555555412 <+30>:    cmpl   $0x1,(%rsp)
   0x0000555555555416 <+34>:    jne    0x555555555422 <phase_2+46>
   0x0000555555555418 <+36>:    mov    $0x1,%ebx
   0x000055555555541d <+41>:    mov    %rsp,%rbp
   0x0000555555555420 <+44>:    jmp    0x555555555433 <phase_2+63>
   0x0000555555555422 <+46>:    callq  0x555555555c89 <explode_bomb>
   0x0000555555555427 <+51>:    jmp    0x555555555418 <phase_2+36>
   0x0000555555555429 <+53>:    add    $0x1,%rbx
   0x000055555555542d <+57>:    cmp    $0x6,%rbx
   0x0000555555555431 <+61>:    je     0x555555555449 <phase_2+85>
   0x0000555555555433 <+63>:    mov    %ebx,%eax
   0x0000555555555435 <+65>:    add    -0x4(%rbp,%rbx,4),%eax
   0x0000555555555439 <+69>:    imul   %ebx,%eax
   0x000055555555543c <+72>:    cmp    %eax,0x0(%rbp,%rbx,4)
---Type <return> to continue, or q <return> to quit---
   0x0000555555555440 <+76>:    je     0x555555555429 <phase_2+53>
   0x0000555555555442 <+78>:    callq  0x555555555c89 <explode_bomb>
   0x0000555555555447 <+83>:    jmp    0x555555555429 <phase_2+53>
   0x0000555555555449 <+85>:    mov    0x18(%rsp),%rax
   0x000055555555544e <+90>:    xor    %fs:0x28,%rax
   0x0000555555555457 <+99>:    jne    0x555555555460 <phase_2+108>
   0x0000555555555459 <+101>:   add    $0x28,%rsp
   0x000055555555545d <+105>:   pop    %rbx
   0x000055555555545e <+106>:   pop    %rbp
   0x000055555555545f <+107>:   retq
   0x0000555555555460 <+108>:   callq  0x555555555010 <__stack_chk_fail@plt>
End of assembler dump.
```

phase_2 함수는 read_six_numbers 함수를 통해 입력 받은 값을 비교한다는 것을 알아낸 후, read_six_numbers에 두번째 break point를 설정해 disas했다.

```
Breakpoint 2, 0x0000555555555cc5 in read_six_numbers ()
(gdb) disas
Dump of assembler code for function read_six_numbers:
=> 0x0000555555555cc5 <+0>:     sub    $0x8,%rsp
   0x0000555555555cc9 <+4>:     mov    %rsi,%rdx
   0x0000555555555ccc <+7>:     lea    0x4(%rsi),%rcx
   0x0000555555555cd0 <+11>:    lea    0x14(%rsi),%rax
   0x0000555555555cd4 <+15>:    push   %rax
   0x0000555555555cd5 <+16>:    lea    0x10(%rsi),%rax
   0x0000555555555cd9 <+20>:    push   %rax
   0x0000555555555cda <+21>:    lea    0xc(%rsi),%r9
   0x0000555555555cde <+25>:    lea    0x8(%rsi),%r8
   0x0000555555555ce2 <+29>:    lea    0x1315(%rip),%rsi        # 0x555555556ffe
   0x0000555555555ce9 <+36>:    mov    $0x0,%eax
   0x0000555555555cee <+41>:    callq  0x5555555550b0 <__isoc99_sscanf@plt>
   0x0000555555555cf3 <+46>:    add    $0x10,%rsp
   0x0000555555555cf7 <+50>:    cmp    $0x5,%eax
   0x0000555555555cfa <+53>:    jle    0x555555555d01 <read_six_numbers+60>
   0x0000555555555cfc <+55>:    add    $0x8,%rsp
   0x0000555555555d00 <+59>:    retq
   0x0000555555555d01 <+60>:    callq  0x555555555c89 <explode_bomb>
End of assembler dump.
(gdb)
```

+36줄을 보면 리턴될 %eax 레지스터에 0을 넣는 것을 볼 수 있다. 따라서 +25까지 입력받은 값을 처리한다고 판단해 0x555555556ffe의 값을 확인해 보았다.

```
(gdb) x/s 0x555555556ffe
0x555555556ffe: "%d %d %d %d %d %d"
(gdb)
```

따라서 입력값이 6개의 정수라는 것을 확인할 수 있었다.

따라서 임의의 답인 1 2 3 4 5 6을 입력한 후 다시 disas 해보았다.

```
0x0000555555555412 <+30>:    cmpl   $0x1,(%rsp)
0x0000555555555416 <+34>:    jne    0x555555555422 <phase_2+46>
0x0000555555555418 <+36>:    mov    $0x1,%ebx
0x000055555555541d <+41>:    mov    %rsp,%rbp
0x0000555555555420 <+44>:    jmp    0x555555555433 <phase_2+63>
0x0000555555555422 <+46>:    callq  0x555555555c89 <explode_bomb>
0x0000555555555427 <+51>:    jmp    0x555555555418 <phase_2+36>
0x0000555555555429 <+53>:    add    $0x1,%rbx
0x000055555555542d <+57>:    cmp    $0x6,%rbx
0x0000555555555431 <+61>:    je     0x555555555449 <phase_2+85>
0x0000555555555433 <+63>:    mov    %ebx,%eax
0x0000555555555435 <+65>:    add    -0x4(%rbp,%rbx,4),%eax
0x0000555555555439 <+69>:    imul   %ebx,%eax
0x000055555555543c <+72>:    cmp    %eax,0x0(%rbp,%rbx,4)
0x0000555555555440 <+76>:    je     0x555555555429 <phase_2+53>
0x0000555555555442 <+78>:    callq  0x555555555c89 <explode_bomb>
0x0000555555555447 <+83>:    jmp    0x555555555429 <phase_2+53>
0x0000555555555449 <+85>:    mov    0x18(%rsp),%rax
0x000055555555544e <+90>:    xor    %fs:0x28,%rax
0x0000555555555457 <+99>:    jne    0x555555555460 <phase_2+108>
0x0000555555555459 <+101>:   add    $0x28,%rsp
0x000055555555545d <+105>:   pop    %rbx
0x000055555555545e <+106>:   pop    %rbp
0x000055555555545f <+107>:   retq
0x0000555555555460 <+108>:   callq  0x555555555010 <__stack_chk_fail@plt>
d of assembler dump.
```

phase_2로 돌아와, read_six_numbers로 입력값을 입력받은 후의 코드를 보았다.

(%rsp)의 값과 1을 비교해 같지 않으면 +46번 줄로 이동해 explode하는 것을 볼 수 있다.

```
(gdb) x/d $rsp
0x7fffffffde50: 1
(gdb)
```

%rsp 레지스터에는 입력한 첫번째 숫자인 1이 들어있다. 즉 첫번째 숫자는 1이어야 하는 것을 알아냈다.

+36줄부터 보면, %ebx 레지스터에 1을 넣고, %rsp(첫번째 input의 주소값)를 %rbp로 옮긴다. 그 후 +63줄로 jump한다.

%ebx(1)를 %eax로 옮긴다. 그 후 %eax(1)의 값에 [%rbp(1)+4*%rbx(1)-0x4]=[%rbp]의 값을 더한다. 즉 2가 된다.

이 때 레지스터 값을들 보면, rax 레지스터에 2가 들어가있는 것을 볼 수 있다.

```
(gdb) i r
rax            0x2      2
rbx            0x1      1
rcx            0x0      0
rdx            0x7fffffffde64   140737488346724
rsi            0x0      0
rdi            0x7fffffffd7c0   140737488345024
rbp            0x7fffffffde50   0x7fffffffde50
rsp            0x7fffffffde50   0x7fffffffde50
r8             0x0      0
r9             0x0      0
r10            0x7ffff7b82cc0   140737349430464
r11            0x55555555700f   93824992243727
r12            0x555555555170   93824992235888
r13            0x7fffffffdf70   140737488346992
```

그 후 +69번줄부터 보면,

```
0x0000555555555439 <+69>:    imul   %ebx,%eax
0x000055555555543c <+72>:    cmp    %eax,0x0(%rbp,%rbx,4)
0x0000555555555440 <+76>:    je     0x555555555429 <phase_2+53>
0x0000555555555442 <+78>:    callq  0x555555555c89 <explode_bomb>
0x0000555555555447 <+83>:    jmp    0x555555555429 <phase_2+53>
0x0000555555555449 <+85>:    mov    0x18(%rsp),%rax
0x000055555555544e <+90>:    xor    %fs:0x28,%rax
0x0000555555555457 <+99>:    jne    0x555555555460 <phase_2+108>
0x0000555555555459 <+101>:   add    $0x28,%rsp
0x000055555555545d <+105>:   pop    %rbx
0x000055555555545e <+106>:   pop    %rbp
0x000055555555545f <+107>:   retq
0x0000555555555460 <+108>:   callq  0x555555555010 <__stack_chk_fail@plt>
d of assembler dump.
```

%eax(2)의 값에 %ebx(1)의 값을 곱한다. 그리고 [%rbp+4*%rbx(1)]의 값과 %eax의 값을 비교해 다르면 +78줄로 이동해 explode되므로, 같아야 함을 알 수 있다.

```
(gdb) x/d $rbp+4*$rbx
0x7fffffffde54: 2
(gdb)
```

여기서 [%rbp +4]는 그 다음 input숫자를 뜻하는 것이므로 두번째 input이 2이어야하는 것을 알 수 있다. 그리고 조건에 따라 +53줄로 이동한다.

```
0x0000555555555429 <+53>:    add    $0x1,%rbx
0x000055555555542d <+57>:    cmp    $0x6,%rbx
0x0000555555555431 <+61>:    je     0x555555555449 <phase_2+85>
0x0000555555555433 <+63>:    mov    %ebx,%eax
0x0000555555555435 <+65>:    add    -0x4(%rbp,%rbx,4),%eax
0x0000555555555439 <+69>:    imul   %ebx,%eax
0x000055555555543c <+72>:    cmp    %eax,0x0(%rbp,%rbx,4)
0x0000555555555440 <+76>:    je     0x555555555429 <phase_2+53>
0x0000555555555442 <+78>:    callq  0x555555555c89 <explode_bomb>
0x0000555555555447 <+83>:    jmp    0x555555555429 <phase_2+53>
0x0000555555555449 <+85>:    mov    0x18(%rsp),%rax
0x000055555555544e <+90>:    xor    %fs:0x28,%rax
0x0000555555555457 <+99>:    jne    0x555555555460 <phase_2+108>
0x0000555555555459 <+101>:   add    $0x28,%rsp
0x000055555555545d <+105>:   pop    %rbx
0x000055555555545e <+106>:   pop    %rbp
0x000055555555545f <+107>:   retq
0x0000555555555460 <+108>:   callq  0x555555555010 <__stack_chk_fail@plt>
d of assembler dump.
```

$rbx(1)에 1을 더하고 그 값을 6과 비교해 같으면 +85줄로 이동, 다르면 +63줄로 진행하는 것을 알 수 있다.

이 경우에는 다르므로 +63줄로 진행한다.

%ebx(2)의 값을 %eax(2)로 이동한 후, %eax의 값에 [%rbp+4*ebx(2) -0x4] = [%rbp+4](2번째 input)의 값을 더한다.

