Lab3 BombLab Report

20220665 유영준

4주차 랩 시간에 GCC, GDB에 대한 간단한 설명과 어셈블리의 간단한 동작 방법에 대해 배웠다. 어셈블리의 syntax는 'opcode+operand' 형식으로 구성되어 있고, opcode는 Unary operator, Binary operator가 존재한다. operand에는 실제 값, 레지스터 값, 메모리 세 종류가 존재한다. 어셈블리의 다양한 instruction에 대해 학습하였고, 예시를 통해 어셈블리의 작동 방식에 대해 알 수 있었다. 그리고 bomblab을 다운받기 위한 서버 접속 방법에 대한 설명을 듣고 해당 랩 시간을 마무리했다. 6주차 랩 시간에는 calling convention과 메모리에서 사용되는 stack의 작동 방식에 대해 학습했다. 해당 랩 시간과 관련하여 BombLab이 과제로 나왔고, phase1부터 phase6 그리고 secret phase까지의 풀이 과정은 다음과 같다. 과제는 xshell 환경에서 진행하였다.

1. Phase1

- 먼저 main 함수를 disassemble 하여 전체 프로그램의 작동 방식을 살펴보았다.

```
0x0000000000400e45 <+136>:   callq  0x4015dc <read_line>
0x0000000000400e4a <+141>:   mov    %rax,%rdi
0x0000000000400e4d <+144>:   callq  0x400ef0 <phase_1>
```

각 phase에 들어가기 전 read_line 함수를 통해 읽은 입력 값을 rdi 레지스터에 넣은 후 phase를 실행하는 것으로 보아 입력 값이 각 phase의 첫 번째 인자로 들어가 함수를 진행시킨다는 것을 알 수 있었다.

- phase_1 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x0000000000400ef0 <+0>:     sub    $0x8,%rsp
   0x0000000000400ef4 <+4>:     mov    $0x4024b0,%esi
   0x0000000000400ef9 <+9>:     callq  0x4012fe <strings_not_equal>
   0x0000000000400efe <+14>:    test   %eax,%eax
   0x0000000000400f00 <+16>:    je     0x400f07 <phase_1+23>
   0x0000000000400f02 <+18>:    callq  0x401564 <explode_bomb>
   0x0000000000400f07 <+23>:    add    $0x8,%rsp
   0x0000000000400f0b <+27>:    retq
End of assembler dump.
```

rsp값을 바꾸어 stack frame를 형성한 뒤 rsi 에 특정 값을 넣은 후 strings_not_equal 함수를 실행시킨다는 것을 알았다. 이때 rsi에 넣은 주소에 어떤 값이 들어 있는지 x/s를 통해 확인해 보았다.

```
(gdb) x/s 0x4024b0
0x4024b0:       "He is evil and fits easily into most overhead storage bins."
```

앞에서 살펴본 것처럼 rdi에는 input string이 들어가있고, rsi에 해당 string이 들어있으므로 strings_not_equal 함수에서 두 string이 동일한지 확인하여 같으면 0, 아니면 1을 return할 것이라고 추측하고, strings_not_equal 함수를 disassemble 해보았다.

```
(gdb) disas strings_not_equal
Dump of assembler code for function strings_not_equal:
   0x00000000004012fe <+0>:     push   %r12
   0x0000000000401300 <+2>:     push   %rbp
   0x0000000000401301 <+3>:     push   %rbx
   0x0000000000401302 <+4>:     mov    %rdi,%rbx
   0x0000000000401305 <+7>:     mov    %rsi,%rbp
   0x0000000000401308 <+10>:    callq  0x4012e1 <string_length>
   0x000000000040130d <+15>:    mov    %eax,%r12d
   0x0000000000401310 <+18>:    mov    %rbp,%rdi
   0x0000000000401313 <+21>:    callq  0x4012e1 <string_length>
   0x0000000000401318 <+26>:    mov    $0x1,%edx
   0x000000000040131d <+31>:    cmp    %eax,%r12d
   0x0000000000401320 <+34>:    jne    0x401360 <strings_not_equal+98>
   0x0000000000401322 <+36>:    movzbl (%rbx),%eax
   0x0000000000401325 <+39>:    test   %al,%al
   0x0000000000401327 <+41>:    je     0x40134d <strings_not_equal+79>
   0x0000000000401329 <+43>:    cmp    0x0(%rbp),%al
   0x000000000040132c <+46>:    je     0x401337 <strings_not_equal+57>
   0x000000000040132e <+48>:    xchg   %ax,%ax
   0x0000000000401330 <+50>:    jmp    0x401354 <strings_not_equal+86>
   0x0000000000401332 <+52>:    cmp    0x0(%rbp),%al
   0x0000000000401335 <+55>:    jne    0x40135b <strings_not_equal+93>
   0x0000000000401337 <+57>:    add    $0x1,%rbx
   0x000000000040133b <+61>:    add    $0x1,%rbp
   0x000000000040133f <+65>:    movzbl (%rbx),%eax
   0x0000000000401342 <+68>:    test   %al,%al
   0x0000000000401344 <+70>:    jne    0x401332 <strings_not_equal+52>
   0x0000000000401346 <+72>:    mov    $0x0,%edx
   0x000000000040134b <+77>:    jmp    0x401360 <strings_not_equal+98>
   0x000000000040134d <+79>:    mov    $0x0,%edx
   0x0000000000401352 <+84>:    jmp    0x401360 <strings_not_equal+98>
   0x0000000000401354 <+86>:    mov    $0x1,%edx
   0x0000000000401359 <+91>:    jmp    0x401360 <strings_not_equal+98>
   0x000000000040135b <+93>:    mov    $0x1,%edx
   0x0000000000401360 <+98>:    mov    %edx,%eax
   0x0000000000401362 <+100>:   pop    %rbx
---Type <return> to continue, or q <return> to quit---
   0x0000000000401363 <+101>:   pop    %rbp
   0x0000000000401364 <+102>:   pop    %r12
   0x0000000000401366 <+104>:   retq
End of assembler dump.
```

예상대로 strings_not_equal 함수에서 두 string이 같으면 0, 아니면 1을 rax에 넣어 return 해준다는 것을 알았다. 해당 함수를 실행시킨 이후 test와 je를 통해 rax 자기 자신의 and 연산 후 그 값이 0이면 bomb을 터트리지 않고 함수를 종료한다는 것을 알았다. 이를 통해 phase_1의 입력 string은 "He is evil and fits easily into most overhead storage bins."임을 알았다.

2. Phase2

- phase_2 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

