CSED211: BombLab Report

20240832 / 채승현

0. x86-64에서 함수의 calling convention

- 함수의 인자: rdi, rsi, rdx, rcx, r8, r9, stack
- 함수의 반환값: rax

1. phase_1

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x0000000000400ef0 <+0>:     sub    $0x8,%rsp
   0x0000000000400ef4 <+4>:     mov    $0x402560,%esi
   0x0000000000400ef9 <+9>:     callq  0x4013ae <strings_not_equal>
   0x0000000000400efe <+14>:    test   %eax,%eax
   0x0000000000400f00 <+16>:    je     0x400f07 <phase_1+23>
   0x0000000000400f02 <+18>:    callq  0x401614 <explode_bomb>
   0x0000000000400f07 <+23>:    add    $0x8,%rsp
   0x0000000000400f0b <+27>:    retq
```

```
(gdb) x/s 0x402560
0x402560:       "You can Russia from land here in Alaska."
```

먼저 esi에 주소 0x402560가 가리키는 값을 담는다. 이후 <strings_not_equal> 함수를 실행하고, eax의 값이 0이 아니라면 <explode_bomb>이 호출되지 않고 phase_1을 통과할 수 있다.

여기서 0x402560에 담긴 값이 string일 것이라 예측하였고 x/s를 통해 확인해보니 위와 같은 “You can Russia from land here in Alaska.”이 확인되었다. 그리고 이는 <strings_not_equal>의 esi 인자로 들어가게 된다.

```
(gdb) disas strings_not_equal
Dump of assembler code for function strings_not_equal:
   0x00000000004013ae <+0>:     push   %r12
   0x00000000004013b0 <+2>:     push   %rbp
   0x00000000004013b1 <+3>:     push   %rbx
   0x00000000004013b2 <+4>:     mov    %rdi,%rbx
   0x00000000004013b5 <+7>:     mov    %rsi,%rbp
   0x00000000004013b8 <+10>:    callq  0x401391 <string_length>
   0x00000000004013bd <+15>:    mov    %eax,%r12d
   0x00000000004013c0 <+18>:    mov    %rbp,%rdi
   0x00000000004013c3 <+21>:    callq  0x401391 <string_length>
   0x00000000004013c8 <+26>:    mov    $0x1,%edx
   0x00000000004013cd <+31>:    cmp    %eax,%r12d
   0x00000000004013d0 <+34>:    jne    0x401410 <strings_not_equal+98>
   0x00000000004013d2 <+36>:    movzbl (%rbx),%eax
   0x00000000004013d5 <+39>:    test   %al,%al
   0x00000000004013d7 <+41>:    je     0x4013fd <strings_not_equal+79>
   0x00000000004013d9 <+43>:    cmp    0x0(%rbp),%al
   0x00000000004013dc <+46>:    je     0x4013e7 <strings_not_equal+57>
   0x00000000004013de <+48>:    xchg   %ax,%ax
   0x00000000004013e0 <+50>:    jmp    0x401404 <strings_not_equal+86>
   0x00000000004013e2 <+52>:    cmp    0x0(%rbp),%al
   0x00000000004013e5 <+55>:    jne    0x40140b <strings_not_equal+93>
   0x00000000004013e7 <+57>:    add    $0x1,%rbx
   0x00000000004013eb <+61>:    add    $0x1,%rbp
   0x00000000004013ef <+65>:    movzbl (%rbx),%eax
   0x00000000004013f2 <+68>:    test   %al,%al
   0x00000000004013f4 <+70>:    jne    0x4013e2 <strings_not_equal+52>
   0x00000000004013f6 <+72>:    mov    $0x0,%edx
   0x00000000004013fb <+77>:    jmp    0x401410 <strings_not_equal+98>
   0x00000000004013fd <+79>:    mov    $0x0,%edx
   0x0000000000401402 <+84>:    jmp    0x401410 <strings_not_equal+98>
   0x0000000000401404 <+86>:    mov    $0x1,%edx
   0x0000000000401409 <+91>:    jmp    0x401410 <strings_not_equal+98>
---Type <return> to continue, or q <return> to quit---
   0x000000000040140b <+93>:    mov    $0x1,%edx
   0x0000000000401410 <+98>:    mov    %edx,%eax
   0x0000000000401412 <+100>:   pop    %rbx
   0x0000000000401413 <+101>:   pop    %rbp
   0x0000000000401414 <+102>:   pop    %r12
   0x0000000000401416 <+104>:   retq
```

```
(gdb) disas string_length
Dump of assembler code for function string_length:
   0x0000000000401391 <+0>:     cmpb   $0x0,(%rdi)
   0x0000000000401394 <+3>:     je     0x4013a8 <string_length+23>
   0x0000000000401396 <+5>:     mov    %rdi,%rdx
   0x0000000000401399 <+8>:     add    $0x1,%rdx
   0x000000000040139d <+12>:    mov    %edx,%eax
   0x000000000040139f <+14>:    sub    %edi,%eax
   0x00000000004013a1 <+16>:    cmpb   $0x0,(%rdx)
   0x00000000004013a4 <+19>:    jne    0x401399 <string_length+8>
   0x00000000004013a6 <+21>:    repz retq
   0x00000000004013a8 <+23>:    mov    $0x0,%eax
   0x00000000004013ad <+28>:    retq
```

<strings_not_equal>을 확인해보자. 여기서 rdi는 우리가 입력한 문자열, rsi(esi)는 “You can Russia from land here in Alaska.”이다. +10에서 우리가 입력한 문자열의 길이, +21에서는 rsi에 담긴 문자열의 길이를 <string_length>를 통해 가져온다고 예상해볼 수 있다. 이후 +31에서 두 문자열의 길이를 cmp를 통해 비교한다. +34에서는 아까 비교한 두 값이 다르다면 +98로 jump하여 이 종료한다. 그렇지 않으면 루프를 돌게 된다. 이때 rdi -> rbx -> eax, rsi -> rbp로 문자열이 이동하며 eax(al 사용)와 rbp의 한 바이트가 같은지 확인(cmp 0x0(%rbp), %al)하게 된다. 같다면 각 string pointer의 값을 1씩 증가시켜 루프를 돌고, 다르면 종료한다. 루프를 모두 돌고 종료하게 된다면 eax에 1, 그렇지 않으면 0이 들어간다. 즉, 1 byte씩 string을 비교해서 같은 지 판단하는 함수다.

<string_length>을 확인해보자. rdi의 1 byte 값을 읽어와 0x0(‘\0’, null)과 비교하여 같으면 +23으로 jump하여 eax의 값이 0인 상태로 함수가 종료된다. 그렇지 않다면 탐색용 pointer를 rdx에 복사하여 1씩 증가시키며 루프를 돈다. 이때 현재 pointer의 위치를 eax에 복사해 놓고, eax에 시작 pointer 주소를 빼서 저장하게 된다. 이는 탐색한 길이에 해당한다. 이후 rdx의 1 byte씩 값을 읽어와 0x0과 비교하며 이와 같다면 루프를 나와 바로 함수를 종료하고, 그렇지 않다면 계속해서 루프를 돈다. 즉, rdi의 레지스터가 가리키는 메모리의 문자열의 길이를 eax에 저장하는 함수다.

결론적으로, **“You can Russia from land here in Alaska.”**를 입력하여 phase_1를 통과할 수 있다.

2. phase_2