```
(gdb) x/d $rbp+4
0x7fffffffde54: 2
```

따라서 %eax의 값은 4가 된다.

그 후, %eax(4)의 값에 %ebx(2)의 값을 곱하면, %eax의 값은 8이 된다.

+72줄을 보면, Mem[%rbp+4*%rbx]의 값을 %eax(8)과 비교해 같으면 다시 +53줄로 돌아가고 다르면 진행해 explode되는 것을 볼 수 있다.

```
(gdb) x/d $rbp+4*$rbx
0x7fffffffde58: 3
```

이 주소에는 3번째 input이 들어있다. 따라서 3번째 input의 값은 8이어야하는 것을 알 수 있다.

+76의 조건에 의해 이렇게 세 번째 input이 8이면 다시 +53줄로 이동해 같은 과정을 반복한다.

따라서 위 반복문을 통해 input은 다음과 같은 조건을 만족해야 한다.

1번 : 1

2번 : (1+1)*2 = 2

3번 : (2+2)*2 = 8

4번 : (3+8)*3 = 33

5번 : (4+33)*4 = 148

6번 : (5+148)*5 = 765

Phase_2의 답은 1 2 8 33 148 765 라는 것을 알아냈다.

```
Phase 1 defused. How about the next one?
1 2 8 33 148 765
That's number 2.  Keep going!
```

Phase_3

Phase_3에 breakpoint를 설정한 후 실행해 phase_3를 살펴보았다.

```
Dump of assembler code for function phase_3:
=> 0x0000555555555465 <+0>:     sub    $0x18,%rsp
   0x0000555555555469 <+4>:     mov    %fs:0x28,%rax
   0x0000555555555472 <+13>:    mov    %rax,0x8(%rsp)
   0x0000555555555477 <+18>:    xor    %eax,%eax
   0x0000555555555479 <+20>:    lea    0x4(%rsp),%rcx
   0x000055555555547e <+25>:    mov    %rsp,%rdx
   0x0000555555555481 <+28>:    lea    0x1b82(%rip),%rsi        # 0x55555555700a
   0x0000555555555488 <+35>:    callq  0x5555555550b0 <__isoc99_sscanf@plt>
   0x000055555555548d <+40>:    cmp    $0x1,%eax
   0x0000555555555490 <+43>:    jle    0x5555555554af <phase_3+74>
   0x0000555555555492 <+45>:    cmpl   $0x8,(%rsp)
   0x0000555555555496 <+49>:    ja     0x555555555558 <phase_3+243>
   0x000055555555549c <+55>:    mov    (%rsp),%eax
   0x000055555555549f <+58>:    lea    0x18da(%rip),%rdx        # 0x555555556d80
   0x00005555555554a6 <+65>:    movslq (%rdx,%rax,4),%rax
   0x00005555555554aa <+69>:    add    %rdx,%rax
   0x00005555555554ad <+72>:    jmpq   *%rax
   0x00005555555554af <+74>:    callq  0x555555555c89 <explode_bomb>
   0x00005555555554b4 <+79>:    jmp    0x555555555492 <phase_3+45>
   0x00005555555554b6 <+81>:    mov    $0x324,%eax
   0x00005555555554bb <+86>:    jmp    0x5555555554c2 <phase_3+93>
   0x00005555555554bd <+88>:    mov    $0x0,%eax
   0x00005555555554c2 <+93>:    shr    %eax
   0x00005555555554c4 <+95>:    sub    $0x1ad,%eax
   0x00005555555554c9 <+100>:   lea    0x2d3(%rax,%rax,1),%edx
   0x00005555555554d0 <+107>:   mov    %edx,%eax
   0x00005555555554d2 <+109>:   shr    $0x1f,%eax
   0x00005555555554d5 <+112>:   add    %edx,%eax
   0x00005555555554d7 <+114>:   sar    %eax
   0x00005555555554d9 <+116>:   sub    $0x321,%eax
   0x00005555555554de <+121>:   lea    0x364(%rax,%rax,1),%eax
   0x00005555555554e5 <+128>:   sar    %eax
   0x00005555555554e7 <+130>:   sub    $0x121,%eax
   0x00005555555554ec <+135>:   lea    0x3c4(%rax,%rax,1),%eax
   0x00005555555554f3 <+142>:   sar    %eax
   0x00005555555554f5 <+144>:   sub    $0x1f4,%eax
```

이전 phase들과 같이 +28줄에 입력값의 형태가 담겨있을 것이라 생각해 출력되는 0x555555556d80의 값을 알아보았다.

```
(gdb) x/s 0x55555555700a
0x55555555700a: "%d %d"
```

또한, +40에서 scan의 리턴값인 %eax를 1과 비교해 +43줄에 따라 1보다 작거나 같으면 explode하는 것으로 보아 입력값이 두개 이상이어야 한다는 것을 알 수 있었다.

입력값으로 임의의 숫자인 10 20을 넣고, phase_3에 breakpoint를 설정 한 후 run해보았다

+45줄에서는 [%rsp]와 8을 비교해, [%rsp]가 더 크면 +243줄로 이동해 explode된다.

```
0x0000555555555558 <+243>:   callq  0x555555555c89 <explode_bomb>
```

이 경우에 %rsp의 메모리 값을 살펴보면,

```
(gdb) x/d $rsp
0x7fffffffde70: 10
```

10, 즉 첫번째 input임을 알 수 있다. 그리고 이는 8보다 크므로 explode할 것이다.

즉 첫번째 input은 8보다 작아야 한다. 7 20을 입력해 다시 run 해보았다.

```
   0x0000555555555496 <+49>:    ja     0x555555555558 <phase_3+243>
=> 0x000055555555549c <+55>:    mov    (%rsp),%eax
   0x000055555555549f <+58>:    lea    0x18da(%rip),%rdx        # 0x555555556d80
   0x00005555555554a6 <+65>:    movslq (%rdx,%rax,4),%rax
   0x00005555555554aa <+69>:    add    %rdx,%rax
   0x00005555555554ad <+72>:    jmpq   *%rax
   0x00005555555554af <+74>:    callq  0x555555555c89 <explode_bomb>
   0x00005555555554b4 <+79>:    jmp    0x555555555492 <phase_3+45>
   0x00005555555554b6 <+81>:    mov    $0x324,%eax
   0x00005555555554bb <+86>:    jmp    0x5555555554c2 <phase_3+93>
   0x00005555555554bd <+88>:    mov    $0x0,%eax
   0x00005555555554c2 <+93>:    shr    %eax
   0x00005555555554c4 <+95>:    sub    $0x1ad,%eax
   0x00005555555554c9 <+100>:   lea    0x2d3(%rax,%rax,1),%edx
   0x00005555555554d0 <+107>:   mov    %edx,%eax
   0x00005555555554d2 <+109>:   shr    $0x1f,%eax
   0x00005555555554d5 <+112>:   add    %edx,%eax
   0x00005555555554d7 <+114>:   sar    %eax
   0x00005555555554d9 <+116>:   sub    $0x321,%eax
   0x00005555555554de <+121>:   lea    0x364(%rax,%rax,1),%eax
   0x00005555555554e5 <+128>:   sar    %eax
   0x00005555555554e7 <+130>:   sub    $0x121,%eax
   0x00005555555554ec <+135>:   lea    0x3c4(%rax,%rax,1),%eax
   0x00005555555554f3 <+142>:   sar    %eax
   0x00005555555554f5 <+144>:   sub    $0x1f4,%eax
   0x00005555555554fa <+149>:   lea    0x3bc(%rax,%rax,1),%eax
   0x0000555555555501 <+156>:   cmpl   $0x6,(%rsp)
   0x0000555555555505 <+160>:   jg     0x55555555550d <phase_3+168>
   0x0000555555555507 <+162>:   cmp    %eax,0x4(%rsp)
   0x000055555555550b <+166>:   je     0x555555555512 <phase_3+173>
   0x000055555555550d <+168>:   callq  0x555555555c89 <explode_bomb>
   0x0000555555555512 <+173>:   mov    0x8(%rsp),%rax
   0x0000555555555517 <+178>:   xor    %fs:0x28,%rax
   0x0000555555555520 <+187>:   jne    0x555555555564 <phase_3+255>
```

explode하지 않고 +55줄로 이동했다.

[%rsp]의 값(7, 첫번째 input)을 %eax에 mov하고 [%rip+0x18da]의 값을 %rdx에 넣는다.

그렇게 +72줄에 따라 *%rax로 jump하면, +229번 줄로 이동하게 된다.

```
=> 0x000055555555554a <+229>:   mov    $0x0,%eax
   0x000055555555554f <+234>:   jmp    0x5555555554f3 <phase_3+142>
   0x0000555555555551 <+236>:   mov    $0x0,%eax
   0x0000555555555556 <+241>:   jmp    0x5555555554fa <phase_3+149>
   0x0000555555555558 <+243>:   callq  0x555555555c89 <explode_bomb>
   0x000055555555555d <+248>:   mov    $0x0,%eax
   0x0000555555555562 <+253>:   jmp    0x555555555501 <phase_3+156>
   0x0000555555555564 <+255>:   callq  0x555555555010 <__stack_chk_fail@plt>
End of assembler dump.
```

그럼 다시 %eax에 0의 값을 넣고 +142줄로 이동한다.

```
=> 0x00005555555554f3 <+142>:   sar    %eax
   0x00005555555554f5 <+144>:   sub    $0x1f4,%eax
   0x00005555555554fa <+149>:   lea    0x3bc(%rax,%rax,1),%eax
   0x0000555555555501 <+156>:   cmpl   $0x6,(%rsp)
   0x0000555555555505 <+160>:   jg     0x55555555550d <phase_3+168>
   0x0000555555555507 <+162>:   cmp    %eax,0x4(%rsp)
   0x000055555555550b <+166>:   je     0x555555555512 <phase_3+173>
   0x000055555555550d <+168>:   callq  0x555555555c89 <explode_bomb>
   0x0000555555555512 <+173>:   mov    0x8(%rsp),%rax
   0x0000555555555517 <+178>:   xor    %fs:0x28,%rax
   0x0000555555555520 <+187>:   jne    0x555555555564 <phase_3+255>
```

+156줄에서 [$rsp]의 값과 6을 비교해 [$rsp]의 값이 더 크면 +168로 이동해 explode하는 것을 볼 수 있다.

```
(gdb) x/d $rsp
0x7fffffffde70: 7
```

그리고 첫번째 인풋인 7이 [%rsp]에 있으므로, 6보다 크다. 따라서 첫번째 input이 6보다 작아야 함을 알았다.

Input을 5 20으로 다시 임의 설정 했다.

```
   0x0000555555555505 <+160>:   jg     0x555555555510d <phase_3+168>
=> 0x0000555555555507 <+162>:   cmp    %eax,0x4(%rsp)
   0x000055555555550b <+166>:   je     0x555555555512 <phase_3+173>
   0x000055555555550d <+168>:   callq  0x555555555c89 <explode_bomb>
   0x0000555555555512 <+173>:   mov    0x8(%rsp),%rax
   0x0000555555555517 <+178>:   xor    %fs:0x28,%rax
   0x0000555555555520 <+187>:   jne    0x555555555564 <phase_3+255>
---Type <return> to continue, or q <return> to quit---
```

+162까지 진행 후, [%rsp+0x4](두 번째 input)와 %eax가 같으면 +173으로 이동하고, 다르면 explode되는 것을 볼 수 있다.

그리고 여기서 %eax의 값은 다음과 같다.