```
(gdb) disas phase_2
Dump of assembler code for function phase_2:
   0x0000000000400f0c <+0>:     push   %rbp
   0x0000000000400f0d <+1>:     push   %rbx
   0x0000000000400f0e <+2>:     sub    $0x28,%rsp
   0x0000000000400f12 <+6>:     mov    %rsp,%rsi
   0x0000000000400f15 <+9>:     callq  0x40159a <read_six_numbers>
   0x0000000000400f1a <+14>:    cmpl   $0x1,(%rsp)
   0x0000000000400f1e <+18>:    je     0x400f40 <phase_2+52>
   0x0000000000400f20 <+20>:    callq  0x401564 <explode_bomb>
   0x0000000000400f25 <+25>:    jmp    0x400f40 <phase_2+52>
   0x0000000000400f27 <+27>:    mov    -0x4(%rbx),%eax
   0x0000000000400f2a <+30>:    add    %eax,%eax
   0x0000000000400f2c <+32>:    cmp    %eax,(%rbx)
   0x0000000000400f2e <+34>:    je     0x400f35 <phase_2+41>
   0x0000000000400f30 <+36>:    callq  0x401564 <explode_bomb>
   0x0000000000400f35 <+41>:    add    $0x4,%rbx
   0x0000000000400f39 <+45>:    cmp    %rbp,%rbx
   0x0000000000400f3c <+48>:    jne    0x400f27 <phase_2+27>
   0x0000000000400f3e <+50>:    jmp    0x400f4c <phase_2+64>
   0x0000000000400f40 <+52>:    lea    0x4(%rsp),%rbx
   0x0000000000400f45 <+57>:    lea    0x18(%rsp),%rbp
   0x0000000000400f4a <+62>:    jmp    0x400f27 <phase_2+27>
   0x0000000000400f4c <+64>:    add    $0x28,%rsp
   0x0000000000400f50 <+68>:    pop    %rbx
   0x0000000000400f51 <+69>:    pop    %rbp
   0x0000000000400f52 <+70>:    retq
End of assembler dump.
```

- rsp값을 바꾼 후 rsi에 rsp값을 넣은 뒤 read_six_numbers 함수를 호출하는 것을 알 수 있다. read_six_numbers를 disassemble 하여 함수의 작동 방식을 살펴보았다.

```
(gdb) disas read_six_numbers
Dump of assembler code for function read_six_numbers:
   0x000000000040159a <+0>:     sub    $0x18,%rsp
   0x000000000040159e <+4>:     mov    %rsi,%rdx
   0x00000000004015a1 <+7>:     lea    0x4(%rsi),%rcx
   0x00000000004015a5 <+11>:    lea    0x14(%rsi),%rax
   0x00000000004015a9 <+15>:    mov    %rax,0x8(%rsp)
   0x00000000004015ae <+20>:    lea    0x10(%rsi),%rax
   0x00000000004015b2 <+24>:    mov    %rax,(%rsp)
   0x00000000004015b6 <+28>:    lea    0xc(%rsi),%r9
   0x00000000004015ba <+32>:    lea    0x8(%rsi),%r8
   0x00000000004015be <+36>:    mov    $0x4027b1,%esi
   0x00000000004015c3 <+41>:    mov    $0x0,%eax
   0x00000000004015c8 <+46>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x00000000004015cd <+51>:    cmp    $0x5,%eax
   0x00000000004015d0 <+54>:    jg     0x4015d7 <read_six_numbers+61>
   0x00000000004015d2 <+56>:    callq  0x401564 <explode_bomb>
   0x00000000004015d7 <+61>:    add    $0x18,%rsp
   0x00000000004015db <+65>:    retq
End of assembler dump.
```

- <+36>에서 rsi 레지스터에 넣는 값을 x/s를 통해 알아보았다.

```
(gdb) x/s 0x4027b1
0x4027b1:       "%d %d %d %d %d %d"
```

이를 통해 read_six_numbers 함수에서는 rsp부터 6개의 공간에 input string의 6개의 정수를 차례로 저장한다는 것을 알았다. 그런 뒤, <+51>부터 만약 scanf를 통해 입력 받은 정수의 개수가 5개 이하일 경우 bomb을 터트린다는 것을 알 수 있다.

- 다시 phase_2 함수로 돌아가서 read_six_numbers 함수 호출 다음 rsp에 들어있는 주소의 값이 1인지 확인하고 만약 1이 아니라면 bomb을 터트린다. 이를 통해 input string의 첫 번째 정수는 1이라는 것을 알 수 있다.

- <+52>로 jump한 뒤, rbx, rbp 레지스터에 각각 rsp+4, rsp+24 값을 넣은 뒤, <+27>부터 <+50>까지 loop를 돌린다는 것을 알 수 있다. rbx-4를 한 값, 즉 input string의 전 정수 값에서 2배를 하여 rbx 값과 일치하는지 확인 후 일치하지 않으면 bomb을 터트리므로 input string의 6개의 정수는 1부터 2배를 한 "1 2 4 8 16 32"라는 것을 알 수 있다. 이때 rbp 레지스터에는 rsp+24 값을 넣어 loop에서 6개의 정수를 확인한 후 종료할 수 있도록 해준다.

3. Phase3

- phase_3 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

```
(gdb) disas phase_3
Dump of assembler code for function phase_3:
   0x0000000000400f53 <+0>:     sub    $0x18,%rsp
   0x0000000000400f57 <+4>:     lea    0x8(%rsp),%rcx
   0x0000000000400f5c <+9>:     lea    0xc(%rsp),%rdx
   0x0000000000400f61 <+14>:    mov    $0x4027bd,%esi
   0x0000000000400f66 <+19>:    mov    $0x0,%eax
   0x0000000000400f6b <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x0000000000400f70 <+29>:    cmp    $0x1,%eax
   0x0000000000400f73 <+32>:    jg     0x400f7a <phase_3+39>
   0x0000000000400f75 <+34>:    callq  0x401564 <explode_bomb>
   0x0000000000400f7a <+39>:    cmpl   $0x7,0xc(%rsp)
   0x0000000000400f7f <+44>:    ja     0x400fe7 <phase_3+148>
   0x0000000000400f81 <+46>:    mov    0xc(%rsp),%eax
   0x0000000000400f85 <+50>:    jmpq   *0x402520(,%rax,8)
   0x0000000000400f8c <+57>:    mov    $0x0,%eax
   0x0000000000400f91 <+62>:    jmp    0x400f98 <phase_3+69>
   0x0000000000400f93 <+64>:    mov    $0x33a,%eax
   0x0000000000400f98 <+69>:    sub    $0x1c1,%eax
   0x0000000000400f9d <+74>:    jmp    0x400fa4 <phase_3+81>
   0x0000000000400f9f <+76>:    mov    $0x0,%eax
   0x0000000000400fa4 <+81>:    add    $0x162,%eax
   0x0000000000400fa9 <+86>:    jmp    0x400fb0 <phase_3+93>
   0x0000000000400fab <+88>:    mov    $0x0,%eax
   0x0000000000400fb0 <+93>:    sub    $0x3b3,%eax
   0x0000000000400fb5 <+98>:    jmp    0x400fbc <phase_3+105>
   0x0000000000400fb7 <+100>:   mov    $0x0,%eax
   0x0000000000400fbc <+105>:   add    $0x3b3,%eax
   0x0000000000400fc1 <+110>:   jmp    0x400fc8 <phase_3+117>
   0x0000000000400fc3 <+112>:   mov    $0x0,%eax
   0x0000000000400fc8 <+117>:   sub    $0x3b3,%eax
   0x0000000000400fcd <+122>:   jmp    0x400fd4 <phase_3+129>
   0x0000000000400fcf <+124>:   mov    $0x0,%eax
   0x0000000000400fd4 <+129>:   add    $0x3b3,%eax
   0x0000000000400fd9 <+134>:   jmp    0x400fe0 <phase_3+141>
   0x0000000000400fdb <+136>:   mov    $0x0,%eax
   0x0000000000400fe0 <+141>:   sub    $0x3b3,%eax
```