```
(gdb) disas phase_2
Dump of assembler code for function phase_2:
   0x000000000400f0c <+0>:     push   %rbp
   0x000000000400f0d <+1>:     push   %rbx
   0x000000000400f0e <+2>:     sub    $0x28,%rsp
   0x000000000400f12 <+6>:     mov    %rsp,%rsi
   0x000000000400f15 <+9>:     callq  0x40164a <read_six_numbers>
   0x000000000400f1a <+14>:    cmpl   $0x0,(%rsp)
   0x000000000400f1e <+18>:    jns    0x400f44 <phase_2+56>
   0x000000000400f20 <+20>:    callq  0x401614 <explode_bomb>
   0x000000000400f25 <+25>:    jmp    0x400f44 <phase_2+56>
   0x000000000400f27 <+27>:    mov    %ebx,%eax
   0x000000000400f29 <+29>:    add    -0x4(%rbp),%eax
   0x000000000400f2c <+32>:    cmp    %eax,0x0(%rbp)
   0x000000000400f2f <+35>:    je     0x400f36 <phase_2+42>
   0x000000000400f31 <+37>:    callq  0x401614 <explode_bomb>
   0x000000000400f36 <+42>:    add    $0x1,%ebx
   0x000000000400f39 <+45>:    add    $0x4,%rbp
   0x000000000400f3d <+49>:    cmp    $0x6,%ebx
   0x000000000400f40 <+52>:    jne    0x400f27 <phase_2+27>
   0x000000000400f42 <+54>:    jmp    0x400f50 <phase_2+68>
   0x000000000400f44 <+56>:    lea    0x4(%rsp),%rbp
   0x000000000400f49 <+61>:    mov    $0x1,%ebx
   0x000000000400f4e <+66>:    jmp    0x400f27 <phase_2+27>
   0x000000000400f50 <+68>:    add    $0x28,%rsp
   0x000000000400f54 <+72>:    pop    %rbx
   0x000000000400f55 <+73>:    pop    %rbp
   0x000000000400f56 <+74>:    retq
```

```
(gdb) disas read_six_numbers
Dump of assembler code for function read_six_numbers:
   0x000000000040164a <+0>:     sub    $0x18,%rsp
   0x000000000040164e <+4>:     mov    %rsi,%rdx
   0x0000000000401651 <+7>:     lea    0x4(%rsi),%rcx
   0x0000000000401655 <+11>:    lea    0x14(%rsi),%rax
   0x0000000000401659 <+15>:    mov    %rax,0x8(%rsp)
   0x000000000040165e <+20>:    lea    0x10(%rsi),%rax
   0x0000000000401662 <+24>:    mov    %rax,(%rsp)
   0x0000000000401666 <+28>:    lea    0xc(%rsi),%r9
   0x000000000040166a <+32>:    lea    0x8(%rsi),%r8
   0x000000000040166e <+36>:    mov    $0x402881,%esi
   0x0000000000401673 <+41>:    mov    $0x0,%eax
   0x0000000000401678 <+46>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x000000000040167d <+51>:    cmp    $0x5,%eax
   0x0000000000401680 <+54>:    jg     0x401687 <read_six_numbers+61>
   0x0000000000401682 <+56>:    callq  0x401614 <explode_bomb>
   0x0000000000401687 <+61>:    add    $0x18,%rsp
   0x000000000040168b <+65>:    retq
```

```
(gdb) x/s 0x402881
0x402881:       "%d %d %d %d %d %d"
```

먼저 phase_2에서 제일 눈에 띄는 함수인 <read_six_numbers>부터 살펴보자. sscanf로 들어가는 esi 레지스터의 값을 확인해본 결과, 6개의 숫자를 인자로 받는 것을 알 수 있었다. rax는 sscanf에서 입력받은 인자의 개수가 저장되는데, 이것이 5보다 커야지 +61로 jump하여 <explode_bomb>이 실행되지 않는다. 즉, 6개의 숫자를 입력해야 한다는 것을 확인할 수 있다. 또한 이렇게 입력된 숫자를 첫 번째 인자로 전달된 주소에 쓴다.

그리고 <phase_2>를 살펴보자. <read_six_numbers>의 sscanf 이후 입력된 6개의 숫자들은 stack에 차례대로 들어가게 된다. 여기서 1번째 값 (%rsp)를 0과 비교한다. 이것이 음수가 아닐 때에만 +56으로 jump하여 <explode_bomb>이 실행되지 않는다. 이후 %rsp + 0x4를 %rbp에, %ebx에는 0x1을 루프 카운터 i로 사용한다. 그리고 +27로 jump해 루프를 시작한다. 먼저 eax에 ebx의 값을 넣는다. +29에서는 rbp가 현재 숫자의 주소를 가리키고 있기에, -0x4(%rbp)는 바로 이전 숫자를 가리키는 것이다. 여기서 6개의 숫자가 array라는 배열의 0번째부터 차례대로 저장되어 있다고 가정했을 때 eax는 i + array[i-1]이 된다. 이 값을 +32에서 0x0(%rbp) 즉, array[i]와 비교하여 같으면 +42로 jump하고, 그렇지 않다면 <explode_bomb>을 실행시킨다. 이후 ebx의 값을 1씩, rbp의 주소를 4씩(입력값이 32비트 정수이므로 1개의 숫자 만큼 건너뛰는 것) 증가시키고, ebx의 값이 6이 아니라면 다시 +27로 jump해 루프를 실행하고, 그렇지 않다면 +68로 jump해 phase_2가 정상적으로 종료된다.

결론적으로, phase_2에서는 6개의 정수를 입력해야 하는데, 첫 번째 수는 음수가 아니여야 한다. 그 다음 숫자부터는 array[i] = i + array[i-1]의 점화식을 만족해야 한다. 이때 첫 번째 수를 0으로 잡고 나머지 5개의 숫자를 구하면 phase_2의 많은 정답 중 하나인 **“0 1 3 6 10 15”**를 구할 수 있고, 이를 입력하여 phase_2를 통과할 수 있다.

3. phase_3 (사진은 다음 페이지에 첨부되어 있다)

먼저 +29에서 sscanf를 호출하는데, phase_2처럼 분석하면 정수, 문자, 정수 (총 3개)가 순서대로 입력되어야 <explode_bomb>이 실행되지 않는다. 이후 +44에서 우리 입력한 첫 번째 정수(+14에서 rdx임을 확인 가능)를 7과 비교한다. 이 값이 7보다 크다면 (+49 ja) +307로 jump하여 <explode_bomb>이 실행된다. 즉, 첫 번째 정수는 0~7의 값을 가진다. (ja는 unsigned에 대한 것) +55

에서는 첫 번째 정수값을 eax에 넣고, +59에서 jmpq *0x4025c0(,%rax,8)를 하게 된다. 이는 switch