```
(gdb) i r
rax            0x156    342
rbx            0x7fffffffdf78   140737488347000
rcx            0x0      0
rdx            0x555555556d80   93824992243072
rsi            0x0      0
rdi            0x7fffffffd800   140737488345088
rbp            0x555555556b30   0x555555556b30 <__libc_csu_init>
rsp            0x7fffffffde70   0x7fffffffde70
r8             0x0      0
r9             0x0      0
r10            0x7ffff7b82cc0   140737349430464
```

따라서 두번째 input은 342이어야 한다는 것을 알 수 있다.

5 342를 넣은 후 다시 이 부분까지 run 해보았다.

```
   0x0000555555555507 <+162>:   cmp    %eax,0x4(%rsp)
   0x000055555555550b <+166>:   je     0x555555555512 <phase_3+173>
   0x000055555555550d <+168>:   callq  0x555555555c89 <explode_bomb>
=> 0x0000555555555512 <+173>:   mov    0x8(%rsp),%rax
   0x0000555555555517 <+178>:   xor    %fs:0x28,%rax
   0x0000555555555520 <+187>:   jne    0x555555555564 <phase_3+255>
---Type <return> to continue, or q <return> to quit---
   0x0000555555555522 <+189>:   add    $0x18,%rsp
   0x0000555555555526 <+193>:   retq
   0x0000555555555527 <+194>:   mov    $0x0,%eax
   0x000055555555552c <+199>:   jmp    0x5555555554c9 <phase_3+100>
   0x000055555555552e <+201>:   mov    $0x0,%edx
   0x0000555555555533 <+206>:   jmp    0x5555555554d0 <phase_3+107>
   0x0000555555555535 <+208>:   mov    $0x0,%eax
   0x000055555555553a <+213>:   jmp    0x5555555554de <phase_3+121>
   0x000055555555553c <+215>:   mov    $0x0,%eax
   0x0000555555555541 <+220>:   jmp    0x5555555554e5 <phase_3+128>
   0x0000555555555543 <+222>:   mov    $0x0,%eax
   0x0000555555555548 <+227>:   jmp    0x5555555554ec <phase_3+135>
   0x000055555555554a <+229>:   mov    $0x0,%eax
   0x000055555555554f <+234>:   jmp    0x5555555554f3 <phase_3+142>
   0x0000555555555551 <+236>:   mov    $0x0,%eax
   0x0000555555555556 <+241>:   jmp    0x5555555554fa <phase_3+149>
   0x0000555555555558 <+243>:   callq  0x555555555c89 <explode_bomb>
   0x000055555555555d <+248>:   mov    $0x0,%eax
   0x0000555555555562 <+253>:   jmp    0x555555555501 <phase_3+156>
   0x0000555555555564 <+255>:   callq  0x555555555010 <__stack_chk_fail@plt>
End of assembler dump.
```

explode하지 않고 +193까지 진행 한 후 ret하는 것을 볼 수 있다.

Phase_3의 답은 5 342 라는 것을 알아냈다.

```
Starting program: /home/parksoohun/system program/Bomb Lab/bomb44/bomb 1.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?
1 2 8 33 148 765
That's number 2.  Keep going!
5 342
Halfway there!
```

Phase_4

Phase_4에 breakpoint 설정 후 disas 해보았다.

```
(gdb) disas
Dump of assembler code for function phase_4:
=> 0x000055555555560a <+0>:     sub    $0x28,%rsp
   0x000055555555560e <+4>:     mov    %fs:0x28,%rax
   0x0000555555555617 <+13>:    mov    %rax,0x18(%rsp)
   0x000055555555561c <+18>:    xor    %eax,%eax
   0x000055555555561e <+20>:    movl   $0x0,0x14(%rsp)
   0x0000555555555626 <+28>:    lea    0x10(%rsp),%rcx
   0x000055555555562b <+33>:    lea    0xc(%rsp),%rdx
   0x0000555555555630 <+38>:    lea    0x19d3(%rip),%rsi        # 0x55555555700a
   0x0000555555555637 <+45>:    callq  0x5555555550b0 <__isoc99_sscanf@plt>
   0x000055555555563c <+50>:    cmp    $0x2,%eax
   0x000055555555563f <+53>:    jne    0x555555555648 <phase_4+62>
   0x0000555555555641 <+55>:    cmpl   $0x15,0xc(%rsp)
   0x0000555555555646 <+60>:    jbe    0x55555555564d <phase_4+67>
   0x0000555555555648 <+62>:    callq  0x555555555c89 <explode_bomb>
   0x000055555555564d <+67>:    lea    0x14(%rsp),%rdx
   0x0000555555555652 <+72>:    mov    0xc(%rsp),%esi
   0x0000555555555656 <+76>:    lea    0x202d43(%rip),%rdi        # 0x5555557583a0 <p4_n1>
   0x000055555555565d <+83>:    callq  0x555555555569 <func4>
   0x0000555555555662 <+88>:    cmp    $0x6,%eax
   0x0000555555555665 <+91>:    jne    0x55555555566e <phase_4+100>
   0x0000555555555667 <+93>:    cmpl   $0x6,0x10(%rsp)
   0x000055555555566c <+98>:    je     0x555555555673 <phase_4+105>
   0x000055555555566e <+100>:   callq  0x555555555c89 <explode_bomb>
   0x0000555555555673 <+105>:   mov    0x18(%rsp),%rax
   0x0000555555555678 <+110>:   xor    %fs:0x28,%rax
   0x0000555555555681 <+119>:   jne    0x555555555688 <phase_4+126>
   0x0000555555555683 <+121>:   add    $0x28,%rsp
   0x0000555555555687 <+125>:   retq
   0x0000555555555688 <+126>:   callq  0x555555555010 <__stack_chk_fail@plt>
```

다른 phase들과 같이 +38의 값이 input일 거라 생각해 0x55555555700a의 값을 확인해 보았다.

```
(gdb) x/s 0x55555555700a
0x55555555700a: "%d %d"
```

입력값은 정수 두개라는 것을 알았다.

```
   0x000055555555563c <+50>:    cmp    $0x2,%eax
   0x000055555555563f <+53>:    jne    0x555555555648 <phase_4+62>
   0x0000555555555641 <+55>:    cmpl   $0x15,0xc(%rsp)
=> 0x0000555555555646 <+60>:    jbe    0x55555555564d <phase_4+67>
   0x0000555555555648 <+62>:    callq  0x555555555c89 <explode_bomb>
   0x000055555555564d <+67>:    lea    0x14(%rsp),%rdx
   0x0000555555555652 <+72>:    mov    0xc(%rsp),%esi
   0x0000555555555656 <+76>:    lea    0x202d43(%rip),%rdi        # 0x5555557583a0 <p4_n1>
   0x000055555555565d <+83>:    callq  0x555555555569 <func4>
   0x0000555555555662 <+88>:    cmp    $0x6,%eax
   0x0000555555555665 <+91>:    jne    0x55555555566e <phase_4+100>
   0x0000555555555667 <+93>:    cmpl   $0x6,0x10(%rsp)
   0x000055555555566c <+98>:    je     0x555555555673 <phase_4+105>
   0x000055555555566e <+100>:   callq  0x555555555c89 <explode_bomb>
   0x0000555555555673 <+105>:   mov    0x18(%rsp),%rax
   0x0000555555555678 <+110>:   xor    %fs:0x28,%rax
   0x0000555555555681 <+119>:   jne    0x555555555688 <phase_4+126>
   0x0000555555555683 <+121>:   add    $0x28,%rsp
   0x0000555555555687 <+125>:   retq
   0x0000555555555688 <+126>:   callq  0x555555555010 <__stack_chk_fail@plt>
End of assembler dump.
(gdb) x/d $rsp+0xc
0x7fffffffde6c: 1
(gdb)
```

Scanf의 리턴값인 %eax의 값은 두개의 정수를 입력했으니 2이므로 +53 넘어 +55로 진행한다.

그리고 [%rcp+0xc](첫번째 인풋)을 0x15(21)과 비교해 이보다 작거나 같으면 +67로 진행하고 아니면 explode하는 것을 알수 있어, 첫번째 input은 21보다 작거나 같아야한다는 것을 알 수 있었다.

그래서 10과 20을 입력 후 실행 해 +67로 진행해 보았다.

```
   0x000055555555564d <+67>:    lea    0x14(%rsp),%rdx
   0x0000555555555652 <+72>:    mov    0xc(%rsp),%esi
   0x0000555555555656 <+76>:    lea    0x202d43(%rip),%rdi        # 0x5555557583a0 <p4_n1>
=> 0x000055555555565d <+83>:    callq  0x555555555569 <func4>
```

+67, 72, 76 과정을 통해, 아래와 같은 결과가 되었다.

```
(gdb) i r
rax            0x2      2
rbx            0x7fffffffdf78   140737488347000
rcx            0x0      0
rdx            0x7fffffffde74   140737488346740
rsi            0xa      10
rdi            0x5555557583a0   93824994345888
rbp            0x555555556b30   0x555555556b30 <__libc_csu_init>
rsp            0x7fffffffde60   0x7fffffffde60
r8             0x0      0
r9             0x0      0
r10            0x7ffff7b82cc0   140737349430464
r11            0x55555555700f   93824992243727
r12            0x555555555170   93824992235888
r13            0x7fffffffdf70   140737488346992
r14            0x0      0
r15            0x0      0
rip            0x55555555565d   0x55555555565d <phase_4+83>
eflags         0x287    [ CF PF SF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) x/d 0x5555557583a0
0x5555557583a0 <p4_n1>: 20
```

그 후 +83줄에 따라 func4를 call한다.

```
0x000055555555565d <+83>:    callq  0x555555555569 <func4>
0x0000555555555662 <+88>:    cmp    $0x6,%eax
0x0000555555555665 <+91>:    jne    0x55555555566e <phase_4+100>
0x0000555555555667 <+93>:    cmpl   $0x6,0x10(%rsp)
0x000055555555566c <+98>:    je     0x555555555673 <phase_4+105>
0x000055555555566e <+100>:   callq  0x555555555c89 <explode_bomb>
0x0000555555555673 <+105>:   mov    0x18(%rsp),%rax
0x0000555555555678 <+110>:   xor    %fs:0x28,%rax
0x0000555555555681 <+119>:   jne    0x555555555688 <phase_4+126>
0x0000555555555683 <+121>:   add    $0x28,%rsp
0x0000555555555687 <+125>:   retq
0x0000555555555688 <+126>:   callq  0x555555555010 <__stack_chk_fail@plt>
```

+88줄에 따라 func4의 리턴값이 저장돼있는 %eax를 6과 비교해서 같지 않으면 +100으로 이동해 explode하는 것을 알수 있으므로, input의 func4 리턴값이 6이어야한다는 것을 알 수 있다. 그리고 +93줄에서 [%rsp+0x10]을 6과 비교해 같으면 +105줄로, 다르면 +100으로 진행해 explode한다는 것을 알 수 있다.

Func4를 disas 해보았다.