```
   0x0000000000400fe5 <+146>:   jmp    0x400ff1 <phase_3+158>
   0x0000000000400fe7 <+148>:   callq  0x401564 <explode_bomb>
   0x0000000000400fec <+153>:   mov    $0x0,%eax
   0x0000000000400ff1 <+158>:   cmpl   $0x5,0xc(%rsp)
   0x0000000000400ff6 <+163>:   jg     0x400ffe <phase_3+171>
   0x0000000000400ff8 <+165>:   cmp    0x8(%rsp),%eax
   0x0000000000400ffc <+169>:   je     0x401003 <phase_3+176>
   0x0000000000400ffe <+171>:   callq  0x401564 <explode_bomb>
   0x0000000000401003 <+176>:   add    $0x18,%rsp
   0x0000000000401007 <+180>:   retq
End of assembler dump.
```

먼저 rsp에 24를 빼서 strack frame을 형성한 후 rcx, rdx를 적절한 위치에 저장한 뒤 scanf 함수를 호출하여 입력 값을 읽는 것을 알 수 있다. x/s를 통해 rsi 레지스터에 어떤 값의 주소가 저장되어 있는지 확인하였다.

```
(gdb) x/s 0x4027bd
0x4027bd:       "%d %d"
```

이를 통해 2개의 정수를 입력해야 한다는 것을 알았고, scanf를 호출한 뒤 rax 값이 1보다 작거나 같은 경우 bomb을 터트린다는 것을 알았고, 첫 번째 정수는 rdx에 저장되고, 두 번째 정수는 rcx에 저장된다는 것을 알았다. rsp+12의 값, 즉 rdx의 값이 7보다 클 경우 <+148>에서 bomb을 터트리는 것으로 보아 입력 string의 첫 번째 정수는 7보다 작거나 같아야 함을 알 수 있다. <+46>부터 rax값에 rsp+12의 주소에 들어있는 값을 저장하고 이를 index로 하여 indirect jump를 하는 것을 알 수 있었다.

- x/8xg (memory) command를 통해 jump table이 어떻게 구성되어 있는지 확인하였다.

```
(gdb) x/8xg 0x402520
0x402520:       0x0000000000400f93      0x0000000000400f8c
0x402530:       0x0000000000400f9f      0x0000000000400fab
0x402540:       0x0000000000400fb7      0x0000000000400fc3
0x402550:       0x0000000000400fcf      0x0000000000400fdb
```

rax 값, 즉 index가 0일 경우 <+64>행으로, 1일 경우 <+57>행으로, 2일 경우 <+76>행으로, 3일 경우 <+88>행으로, 4일 경우 <+100>행으로, 5일 경우 <+112>행으로, 6일 경우 <+124>행으로, 7일 경우 <+136>행으로 jump를 한다는 것을 알 수 있다. 각각의 경우 jump를 한 뒤 rax 값을 필요한 경우 초기화 시킨 뒤 특정 값을 더하거나 빼는 것을 알 수 있다.

- 모든 instruction을 수행한 뒤 rsp+12 주소에 들어있는 값과 5를 비교하여 만약 5보다 크면 bomb을 터트리는 것으로 보아 첫 번째 정수는 5보다 작거나 같아야 한다는 것을 알 수 있었다. 각각의 index의 경우 16진수를 10진수로 바꾸어 연산한 결과가 rax에 저장되어 있고, 그 값을 rsp+8의 값, 즉 rcx와 비교하여 같지 않으면 bomb을 터트리는 것을 알 수 있다. 이를 통해 첫 번째 정수에는 5보다 작거나 같은 정수가 들어오고 두 번째 정수에는 첫 번째 정수를 index로 하여 연산한 결과가 들어가야 한다는 것을 알게되었다.
- 가능한 첫 번째 값을 x1이라 하고, 두 번째 값을 x2라 할 때 연산 과정은 다음과 같다.

  x1 = 0인 경우: x2 = 826-449+354-947+947-947+947-947 = -216

x1 = 1인 경우: x2 = 0-449+354-947+947-947+947-947 = -1042

x1=2인 경우: x2 = 0+354-947+947-947+947-947 = -593

x1=3인 경우: x2= 0-947+947-947+947-947 = -947

x1=4인 경우: x2= 0+947-947+947-947 = 0

x1=5인 경우: x2= 0-947+947-947 = -947

따라서 가능한 입력 정수의 순서쌍은 다음과 같다.

(0 -216), (1 -1042), (2 -593), (3 -947), (4 0), (5 -947)

4. Phase4

- phase_4 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

```
(gdb) disas phase_4
Dump of assembler code for function phase_4:
   0x0000000000401040 <+0>:     sub    $0x18,%rsp
   0x0000000000401044 <+4>:     lea    0xc(%rsp),%rcx
   0x0000000000401049 <+9>:     lea    0x8(%rsp),%rdx
   0x000000000040104e <+14>:    mov    $0x4027bd,%esi
   0x0000000000401053 <+19>:    mov    $0x0,%eax
   0x0000000000401058 <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x000000000040105d <+29>:    cmp    $0x2,%eax
   0x0000000000401060 <+32>:    jne    0x40106e <phase_4+46>
   0x0000000000401062 <+34>:    mov    0xc(%rsp),%eax
   0x0000000000401066 <+38>:    sub    $0x2,%eax
   0x0000000000401069 <+41>:    cmp    $0x2,%eax
   0x000000000040106c <+44>:    jbe    0x401073 <phase_4+51>
   0x000000000040106e <+46>:    callq  0x401564 <explode_bomb>
   0x0000000000401073 <+51>:    mov    0xc(%rsp),%esi
   0x0000000000401077 <+55>:    mov    $0x9,%edi
   0x000000000040107c <+60>:    callq  0x401008 <func4>
   0x0000000000401081 <+65>:    cmp    0x8(%rsp),%eax
   0x0000000000401085 <+69>:    je     0x40108c <phase_4+76>
   0x0000000000401087 <+71>:    callq  0x401564 <explode_bomb>
   0x000000000040108c <+76>:    add    $0x18,%rsp
   0x0000000000401090 <+80>:    retq
End of assembler dump.
```

먼저 rsp에 24를 빼서 stack frame을 형성한 뒤 rsp+8, rsp+12에 각각 rdx, rcx 값을 저장한 뒤 rsi에 특정 메모리의 값을 저장하고 rax에 0을 넣은 후 scanf 함수를 호출하는 것을 알 수 있다. x/s를 통해 rsi에 저장된 값을 알아보면 다음과 같다.

