<BombLab> 202055516 김명서

#Phase 1 : string comparison

```
Dump of assembler code for function phase_1:
   0x0000000000001254 <+0>:     sub    $0x8,%rsp
   0x0000000000001258 <+4>:     lea    0x1811(%rip),%rsi        # 0x2a70
   0x000000000000125f <+11>:    callq  0x172d <strings_not_equal>
   0x0000000000001264 <+16>:    test   %eax,%eax
   0x0000000000001266 <+18>:    jne    0x126d <phase_1+25>
   0x0000000000001268 <+20>:    add    $0x8,%rsp
   0x000000000000126c <+24>:    retq
   0x000000000000126d <+25>:    callq  0x1a31 <explode_bomb>
   0x0000000000001272 <+30>:    jmp    0x1268 <phase_1+20>
```

phase_1의 전체 구조를 살펴보면 <strings_not_equal>함수에 들어가 입력값과 정답을 비교를 한 후 결과가 같지 않으면 <explode_bomb>을 실행하는 것으로 보인다. 그래서 <stings_not_equal>함수에 break를 걸고 내부를 살펴보았다.

```
Dump of assembler code for function strings_not_equal:
=> 0x000055555555572d <+0>:     push   %r12
   0x000055555555572f <+2>:     push   %rbp
   0x0000555555555730 <+3>:     push   %rbx
   0x0000555555555731 <+4>:     mov    %rdi,%rbx
   0x0000555555555734 <+7>:     mov    %rsi,%rbp
   0x0000555555555737 <+10>:    callq  0x555555555710 <string_length>
```

<+4>, <+7>을 통해 <stings_not_equal>함수는 2개의 매개변수를 사용한다는 것을 알 수 있다. 그래서 인자를 받는 두 레지스터 %rdi와 %rsi의 값을 출력하였다.

```
(gdb) x/s $rdi
0x5555557586c0 <input_strings>: "test"
(gdb) x/s $rsi
0x555555556a70: "You can Russia from land here in Alaska."
```

출력 결과, %rdi에는 내가 입력한 문자가, %rsi에는 정답 문자가 각각 들어 있었다.

#Phase 2 : loops

```
Dump of assembler code for function phase_2:
=> 0x0000555555555274 <+0>:     push   %rbp
   0x0000555555555275 <+1>:     push   %rbx
   0x0000555555555276 <+2>:     sub    $0x28,%rsp
   0x000055555555527a <+6>:     mov    %fs:0x28,%rax
   0x0000555555555283 <+15>:    mov    %rax,0x18(%rsp)
   0x0000555555555288 <+20>:    xor    %eax,%eax
   0x000055555555528a <+22>:    mov    %rsp,%rsi
   0x000055555555528d <+25>:    callq  0x555555555a6d <read_six_numbers>
   0x0000555555555292 <+30>:    cmpl   $0x0,(%rsp)
   0x0000555555555296 <+34>:    js     0x5555555552a2 <phase_2+46>
   0x0000555555555298 <+36>:    mov    $0x1,%ebx
   0x000055555555529d <+41>:    mov    %rsp,%rbp
   0x00005555555552a0 <+44>:    jmp    0x5555555552b3 <phase_2+63>
   0x00005555555552a2 <+46>:    callq  0x555555555a31 <explode_bomb>
   0x00005555555552a7 <+51>:    jmp    0x555555555298 <phase_2+36>
   0x00005555555552a9 <+53>:    add    $0x1,%rbx
   0x00005555555552ad <+57>:    cmp    $0x6,%rbx
   0x00005555555552b1 <+61>:    je     0x5555555552c6 <phase_2+82>
   0x00005555555552b3 <+63>:    mov    %ebx,%eax
   0x00005555555552b5 <+65>:    add    -0x4(%rbp,%rbx,4),%eax
   0x00005555555552b9 <+69>:    cmp    %eax,0x0(%rbp,%rbx,4)
   0x00005555555552bd <+73>:    je     0x5555555552a9 <phase_2+53>
   0x00005555555552bf <+75>:    callq  0x555555555a31 <explode_bomb>
   0x00005555555552c4 <+80>:    jmp    0x5555555552a9 <phase_2+53>
```

Phase_2의 전반적인 흐름을 살펴보면 다음과 같다.