```
0x000055555555559c <+51>:    cmp    $0x1,%ecx
0x000055555555559f <+54>:    cmovne %eax,%ebx
0x00005555555555a2 <+57>:    cmp    %esi,%eax
0x00005555555555a4 <+59>:    je     0x5555555555db <func4+114>
0x00005555555555a6 <+61>:    mov    %rdx,%r13
0x00005555555555a9 <+64>:    mov    %esi,%r14d
0x00005555555555ac <+67>:    mov    %rdi,%rbp
0x00005555555555af <+70>:    mov    0x8(%rdi),%rdi
0x00005555555555b3 <+74>:    test   %rdi,%rdi
0x00005555555555b6 <+77>:    je     0x5555555555e3 <func4+122>
0x00005555555555b8 <+79>:    cmpl   $0x0,0x4(%rdi)
0x00005555555555bc <+83>:    je     0x5555555555f0 <func4+135>
0x00005555555555be <+85>:    mov    0x10(%rbp),%rdi
0x00005555555555c2 <+89>:    test   %rdi,%rdi
0x00005555555555c5 <+92>:    je     0x5555555555cd <func4+100>
0x00005555555555c7 <+94>:    cmpl   $0x0,0x4(%rdi)
0x00005555555555cb <+98>:    je     0x5555555555fa <func4+145>
0x00005555555555cd <+100>:   add    %r12d,%ebx
0x00005555555555d0 <+103>:   mov    %ebx,%eax
0x00005555555555d2 <+105>:   pop    %rbx
0x00005555555555d3 <+106>:   pop    %rbp
0x00005555555555d4 <+107>:   pop    %r12
0x00005555555555d6 <+109>:   pop    %r13
0x00005555555555d8 <+111>:   pop    %r14
0x00005555555555da <+113>:   retq
0x00005555555555db <+114>:   movl   $0x1,(%rdx)
0x00005555555555e1 <+120>:   jmp    0x5555555555d0 <func4+103>
Type <return> to continue, or q <return> to quit---
0x00005555555555e3 <+122>:   cmpq   $0x0,0x10(%rbp)
0x00005555555555e8 <+127>:   je     0x5555555555d0 <func4+103>
0x00005555555555ea <+129>:   mov    0x10(%rbp),%rdi
0x00005555555555ee <+133>:   jmp    0x5555555555c7 <func4+94>
0x00005555555555f0 <+135>:   callq  0x555555555569 <func4>
0x00005555555555f5 <+140>:   mov    %eax,%r12d
0x00005555555555f8 <+143>:   jmp    0x5555555555be <func4+85>
0x00005555555555fa <+145>:   mov    %r13,%rdx
0x00005555555555fd <+148>:   mov    %r14d,%esi
0x0000555555555600 <+151>:   callq  0x555555555569 <func4>
0x0000555555555605 <+156>:   add    %eax,%r12d
0x0000555555555608 <+159>:   jmp    0x5555555555cd <func4+100>
```

함수의 끝 부분을 보면,

+59줄에서 조건을 만족하면 +114로 jump한 후 +103으로 jump해 ret하고,

+77줄에서 조건을 만족하면 +122로 jump한 후, 다시 +127의 조건을 만족하면 +103으로 jump해 ret하고,

+92에서 조건을 만족하면 +100으로 점프해 ret하게 되고,

+83줄에서 조건을 만족하면 +135로 점프해 다시 함수를 call하게 된다. 세부 라인을 살펴보았다.

0x000055555555558a <+33>: shr $0x1f,%r8d

%r8의 값은 20이고 이 값을 오른쪽으로 0x1f(31)만큼 shift한다. 즉, 32바이트의 숫자가 음수였다면 1이 될 것이고, 양수였다면 0이 될 것이다.

0x000055555555558e <+37>: lea (%rax,%r8,1),%ecx

+%ecx에 [%rax + %r8]의 값을 넣는다. [%rax+1*%r8]의 값은 20이므로, %ecx에 20이 들어가게 된다.

0x0000555555555592 <+41>: and $0x1,%ecx

0x14(20)와 1을 and 연산하게 되면, 0이 된다.

0x0000555555555595 <+44>: sub %r8d,%ecx

그 후, 0에서 0을 빼게 되면 0이 된다.

0x0000555555555598 <+47>: mov %eax,%ebx

%eax에 있던 두번째 input값인 20을 %ebx로 옮긴다.

0x000055555555559a <+49>: neg %ebx

그리고 그 값을 2의 보수를 계산하면 부호가 반전되므로 %ebx의 값은 -20이 된다.

0x000055555555559c <+51>: cmp $0x1,%ecx

%ecx는 +44줄에 의해서 0이 되어있었으므로, 1과 cmp하게 되면, 같지 않으므로, ZF가 0으로 set된다.

0x000055555555559f <+54>: cmovne %eax,%ebx

현재 %eax의 값은 20, %ebx의 값은 -20이다. 이 두 값을 cmovne(mov if not equal (ZF=0))를 하게 되면, %ebx의 값에 %eax의 값인 20이 mov되게 된다.

0x00005555555555a2 <+57>: cmp %esi,%eax

위의 과정들에 따라 %eax에는 20, %esi에는 10이 들어가 있으므로, 값이 다르다.

```
> 0x00005555555555a2 <+57>:    cmp    %esi,%eax
  0x00005555555555a4 <+59>:    je     0x5555555555db <func4+114>
  0x00005555555555a6 <+61>:    mov    %rdx,%r13
  0x00005555555555a9 <+64>:    mov    %esi,%r14d
  0x00005555555555ac <+67>:    mov    %rdi,%rbp
  0x00005555555555af <+70>:    mov    0x8(%rdi),%rdi
  0x00005555555555b3 <+74>:    test   %rdi,%rdi
  0x00005555555555b6 <+77>:    je     0x5555555555e3 <func4+122>
  0x00005555555555b8 <+79>:    cmpl   $0x0,0x4(%rdi)
  0x00005555555555bc <+83>:    je     0x5555555555f0 <func4+135>
  0x00005555555555be <+85>:    mov    0x10(%rbp),%rdi
  0x00005555555555c2 <+89>:    test   %rdi,%rdi
  0x00005555555555c5 <+92>:    je     0x5555555555cd <func4+100>
  0x00005555555555c7 <+94>:    cmpl   $0x0,0x4(%rdi)
  0x00005555555555cb <+98>:    je     0x5555555555fa <func4+145>
  0x00005555555555cd <+100>:   add    %r12d,%ebx
  0x00005555555555d0 <+103>:   mov    %ebx,%eax
  0x00005555555555d2 <+105>:   pop    %rbx
  0x00005555555555d3 <+106>:   pop    %rbp
  0x00005555555555d4 <+107>:   pop    %r12
  0x00005555555555d6 <+109>:   pop    %r13
  0x00005555555555d8 <+111>:   pop    %r14
  0x00005555555555da <+113>:   retq
  0x00005555555555db <+114>:   movl   $0x1,(%rdx)
  0x00005555555555e1 <+120>:   jmp    0x5555555555d0 <func4+103>
  0x00005555555555e3 <+122>:   cmpq   $0x0,0x10(%rbp)
```

+59줄에 의해 같으면 114줄로 이동하고, 다르면 +61줄로 진행한다.

하지만 여기서 +114로 이동하면 +120에서 +103으로 jump해 +113까지 조건문없이 진행해 ret한다는 것을 알 수 있었다.

Phase_4를 disas했을 때, 이 func4의 결과값이 6이어야한다는 점을 알아냈기 때문에, 첫번째 input과 두번째 input이 같아야하고, 첫번째 input이 21이하이며, 두번째 input이 첫번째와 똑같다면, +57에서 cmp 결과가 같다고 되어 +114줄로 이동할 수 있음을 알아냈다.

그래서 phase_4를 해체하기 위해선 6 6이 맞는 input이라는 것을 알아냈다.

```
Halfway there!
6 6
So you got that one.  Try this one.
```

Phase_5

Phase_5에 breakpoint 설정 후, disas해 살펴보았다.

```
ump of assembler code for function phase_5:
> 0x000055555555568d <+0>:     sub    $0x28,%rsp
  0x0000555555555691 <+4>:     mov    %fs:0x28,%rax
  0x000055555555569a <+13>:    mov    %rax,0x18(%rsp)
  0x000055555555569f <+18>:    xor    %eax,%eax
  0x00005555555556a1 <+20>:    lea    0xc(%rsp),%rcx
  0x00005555555556a6 <+25>:    lea    0x8(%rsp),%rdx
  0x00005555555556ab <+30>:    lea    0x14(%rsp),%r9
  0x00005555555556b0 <+35>:    lea    0x10(%rsp),%r8
  0x00005555555556b5 <+40>:    lea    0x1948(%rip),%rsi        # 0x555555557004
  0x00005555555556bc <+47>:    callq  0x5555555550b0 <__isoc99_sscanf@plt>
  0x00005555555556c1 <+52>:    cmp    $0x3,%eax
  0x00005555555556c4 <+55>:    jle    0x555555555700 <phase_5+115>
  0x00005555555556c6 <+57>:    cmpl   $0x3,0xc(%rsp)
  0x00005555555556cb <+62>:    jne    0x555555555707 <phase_5+122>
  0x00005555555556cd <+64>:    mov    0xc(%rsp),%r9d
  0x00005555555556d2 <+69>:    mov    0x8(%rsp),%eax
  0x00005555555556d6 <+73>:    mov    $0x0,%r8d
  0x00005555555556dc <+79>:    mov    $0x0,%edi
  0x00005555555556e1 <+84>:    mov    $0x0,%ecx
  0x00005555555556e6 <+89>:    mov    $0x0,%edx
  0x00005555555556eb <+94>:    lea    0x16ce(%rip),%rsi        # 0x555555556dc0-
<array.3444>
  0x00005555555556f2 <+101>:   mov    $0x7,%r11d
  0x00005555555556f8 <+107>:   mov    $0x1,%r10d
  0x00005555555556fe <+113>:   jmp    0x555555555714 <phase_5+135>
  0x0000555555555700 <+115>:   callq  0x555555555c89 <explode_bomb>
  0x0000555555555705 <+120>:   jmp    0x5555555556c6 <phase_5+57>
  0x0000555555555707 <+122>:   callq  0x555555555c89 <explode_bomb>
  0x000055555555570c <+127>:   jmp    0x5555555556cd <phase_5+64>
  0x000055555555570e <+129>:   mov    %r11d,%eax
  0x0000555555555711 <+132>:   mov    %r10d,%r8d
  0x0000555555555714 <+135>:   cmp    %edi,%r9d
  0x0000555555555717 <+138>:   jle    0x555555555738 <phase_5+171>
  0x0000555555555719 <+140>:   add    $0x1,%edi
  0x000055555555571c <+143>:   imul   %edi,%eax
  0x000055555555571f <+146>:   and    $0xf,%eax
  0x0000555555555722 <+149>:   cmp    $0x7,%eax
  0x0000555555555725 <+152>:   je     0x55555555570e <phase_5+129>
  0x0000555555555727 <+154>:   add    $0x1,%edx
  0x000055555555572a <+157>:   cltq
  0x000055555555572c <+159>:   mov    (%rsi,%rax,4),%eax
  0x000055555555572f <+162>:   add    %eax,%ecx
  0x0000555555555731 <+164>:   cmp    $0x7,%eax
  0x0000555555555734 <+167>:   jne    0x555555555727 <phase_5+154>
```