```
(gdb) x/s 0x4027bd
0x4027bd:       "%d %d"
```

scanf 함수를 호출한 뒤 rax 값이 2가 아니면 bomb을 터트리는 것으로 보아 입력 string에는 2개의 정수가 포함되어야 함을 알 수 있고, 첫 번째 정수는 rdx에, 두 번째 정수는 rcx에 저장되는 것을 알 수 있다.

- scanf 함수를 호출한 뒤 <+34>부터 rax에 rsp+12에 저장된 메모리 안의 값, 즉 두 번째 입력 값을 저장하고 rax에 2를 빼준 뒤 2와 비교하여 2보다 크면 bomb을 터트리는 것을 알 수 있다. 이를 통해 두 번째 정수에는 4보다 작거나 같은 값이 들어와야 한다는 것을 알았다.

- <+51>부터는 rsi에는 rsp+12에 저장된 메모리 안의 수, 즉 두 번째 정수를 저장하고, rdi에 9을 저장한 뒤 func4 함수를 호출하는 것을 알 수 있다. 그리고 이 함수의 return 값을 rsp+8에 저장된 메모리의 수와 비교하여 같지 않으면 bomb을 터트리는 것으로 보아 첫 번째 정수에는 func4 함수에 9와 두 번째 정수를 인자로 한 것의 return 값이 들어가야 한다는 것을 알 수 있다.

- func4 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

```
(gdb) disas func4
Dump of assembler code for function func4:
   0x0000000000401008 <+0>:     push   %r12
   0x000000000040100a <+2>:     push   %rbp
   0x000000000040100b <+3>:     push   %rbx
   0x000000000040100c <+4>:     mov    %edi,%ebx
   0x000000000040100e <+6>:     test   %edi,%edi
   0x0000000000401010 <+8>:     jle    0x401036 <func4+46>
   0x0000000000401012 <+10>:    mov    %esi,%ebp
   0x0000000000401014 <+12>:    mov    %esi,%eax
   0x0000000000401016 <+14>:    cmp    $0x1,%edi
   0x0000000000401019 <+17>:    je     0x40103b <func4+51>
   0x000000000040101b <+19>:    lea    -0x1(%rdi),%edi
   0x000000000040101e <+22>:    callq  0x401008 <func4>
   0x0000000000401023 <+27>:    lea    (%rax,%rbp,1),%r12d
   0x0000000000401027 <+31>:    lea    -0x2(%rbx),%edi
   0x000000000040102a <+34>:    mov    %ebp,%esi
   0x000000000040102c <+36>:    callq  0x401008 <func4>
   0x0000000000401031 <+41>:    add    %r12d,%eax
   0x0000000000401034 <+44>:    jmp    0x40103b <func4+51>
   0x0000000000401036 <+46>:    mov    $0x0,%eax
   0x000000000040103b <+51>:    pop    %rbx
   0x000000000040103c <+52>:    pop    %rbp
   0x000000000040103d <+53>:    pop    %r12
   0x000000000040103f <+55>:    retq
End of assembler dump.
```

함수를 보았을 때 rdi, 즉 첫 번째 인자가 0보다 작거나 같은 경우 0을 return하고, 1인 경우 1을 return하고 그렇지 않은 경우 rdi에 1을 빼서 다시 func4를 호출하는 것을 알 수 있다. 그런 다음 r12에 함수의 return 값에 rbp, 즉 두 번째 인자를 더하여 저장하고, 첫 번째 인자

에 2를 뺀 뒤 func4 함수를 한 번 더 호출하는 것을 알 수 있다. 그런 다음 return 값에 r12를 더하는 것을 알 수 있다. rdi가 2 이상일 경우를 점화식의 형태로 나타내면 다음과 같다.

func4(di, si) = func4(di-1, si) + func4(di-2, si) + si

- 우리가 구해야 하는 것은 func4(9, si)이므로 이를 식으로 표현하면

  func4(9, si) = func4(8, si) + func4(7, si) + si

  위에서 종료 조건을 통해 func4(1, si) = si, func4(0, si) = 0이라는 것을 알기 때문에 이를 이용하여 func4(9, si) 값을 계산하면 그 값은 88*si가 된다.

- func4 함수 호출이 종료된 후 return 값을 첫 번째 인자와 같은지 비교하기 때문에 두 번째 정수를 x라 할 때 첫 번째 정수는 88*x의 형태여야 한다. 따라서 가능한 입력 정수의 순서쌍은 다음과 같다.

  (176 2), (264 3), (352 4)

  이때 두 번째 정수가 1인 경우는 불가능하다. <phase_4+38>에서 두 번째 정수를 rax에 넣은 뒤 rax-2를 한 뒤 2와 비교하기 때문이다.

5. Phase5

- phase_5 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

```
(gdb) disas phase_5
Dump of assembler code for function phase_5:
   0x0000000000401091 <+0>:     push   %rbx
   0x0000000000401092 <+1>:     sub    $0x10,%rsp
   0x0000000000401096 <+5>:     mov    %rdi,%rbx
   0x0000000000401099 <+8>:     callq  0x4012e1 <string_length>
   0x000000000040109e <+13>:    cmp    $0x6,%eax
   0x00000000004010a1 <+16>:    je     0x4010e2 <phase_5+81>
   0x00000000004010a3 <+18>:    callq  0x401564 <explode_bomb>
   0x00000000004010a8 <+23>:    jmp    0x4010e2 <phase_5+81>
   0x00000000004010aa <+25>:    movzbl (%rbx,%rax,1),%edx
   0x00000000004010ae <+29>:    and    $0xf,%edx
   0x00000000004010b1 <+32>:    movzbl 0x402560(%rdx),%edx
   0x00000000004010b8 <+39>:    mov    %dl,(%rsp,%rax,1)
   0x00000000004010bb <+42>:    add    $0x1,%rax
   0x00000000004010bf <+46>:    cmp    $0x6,%rax
   0x00000000004010c3 <+50>:    jne    0x4010aa <phase_5+25>
   0x00000000004010c5 <+52>:    movb   $0x0,0x6(%rsp)
   0x00000000004010ca <+57>:    mov    $0x402516,%esi
   0x00000000004010cf <+62>:    mov    %rsp,%rdi
   0x00000000004010d2 <+65>:    callq  0x4012fe <strings_not_equal>
   0x00000000004010d7 <+70>:    test   %eax,%eax
   0x00000000004010d9 <+72>:    je     0x4010e9 <phase_5+88>
   0x00000000004010db <+74>:    callq  0x401564 <explode_bomb>
   0x00000000004010e0 <+79>:    jmp    0x4010e9 <phase_5+88>
   0x00000000004010e2 <+81>:    mov    $0x0,%eax
   0x00000000004010e7 <+86>:    jmp    0x4010aa <phase_5+25>
   0x00000000004010e9 <+88>:    add    $0x10,%rsp
   0x00000000004010ed <+92>:    pop    %rbx
   0x00000000004010ee <+93>:    retq
End of assembler dump.
```

먼저 rsp에 16을 빼서 stack frame을 형성한 뒤, rbx에 rdi의 값을 넣은 후 string_length 함수를 호출하는 것을 알 수 있다. string_length 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