```
(gdb) disas phase_3
Dump of assembler code for function phase_3:
   0x0000000000400f57 <+0>:     sub    $0x18,%rsp
   0x0000000000400f5b <+4>:     lea    0x8(%rsp),%r8
   0x0000000000400f60 <+9>:     lea    0x7(%rsp),%rcx
   0x0000000000400f65 <+14>:    lea    0xc(%rsp),%rdx
   0x0000000000400f6a <+19>:    mov    $0x4025b6,%esi
   0x0000000000400f6f <+24>:    mov    $0x0,%eax
   0x0000000000400f74 <+29>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x0000000000400f79 <+34>:    cmp    $0x2,%eax
   0x0000000000400f7c <+37>:    jg     0x400f83 <phase_3+44>
   0x0000000000400f7e <+39>:    callq  0x401614 <explode_bomb>
   0x0000000000400f83 <+44>:    cmpl   $0x7,0xc(%rsp)
   0x0000000000400f88 <+49>:    ja     0x40108a <phase_3+307>
   0x0000000000400f8e <+55>:    mov    0xc(%rsp),%eax
   0x0000000000400f92 <+59>:    jmpq   *0x4025c0(,%rax,8)
   0x0000000000400f99 <+66>:    mov    $0x62,%eax
   0x0000000000400f9e <+71>:    cmpl   $0x2da,0x8(%rsp)
   0x0000000000400fa6 <+79>:    je     0x401094 <phase_3+317>
   0x0000000000400fac <+85>:    callq  0x401614 <explode_bomb>
   0x0000000000400fb1 <+90>:    mov    $0x62,%eax
   0x0000000000400fb6 <+95>:    jmpq   0x401094 <phase_3+317>
   0x0000000000400fbb <+100>:   mov    $0x6c,%eax
   0x0000000000400fc0 <+105>:   cmpl   $0x2e7,0x8(%rsp)
   0x0000000000400fc8 <+113>:   je     0x401094 <phase_3+317>
   0x0000000000400fce <+119>:   callq  0x401614 <explode_bomb>
   0x0000000000400fd3 <+124>:   mov    $0x6c,%eax
   0x0000000000400fd8 <+129>:   jmpq   0x401094 <phase_3+317>
   0x0000000000400fdd <+134>:   mov    $0x75,%eax
   0x0000000000400fe2 <+139>:   cmpl   $0x1ca,0x8(%rsp)
   0x0000000000400fea <+147>:   je     0x401094 <phase_3+317>
   0x0000000000400ff0 <+153>:   callq  0x401614 <explode_bomb>
   0x0000000000400ff5 <+158>:   mov    $0x75,%eax
   0x0000000000400ffa <+163>:   jmpq   0x401094 <phase_3+317>
---Type <return> to continue, or q <return> to quit---
   0x0000000000400fff <+168>:   mov    $0x78,%eax
   0x0000000000401004 <+173>:   cmpl   $0x164,0x8(%rsp)
   0x000000000040100c <+181>:   je     0x401094 <phase_3+317>
   0x0000000000401012 <+187>:   callq  0x401614 <explode_bomb>
   0x0000000000401017 <+192>:   mov    $0x78,%eax
   0x000000000040101c <+197>:   jmp    0x401094 <phase_3+317>
   0x000000000040101e <+199>:   mov    $0x6e,%eax
   0x0000000000401023 <+204>:   cmpl   $0x3a5,0x8(%rsp)
   0x000000000040102b <+212>:   je     0x401094 <phase_3+317>
   0x000000000040102d <+214>:   callq  0x401614 <explode_bomb>
   0x0000000000401032 <+219>:   mov    $0x6e,%eax
   0x0000000000401037 <+224>:   jmp    0x401094 <phase_3+317>
   0x0000000000401039 <+226>:   mov    $0x61,%eax
   0x000000000040103e <+231>:   cmpl   $0x376,0x8(%rsp)
   0x0000000000401046 <+239>:   je     0x401094 <phase_3+317>
   0x0000000000401048 <+241>:   callq  0x401614 <explode_bomb>
   0x000000000040104d <+246>:   mov    $0x61,%eax
   0x0000000000401052 <+251>:   jmp    0x401094 <phase_3+317>
   0x0000000000401054 <+253>:   mov    $0x68,%eax
   0x0000000000401059 <+258>:   cmpl   $0x9c,0x8(%rsp)
   0x0000000000401061 <+266>:   je     0x401094 <phase_3+317>
   0x0000000000401063 <+268>:   callq  0x401614 <explode_bomb>
   0x0000000000401068 <+273>:   mov    $0x68,%eax
   0x000000000040106d <+278>:   jmp    0x401094 <phase_3+317>
   0x000000000040106f <+280>:   mov    $0x75,%eax
   0x0000000000401074 <+285>:   cmpl   $0x362,0x8(%rsp)
   0x000000000040107c <+293>:   je     0x401094 <phase_3+317>
   0x000000000040107e <+295>:   callq  0x401614 <explode_bomb>
   0x0000000000401083 <+300>:   mov    $0x75,%eax
   0x0000000000401088 <+305>:   jmp    0x401094 <phase_3+317>
   0x000000000040108a <+307>:   callq  0x401614 <explode_bomb>
   0x000000000040108f <+312>:   mov    $0x6b,%eax
   0x0000000000401094 <+317>:   cmp    0x7(%rsp),%al
---Type <return> to continue, or q <return> to quit---
   0x0000000000401098 <+321>:   je     0x40109f <phase_3+328>
   0x000000000040109a <+323>:   callq  0x401614 <explode_bomb>
   0x000000000040109f <+328>:   add    $0x18,%rsp
   0x00000000004010a3 <+332>:   retq
```

```
(gdb) x/s 0x4025b6
0x4025b6:       "%d %c %d"
```

```
(gdb) x /8xg 0x4025c0
0x4025c0:       0x0000000000400f99      0x0000000000400fbb
0x4025d0:       0x0000000000400fdd      0x0000000000400fff
0x4025e0:       0x000000000040101e      0x0000000000401039
0x4025f0:       0x0000000000401054      0x000000000040106f
```

이므로, jump table인 0x4025c0를 확인해주었다. (명령어는 lecture note에 나온 것을 사용하였고, rax가 0~7이므로 숫자는 그대로 나뒀다) 그 결과 0~7이 linear하게 jump된다는 것을 확인할 수 있었다. jump 되어진 각 부분의 구조는 숫자만 다를 뿐 로직은 동일하다. eax가 0인 경우만 보면, +66으로 jump하고, eax에 0x62를 넣는다. 이후 입력한 두 번째 정수(+4에서 r8 확인 가능)와 0x2da (10진수로 730)를 비교하여 같을 때에만 <explode_bomb>이 실행되지 않고 +317로 jump하게 된다. 이후 우리가 입력한 문자(+4에서 rcx 확인 가능)와 eax(%al이므로, 문자여서 1byte)를 비교하여 같을 때(0x62 = 'b')에만 <explode_bomb>이 실행되지 않고 정상적으로 phase_3가 종료된다. 가능한 8개의 정답 중 첫 번째 숫자가 0인 경우를 골랐고, 이때의 정답은 **"0 b 730"**이다. 이를 입력하면 phase_3를 통과할 수 있다.

4. phase_4