1.<read_six_number> 함수를 통해 6개의 숫자를 입력으로 받음을 알 수 있다.

2. <+30>, <+34>,<+46>을 통해서 처음 입력 받을 숫자가 0 이상의 숫자임을 알 수 있다.

3. <+36>, <+41>, <+44>, <+63>을 통해 %ebx와 %eax에는 0x1이, %rbp에는 %rsp와 동일한 값(주소)가 들어 있음을

알 수 있다.

4. <+65>, <+69>, <+73>을 통해서 -0x4(%rbp, %rbx, 4)의 값 + %eax 와 (%rbp, %rbx, 4)의 값이 동일해야 한다.

5. <+73>, <+53>, <+57>을 통해 %rbx는 매 루프마다 1씩 증가하는 값을 가지며 %rbx의 값이 6이 될 때 루프를 빠져나온다.

따라서 위의 조건들을 정리하면 다음과 같다.

x	<- (%rbp)
x+1	<- (%rbp + 4)
(x+1)+2	<- (%rbp + 8)
((x+1)+2)+3	<- (%rbp + 12)
(((x+1)+2)+3)+4	<- (%rbp + 16)
((((x+1)+2)+3)+4)+5	<- (%rbp + 20)

X는 0이상의 수 이며, x=0 일 때 0, 1, 3, 6, 10, 15 를 정답으로 가진다.

#Phase 3 : conditionals/switches

```
(gdb) x/s 0x555555556d8d
0x555555556d8d: "%d %d"
```

먼저 위의 주소를 출력해보면 입력이 정수 2개 인 것을 알 수 있다. 또한 %rsp을 출력하면 %rsp의 값이 첫번째로 입

```
Dump of assembler code for function phase_3:
=> 0x00005555555552e2 <+0>:     sub    $0x18,%rsp
   0x00005555555552e6 <+4>:     mov    %fs:0x28,%rax
   0x00005555555552ef <+13>:    mov    %rax,0x8(%rsp)
   0x00005555555552f4 <+18>:    xor    %eax,%eax
   0x00005555555552f6 <+20>:    lea    0x4(%rsp),%rcx
   0x00005555555552fb <+25>:    mov    %rsp,%rdx
   0x00005555555552fe <+28>:    lea    0x1a88(%rip),%rsi        # 0x555555556d8d
   0x0000555555555305 <+35>:    callq  0x555555554f30 <__isoc99_sscanf@plt>
   0x000055555555530a <+40>:    cmp    $0x1,%eax
   0x000055555555530d <+43>:    jle    0x55555555532c <phase_3+74>
   0x000055555555530f <+45>:    cmpl   $0x7,(%rsp)
   0x0000555555555313 <+49>:    ja     0x5555555553b2 <phase_3+208>
   0x0000555555555319 <+55>:    mov    (%rsp),%eax
   0x000055555555531c <+58>:    lea    0x17bd(%rip),%rdx        # 0x555555556ae0
   0x0000555555555323 <+65>:    movslq (%rdx,%rax,4),%rax
   0x0000555555555327 <+69>:    add    %rdx,%rax
   0x000055555555532a <+72>:    jmpq   *%rax
   0x000055555555532c <+74>:    callq  0x555555555a31 <explode_bomb>
   0x0000555555555331 <+79>:    jmp    0x55555555530f <phase_3+45>
   0x0000555555555333 <+81>:    mov    $0x3bc,%eax
   0x0000555555555338 <+86>:    jmp    0x55555555533f <phase_3+93>
   0x000055555555533a <+88>:    mov    $0x0,%eax
```

력한 값과 동일함을 알 수 있다.