```
(gdb) disas string_length
Dump of assembler code for function string_length:
   0x00000000004012e1 <+0>:     cmpb   $0x0,(%rdi)
   0x00000000004012e4 <+3>:     je     0x4012f8 <string_length+23>
   0x00000000004012e6 <+5>:     mov    %rdi,%rdx
   0x00000000004012e9 <+8>:     add    $0x1,%rdx
   0x00000000004012ed <+12>:    mov    %edx,%eax
   0x00000000004012ef <+14>:    sub    %edi,%eax
   0x00000000004012f1 <+16>:    cmpb   $0x0,(%rdx)
   0x00000000004012f4 <+19>:    jne    0x4012e9 <string_length+8>
   0x00000000004012f6 <+21>:    repz retq
   0x00000000004012f8 <+23>:    mov    $0x0,%eax
   0x00000000004012fd <+28>:    retq
End of assembler dump.
```

loop문을 사용하여 입력 받은 string의 길이를 구하여 return하는 함수임을 알 수 있다.

- <phase_5+13>부터 길이가 저장된 rax를 6과 비교하여 같지 않으면 bomb을 터트리는 것으로 보아 입력 해야하는 string의 길이가 6이라는 것을 알 수 있다.
- rax에 0을 넣은 후 <+25>부터 loop에 들어가는 것을 알 수 있다. rdx에 rbx값, 즉 입력 string의 주소에 rax를 더한 값을 넣고 이를 0xf와 and 연산을 한 값을 다시 rdx에 넣은 후 <+32>에서 특정 메모리 주소에 rdx를 더한 값을 다시 rdx에 넣는다. 그리고 이를 rsp+rax의 주소에 넣는 것을 알 수 있다. 그런 다음 rax를 6이 될 때까지 1씩 더해주면서 반복해주는 것을 알 수 있다. x/s를 통해 메모리 주소의 값을 확인해보면 다음과 같다.

```
(gdb) x/s 0x402560
0x402560 <array.3159>:  "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
```

이를 통해 함수를 해석해보면 길이가 6인 string을 입력 받아 string의 한 자리씩 0xf와 and 연산을 한 값을 n이라 하면, 위 string의 n+1번째 수가 rsp+rax에 저장되어 결국 rsp에 새로운 string이 생성되는 것을 알 수 있다. 이때 string의 주소는 string의 첫 자리의 주소와 같고, rsp+6의 값에 0을 저장하여 string을 형성하는 것을 알 수 있다.

- 아스키코드를 통해 각 문자에 해당하는 수를 확인하여 0xf와 연산하면 아스키코드를 16으로 나눈 나머지가 나오는 것을 알 수 있다. 예를 들어 a는 아스키코드로 97이고 이를 0xf와 and 연산하면 결과는 1이 된다. 그리고 이는 위의 string의 2번째 수인 a를 rsp에 저장하게 만든다.
- rsp에 string을 형성한 뒤 rdi에 넣고, rsi에 특정 값을 넣은 후 strings_not_equal 함수를 실행시켜 두 string이 같은지 확인한다. 이 함수는 phase_1에서 살펴본 바와 같이 두 string이 같으면 0, 다르면 1을 return하는 함수이다. 그리고 그 값이 0인지 test를 통해

확인하여 0이 아니면 bomb을 터트리는 것을 확인할 수 있다.

- x/s를 통해 rsi에 저장된 값을 확인해보면 다음과 같다.

```
(gdb) x/s 0x402516
0x402516:       "flyers"
```

- rsp에 저장된 string이 flyers가 되도록 역추적하여 원래의 string이 무엇인지 찾아야 한다. 예를 들어 flyers의 첫 번째인 f는 위의 0x452560에 저장된 string에서 10번째에 해당하므로 0xf와 and 연산의 결과가 9가 되는 문자를 찾으면 아스키코드가 105인 i가 나온다. 이와 같은 방법으로 나머지 string을 완성시켜주면 입력 string에는 "ionefg"가 들어가야 한다는 것을 알 수 있다.

6. Phase6

- phase_6 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

```
(gdb) disas phase_6
Dump of assembler code for function phase_6:
   0x00000000004010ef <+0>:     push   %r13
   0x00000000004010f1 <+2>:     push   %r12
   0x00000000004010f3 <+4>:     push   %rbp
   0x00000000004010f4 <+5>:     push   %rbx
   0x00000000004010f5 <+6>:     sub    $0x58,%rsp
   0x00000000004010f9 <+10>:    lea    0x30(%rsp),%rsi
   0x00000000004010fe <+15>:    callq  0x40159a <read_six_numbers>
   0x0000000000401103 <+20>:    lea    0x30(%rsp),%r13
   0x0000000000401108 <+25>:    mov    $0x0,%r12d
   0x000000000040110e <+31>:    mov    %r13,%rbp
   0x0000000000401111 <+34>:    mov    0x0(%r13),%eax
   0x0000000000401115 <+38>:    sub    $0x1,%eax
   0x0000000000401118 <+41>:    cmp    $0x5,%eax
   0x000000000040111b <+44>:    jbe    0x401122 <phase_6+51>
   0x000000000040111d <+46>:    callq  0x401564 <explode_bomb>
   0x0000000000401122 <+51>:    add    $0x1,%r12d
   0x0000000000401126 <+55>:    cmp    $0x6,%r12d
   0x000000000040112a <+59>:    jne    0x401133 <phase_6+68>
   0x000000000040112c <+61>:    mov    $0x0,%esi
   0x0000000000401131 <+66>:    jmp    0x401175 <phase_6+134>
   0x0000000000401133 <+68>:    mov    %r12d,%ebx
   0x0000000000401136 <+71>:    movslq %ebx,%rax
   0x0000000000401139 <+74>:    mov    0x30(%rsp,%rax,4),%eax
   0x000000000040113d <+78>:    cmp    %eax,0x0(%rbp)
   0x0000000000401140 <+81>:    jne    0x401147 <phase_6+88>
   0x0000000000401142 <+83>:    callq  0x401564 <explode_bomb>
   0x0000000000401147 <+88>:    add    $0x1,%ebx
   0x000000000040114a <+91>:    cmp    $0x5,%ebx
   0x000000000040114d <+94>:    jle    0x401136 <phase_6+71>
   0x000000000040114f <+96>:    add    $0x4,%r13
   0x0000000000401153 <+100>:   jmp    0x40110e <phase_6+31>
   0x0000000000401155 <+102>:   mov    0x8(%rdx),%rdx
   0x0000000000401159 <+106>:   add    $0x1,%eax
   0x000000000040115c <+109>:   cmp    %ecx,%eax
   0x000000000040115e <+111>:   jne    0x401155 <phase_6+102>
```