```
(gdb) x/s 0x555555557004
0x555555557004:  "%d %d %d %d"
```

이전 phase들과 같은 방법으로, scanf 줄 전의 #0x555555557004의 문자열을 확인해 input이 정수 네 개라는 것을 알아냈다. 또한, +52줄부터 보면, scan의 리턴값이 저장돼있는 %eax를 3과 비교해 3보다 작거나 같으면 +122로 jump해 explode하는 것을 알 수 있었다. 그래서 임의의 input 값으로 10 20 30 40을 입력 후 disas 해보았다.

0x00005555555556c6 <+57>: cmpl $0x3,0xc(%rsp)

0x00005555555556cb <+62>: jne 0x555555555707 <phase_5+122>

그 밑의 줄을 보면, [%rsp+0xc] (두번째 input, 20)을 3과 비교해 같지 않으면 +122줄로 이동해 explode한다. 즉 두번째 input은 3이어야한다. 다시 임의의 input을 10 3 30 40으으로 입력 후 살펴보았다.

0x00005555555556cd <+64>: mov 0xc(%rsp),%r9d

0x00005555555556d2 <+69>: mov 0x8(%rsp),%eax

이 두 줄에 따라, %r9d 레지스터에는 두번째 input, 3의 값이 들어가며, %eax에는 첫 번째 input, 10이 들어간다.

0x00005555555556d6 <+73>: mov $0x0,%r8d

0x00005555555556dc <+79>: mov $0x0,%edi

0x00005555555556e1 <+84>: mov $0x0,%ecx

0x00005555555556e6 <+89>: mov $0x0,%edx

그 후 이 네 줄에 의해, %r8, %edi, %ecx, %edx에는 0의 값이 들어간다.

0x00005555555556eb <+94>: lea 0x16ce(%rip),%rsi # 0x555555556dc0 <array.3444>

```
(gdb) x/d 0x555555556dc0
0x555555556dc0 <array.3444>:    11
```

%rsi에는 위의 값이 들어가게 된다.

0x00005555555556f2 <+101>: mov $0x7,%r11d

0x00005555555556f8 <+107>: mov $0x1,%r10d

0x00005555555556fe <+113>: jmp 0x555555555714 <phase_5+135>

그 후 세 줄에 의해, %r11d에는 7, $r10d에는 1이 들어가며, 135로 jump한다.

0x0000555555555714 <+135>: cmp %edi,%r9d

0x0000555555555717 <+138>: jle 0x555555555738 <phase_5+171>

현재, %r9d에는 위의 과정에 따라 두번째 input인 3이 들어있고, %edi에는 0이 들어있다. +138의 조건을 만족시키지 못하므로, jump하지 않고 진행한다.

0x0000555555555719 <+140>: add $0x1,%edi

%edi에는 0이 있었으므로 1이 더해져 1이 저장된다.

0x000055555555571c <+143>: imul %edi,%eax

%eax(첫번째 input, 10)에 %edi의 1을 곱하면 10이 된다.

0x000055555555571f <+146>: and $0xf,%eax

%eax(10)과 0xf(15)를 and 연산 하게 되면, 끝의 네 개의 bit들은 그대로 남으며, 그 상위 비트들은 0으로 남게된다. 즉 숫자가 15 이상이라면, 하위 네 개의 bit들만이 남게 된다. 이 경우는 10이므로 그대로 10의 숫자가 남게 된다.

0x0000555555555722 <+149>: cmp $0x7,%eax

0x0000555555555725 <+152>: je 0x55555555570e <phase_5+129>

이 때 10과 7을 cmp하면, 같지 않으므로 +154줄로 진행한다.

현재의 레지스터 값들은 아래와 같다.

```
rax            0xa      10
rbx            0x7fffffffdf78   140737488347000
rcx            0x0      0
rdx            0x0      0
rsi            0x555555556dc0   93824992243136
rdi            0x1      1
rbp            0x555555556b30   0x555555556b30 <__libc_csu_init>
rsp            0x7fffffffde60   0x7fffffffde60
r8             0x0      0
r9             0x3      3
r10            0x1      1
r11            0x7      7
r12            0x555555555170   93824992235888
r13            0x7fffffffdf70   140737488346992
r14            0x0      0
r15            0x0      0
rip            0x555555555722   0x555555555722 <phase_5+149>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

0x0000555555555727 <+154>: add $0x1,%edx

%rdx의 값은 1이 된다.

0x000055555555572a <+157>: cltq

%eax의 값을 %rax까지 sign extension시킨다. 값에 영향이 없다.

0x000055555555572c <+159>: mov (%rsi,%rax,4),%eax

[%rsi + 4*%rax]의 값을 %eax로 이동시킨다.

```
(gdb) x/d $rsi+4*$rax
0x555555556de8 <array.3444+40>: 2
```

0x000055555555572f <+162>: add %eax,%ecx

%ecx의 값은 현재 0이고, %eax의 값을 더하게 되면 2가 된다.

0x0000555555555731 <+164>: cmp $0x7,%eax

0x0000555555555734 <+167>: jne 0x555555555727 <phase_5+154>

%eax(2)와 7을 비교하면 같지 않으므로 +154로 jump해 반복한다..

이 반복이 끝나려면, +167에서 조건을 만족하지 않아 +129로 jump한 후, +138줄의 아래 조건을 만족해 +171로 jump해야 한다.

즉, +154 ~ +167의 과정을, %eax의 값이 7이 될때까지 반복하며, 한번 loop를 돌 때마다 %edx에 1이 더해진다.

아래의 array.3444의 요소들에 근거해 계산해보면 다음과 같다.

```
(gdb) x/d $rsi+8
0x555555556dc8 <array.3444+8>:  0
(gdb) x/d $rsi+12
0x555555556dcc <array.3444+12>: 1
(gdb) x/d $rsi+16
0x555555556dd0 <array.3444+16>: 10
(gdb) x/d $rsi+20
0x555555556dd4 <array.3444+20>: 12
(gdb) x/d $rsi+24
0x555555556dd8 <array.3444+24>: 8
(gdb) x/d $rsi+28
0x555555556ddc <array.3444+28>: 14
(gdb) x/d $rsi+32
0x555555556de0 <array.3444+32>: 9
(gdb) x/d $rsi+36
0x555555556de4 <array.3444+36>: 4
(gdb) x/d $rsi+40
0x555555556de8 <array.3444+40>: 2
(gdb) x/d $rsi+44
0x555555556dec <array.3444+44>: 15
(gdb) x/d $rsi+48
0x555555556df0 <array.3444+48>: 3
(gdb) x/d $rsi+52
0x555555556df4 <array.3444+52>: 6
(gdb) x/d $rsi+56
0x555555556df8 <array.3444+56>: 5
(gdb) x/d $rsi+60
0x555555556dfc <array.3444+60>: 7
```

따라서 %eax = 15라면, (array.3444+4*15 = 7이기 때문에)바로 이 루프를 통과할 수 있음을 알 수 있었다.

그 후, +169에 따라 +129로 jump한다. 이 때의 레지스터 값은 아래와 같다.

```
(gdb) i r
rax            0x7      7
rbx            0x7fffffffdf78   140737488347000
rcx            0x7      7
rdx            0x1      1
rsi            0x555555556dc0   93824992243136
rdi            0x1      1
rbp            0x555555556b30   0x555555556b30 <__libc_csu_init>
rsp            0x7fffffffde60   0x7fffffffde60
r8             0x0      0
r9             0x3      3
r10            0x1      1
r11            0x7      7
r12            0x555555555170   93824992235888
r13            0x7fffffffdf70   140737488346992
r14            0x0      0
r15            0x0      0
rip            0x55555555570e   0x55555555570e <phase_5+129>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

0x000055555555570e <+129>: mov %r11d,%eax

0x0000555555555711 <+132>: mov %r10d,%r8d

0x0000555555555714 <+135>: cmp %edi,%r9d

0x0000555555555717 <+138>: jle 0x555555555738 <phase_5+171>

0x0000555555555719 <+140>: add $0x1,%edi

0x000055555555571c <+143>: imul %edi,%eax

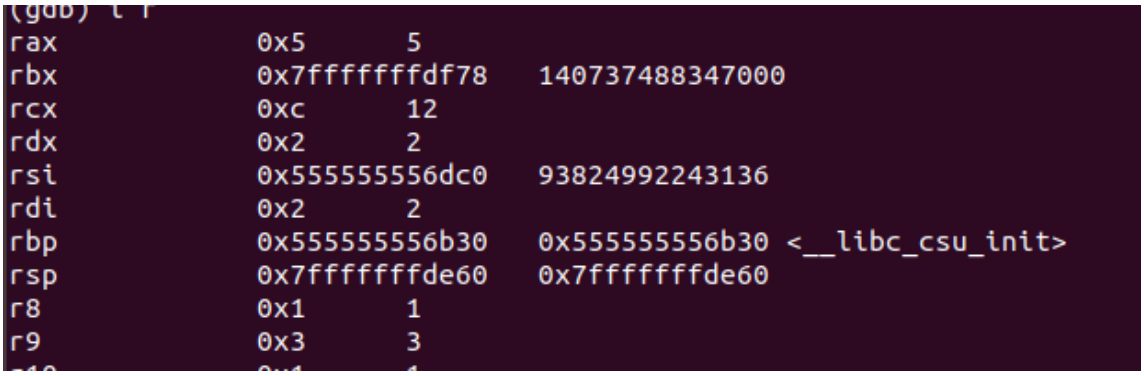
0x000055555555571f <+146>: and $0xf,%eax

0x0000555555555722 <+149>: cmp $0x7,%eax

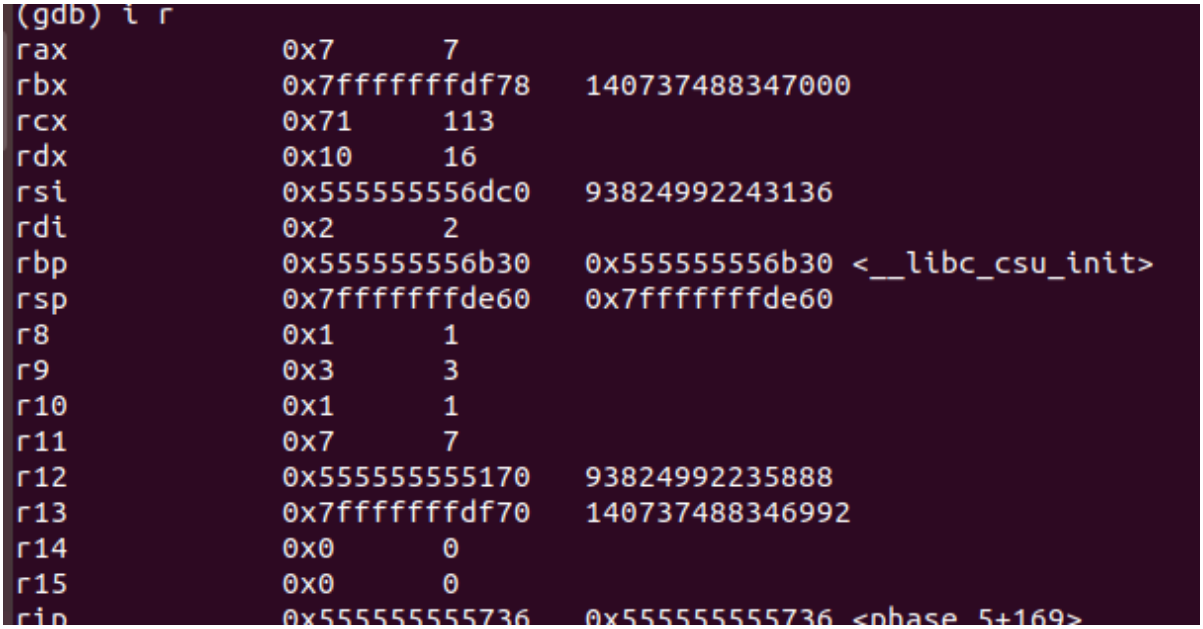
0x0000555555555725 <+152>: je 0x55555555570e <phase_5+129>

그 후, +129부터의 과정에 따라 %eax에 7, %r8d에 1이 이동된다.

이 때도, 반복을 하게 되면 반복에 따른 +169에서의 레지스터값은 아래와 같다.



그렇다면 다시 rax의 값이 15가 될 때, 이 루프를 탈출한다. 탈출할 때의 레지스터값은 아래와 같다



그리고 다시+129로 가 루프를 반복한다. 하지만 이 때 %edi의 값은 2로, 아직 %r9d의 값보다 작으므로 또 다시 반복된다. 다시 모든 똑 같은 과정을 거친 후 탈출했을 때의 레지스터 값은 아래와 같다.