그리고 <+55>, <+65>, <+69>를 통해서 %rsp의 값에 따라 jump하는 위치가 달라진다. 만약 첫번째 입력 값이 1이면 <+88>의 위치로 이동한다.

```
0x000000000000133a <+88>:    mov    $0x0,%eax
0x000000000000133f <+93>:    sub    $0x2ad,%eax
0x0000000000001344 <+98>:    add    $0x2c9,%eax
0x0000000000001349 <+103>:   sub    $0xbf,%eax
0x000000000000134e <+108>:   add    $0xbf,%eax
0x0000000000001353 <+113>:   sub    $0xbf,%eax
0x0000000000001358 <+118>:   add    $0xbf,%eax
0x000000000000135d <+123>:   sub    $0xbf,%eax
```

첫번째 입력 값이 1일 때 <+88>부터 <+123>까지 계산을 하면 %eax의 값은 -163이다

```
0x0000000000001362 <+128>:   cmpl   $0x5,(%rsp)
0x0000000000001366 <+132>:   jg     0x136e <phase_3+140>
0x0000000000001368 <+134>:   cmp    %eax,0x4(%rsp)
0x000000000000136c <+138>:   je     0x1373 <phase_3+145>
0x000000000000136e <+140>:   callq  0x1a31 <explode_bomb>
0x0000000000001373 <+145>:   mov    0x8(%rsp),%rax
```

<+134>, <+138>을 통해서 0x4(%rsp) 즉, 두번째 입력 값이 %eax의 값과 동일해야 함을 알 수 있다. 따라서 phase_3의 입력값으로는 1, -163 등이 올 수 있다.

#Phase 4 : recursive calls and the stack discipline

```
(gdb) x/s 0x555555556d8d
0x555555556d8d: "%d %d"
```

위의 주소를 출력해보면 입력으로 두개의 정수를 받음을 알 수 있다.

```
Dump of assembler code for function phase_4:
=> 0x00005555555553fc <+0>:     sub    $0x18,%rsp
   0x0000555555555400 <+4>:     mov    %fs:0x28,%rax
   0x0000555555555409 <+13>:    mov    %rax,0x8(%rsp)
   0x000055555555540e <+18>:    xor    %eax,%eax
   0x0000555555555410 <+20>:    mov    %rsp,%rcx
   0x0000555555555413 <+23>:    lea    0x4(%rsp),%rdx
   0x0000555555555418 <+28>:    lea    0x196e(%rip),%rsi        # 0x555555556d8d
   0x000055555555541f <+35>:    callq  0x555555554f30 <__isoc99_sscanf@plt>
   0x0000555555555424 <+40>:    cmp    $0x2,%eax
   0x0000555555555427 <+43>:    jne    0x555555555434 <phase_4+56>
   0x0000555555555429 <+45>:    mov    (%rsp),%eax
   0x000055555555542c <+48>:    sub    $0x2,%eax
   0x000055555555542f <+51>:    cmp    $0x2,%eax
   0x0000555555555432 <+54>:    jbe    0x555555555439 <phase_4+61>
   0x0000555555555434 <+56>:    callq  0x555555555a31 <explode_bomb>
   0x0000555555555439 <+61>:    mov    (%rsp),%esi
   0x000055555555543c <+64>:    mov    $0x9,%edi
   0x0000555555555441 <+69>:    callq  0x5555555553c3 <func4>
   0x0000555555555446 <+74>:    cmp    %eax,0x4(%rsp)
   0x000055555555544a <+78>:    je     0x555555555451 <phase_4+85>
   0x000055555555544c <+80>:    callq  0x555555555a31 <explode_bomb>
```

또한 Phase_4의 전반적인 흐름을 살펴보면 %rsp에는 두번째로 입력한 값이 저장되고 이는 %eax에 저장된다. <+48>, <+51>, <+54>를 통해 %rsp의 값이 4 이하여야 함을 알 수 있다. 또한 <+69>, <+74>, <+78>을 통해 <func4>에서 계산 한 결과가 %eax에 저장되고 이 값이 0x4(%rsp) 즉, 첫번째로 입력한 값과 동일해야 함을 알 수 있다.

```
Breakpoint 8, 0x000055555555544a in phase_4 ()
(gdb) info registers
rax            0xb0     176
```

입력 값이 1, 2였을 때 <+78>에서 break를 걸고 %eax의 값을 살펴보면 176이다. 따라서 176, 2 등이 정답 임을 알 수 있다.