```
   0x0000000000401160 <+113>:   jmp    0x401167 <phase_6+120>
   0x0000000000401162 <+115>:   mov    $0x6042f0,%edx
   0x0000000000401167 <+120>:   mov    %rdx,(%rsp,%rsi,2)
   0x000000000040116b <+124>:   add    $0x4,%rsi
   0x000000000040116f <+128>:   cmp    $0x18,%rsi
   0x0000000000401173 <+132>:   je     0x40118a <phase_6+155>
   0x0000000000401175 <+134>:   mov    0x30(%rsp,%rsi,1),%ecx
   0x0000000000401179 <+138>:   cmp    $0x1,%ecx
   0x000000000040117c <+141>:   jle    0x401162 <phase_6+115>
   0x000000000040117e <+143>:   mov    $0x1,%eax
   0x0000000000401183 <+148>:   mov    $0x6042f0,%edx
   0x0000000000401188 <+153>:   jmp    0x401155 <phase_6+102>
   0x000000000040118a <+155>:   mov    (%rsp),%rbx
   0x000000000040118e <+159>:   lea    0x8(%rsp),%rax
   0x0000000000401193 <+164>:   lea    0x30(%rsp),%rsi
   0x0000000000401198 <+169>:   mov    %rbx,%rcx
   0x000000000040119b <+172>:   mov    (%rax),%rdx
   0x000000000040119e <+175>:   mov    %rdx,0x8(%rcx)
   0x00000000004011a2 <+179>:   add    $0x8,%rax
   0x00000000004011a6 <+183>:   cmp    %rsi,%rax
   0x00000000004011a9 <+186>:   je     0x4011b0 <phase_6+193>
   0x0000000000401ab <+188>:    mov    %rdx,%rcx
   0x0000000000401ae <+191>:    jmp    0x40119b <phase_6+172>
   0x00000000004011b0 <+193>:   movq   $0x0,0x8(%rdx)
   0x00000000004011b8 <+201>:   mov    $0x5,%ebp
   0x00000000004011bd <+206>:   mov    0x8(%rbx),%rax
   0x00000000004011c1 <+210>:   mov    (%rax),%eax
   0x00000000004011c3 <+212>:   cmp    %eax,(%rbx)
   0x00000000004011c5 <+214>:   jle    0x4011cc <phase_6+221>
   0x00000000004011c7 <+216>:   callq  0x401564 <explode_bomb>
   0x00000000004011cc <+221>:   mov    0x8(%rbx),%rbx
   0x00000000004011d0 <+225>:   sub    $0x1,%ebp
   0x00000000004011d3 <+228>:   jne    0x4011bd <phase_6+206>
   0x0000000000401d5 <+230>:    add    $0x58,%rsp
   0x0000000000401d9 <+234>:    pop    %rbx
   0x0000000000401da <+235>:    pop    %rbp
```

```
   0x00000000004011db <+236>:   pop    %r12
   0x00000000004011dd <+238>:   pop    %r13
   0x00000000004011df <+240>:   retq
End of assembler dump.
```

먼저 rsp에 88을 빼서 stack frame을 형성한 것을 알 수 있다. read_six_numbers는 phase_2에서 확인해본 바와 같다. rsi에 rsp+48의 주소 값을 넣기 때문에 rsp+48부터 4byte씩 6개의 정수를 저장한다는 것을 알 수 있다.

- <+31>부터 <+100>까지 이중 loop를 돌리고 있다는 것을 알 수 있는데, 입력 받은 6개의 정수를 0번째부터 5번째라 할 때, 큰 loop에서는 rax에 n번째 정수 값을 넣어 <+38>부터 <+44>까지 그 값에 1을 뺀 값을 5보다 작거나 같은지 확인하고 있다. 즉, 입력 받은 6개의 정수가 모두 1이상 5이하의 정수라는 것을 알 수 있다. 그리고 0~5번째까지 모두 확인한 후에 r12에 들어있는 값이 6이 되었을 때 loop를 탈출하는 것을 알 수 있다.

- <+71>부터 <+94>까지는 큰 loop 안에 작은 loop를 형성하고 있는데, rbp에 n번째 수를 저장하고 있다면, rax에는 n+1번째부터 마지막까지의 수를 저장하도록 하여 두 수가 같을 경우 bomb을 터트린다. 즉, 두 loop에서 확인하고 있는 것은 입력 값이 서로 다른 6개의 1이상 6이하의 정수라는 것이다. 따라서 입력해야하는 6개의 수는 1, 2, 3, 4, 5, 6의 적절한 순서 조합이라는 것을 알았다.

- 앞의 loop에서 빠져나갈 때 <+134>행으로 이동하게 된다. <+102>부터 <+153>까지도 하나의 큰 loop를 형성하고 있는데, 작동 방식을 살펴보면 다음과 같다. <+134>에서 rcx에 48+(rsp+1*rsi) 연산을 통해 rcx에 입력한 n번째 정수를 저장한다는 것을 알 수 있다. rsi는 <+124>에서 4씩 더하고 있으므로 index의 역할을 한다고 할 수 있다. rcx가 1일 경우 <+115>에서 rdx에 0x6042f0에 저장된 값을 넣고, 이를 다시 rsp+2*rsi의 주소에 저장하는 것을 알 수 있다. rsi가 4씩 변하므로 rsp부터는 8byte 크기만큼의 pointer array를 형성하고 있다는 것을 알 수 있다.

- 0x6042f0에는 어떤 값이 저장되어 있는지 다음과 같이 확인해 보았다.

```
(gdb) x/24gw 0x6042f0
0x6042f0 <node1>:       223     1       6308608 0
0x604300 <node2>:       771     2       6308624 0
0x604310 <node3>:       742     3       6308640 0
0x604320 <node4>:       130     4       6308656 0
0x604330 <node5>:       993     5       6308672 0
0x604340 <node6>:       119     6       0       0
```

  node1부터 6까지에는 세 자리의 정수와 1부터 6까지의 정수가 저장되어 있음을 확인할 수 있었다. rcx가 1일 경우, node1 pointer가 pointer array에 저장되고, 그 외의 경우에는 <+102>부터 <+111>까지 loop를 돌려 해당 숫자의 node pointer를 pointer array에 저장한다는 것을 알 수 있다. 즉, n번째 수가 X인 경우 nodeX를 pointer array의 n번째에 저장한다. <+128>과 <+132>에서 rsi 값을 24와 비교하여 같으면 loop를 탈출하는 것을 알 수 있다.

- loop를 탈출한 다음 rbx, rax, rsi를 각각 initialize 해준 뒤, <+172>부터 <+191>까지 loop에 들어가는 것을 알 수 있다. rcx, rdx에는 각각 array의 n번째, n+1번째 노드의 값이 들어가고, rax에 n+1번째 노드의 주소가 들어가도록 하여 <+175>행에서 해당 주소에 각 노드의 세 자리 정수 값이 저장되도록 하였다. rsi에는 array의 마지막 주소의 다음 주소로 하여 loop가 끝날 수 있도록 했다.

- 그런 다음 <+201>에서 rbp를 5로 초기화 해준 뒤, <+206>부터 <+228>까지 loop에 들어간다. loop를 실행할 때마다 rbp 값을 1씩 감소시킨 뒤 0이 되면 loop를 탈출한다. 해당 loop에서 rbx는 array의 n번째의 주소를 의미하고 rax는 n+1번째에 들어있는 값을 의미한다. rbx에 저장된 메모리에 들어있는 값과 rax와 비교하여 rbx에 저장된 메모리에 들어있는 값이 rax값보다 작으면 bomb을 터트린다. 이를 통해 node 안에 저장된 수들이 점점 커지는 순서로 node를 배열해야 한다는 것을 알았다. node1~6까지 저장된 수는 각각 223, 771, 742, 130, 993, 119이므로 해당 수들을 커지는 순으로 배열하기 위한 입력 값은 "6 4 1 3 2 5"임을 알았다.

7. Secret Phase

- main 함수에서 호출하는 phase_defused 함수를 disassemble 하였을 때 <+107> 행에 secret_phase가 존재한다는 것을 알 수 있었다.