```
(gdb) i r
rax            0x7      7
rbx            0x7fffffffdf78   140737488347000
rcx            0xd6     214
rdx            0x1e     30
rsi            0x555555556dc0   93824992243136
rdi            0x3      3
rbp            0x555555556b30   0x555555556b30 <__libc_csu_init>
rsp            0x7fffffffde60   0x7fffffffde60
r8             0x1      1
r9             0x3      3
r10            0x1      1
r11            0x7      7
r12            0x555555555170   93824992235888
r13            0x7fffffffdf70   140737488346992
r14            0x0      0
r15            0x0      0
rip            0x55555555570e   0x55555555570e <phase_5+129>
eflags         0x246    [ PF ZF IF ]
```

이제 %edi의 값이 3이므로 이제 +138줄의 jle 조건을 만족해 +171로 jump한다. 그 이후의 코드는 다음과 같다.

```
0x0000555555555738 <+171>:   test   %r8b,%r8b
-Type <return> to continue, or q <return> to quit---
0x000055555555573b <+174>:   jne    0x555555555763 <phase_5+214>
0x000055555555573d <+176>:   cmp    %edx,0x10(%rsp)
0x0000555555555741 <+180>:   jne    0x555555555749 <phase_5+188>
0x0000555555555743 <+182>:   cmp    %ecx,0x14(%rsp)
0x0000555555555747 <+186>:   je     0x55555555574e <phase_5+193>
0x0000555555555749 <+188>:   callq  0x555555555c89 <explode_bomb>
0x000055555555574e <+193>:   mov    0x18(%rsp),%rax
0x0000555555555753 <+198>:   xor    %fs:0x28,%rax
0x000055555555575c <+207>:   jne    0x555555555769 <phase_5+220>
0x000055555555575e <+209>:   add    $0x28,%rsp
0x0000555555555762 <+213>:   retq
0x0000555555555763 <+214>:   mov    %eax,0x8(%rsp)
0x0000555555555767 <+218>:   jmp    0x55555555573d <phase_5+176>
0x0000555555555769 <+220>:   callq  0x555555555010 <__stack_chk_fail@plt>
d of assembler dump.
```

+174에 따라 +214로 jump한 후,

0x0000555555555763 <+214>: mov %eax,0x8(%rsp)

0x0000555555555767 <+218>: jmp 0x55555555573d <phase_5+176>

위의 두 줄에 따라, [%rsp]+0x8에 %eax의 값인 7을 이동시키고 +176으로 jump한다.

0x000055555555573d <+176>: cmp %edx,0x10(%rsp)

0x0000555555555741 <+180>: jne 0x555555555749 <phase_5+188>

%edx의 값은 현재 30이며, [%rsp+0x10]은 세번째 input을 가리킨다. 따라서 만족하므로 +188로 jump하지 않고, +182로 진행한다.

0x0000555555555743 <+182>: cmp %ecx,0x14(%rsp)

0x0000555555555747 <+186>: je 0x55555555574e <phase_5+193>

[%rsp+0x14]는 네번째 input이 저장된 메모리이며, 현재 %ecx의 값은214이다. 즉 네 번째 input은 214여야 한다.

그렇다면, +186의 조건을 만족시켜, +193으로 jump한다.

0x000055555555574e <+193>: mov 0x18(%rsp),%rax

0x0000555555555753 <+198>: xor %fs:0x28,%rax

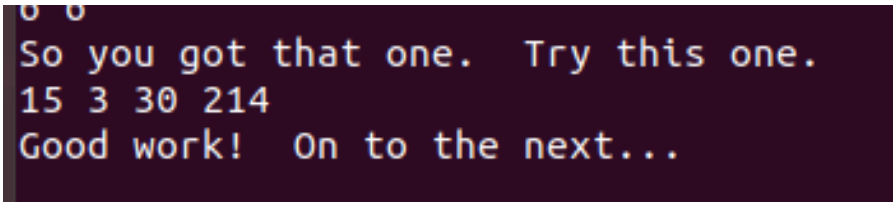
0x000055555555575c <+207>: jne 0x555555555769 <phase_5+220>

0x000055555555575e <+209>: add $0x28,%rsp

0x0000555555555762 <+213>: retq

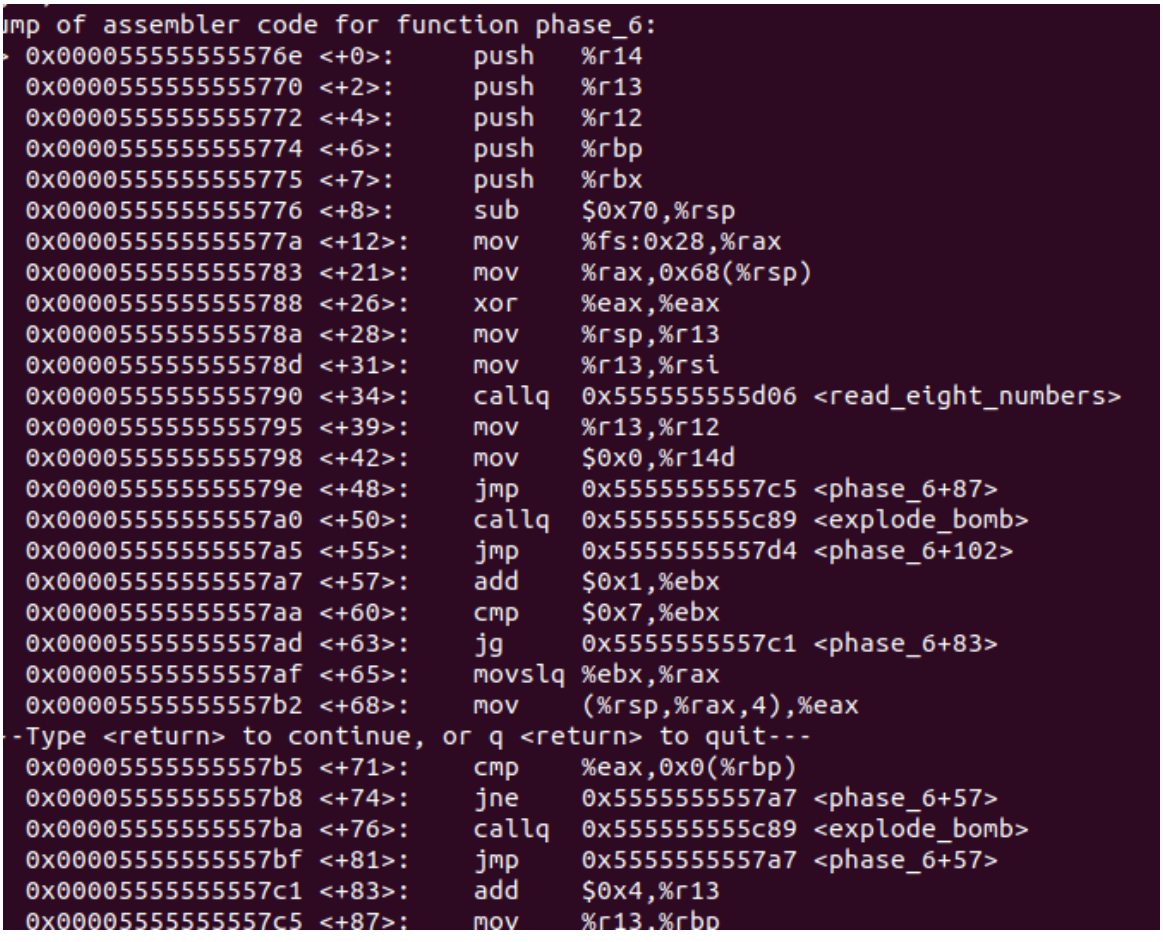
그 후 +213으로 진행해 ret하는 것을 볼 수 있다.

따라서 phase_5를 해체하기 위해선, 15 3 30 214가 알맞은 input이라는 것을 알아냈다.



Phase_6

Phase_6에 breakpoint를 설정하고 disas 해보았다.



+34의 read_eight_numbers 함수를 call하는 것으로 보아, 8개의 정수를 입력하는 함수라는 것을 예상할 수 있었

다. 그래서 read_eight_numbers에 두 번째 breakpoint를 설정 한 후 이 함수를 disas해보았다.

```
0x0000555555555d06 <+0>:     sub    $0x8,%rsp
0x0000555555555d0a <+4>:     mov    %rsi,%rdx
0x0000555555555d0d <+7>:     lea    0x4(%rsi),%rcx
0x0000555555555d11 <+11>:    lea    0x1c(%rsi),%rax
0x0000555555555d15 <+15>:    push   %rax
0x0000555555555d16 <+16>:    lea    0x18(%rsi),%rax
0x0000555555555d1a <+20>:    push   %rax
0x0000555555555d1b <+21>:    lea    0x14(%rsi),%rax
0x0000555555555d1f <+25>:    push   %rax
0x0000555555555d20 <+26>:    lea    0x10(%rsi),%rax
0x0000555555555d24 <+30>:    push   %rax
0x0000555555555d25 <+31>:    lea    0xc(%rsi),%r9
0x0000555555555d29 <+35>:    lea    0x8(%rsi),%r8
0x0000555555555d2d <+39>:    lea    0x12c4(%rip),%rsi        # 0x555555556ff8
0x0000555555555d34 <+46>:    mov    $0x0,%eax
0x0000555555555d39 <+51>:    callq  0x5555555550b0 <__isoc99_sscanf@plt>
0x0000555555555d3e <+56>:    add    $0x20,%rsp
0x0000555555555d42 <+60>:    cmp    $0x7,%eax
0x0000555555555d45 <+63>:    jle    0x555555555d4c <read_eight_numbers+70>
0x0000555555555d47 <+65>:    add    $0x8,%rsp
0x0000555555555d4b <+69>:    retq
0x0000555555555d4c <+70>:    callq  0x555555555c89 <explode_bomb>
```

다른 phase들과 마찬가지로 +39줄의 0x555555556ff8에 인풋의 형태가 있을 것이라 생각해 저 주소의 내용을 확인해 보았다.

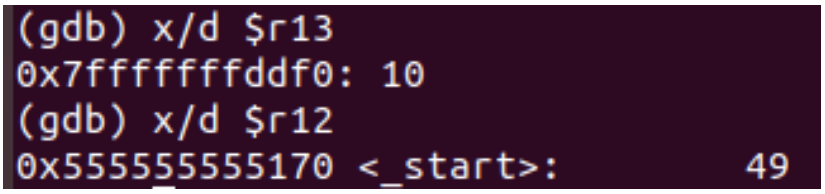
```
(gdb) x/s 0x555555556ff8
0x555555556ff8: "%d %d %d %d %d %d %d %d"
```

그리고 +60, 63의 줄로 보아, +51의 scan함수의 리턴값인 %eax를 7과 비교에 7보다 작으면 +70으로 이동해 explode하는 것으로 보아 입력값은 8개의 정수이다. 그래서 임의의 값 8 7 6 5 4 3 2 1을 입력 후 run 해보았다.