```
Dump of assembler code for function phase_4:
   0x00000000004010e2 <+0>:     sub    $0x18,%rsp
   0x00000000004010e6 <+4>:     lea    0x8(%rsp),%rcx
   0x00000000004010eb <+9>:     lea    0xc(%rsp),%rdx
   0x00000000004010f0 <+14>:    mov    $0x40288d,%esi
   0x00000000004010f5 <+19>:    mov    $0x0,%eax
   0x00000000004010fa <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x00000000004010ff <+29>:    cmp    $0x2,%eax
   0x0000000000401102 <+32>:    jne    0x40110b <phase_4+41>
   0x0000000000401104 <+34>:    cmpl   $0xe,0xc(%rsp)
   0x0000000000401109 <+39>:    jbe    0x401110 <phase_4+46>
   0x000000000040110b <+41>:    callq  0x401614 <explode_bomb>
   0x0000000000401110 <+46>:    mov    $0xe,%edx
   0x0000000000401115 <+51>:    mov    $0x0,%esi
   0x000000000040111a <+56>:    mov    0xc(%rsp),%edi
   0x000000000040111e <+60>:    callq  0x4010a4 <func4>
   0x0000000000401123 <+65>:    test   %eax,%eax
   0x0000000000401125 <+67>:    jne    0x40112e <phase_4+76>
   0x0000000000401127 <+69>:    cmpl   $0x0,0x8(%rsp)
   0x000000000040112c <+74>:    je     0x401133 <phase_4+81>
   0x000000000040112e <+76>:    callq  0x401614 <explode_bomb>
   0x0000000000401133 <+81>:    add    $0x18,%rsp
   0x0000000000401137 <+85>:    retq
```

```
Dump of assembler code for function func4:
   0x00000000004010a4 <+0>:     sub    $0x8,%rsp
   0x00000000004010a8 <+4>:     mov    %edx,%eax
   0x00000000004010aa <+6>:     sub    %esi,%eax
   0x00000000004010ac <+8>:     mov    %eax,%ecx
   0x00000000004010ae <+10>:    shr    $0x1f,%ecx
   0x00000000004010b1 <+13>:    add    %ecx,%eax
   0x00000000004010b3 <+15>:    sar    %eax
   0x00000000004010b5 <+17>:    lea    (%rax,%rsi,1),%ecx
   0x00000000004010b8 <+20>:    cmp    %edi,%ecx
   0x00000000004010ba <+22>:    jle    0x4010c8 <func4+36>
   0x00000000004010bc <+24>:    lea    -0x1(%rcx),%edx
   0x00000000004010bf <+27>:    callq  0x4010a4 <func4>
   0x00000000004010c4 <+32>:    add    %eax,%eax
   0x00000000004010c6 <+34>:    jmp    0x4010dd <func4+57>
   0x00000000004010c8 <+36>:    mov    $0x0,%eax
   0x00000000004010cd <+41>:    cmp    %edi,%ecx
   0x00000000004010cf <+43>:    jge    0x4010dd <func4+57>
   0x00000000004010d1 <+45>:    lea    0x1(%rcx),%esi
   0x00000000004010d4 <+48>:    callq  0x4010a4 <func4>
   0x00000000004010d9 <+53>:    lea    0x1(%rax,%rax,1),%eax
   0x00000000004010dd <+57>:    add    $0x8,%rsp
   0x00000000004010e1 <+61>:    retq
```

```
(gdb) x/s 0x40288d
0x40288d:       "%d %d"
```

phase_4를 살펴보면, 먼저 +24에서 sscanf를 호출하는데, phase_2처럼 분석하면 정수 2개가 입력되어야 <explode_bomb>이 실행되지 않는다. 이후 +34에서 입력한 첫 번째 정수(+9에서 rdx 확인 가능)와 0xe (10진수로 14)를 비교한다. jne이므로 첫 번째 정수는 14보다 작거나 같아야

<explode_bomb>이 실행되지 않는다. 이후 +60에서는 <func4>를 호출하는데, edi에는 첫 번째 정수, esi에는 0, edx에는 14가 들어간다. 즉, func4(첫 번째 숫자, 0, 14)을 실행시킨다. 이후 func4의 반환 값이 0이 아닐 경우(+65, +67) <explode_bomb>이 실행되는 것을 알 수 있다. 또한 +69에서 두 번째 정수(+4에서 rcx 확인 가능)가 0일 경우에만 <explode_bomb>이 실행되지 않고, phase_4가 종료된다.

이제 <func4>를 살펴보자. 이 함수는 +27, +48에서 <func4> (자기자신)을 다시 호출하는 재귀 함수이다. 세 개의 인자(edi, esi, edx)를 각각 n, low, high라고 하였을 때 mid = low + (high – low) / 2를 통해(+4 ~ +15) 중간값을 계산하는 것을 확인할 수 있다. <func4>가 0을 반환하는 경우는 +36이 실행되고 난 후 func4가 종료되었을 때이다. 이는 +20의 결과가 jle, jge 조건을 모두 만족할 때, 즉 n == mid일 때만 실행됨을 알 수 있다. n < mid인 경우엔 func4(n, low, mid - 1) * 2가 반환되고, n > mid인 경우는 func4(n, mid + 1, high) * 2 + 1이 반환한다. 이때 func4의 반환값이 0인 경우는 여러 경우가 있는데, 그 중 n = 7일 때 func4가 한 번에 0을 반환하게 된다.

결론적으로, 여러 개의 답 중 하나인 **“7 0”**을 입력하여 phase_4를 통과할 수 있다.

5. phase_5

```
Dump of assembler code for function phase_5:
   0x0000000000401138 <+0>:     push   %rbx
   0x0000000000401139 <+1>:     mov    %rdi,%rbx
   0x000000000040113c <+4>:     callq  0x401391 <string_length>
   0x0000000000401141 <+9>:     cmp    $0x6,%eax
   0x0000000000401144 <+12>:    je     0x40114b <phase_5+19>
   0x0000000000401146 <+14>:    callq  0x401614 <explode_bomb>
   0x000000000040114b <+19>:    mov    $0x0,%eax
   0x0000000000401150 <+24>:    mov    $0x0,%edx
   0x0000000000401155 <+29>:    movzbl (%rbx,%rax,1),%ecx
   0x0000000000401159 <+33>:    and    $0xf,%ecx
   0x000000000040115c <+36>:    add    0x402600(,%rcx,4),%edx
   0x0000000000401163 <+43>:    add    $0x1,%rax
   0x0000000000401167 <+47>:    cmp    $0x6,%rax
   0x000000000040116b <+51>:    jne    0x401155 <phase_5+29>
   0x000000000040116d <+53>:    cmp    $0x2c,%edx
   0x0000000000401170 <+56>:    je     0x401177 <phase_5+63>
   0x0000000000401172 <+58>:    callq  0x401614 <explode_bomb>
   0x0000000000401177 <+63>:    pop    %rbx
   0x0000000000401178 <+64>:    retq
```