#Phase 5 : pointers

```
   0x000055555555546f <+4>:     callq  0x555555555710 <string_length>
   0x0000555555555474 <+9>:     cmp    $0x6,%eax
   0x0000555555555477 <+12>:    jne    0x5555555554aa <phase_5+63>
   0x0000555555555479 <+14>:    mov    %rbx,%rax
   0x000055555555547c <+17>:    lea    0x6(%rbx),%rdi
   0x0000555555555480 <+21>:    mov    $0x0,%ecx
   0x0000555555555485 <+26>:    lea    0x1674(%rip),%rsi        # 0x555555556b00
<array.3415>
   0x000055555555548c <+33>:    movzbl (%rax),%edx
   0x000055555555548f <+36>:    and    $0xf,%edx
   0x0000555555555492 <+39>:    add    (%rsi,%rdx,4),%ecx
   0x0000555555555495 <+42>:    add    $0x1,%rax
   0x0000555555555499 <+46>:    cmp    %rdi,%rax
   0x000055555555549c <+49>:    jne    0x55555555548c <phase_5+33>
```

Phase_5의 흐름을 살펴보면 <+4>, <+9>,<+12>를 통해서 입력 받는 문자의 길이가 6이여야 함을 알 수 있다. 그리고 <+42>, <+46>, <+49>를 통해 입력된 문자를 하나씩 대응하는 숫자로 변경하여 더함을 알 수 있다.

```
   0x000055555555549c <+49>:    jne    0x55555555548c <phase_5+33>
   0x000055555555549e <+51>:    cmp    $0x2a,%ecx
   0x00005555555554a1 <+54>:    je     0x5555555554a8 <phase_5+61>
   0x00005555555554a3 <+56>:    callq  0x555555555a31 <explode_bomb>
```

그리고 <+51>에 의해 최종적으로 더한 값이 42가 되어야 한다. 숫자들의 규칙을 잘 이해가 되지 않아서 <+42>에 break를 걸고 a부터 하나씩 대응 값을 출력해보았다. i의 값이 7에 대응됨을 알게 되었고 입력으로 iiiiii를 입력하면 정답 임을 알 수 있다.

#phase 6 : linked lists/pointers/structs

```
0x00005555555554b9 <+8>:     sub    $0x60,%rsp
0x00005555555554bd <+12>:    mov    %fs:0x28,%rax
0x00005555555554c6 <+21>:    mov    %rax,0x58(%rsp)
0x00005555555554cb <+26>:    xor    %eax,%eax
0x00005555555554cd <+28>:    mov    %rsp,%r13
0x00005555555554d0 <+31>:    mov    %r13,%rsi
0x00005555555554d3 <+34>:    callq  0x555555555a6d <read_six_numbers>
0x00005555555554d8 <+39>:    mov    %r13,%r12
```

<+34>를 통해서 입력이 6개의 숫자임을 알 수 있다.

```
0x000055555555550b <+90>:    mov    0x0(%r13),%eax
0x000055555555550f <+94>:    sub    $0x1,%eax
0x0000555555555512 <+97>:    cmp    $0x5,%eax
0x0000555555555515 <+100>:   ja     0x5555555554e3 <phase_6+50>
0x0000555555555517 <+102>:   add    $0x1,%r14d
0x000055555555551b <+106>:   cmp    $0x6,%r14d
0x000055555555551f <+110>:   je     0x555555555526 <phase_6+117>
```