```
0x0000555555555795 <+39>:    mov    %r13,%r12
0x0000555555555798 <+42>:    mov    $0x0,%r14d
0x000055555555579e <+48>:    jmp    0x5555555557c5 <phase_6+87>
0x00005555555557a0 <+50>:    callq  0x555555555c89 <explode_bomb>
0x00005555555557a5 <+55>:    jmp    0x5555555557d4 <phase_6+102>
0x00005555555557a7 <+57>:    add    $0x1,%ebx
0x00005555555557aa <+60>:    cmp    $0x7,%ebx
0x00005555555557ad <+63>:    jg     0x5555555557c1 <phase_6+83>
0x00005555555557af <+65>:    movslq %ebx,%rax
0x00005555555557b2 <+68>:    mov    (%rsp,%rax,4),%eax
0x00005555555557b5 <+71>:    cmp    %eax,0x0(%rbp)
0x00005555555557b8 <+74>:    jne    0x5555555557a7 <phase_6+57>
0x00005555555557ba <+76>:    callq  0x555555555c89 <explode_bomb>
0x00005555555557bf <+81>:    jmp    0x5555555557a7 <phase_6+57>
0x00005555555557c1 <+83>:    add    $0x4,%r13
0x00005555555557c5 <+87>:    mov    %r13,%rbp
0x00005555555557c8 <+90>:    mov    0x0(%r13),%eax
0x00005555555557cc <+94>:    sub    $0x1,%eax
0x00005555555557cf <+97>:    cmp    $0x7,%eax
0x00005555555557d2 <+100>:   ja     0x5555555557a0 <phase_6+50>
0x00005555555557d4 <+102>:   add    $0x1,%r14d
0x00005555555557d8 <+106>:   cmp    $0x8,%r14d
0x00005555555557dc <+110>:   je     0x5555555557e3 <phase_6+117>
0x00005555555557de <+112>:   mov    %r14d,%ebx
0x00005555555557e1 <+115>:   jmp    0x5555555557af <phase_6+65>
0x00005555555557e3 <+117>:   lea    0x20(%r12),%rcx
0x00005555555557e8 <+122>:   mov    $0x9,%edx
0x00005555555557ed <+127>:   mov    %edx,%eax
0x00005555555557ef <+129>:   sub    (%r12),%eax
0x00005555555557f3 <+133>:   mov    %eax,(%r12)
0x00005555555557f7 <+137>:   add    $0x4,%r12
0x00005555555557fb <+141>:   cmp    %rcx,%r12
```

그 후의 코드는 위와 같다.

+39에 따라,



첫번째 input의 주소값을 _start함수의 처음에 이동시킨다. 그리고 +48에 따라 +87로 jump한다.

0x00005555555557c5 <+87>: mov %r13,%rbp

%rbp에 %r13(첫번째 input의 주소값, 8)이 이동된다.

0x00005555555557c8 <+90>: mov 0x0(%r13),%eax

%rax 레지스터의 값은 8이 된다.

0x00005555555557cc <+94>: sub $0x1,%eax

%rax 레지스터의 값은 7가 된다.

0x00005555555557cf <+97>: cmp $0x7,%eax

0x00005555555557d2 <+100>: ja 0x5555555557a0 <phase_6+50>

%eax가 7보다 크면 +50으로 이동해 explode하므로, 첫번째 input은 8 이하이어야 한다는 것을 알 수 있었다.

그리고 뒤의

0x00005555555557d4 <+102>: add $0x1,%r14d

%r14의 레지스터값은 0이다. +102 줄에 의해 1이 더해져 1이 된다.

0x00005555555557d8 <+106>: cmp $0x8,%r14d

0x00005555555557dc <+110>: je 0x5555555557e3 <phase_6+117>

코드로 보아, 8번이 반복되어야 +117로 jump할 수 있다는 것을 예상했다.

0x00005555555557de <+112>: mov %r14d,%ebx

0x00005555555557e1 <+115>: jmp 0x5555555557af <phase_6+65>

그 이후의 +112, +115줄에 의해, %r14의 레지스터 값이 %ebx로 이동하고 +65로 jump한다.

0x00005555555557af <+65>: movslq %ebx,%rax

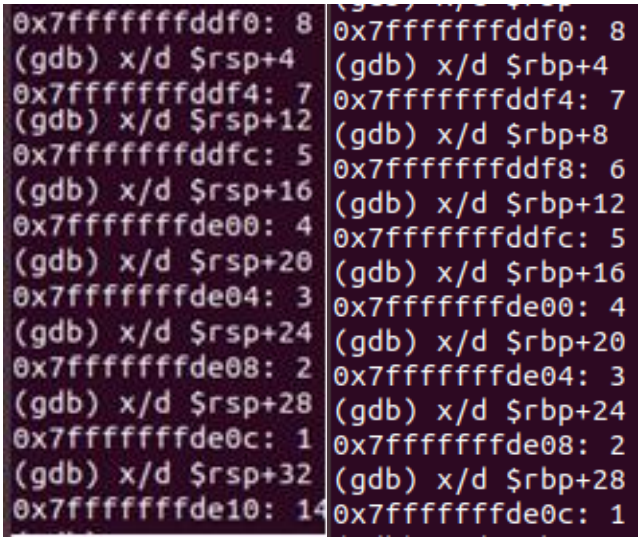
0x00005555555557b2 <+68>: mov (%rsp,%rax,4),%eax

0x00005555555557b5 <+71>: cmp %eax,0x0(%rbp)

0x00005555555557b8 <+74>: jne 0x5555555557a7 <phase_6+57>

현재 %rbx에는 1, %rax에는 7의 값이 있다. [%rsp+4*%rax]의 값은 여덟번째 input인 1이다. 그 값을 %eax로 이동

시킨다. 현재 [%rbp]에는 첫번째 input인 8의 값이 있다.



+74에 의해 첫번째 input과 여덟번째 input이 같지 않다면 +57로 jump하고 같으면 explode한다.

0x00005555555557a7 <+57>: add $0x1,%ebx

0x00005555555557aa <+60>: cmp $0x7,%ebx

0x00005555555557ad <+63>: jg 0x5555555557c1 <phase_6+83>

0x00005555555557af <+65>: movslq %ebx,%rax

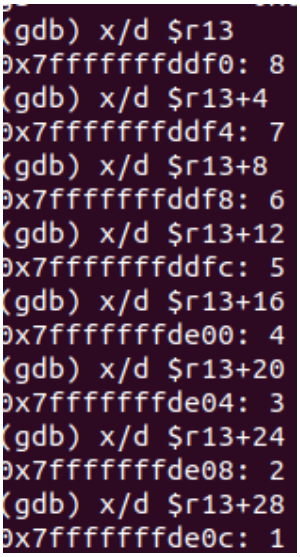
0x00005555555557b2 <+68>: mov (%rsp,%rax,4),%eax

0x00005555555557b5 <+71>: cmp %eax,0x0(%rbp)

0x00005555555557b8 <+74>: jne 0x5555555557a7 <phase_6+57>]

현재 위의 과정을 한번 거쳤기 때문에 %rbx 레지스터값은 1이다. 그리고 한번 진행될 때마다, %rax의 값이 1 늘어나니, [%rsp+4*%rax]의 값은 계속 하나 앞의 input을 가리키게 된다.

여덟번의 과정을 거치게 되면, 즉, 첫번째 input과 같은 다른 input이 없음을 확인하는 과정이며, 이 과정이 끝나면, +63줄에서 +83줄로 jump하게 된다.



0x00005555555557c1 <+83>: add $0x4,%r13

0x00005555555557c5 <+87>: mov %r13,%rbp

0x00005555555557c8 <+90>: mov 0x0(%r13),%eax

0x00005555555557cc <+94>: sub $0x1,%eax

0x00005555555557cf <+97>: cmp $0x7,%eax

0x00005555555557d2 <+100>: ja 0x5555555557a0 <phase_6+50>

우측의 레지스터값에 따라, 이 과정은 다음 input을 %eax에 대입한 후, 1을 빼고 그 값이 7보다 크면 +50으로 이동해 explode한다. 현재 모든 input input이 8 이하이므로, +100에서 jump하지 않고 +102로 진행한다.

0x00005555555557d4 <+102>: add $0x1,%r14d

0x00005555555557d8 <+106>: cmp $0x8,%r14d

0x00005555555557dc <+110>: je 0x5555555557e3 <phase_6+117>

0x00005555555557de <+112>: mov %r14d,%ebx

0x00005555555557e1 <+115>: jmp 0x5555555557af <phase_6+65>

현재 %r14는 1이므로, +102~105 과정을 7번 더 거친 후+110의 조건을 만족해 +117로 이동했다.

0x00005555555557e3 <+117>: lea 0x20(%r12),%rcx

0x00005555555557e8 <+122>: mov $0x9,%edx

0x00005555555557ed <+127>: mov %edx,%eax

0x00005555555557ef <+129>: sub (%r12),%eax

0x00005555555557f3 <+133>: mov %eax,(%r12)

0x00005555555557f7 <+137>: add $0x4,%r12

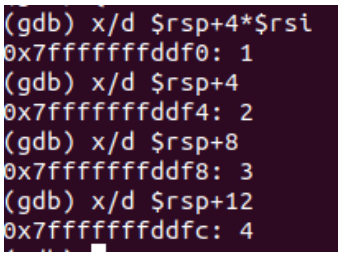
0x00005555555557fb <+141>: cmp %rcx,%r12

0x00005555555557fe <+144>: jne 0x5555555557ed <phase_6+127>

위의 과정을 설명하면, %edx에 9가 저장되고, 그 값을 %eax로 이동한다. 그리고 9에서 [%r12](여덟번째 input)을 뺀 후, 그 값을 다시 [%r12]에 저장한다. 그리고 %r12에 4를 더하면 일곱 번째 input의 주소를 가리키는 값을 갖게 된다. 그리고 %r12와 $rcx를 비교해, 같지 않으면 다시 +127로 돌아가고 같으면 +146으로 진행한다. 즉, 여덟 개의 input들의 9와의 차이 만큼으로 재저장된다. 즉 input은 이제 1 2 3 4 5 6 7 8이다. 그 후 +146으로 진행한다.

0x0000555555555800 <+146>: mov $0x0,%esi

0x0000555555555805 <+151>: jmp 0x555555555821 <phase_6+179>



%esi에 0을 저장 후 +179로 jump한다.

0x0000555555555821 <+179>: mov (%rsp,%rsi,4),%ecx

0x0000555555555824 <+182>: mov $0x1,%eax

0x0000555555555829 <+187>: lea 0x202b00(%rip),%rdx # 0x555555758330 <node1>

0x0000555555555830 <+194>: cmp $0x1,%ecx

0x0000555555555833 <+197>: jg 0x555555555807 <phase_6+153>

0x0000555555555835 <+199>: jmp 0x555555555812 <phase_6+164>

첫번째 수인 1을 %ecx로 옮긴 후, %eax에 1을 옮긴다. 그리고 [%rip+0x202b00] 값을 %rdx에 저장한다. 이는 node1이 된다. 그리고 %ecx를 1과 비교해 더 크면 +153으로 돌아간다. 현재 %ecx는 1이므로 +199로 진행해

+164로 jump한다.

0x0000555555555812 <+164>: mov %rdx,0x20(%rsp,%rsi,8)

0x0000555555555817 <+169>: add $0x1,%rsi

0x000055555555581b <+173>: cmp $0x8,%rsi

0x000055555555581f <+177>: je 0x555555555837 <phase_6+201>

0x0000555555555821 <+179>: mov (%rsp,%rsi,4),%ecx

0x0000555555555824 <+182>: mov $0x1,%eax

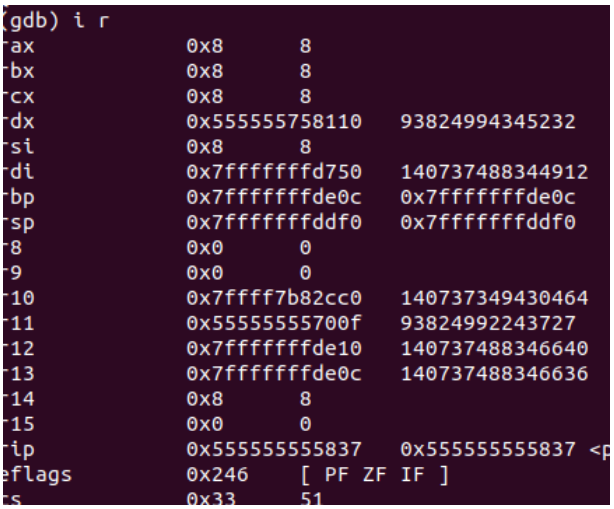
0x0000555555555829 <+187>: lea 0x202b00(%rip),%rdx # 0x555555758330 <node1>

0x0000555555555830 <+194>: cmp $0x1,%ecx

0x0000555555555833 <+197>: jg 0x555555555807 <phase_6+153>

0x0000555555555835 <+199>: jmp 0x555555555812 <phase_6+164>

위의 과정에 따라 %rsi에 1이 더해져 1이 되고, +177의 조건을 만족하지 못하므로 +179로 진행한다. %ecx에 [%rsp+4](두 번째 숫자)를 옮기고, %eax에 1을 저장하고, 그리고 다시 [%rip+0x202b00]을 %rdx에 저장한후, %ecx와 1을 비교하면, 이제 %ecx가 2이므로, +197의 조건을 만족해 +153으로 돌아간다. 그리고 다시 +153부터 이 과정들을 반복한다. 그러다가 +1177에서 %rsi의 값이 8이 될 때, 즉 이 과정이 7번 더 반복된 후, +201로 jump 한다. 그리고 이 때의 레지스터값은 아래와 같다.



0x0000555555555837 <+201>: mov 0x20(%rsp),%rbx

0x000055555555583c <+206>: lea 0x20(%rsp),%rax

0x0000555555555841 <+211>: lea 0x58(%rsp),%rsi

0x0000555555555846 <+216>: mov %rbx,%rcx

0x0000555555555849 <+219>: mov 0x8(%rax),%rdx

0x000055555555584d <+223>: mov %rdx,0x8(%rcx)

0x0000555555555851 <+227>: add $0x8,%rax

0x0000555555555855 <+231>: mov %rdx,%rcx

0x0000555555555858 <+234>: cmp %rax,%rsi

0x000055555555585b <+237>: jne 0x555555555849 <phase_6+219>

현재 %rax 레지스터값은 0x7fffffffde18, %rsi 레지스터 값은 0x7fffffffde48이다. 이 루프를 돌며, +227에 의해, %rax에 8을 더해가므로, 이 과정이 6번 진행되면, +237에서 조건을 만족시키지 않아 탈출할 수 있다.