```
(gdb) x/16w 0x402600
0x402600 <array.3162>:  2       10      6       1
0x402610 <array.3162+16>:       12      16      9       3
0x402620 <array.3162+32>:       4       7       14      5
0x402630 <array.3162+48>:       11      8       15      13
```

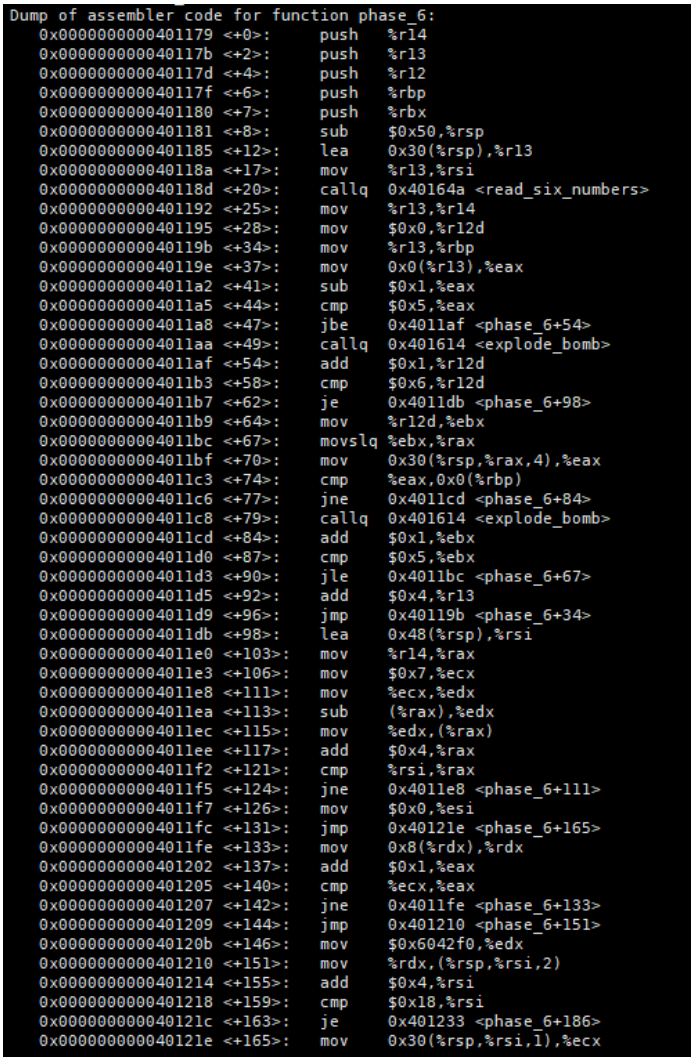
phase_5를 살펴보면, 먼저 <string_length>를 통해 우리가 입력한 string의 길이를 eax에 넣는다. 길이가 6이 아니라면 <explode_bomb>이 실행된다. 즉, 입력하는 string의 길이는 6글자여야 한다. 이후 eax(루프 카운터, i로 후술), edx(누적 합)를 0으로 한다. 이후 입력한 string(rbx)의 i번째 문자를 ecx에 넣고, 하위 4비트(문자는 아스키코드, 즉 숫자로 표현됨)만을 남긴다. 이때 ecx는 0~15의 값을 가지게 된다. 이후 0x402600 주소에 있는 배열에 저장된 ecx번째 숫자를 불러와 edx에 더한다. (+36) 이후에는 i에 1만큼 더하고 i가 6이 되지 않았다면 +29부터 루프를 다시 실행한다. 루프가 끝나게 되면, edx에는 6개의 문자의 하위 4비트에 해당하는 0x402600 주소에 있는 배열에 저장된 정수의 합이 저장되게 된다. 그래서 0x402600 주소에 있는 배열에 저장된 16개의 32비트 정수를 확인해주었다. 또한 마지막으로 그렇게 누적된 값 edx가 0x2c (10진수로 44)와 같아야만 <explode_bomb>이 실행되지 않고 phase_5를 통과할 수 있게 된다.

우선 배열의 값 중 6개의 합이 44가 되는 경우를 찾아주었다. 내가 찾은 경우는 10, 6, 12, 9, 3, 4였다. 이를 인덱스로 다시 표현하면 1, 2, 4, 6, 7, 8이다. 여기서 ‘1’의 하위 4비트가 1임을 이용하면 “124678”이 정답 중 하나가 됨을 알 수 있다.

결론적으로, 여러 개의 답 중 하나인 **“124678”**을 입력하여 phase_5를 통과할 수 있다.

6. phase_6 (사진은 다음 페이지에 첨부되어 있다)

phase_6 함수가 아주 길지만 jump를 기준으로 나누어서 보면 구조를 대략적으로 파악해볼 수 있다. 처음부터 살펴보면, 먼저 여러 차례 설명했던 <read_six_numbers>를 통해 6개의 정수를 입력 받고 이것이 stack에 담긴다. 이후 eax에 r13의 레지스터 값을 넣고 이것이 1~6 범위 내의 정