<+90>, <+94>, <+97>을 통해 모든 입력이 6보다 같거나 작으며, <+71>, <+74>에 의해 각각의 값이 전부 달라야 함을 알 수 있다.

```
0x0000555555555562 <+177>:   je     0x55555555557a <phase_6+201>
0x0000555555555564 <+179>:   mov    (%rsp,%rsi,4),%ecx
0x0000555555555567 <+182>:   mov    $0x1,%eax
0x000055555555556c <+187>:   lea    0x202cbd(%rip),%rdx        # 0x5555557582
<node1>
0x0000555555555573 <+194>:   cmp    $0x1,%ecx
```

<+173>, <+177>, <+179>을 통해 <%rsi 값을 0에서 5까지 1씩 증가하며 loop를 돌며 일치하는 입력 값에 대해 대응 주소 값을 찾는다.

```
0x00005555555555ac <+251>:   movq   $0x0,0x8(%rax)
0x00005555555555b4 <+259>:   mov    $0x5,%ebp
0x00005555555555b9 <+264>:   jmp    0x5555555555c4 <phase_6+275>
0x00005555555555bb <+266>:   mov    0x8(%rbx),%rbx
0x00005555555555bf <+270>:   sub    $0x1,%ebp
0x00005555555555c2 <+273>:   je     0x5555555555d5 <phase_6+292>
0x00005555555555c4 <+275>:   mov    0x8(%rbx),%rax
0x00005555555555c8 <+279>:   mov    (%rax),%eax
0x00005555555555ca <+281>:   cmp    %eax,(%rbx)
0x00005555555555cc <+283>:   jge    0x5555555555bb <phase_6+266>
```

<+275>, <+279>, <+281>등에 의하면 %rbx에는 위에서 구한 주소 중 현재 위치의 입력 값에 해당하는 주소를, %eax에는 다음 위치의 입력 값에 해당하는 주소를 넣는다. 그리고 두 주소가 가진 값들을 비교해 (%rbx)>(%eax)일 때 정답이 된다.

```
(gdb) x/d $rdx
0x555555758230 <node1>: 214
```

```
(gdb) x/d $rdx
0x555555758240 <node2>: 356
```

```
(gdb) x/d $rdx
0x555555758250 <node3>: 275
```

```
(gdb) x/d $rdx
0x555555758260 <node4>: 977
```

```
(gdb) x/d $rdx
0x555555758270 <node5>: 392
```

```
(gdb) x/d $rdx
0x555555758110 <node6>: 163
```

따라서 각 node에 들어 있는 값을 찾아보면 차례대로 214, 356, 275, 977, 392, 163이고 이를 큰 숫자대로 나열하면 node4 > node5 > node2 > node3 > node1 > node6이다. 따라서 node들에 대응하는 입력 값을 나열한 3-2-5-4-6-1이 정답이 된다.

#secret phase

disas phase_defused를 통해서 phase_defused 함수 내부를 살펴보면 seceret_phase가 있는 것을 알 수 있다.

```
   0x0000555555555c37 <+69>:    lea    0x8(%rsp),%rdx
   0x0000555555555c3c <+74>:    lea    0x10(%rsp),%r8
   0x0000555555555c41 <+79>:    lea    0x118f(%rip),%rsi        # 0x555555556dd7
   0x0000555555555c48 <+86>:    lea    0x202b61(%rip),%rdi        # 0x5555557587
```

secret_phase 내부를 살펴보면 %rdi와 %rsi가 있는 것으로 보아 두개의 인자의 값을 받음을 생각할 수 있는데 출력해보면 phase_4에서 입력한 값과 입력 인자로 %d, %d, %s로 뒤에 한 개를 추가로 입력 받음을 알 수 있다.

```
(gdb) x/s 0x5555557587b0
0x5555557587b0 <input_strings+240>:     "176 2"
(gdb) x/s 0x555555556dd7
0x555555556dd7: "%d %d %s"
```

```
0x0000555555555c7d <+139>:   lea    0x115c(%rip),%rsi        # 0x555555556de0
0x0000555555555c84 <+146>:   callq  0x55555555572d <strings_not_equal>
```