```
(gdb) x/d $rdx+8
0x555555758118 <node8+8>:       0
```

0x000055555555585d <+239>: movq $0x0,0x8(%rdx)

0x0000555555555865 <+247>: mov $0x7,%ebp

0x000055555555586a <+252>: jmp 0x555555555875 <phase_6+263>

[%rdx+8](<node8+8>, 0) 에 0을 옮기고, %ebp에 7을 옮긴다. 그 후 +263으로 jump한다.

0x0000555555555875 <+263>: mov 0x8(%rbx),%rax

0x0000555555555879 <+267>: mov (%rax),%eax

0x000055555555587b <+269>: cmp %eax,(%rbx)

0x000055555555587d <+271>: jge 0x55555555586c <phase_6+254>

0x000055555555587f <+273>: callq 0x555555555c89 <explode_bomb>

%eax에는 [%rax+8]의 값이 들어가며, [%rbx]의 값이 이 값보다 크거나 같으면 +254로 돌아가고, 작으면 explode한다. 현재 %rax의 값은 227, [%rbx]의 값은 938이다. 즉 더 크므로 +254로 돌아간다.

```
(gdb) x/d $rbx-16
0x555555758330 <node1>: 938
(gdb) x/d $rbx
0x555555758340 <node2>: 227
(gdb) x/d $rbx+16
0x555555758350 <node3>: 504
(gdb) x/d $rbx+32
0x555555758360 <node4>: 891
(gdb) x/d $rbx+48
0x555555758370 <node5>: 447
(gdb) x/d $rbx+64
0x555555758380 <node6>: 769
(gdb) x/d $rbx+80
0x555555758390 <node7>: 121
(gdb) x/d $rbx+96
```

0x000055555555586c <+254>: mov 0x8(%rbx),%rbx

0x0000555555555870 <+258>: sub $0x1,%ebp

0x0000555555555873 <+261>: je 0x555555555886 <phase_6+280>

0x0000555555555875 <+263>: mov 0x8(%rbx),%rax

0x0000555555555879 <+267>: mov (%rax),%eax

0x000055555555587b <+269>: cmp %eax,(%rbx)

0x000055555555587d <+271>: jge 0x55555555586c <phase_6+254>

0x000055555555587f <+273>: callq 0x555555555c89 <explode_bomb>

즉, 이 루프를 계속 반복하다, +261에서 조건을 만족하게 되면 +280으로 jump한다.

현재 %rbp 레지스터의 값은 7이다. 즉 +258줄을 7번 더 거치면 +261의 조건을 만족하게 된다.

그리고 현재 node들의 값은 우측상단의 값과 같고 <node8> = 795이다. 그리고 비교 순서는

7의 input 순서의 노드 값 < 8의 input순서의 노드 값

6의 input 순서 의 노드 값 < 7의 input 순서 의 노드 값

5의 input 순서 의 노드 값 < 6의 input 순서 의 노드 값

4의 input 순서 의 노드 값 < 5의 input 순서 의 노드 값

3의 input 순서 의 노드 값 < 4의 input 순서 의 노드 값

2의 input 순서 의 노드 값 < 3의 input 순서 의 노드 값

1의 input 순서 의 노드 값 < 2의 input 순서 의 노드 값

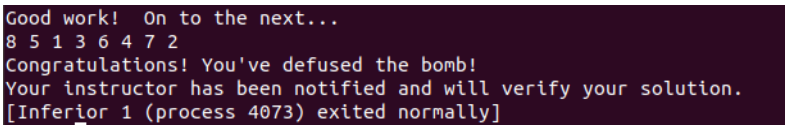
이어야 한다. 그리고 노드의 값의 크기를 비교하면,

7 < 2 < 5 < 3 < 6 < 8 < 4 < 1 이다. 즉 올바른 input은 이다. 즉 올바른 입력값은 8 5 1 3 6 4 7 2이다.

그 후는 +280부터 진행해, +308에서 ret하는 것을 알 수 있었다.

```
0x0000555555555886 <+280>:     mov      0x68(%rsp),%rax
0x000055555555588b <+285>:     xor      %fs:0x28,%rax
0x0000555555555894 <+294>:     jne      0x5555555558a3 <phase_6+309>
0x0000555555555896 <+296>:     add      $0x70,%rsp
0x000055555555589a <+300>:     pop      %rbx
0x000055555555589b <+301>:     pop      %rbp
0x000055555555589c <+302>:     pop      %r12
0x000055555555589e <+304>:     pop      %r13
0x00005555555558a0 <+306>:     pop      %r14
0x00005555555558a2 <+308>:     retq
```

Phase_6를 해체하기 위한 올바른 입력값은 8 5 1 3 6 4 7 2이다.



- Progress and unique experience of work

시스템 프로그램이라는 수업을 들으며 어셈블리에 대해 처음 접해보았다. 이번 bomb lab 과제를 하며 정말 많은 시간을 쏟은 것 같다. 하지만 이렇게 많은 시간을 쏟으며 힘들다는 생각보다는 너무나 재미있고 뿌듯하다는 감정이 많이 들었다. 평소 공부했던 C언어 등의 다른 언어들과는 많이 다른 느낌의 재미였다. 결국 내가 작성한 C코드도 이 어셈블리 코드로 변환이 된 후 실행된다는 점, 그리고 어셈블리 코드를 보며 코드로 어떤 코드였을까에 대한 생각을 하며 과제를 한다는 점이 가장 나의 흥미를 자극했다.

물론 C코드로의 완벽한 구상이 머릿속에 일어나지 못한 phase들도 존재했다. 그저 레지스터 값을 보며 추측하는 부분들도 많이 있어, 나의 부족함을 느끼기도 하였지만, 그 부족함을 채우기 위해 교안도 찾아보고, 구글 검색도 해보며 공부를 하니, 정말 배운 것도 많고 남는 것도 많은 과제였다고 생각한다.

또한, 이런 어셈블리에 대한 공부가, 나의 C언어 코딩 스킬에도 많은 도움을 준 것 같다. C코드를 작성할 때, 나도 모르게 이 코드가 어셈블리로 어떻게 변환이 되겠구나에 대한 생각을 하게 되니, 조금 더 효율적인 코드는 어떤 코드일까, 이 루프보다는 이런 식의 루프가 낫다라는 판단이 스스로 되고 있음을 느끼며 신기하고 매우 가치있는 과제였다고 생각한다.

앞으로 나의 학부 공부에서 어셈블리가 얼마나 많은 부분을 차지할지는 모르겠지만, 어셈블리에 대한 공부는 시스템을 이해하는 데에 있어서 필수적이라는 점을 알게 되었고, 추가적인 공부를 하고 싶다는 마음이 많이 자리잡았다. 또한, 리눅스 터미널을 처음 쓰다보니, 익숙치 못해 explode가 두 번 일어났다. 점수가 1점이 깎인 부분은 많이 아쉽지만, 이 과제를 통해 리눅스 터미널, gdb에 대한 이해도나 숙련도가 높아졌다는 점에서도 아주 의미있고 특별한 경험이었다고 생각한다.