```
Dump of assembler code for function phase_6:
   0x0000000000401179 <+0>:     push   %r14
   0x000000000040117b <+2>:     push   %r13
   0x000000000040117d <+4>:     push   %r12
   0x000000000040117f <+6>:     push   %rbp
   0x0000000000401180 <+7>:     push   %rbx
   0x0000000000401181 <+8>:     sub    $0x50,%rsp
   0x0000000000401185 <+12>:    lea    0x30(%rsp),%r13
   0x000000000040118a <+17>:    mov    %r13,%rsi
   0x000000000040118d <+20>:    callq  0x40164a <read_six_numbers>
   0x0000000000401192 <+25>:    mov    %r13,%r14
   0x0000000000401195 <+28>:    mov    $0x0,%r12d
   0x000000000040119b <+34>:    mov    %r13,%rbp
   0x000000000040119e <+37>:    mov    0x0(%r13),%eax
   0x00000000004011a2 <+41>:    sub    $0x1,%eax
   0x00000000004011a5 <+44>:    cmp    $0x5,%eax
   0x00000000004011a8 <+47>:    jbe    0x4011af <phase_6+54>
   0x00000000004011aa <+49>:    callq  0x401614 <explode_bomb>
   0x00000000004011af <+54>:    add    $0x1,%r12d
   0x00000000004011b3 <+58>:    cmp    $0x6,%r12d
   0x00000000004011b7 <+62>:    je     0x4011db <phase_6+98>
   0x00000000004011b9 <+64>:    mov    %r12d,%ebx
   0x00000000004011bc <+67>:    movslq %ebx,%rax
   0x00000000004011bf <+70>:    mov    0x30(%rsp,%rax,4),%eax
   0x00000000004011c3 <+74>:    cmp    %eax,0x0(%rbp)
   0x00000000004011c6 <+77>:    jne    0x4011cd <phase_6+84>
   0x00000000004011c8 <+79>:    callq  0x401614 <explode_bomb>
   0x00000000004011cd <+84>:    add    $0x1,%ebx
   0x00000000004011d0 <+87>:    cmp    $0x5,%ebx
   0x00000000004011d3 <+90>:    jle    0x4011bc <phase_6+67>
   0x00000000004011d5 <+92>:    add    $0x4,%r13
   0x00000000004011d9 <+96>:    jmp    0x40119b <phase_6+34>
   0x00000000004011db <+98>:    lea    0x48(%rsp),%rsi
   0x00000000004011e0 <+103>:   mov    %r14,%rax
   0x00000000004011e3 <+106>:   mov    $0x7,%ecx
   0x00000000004011e8 <+111>:   mov    %ecx,%edx
   0x00000000004011ea <+113>:   sub    (%rax),%edx
   0x00000000004011ec <+115>:   mov    %edx,(%rax)
   0x00000000004011ee <+117>:   add    $0x4,%rax
   0x00000000004011f2 <+121>:   cmp    %rsi,%rax
   0x00000000004011f5 <+124>:   jne    0x4011e8 <phase_6+111>
   0x00000000004011f7 <+126>:   mov    $0x0,%esi
   0x00000000004011fc <+131>:   jmp    0x40121e <phase_6+165>
   0x00000000004011fe <+133>:   mov    0x8(%rdx),%rdx
   0x0000000000401202 <+137>:   add    $0x1,%eax
   0x0000000000401205 <+140>:   cmp    %ecx,%eax
   0x0000000000401207 <+142>:   jne    0x4011fe <phase_6+133>
   0x0000000000401209 <+144>:   jmp    0x401210 <phase_6+151>
   0x000000000040120b <+146>:   mov    $0x6042f0,%edx
   0x0000000000401210 <+151>:   mov    %rdx,(%rsp,%rsi,2)
   0x0000000000401214 <+155>:   add    $0x4,%rsi
   0x0000000000401218 <+159>:   cmp    $0x18,%rsi
   0x000000000040121c <+163>:   je     0x401233 <phase_6+186>
   0x000000000040121e <+165>:   mov    0x30(%rsp,%rsi,1),%ecx
   0x0000000000401222 <+169>:   cmp    $0x1,%ecx
---Type <return> to continue, or q <return> to quit---
   0x0000000000401225 <+172>:   jle    0x40120b <phase_6+146>
   0x0000000000401227 <+174>:   mov    $0x1,%eax
   0x000000000040122c <+179>:   mov    $0x6042f0,%edx
   0x0000000000401231 <+184>:   jmp    0x4011fe <phase_6+133>
   0x0000000000401233 <+186>:   mov    (%rsp),%rbx
   0x0000000000401237 <+190>:   lea    0x8(%rsp),%rax
   0x000000000040123c <+195>:   lea    0x30(%rsp),%rsi
   0x0000000000401241 <+200>:   mov    %rbx,%rcx
   0x0000000000401244 <+203>:   mov    (%rax),%rdx
   0x0000000000401247 <+206>:   mov    %rdx,0x8(%rcx)
   0x000000000040124b <+210>:   add    $0x8,%rax
   0x000000000040124f <+214>:   cmp    %rsi,%rax
   0x0000000000401252 <+217>:   je     0x401259 <phase_6+224>
   0x0000000000401254 <+219>:   mov    %rdx,%rcx
   0x0000000000401257 <+222>:   jmp    0x401244 <phase_6+203>
   0x0000000000401259 <+224>:   movq   $0x0,0x8(%rdx)
   0x0000000000401261 <+232>:   mov    $0x5,%ebp
   0x0000000000401266 <+237>:   mov    0x8(%rbx),%rax
   0x000000000040126a <+241>:   mov    (%rax),%eax
   0x000000000040126c <+243>:   cmp    %eax,(%rbx)
   0x000000000040126e <+245>:   jge    0x401275 <phase_6+252>
   0x0000000000401270 <+247>:   callq  0x401614 <explode_bomb>
   0x0000000000401275 <+252>:   mov    0x8(%rbx),%rbx
   0x0000000000401279 <+256>:   sub    $0x1,%ebp
   0x000000000040127c <+259>:   jne    0x401266 <phase_6+237>
   0x000000000040127e <+261>:   add    $0x50,%rsp
   0x0000000000401282 <+265>:   pop    %rbx
   0x0000000000401283 <+266>:   pop    %rbp
   0x0000000000401284 <+267>:   pop    %r12
   0x0000000000401286 <+269>:   pop    %r13
   0x0000000000401288 <+271>:   pop    %r14
   0x000000000040128a <+273>:   retq
```

```
(gdb) x/24w 0x6042f0
0x6042f0 <node1>:       777     1       6308608 0
0x604300 <node2>:       872     2       6308624 0
0x604310 <node3>:       341     3       6308640 0
0x604320 <node4>:       555     4       6308656 0
0x604330 <node5>:       830     5       6308672 0
0x604340 <node6>:       769     6       0       0
```

수인지 확인한다. 만약 범위를 벗어난다면 <explode_bomb>이 실행된다. 이는 사실 입력된 정수가 전부 1~6사이의 정수인지 확인하는 코드인데, +92에서 r13의 주소에 0x4만큼 더하고(즉, stack에 저장된 입력된 32비트 정수를 차례대로 확인) +96에서 +34로 jump하는 것을 통해 알 수 있다. 즉, 입력된 정수 6개는 모두 **1~6사이의 정수**여야 한다. 또한 r12d가 바깥쪽 루프 카운터 i(0부터 5까지, +54 ~ +58에서 확인, 6이 되면 +98로 jump)이고, r13, rbp(+34)가 numbers[i]라고 생각할 수 있다. +67 ~ +90을 확인해보면 ebx가 안쪽 루프 카운터 j의 역할을 하게 되고, i+1부터 시작하여(+54에서 1이 증가된 i를 +64에서 받아옴) 5까지 증가(+84, +87)하게 된다. 안쪽 루프는 입력된 숫자들 사이에 중복이 있는지 검사하는 것으로(+67 ~ +77), 중복이 없어야 <explode_bomb>이 실행되지 않는다. 즉, 입력된 6개의 정수는 모두 달라야 하므로 **1~6의 순열**이여야 한다.

이제 첫 번째 이중 루프를 나온 후를 살펴보자. +98에서부터 다시 루프를 돌며 stack에 저장된 6개의 숫자 x를 7-x로 변환하여서 원래 위치에 덮어쓴다는 것을 확인할 수 있다.

두 번째 루프를 나온 후 다시 나오는 세 번째 루프는 정말 복잡해 보인다. 먼저 +146에서 확인할 수 있는 0x6042f0가 의심스러워 이를 x/24w를 통해 출력해보니, linked list임을 확인할 수 있었다. 이것을 알게 된 후 일단 +237부터 시작하는 마지막 루프를 보았다. 이는 linked list가 내림차순으로 정렬되어 있는지 확인하는 코드로, 현재 노드(rbx)의 값이 다음 노드(ebx)의 값보다 크거나 같아야(+241 ~ +245) <explode_bomb>이 실행되지 않는다. 즉 세 번째 루프를 나와 변형된 linked list가 내림차순으로 정렬되어 있어야 한다는 것을 알 수 있다.

여기서 나는 매우 복잡해 보이는 세 번째 루프를 해석하기보다, br explode_bomb을 통하여 bomb이

터지지 않도록 한 후 지금까지 얻은 정보를 통해 답을 유추해 입력해보기로 하였다. 입력된 1~6의 순열(첫 번째 루프에서 확인)의 각 숫자인 x를 7-x로 바꾼 후(두 번째 루프에서 확인) 이에 따라 linked list를 정렬하여(세 번째 루프의 동작을 예측한 것, explode_bomb이 호출되지 않음으로 linked list를 변환하는 작업으로 추정, 여기에 더해 x/24w 0x6042f0의 출력값에서 각 노드에 부여된 1~6을 확인하였기에 이렇게 추측함) 그것이 내림차순으로 정렬되었을 때(마지막 루프에서 확인) phase_6이 통과될 수 있을 것이라고 생각하였다. 이에 따라 내가 유추한 답을 계산해보면, 원래 주어진 linked list의 node가 2, 5, 1, 6, 4, 3 순서로 되었을 때이다. 이를 y = 7 – x를 통해 변환하여 얻은 최종 결과는 “5 2 6 1 3 4” 였으며, 이를 입력하였을 때 phase_6을 통과할 수 있었다.

결론적으로 정답인 **“5 2 6 1 3 4”**를 입력하여 phase_6을 통과할 수 있다.

7. secret_phase