또한 뒤에 받은 입력 값과 정답을 비교하는 함수가 있는데 정답을 출력하면 다음과 같다

```
(gdb) x/s 0x555555556de0
0x555555556de0: "DrEvil"
```

```
0x0000555555555651 <+27>:    lea    -0x1(%rax),%eax
0x0000555555555654 <+30>:    cmp    $0x3e8,%eax
0x0000555555555659 <+35>:    ja     0x555555555686 <secret_phase+80>
```

Secret_phase 내부를 살펴보면 <+27>, <+30>에 의해 입력 값이 1001보다 같거나 작은 수여야 함을 알 수 있다.

```
0x0000555555555664 <+46>:    callq  0x5555555555f7 <fun7>
0x0000555555555669 <+51>:    cmp    $0x7,%eax
0x000055555555566c <+54>:    je     0x555555555673 <secret_phase+61>
0x000055555555566e <+56>:    callq  0x555555555a31 <explode_bomb>
```

```
   0x0000555555555602 <+11>:    cmp    %esi,%edx
   0x0000555555555604 <+13>:    jg     0x555555555614 <fun7+29>
   0x0000555555555606 <+15>:    mov    $0x0,%eax
   0x000055555555560b <+20>:    cmp    %esi,%edx
   0x000055555555560d <+22>:    jne    0x555555555621 <fun7+42>
   0x000055555555560f <+24>:    add    $0x8,%rsp
   0x0000555555555613 <+28>:    retq
   0x0000555555555614 <+29>:    mov    0x8(%rdi),%rdi
   0x0000555555555618 <+33>:    callq  0x5555555555f7 <fun7>
   0x000055555555561d <+38>:    add    %eax,%eax
   0x000055555555561f <+40>:    jmp    0x55555555560f <fun7+24>
   0x0000555555555621 <+42>:    mov    0x10(%rdi),%rdi
   0x0000555555555625 <+46>:    callq  0x5555555555f7 <fun7>
   0x000055555555562a <+51>:    lea    0x1(%rax,%rax,1),%eax
   0x000055555555562e <+55>:    jmp    0x55555555560f <fun7+24>
```

그리고 함수 fun7을 통해 나온 결과가 7일 때가 정답이 됨을 알 수 있다.

그래서 Fun7의 내부를 살펴보면 <+11>, <+13>, <+38>, <+51>을 통해 %esi와 %edx의 값을 비교하는 데 %edx가 크면 %eax를 두배 하고, 작으면 %eax를 두배 하고 1을 더해주는 것을 알 수 있다. 이 때 처음 %edx값은 <n1>의 값이 되고 fun7이 재귀 되기 전에 다른 값으로 대체 된다. 이 값을 변화를 출력하면 다음과 같다.

```
0x555555758150 <n1>:     36       93824994345328
0x555555758160 <n1+16>: 93824994345360   0
0x555555758170 <n21>:    8        93824994345456
0x555555758180 <n21+16>:          93824994345392   0
0x555555758190 <n22>:    50       93824994345424
0x5555557581a0 <n22+16>:          93824994345488   0
0x5555557581b0 <n32>:    22       93824994345136
0x5555557581c0 <n32+16>:          93824994345072   0
0x5555557581d0 <n33>:    45       93824994344976
0x5555557581e0 <n33+16>:          93824994345168   0
0x5555557581f0 <n31>:    6        93824994345008
0x555555758200 <n31+16>:          93824994345104   0
0x555555758210 <n34>:    107      93824994345040
```

```
(gdb) x/d $rdi
0x5555557580f0 <n48>:    1001
```

결과가 7이여야하므로 입력 값은 36보다 크면서 50보다 크고, 107보다 크면서 1001과 같은 값 즉, 1001이여야지 정답임을 알 수 있다.