```
(gdb) disas phase_defused
Dump of assembler code for function phase_defused:
   0x0000000000401702 <+0>:     sub    $0x68,%rsp
   0x0000000000401706 <+4>:     mov    $0x1,%edi
   0x000000000040170b <+9>:     callq  0x4014a0 <send_msg>
   0x0000000000401710 <+14>:    cmpl   $0x6,0x203085(%rip)        # 0x60479c <num_input_strings>
   0x0000000000401717 <+21>:    jne    0x401786 <phase_defused+132>
   0x0000000000401719 <+23>:    lea    0x10(%rsp),%r8
   0x000000000040171e <+28>:    lea    0x8(%rsp),%rcx
   0x0000000000401723 <+33>:    lea    0xc(%rsp),%rdx
   0x0000000000401728 <+38>:    mov    $0x402807,%esi
   0x000000000040172d <+43>:    mov    $0x6048b0,%edi
   0x0000000000401732 <+48>:    mov    $0x0,%eax
   0x0000000000401737 <+53>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x000000000040173c <+58>:    cmp    $0x3,%eax
   0x000000000040173f <+61>:    jne    0x401772 <phase_defused+112>
   0x0000000000401741 <+63>:    mov    $0x402810,%esi
   0x0000000000401746 <+68>:    lea    0x10(%rsp),%rdi
   0x000000000040174b <+73>:    callq  0x4012fe <strings_not_equal>
   0x0000000000401750 <+78>:    test   %eax,%eax
   0x0000000000401752 <+80>:    jne    0x401772 <phase_defused+112>
   0x0000000000401754 <+82>:    mov    $0x402668,%edi
   0x0000000000401759 <+87>:    callq  0x400b40 <puts@plt>
   0x000000000040175e <+92>:    mov    $0x402690,%edi
   0x0000000000401763 <+97>:    callq  0x400b40 <puts@plt>
   0x0000000000401768 <+102>:   mov    $0x0,%eax
   0x000000000040176d <+107>:   callq  0x40121e <secret_phase>
   0x0000000000401772 <+112>:   mov    $0x4026c8,%edi
   0x0000000000401777 <+117>:   callq  0x400b40 <puts@plt>
   0x000000000040177c <+122>:   mov    $0x4026f8,%edi
   0x0000000000401781 <+127>:   callq  0x400b40 <puts@plt>
   0x0000000000401786 <+132>:   add    $0x68,%rsp
   0x000000000040178a <+136>:   retq
End of assembler dump.
```

- <+38>과 <+43>에서 rsi에 들어있는 주소에 어떤 값이 들어있는지 확인하면 다음과 같다.

```
(gdb) x/s 0x402807
0x402807:       "%d %d %s"
```

그리고 rdi에 들어있는 주소의 값은 phase_4에서 입력한 string이 나왔다. scanf를 호출한 뒤에 <+58>에서 rax 값이 3이면 rsi에 특정 메모리 주소를 저장하고 strings_not_equal 함수를 호출하는 것을 알았다. 해당 메모리에 어떤 값이 저장 되어있는지 확인하면 다음과 같다.

```
(gdb) x/s 0x402810
0x402810:       "DrEvil"
```

그리고 두 string이 같으면 secret_phase 함수를 호출한다. 이를 통해 phase_4의 입력을 받을 때 "DrEvil"을 추가로 입력하면 secret_phase가 호출된다는 것을 알았다.

- secret_phase 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

```
(gdb) disas secret_phase
Dump of assembler code for function secret_phase:
   0x000000000040121e <+0>:     push   %rbx
   0x000000000040121f <+1>:     callq  0x4015dc <read_line>
   0x0000000000401224 <+6>:     mov    $0xa,%edx
   0x0000000000401229 <+11>:    mov    $0x0,%esi
   0x000000000040122e <+16>:    mov    %rax,%rdi
   0x0000000000401231 <+19>:    callq  0x400c00 <strtol@plt>
   0x0000000000401236 <+24>:    mov    %rax,%rbx
   0x0000000000401239 <+27>:    lea    -0x1(%rax),%eax
   0x000000000040123c <+30>:    cmp    $0x3e8,%eax
   0x0000000000401241 <+35>:    jbe    0x401248 <secret_phase+42>
   0x0000000000401243 <+37>:    callq  0x401564 <explode_bomb>
   0x0000000000401248 <+42>:    mov    %ebx,%esi
   0x000000000040124a <+44>:    mov    $0x604110,%edi
   0x000000000040124f <+49>:    callq  0x4011e0 <fun7>
   0x0000000000401254 <+54>:    cmp    $0x7,%eax
   0x0000000000401257 <+57>:    je     0x40125e <secret_phase+64>
   0x0000000000401259 <+59>:    callq  0x401564 <explode_bomb>
   0x000000000040125e <+64>:    mov    $0x4024f0,%edi
   0x0000000000401263 <+69>:    callq  0x400b40 <puts@plt>
   0x0000000000401268 <+74>:    callq  0x401702 <phase_defused>
   0x000000000040126d <+79>:    pop    %rbx
   0x000000000040126e <+80>:    retq
End of assembler dump.
```

- 먼저 read_line 함수를 호출하여 입력 받은 string을 rdi에 저장하고, rsi, rdx에 각각 0, 10을 저장한 뒤 strtol 함수를 호출한다. strtol 함수는 string을 long 타입의 정수로 변환해주는 함수이다. 변환해준 정수를 rbx에 저장하는 것을 알 수 있다.

- rax에 1을 빼준 뒤 1000과 비교한 뒤 1000보다 크면 bomb을 터트리는 것으로 보아 입력하는 정수 값은 1이상 1001이하라는 것을 알았다.

- 그 다음 rdi에 특정 메모리 주소를 넣고 rsi에 변환된 정수를 넣어 fun7 함수를 호출하는 것을 알았다. 그 다음 rax를 7과 비교하여 같지 않으면 bomb을 터트리는 것으로 보아 fun7 함수의 return 값이 7이 되도록 하는 적절한 정수를 입력해야 한다는 것을 알았다.

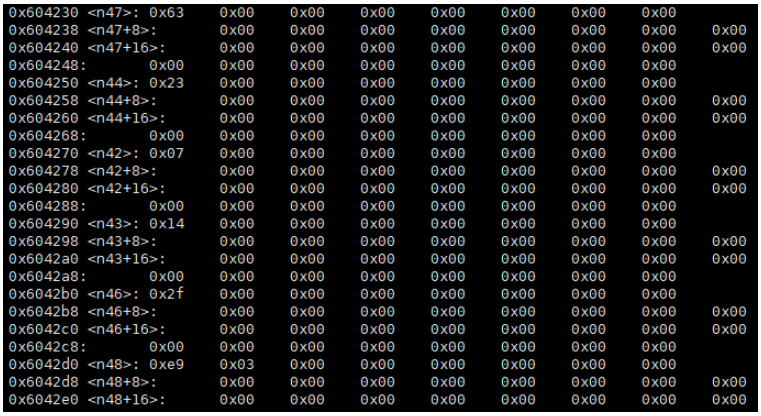
- fun7 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