```
Dump of assembler code for function phase_defused:
   0x00000000004017b2 <+0>:     sub    $0x68,%rsp
   0x00000000004017b6 <+4>:     mov    $0x1,%edi
   0x00000000004017bb <+9>:     callq  0x401550 <send_msg>
   0x00000000004017c0 <+14>:    cmpl   $0x6,0x202fd5(%rip)        # 0x60479c <num_input_strings>
   0x00000000004017c7 <+21>:    jne    0x401836 <phase_defused+132>
   0x00000000004017c9 <+23>:    lea    0x10(%rsp),%r8
   0x00000000004017ce <+28>:    lea    0x8(%rsp),%rcx
   0x00000000004017d3 <+33>:    lea    0xc(%rsp),%rdx
   0x00000000004017d8 <+38>:    mov    $0x4028d7,%esi
   0x00000000004017dd <+43>:    mov    $0x6048b0,%edi
   0x00000000004017e2 <+48>:    mov    $0x0,%eax
   0x00000000004017e7 <+53>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x00000000004017ec <+58>:    cmp    $0x3,%eax
   0x00000000004017ef <+61>:    jne    0x401822 <phase_defused+112>
   0x00000000004017f1 <+63>:    mov    $0x4028e0,%esi
   0x00000000004017f6 <+68>:    lea    0x10(%rsp),%rdi
   0x00000000004017fb <+73>:    callq  0x4013ae <strings_not_equal>
   0x0000000000401800 <+78>:    test   %eax,%eax
   0x0000000000401802 <+80>:    jne    0x401822 <phase_defused+112>
   0x0000000000401804 <+82>:    mov    $0x402738,%edi
   0x0000000000401809 <+87>:    callq  0x400b40 <puts@plt>
   0x000000000040180e <+92>:    mov    $0x402760,%edi
   0x0000000000401813 <+97>:    callq  0x400b40 <puts@plt>
   0x0000000000401818 <+102>:   mov    $0x0,%eax
   0x000000000040181d <+107>:   callq  0x4012c9 <secret_phase>
   0x0000000000401822 <+112>:   mov    $0x402798,%edi
   0x0000000000401827 <+117>:   callq  0x400b40 <puts@plt>
   0x000000000040182c <+122>:   mov    $0x4027c8,%edi
   0x0000000000401831 <+127>:   callq  0x400b40 <puts@plt>
   0x0000000000401836 <+132>:   add    $0x68,%rsp
   0x000000000040183a <+136>:   retq
```

```
(gdb) r
Starting program: /home/std/cotmdgus/bomb26/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
You can Russia from land here in Alaska.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2.  Keep going!
0 b 730
Halfway there!
7 0
So you got that one.  Try this one.
124678
Good work!  On to the next...
5 2 6 1 3 4

Breakpoint 2, 0x00000000004017e7 in phase_defused ()
Missing separate debuginfos, use: debuginfo-install glibc-2.17-326.el7_9.x86_64
(gdb) x/s 0x6048b0
0x6048b0 <input_strings+240>:   "7 0"
```

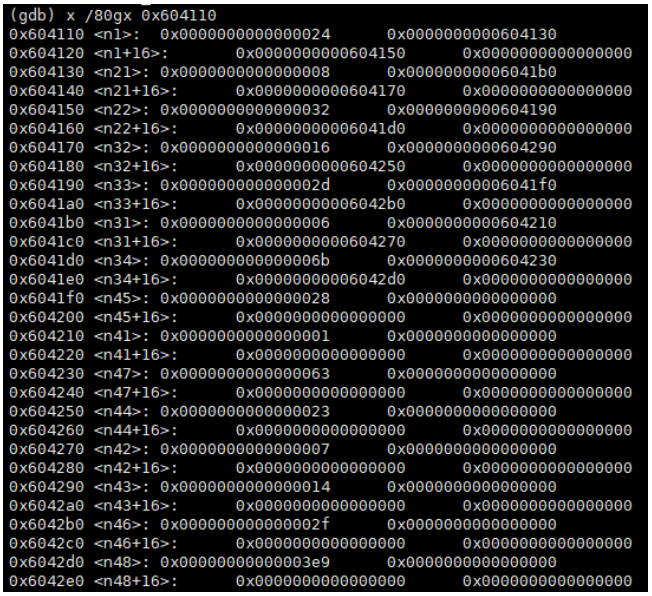
```
(gdb) x/s 0x4028d7
0x4028d7:       "%d %d %s"
```

```
(gdb) x/s 0x4028e0
0x4028e0:       "DrJisungPark"
```

먼저 <phase_defused>를 살펴보면, 놀랍게도 <secret_phase>가 존재함을 확인할 수 있다. 여기서 <secret_phase>의 실행 조건을 찾아보았다. 먼저 sscanf의 인자로 들어가는 esi(0x4028d7) 주소에 담긴 값을 확인해본 결과 정수 2개와 string 1개가 입력되는 것을 알 수 있었다. 또한 0x6048b0 주소에 담긴 값을 확인해보니 empty string이 출력되었다. 마지막으로 0x4028e0 주소에 담긴 값을 확인해보니 “DrJisungPark”이 나왔다. 이 string과 앞에서 입력된 string이 <strings_not_equal>의 인자로 들어가게 되고, 두 string이 같은 경우에만 +112로 jump하지 않고, <secret_phase>가 실행되는 것처럼 보였다. 또한 0x6048b0 주소에 무슨 값이 담기는 건지 확인하기 위하여 *phase_defused +53에 break point를 걸어주고, bomb을 실행해 phase 6까지 통과하였다. 이후 x/s 0x6048b0 주소에 담긴 값을 확인하여보니 “7 0”이 나왔고, 이는 phase 4에서 입력한 값이었다. 이에 따라 나는 phase 4에서 “7 0 DrJisungPark”을 입력하면 secret_phase에 들어갈 수 있을 것 같았고, explode_bomb에 break point를 걸고 실험을 해보았다. 나머지 phase에서는 전부 앞에서 구했던 정답을 입력하였고, phase 6을 통과하고 나니 실제로 secret_phase에 진입하는데 성공하였다.

```
Dump of assembler code for function secret_phase:
   0x00000000004012c9 <+0>:     push   %rbx
   0x00000000004012ca <+1>:     callq  0x40168c <read_line>
   0x00000000004012cf <+6>:     mov    $0xa,%edx
   0x00000000004012d4 <+11>:    mov    $0x0,%esi
   0x00000000004012d9 <+16>:    mov    %rax,%rdi
   0x00000000004012dc <+19>:    callq  0x400c00 <strtol@plt>
   0x00000000004012e1 <+24>:    mov    %rax,%rbx
   0x00000000004012e4 <+27>:    lea    -0x1(%rax),%eax
   0x00000000004012e7 <+30>:    cmp    $0x3e8,%eax
   0x00000000004012ec <+35>:    jbe    0x4012f3 <secret_phase+42>
   0x00000000004012ee <+37>:    callq  0x401614 <explode_bomb>
   0x00000000004012f3 <+42>:    mov    %ebx,%esi
   0x00000000004012f5 <+44>:    mov    $0x604110,%edi
   0x00000000004012fa <+49>:    callq  0x40128b <fun7>
   0x00000000004012ff <+54>:    cmp    $0x7,%eax
   0x0000000000401302 <+57>:    je     0x401309 <secret_phase+64>
   0x0000000000401304 <+59>:    callq  0x401614 <explode_bomb>
   0x0000000000401309 <+64>:    mov    $0x402590,%edi
   0x000000000040130e <+69>:    callq  0x400b40 <puts@plt>
   0x0000000000401313 <+74>:    callq  0x4017b2 <phase_defused>
   0x0000000000401318 <+79>:    pop    %rbx
   0x0000000000401319 <+80>:    retq
```

```
Dump of assembler code for function fun7:
   0x000000000040128b <+0>:     sub    $0x8,%rsp
   0x000000000040128f <+4>:     test   %rdi,%rdi
   0x0000000000401292 <+7>:     je     0x4012bf <fun7+52>
   0x0000000000401294 <+9>:     mov    (%rdi),%edx
   0x0000000000401296 <+11>:    cmp    %esi,%edx
   0x0000000000401298 <+13>:    jle    0x4012a7 <fun7+28>
   0x000000000040129a <+15>:    mov    0x8(%rdi),%rdi
   0x000000000040129e <+19>:    callq  0x40128b <fun7>
   0x00000000004012a3 <+24>:    add    %eax,%eax
   0x00000000004012a5 <+26>:    jmp    0x4012c4 <fun7+57>
   0x00000000004012a7 <+28>:    mov    $0x0,%eax
   0x00000000004012ac <+33>:    cmp    %esi,%edx
   0x00000000004012ae <+35>:    je     0x4012c4 <fun7+57>
   0x00000000004012b0 <+37>:    mov    0x10(%rdi),%rdi
   0x00000000004012b4 <+41>:    callq  0x40128b <fun7>
   0x00000000004012b9 <+46>:    lea    0x1(%rax,%rax,1),%eax
   0x00000000004012bd <+50>:    jmp    0x4012c4 <fun7+57>
   0x00000000004012bf <+52>:    mov    $0xffffffff,%eax
   0x00000000004012c4 <+57>:    add    $0x8,%rsp
   0x00000000004012c8 <+61>:    retq
```



```
(gdb) x /80gx 0x604110
0x604110 <n1>:  0x0000000000000024      0x0000000000604130
0x604120 <n1+16>:       0x0000000000604150      0x0000000000000000
0x604130 <n21>: 0x0000000000000008      0x00000000006041b0
0x604140 <n21+16>:      0x0000000000604170      0x0000000000000000
0x604150 <n22>: 0x0000000000000032      0x0000000000604190
0x604160 <n22+16>:      0x00000000006041d0      0x0000000000000000
0x604170 <n32>: 0x0000000000000016      0x0000000000604290
0x604180 <n32+16>:      0x0000000000604250      0x0000000000000000
0x604190 <n33>: 0x000000000000002d      0x00000000006041f0
0x6041a0 <n33+16>:      0x00000000006042b0      0x0000000000000000
0x6041b0 <n31>: 0x0000000000000006      0x0000000000604210
0x6041c0 <n31+16>:      0x0000000000604270      0x0000000000000000
0x6041d0 <n34>: 0x000000000000006b      0x0000000000604230
0x6041e0 <n34+16>:      0x00000000006042d0      0x0000000000000000
0x6041f0 <n45>: 0x0000000000000028      0x0000000000000000
0x604200 <n45+16>:      0x0000000000000000      0x0000000000000000
0x604210 <n41>: 0x0000000000000001      0x0000000000000000
0x604220 <n41+16>:      0x0000000000000000      0x0000000000000000
0x604230 <n47>: 0x0000000000000063      0x0000000000000000
0x604240 <n47+16>:      0x0000000000000000      0x0000000000000000
0x604250 <n44>: 0x0000000000000023      0x0000000000000000
0x604260 <n44+16>:      0x0000000000000000      0x0000000000000000
0x604270 <n42>: 0x0000000000000007      0x0000000000000000
0x604280 <n42+16>:      0x0000000000000000      0x0000000000000000
0x604290 <n43>: 0x0000000000000014      0x0000000000000000
0x6042a0 <n43+16>:      0x0000000000000000      0x0000000000000000
0x6042b0 <n46>: 0x000000000000002f      0x0000000000000000
0x6042c0 <n46+16>:      0x0000000000000000      0x0000000000000000
0x6042d0 <n48>: 0x00000000000003e9      0x0000000000000000
0x6042e0 <n48+16>:      0x0000000000000000      0x0000000000000000
```

이제 secret_phase를 살펴보자. 먼저 <read_line>을 통해 한 줄의 string을 입력받는다. 이후 이를 <strtol@plt>를 통해 10진수 정수로 변환한다. 이 값을 N이라고 했을 때 N – 1 <= 1000인 경우에만 explode_bomb이 실행되지 않는다. (jbe이므로 unsigned) 즉, 하나의 정수를 입력하는데, 이것이 0~1001 범위 내의 정수여야 한다. 이후로는 이것과 함께 x604110 주소의 값이 fun7의 인자로 들어가므로 의심스럽기에 이를 확인해주었다. 여기서 나는 이것이 이진 트리인 것을 알 수 있었다. 또한 fun7의 반환값이 7일 때에만 explode_bomb이 실행되지 않는다는 것을 +54에서 알 수 있었다.

마지막으로 fun7을 살펴보자. 우선 fun7은 재귀함수임을 알 수 있었고, 현재 노드의 주소와 탐색할 노드의 주소를 인자로 받고 있는 것을 확인하였다. 주어진 이진 트리는 왼쪽 노드의 값 < 현재 노드의 값 < 오른쪽 노드의 값의 구조를 하고 있었다. 여기서 fun7은 이 특징을 이용해 입력된 값(N)을 가지고 있는 노드를 찾는 함수라는 것을 알 수 있다. 또한 fun7(찾을 값, 현재 노드)로 호출한다고 해보자. N이 현재 노드의 값보다 작은 경우에는 2 * fun7(N, 왼쪽 노드), N이 현재 노드의 값보다 큰 경우에는 2 * fun7(N, 오른쪽 노드) + 0x1가 반환된다. 마지막으로 N이 현재 노드의 값과 같은 경우 0을 반환한다. 만약 현재 노드가 NULL이라면 -1을 반환한다. fun7의 반환값이 7이 되기 위해서는 n1에서 시작하여 오른쪽으로 3번 탐색한 노드의 값과 N이 같은 값이어야 한다. 해당 위치에 있는 노드는 n48으로, 0x3e9 (10진수로 1001)을 가지고 있다. 즉 N = 1001일 때 fun7이 7을 반환하게 된다.

결론적으로 정답인 **“1001”**을 입력하면 secret_phase를 통과할 수 있다.