```
(gdb) disas fun7
Dump of assembler code for function fun7:
   0x00000000004011e0 <+0>:     sub    $0x8,%rsp
   0x00000000004011e4 <+4>:     test   %rdi,%rdi
   0x00000000004011e7 <+7>:     je     0x401214 <fun7+52>
   0x00000000004011e9 <+9>:     mov    (%rdi),%edx
   0x00000000004011eb <+11>:    cmp    %esi,%edx
   0x00000000004011ed <+13>:    jle    0x4011fc <fun7+28>
   0x00000000004011ef <+15>:    mov    0x8(%rdi),%rdi
   0x00000000004011f3 <+19>:    callq  0x4011e0 <fun7>
   0x00000000004011f8 <+24>:    add    %eax,%eax
   0x00000000004011fa <+26>:    jmp    0x401219 <fun7+57>
   0x00000000004011fc <+28>:    mov    $0x0,%eax
   0x0000000000401201 <+33>:    cmp    %esi,%edx
   0x0000000000401203 <+35>:    je     0x401219 <fun7+57>
   0x0000000000401205 <+37>:    mov    0x10(%rdi),%rdi
   0x0000000000401209 <+41>:    callq  0x4011e0 <fun7>
   0x000000000040120e <+46>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401212 <+50>:    jmp    0x401219 <fun7+57>
   0x0000000000401214 <+52>:    mov    $0xffffffff,%eax
   0x0000000000401219 <+57>:    add    $0x8,%rsp
   0x000000000040121d <+61>:    retq
End of assembler dump.
```

첫 번째 인자가 0일 경우 rax에 0xffffffff을 넣어 return하는 것을 알 수 있다. rdx에 첫 번째 인자로 받은 주소에 들어있는 값을 넣고, rdx가 두 번째 인자보다 클 경우 rdi에 rdi의 주소에 8을 더한 주소에 들어있는 값을 넣은 뒤 다시 fun7 함수를 호출하고, return 값에 2를 곱한다. 작거나 같을 경우에는 rax에 0을 넣고 rdx와 rsi가 같으면 함수를 return, 아니면 rdi에 16을 더한 주소에 들어있는 값을 넣고 fun7 함수를 호출한 뒤 return 값에 2를 곱하고 1을 더하는 연산을 수행한다.

- fun7 함수의 첫 번째 인자로 들어오는 0x604110에 어떤 값이 들어있는지 확인하였다.

```
(gdb) x/600x 0x604110
0x604110 <n1>:  0x24    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604118 <n1+8>:        0x30    0x41    0x60    0x00    0x00    0x00    0x00    0x00
0x604120 <n1+16>:       0x50    0x41    0x60    0x00    0x00    0x00    0x00    0x00
0x604128:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604130 <n21>: 0x08    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604138 <n21+8>:       0xb0    0x41    0x60    0x00    0x00    0x00    0x00    0x00
0x604140 <n21+16>:      0x70    0x41    0x60    0x00    0x00    0x00    0x00    0x00
0x604148:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604150 <n22>: 0x32    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604158 <n22+8>:       0x90    0x41    0x60    0x00    0x00    0x00    0x00    0x00
0x604160 <n22+16>:      0xd0    0x41    0x60    0x00    0x00    0x00    0x00    0x00
0x604168:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604170 <n32>: 0x16    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604178 <n32+8>:       0x90    0x42    0x60    0x00    0x00    0x00    0x00    0x00
0x604180 <n32+16>:      0x50    0x42    0x60    0x00    0x00    0x00    0x00    0x00
0x604188:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604190 <n33>: 0x2d    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604198 <n33+8>:       0xf0    0x41    0x60    0x00    0x00    0x00    0x00    0x00
0x6041a0 <n33+16>:      0xb0    0x42    0x60    0x00    0x00    0x00    0x00    0x00
0x6041a8:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6041b0 <n31>: 0x06    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6041b8 <n31+8>:       0x10    0x42    0x60    0x00    0x00    0x00    0x00    0x00
0x6041c0 <n31+16>:      0x70    0x42    0x60    0x00    0x00    0x00    0x00    0x00
0x6041c8:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6041d0 <n34>: 0x6b    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6041d8 <n34+8>:       0x30    0x42    0x60    0x00    0x00    0x00    0x00    0x00
0x6041e0 <n34+16>:      0xd0    0x42    0x60    0x00    0x00    0x00    0x00    0x00
0x6041e8:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6041f0 <n45>: 0x28    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x6041f8 <n45+8>:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604200 <n45+16>:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604208:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604210 <n41>: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604218 <n41+8>:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604220 <n41+16>:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604228:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```



메모리에는 먼저 <nX>에 정수가 들어있고, <nX+8>과 <nX+16>에 주소가 little endian의 방식으로 적혀 있는 것을 알 수 있다. <nX>를 하나의 노드라고 하면 이 메모리에는 tree 형식으로 값이 저장되어 있다.

- 메모리에 들어있는 값과 fun7 함수를 종합적으로 볼 때, fun7 함수는 두 번째 인자와 노드의 값을 비교하여 다음 노드의 값을 두 번째 인자로 하여 다시 fun7 함수를 호출하는 것을 반복하면서 위에서 언급한 규칙대로 return 값을 변화시킨다.

- fun7 함수의 최종 return 값이 7이 되어야 하는데, 이 값은 0, 2*0+1 = 1, 2*1+1 = 3, 2*3+1 = 7의 과정을 거쳐 만들 수 있다. 이를 위해서는 tree에서 오른쪽으로 이동하는 동작을 3번 수행해야 하므로 두 번째 인자로는 <n48> 노드의 값이 들어있어야 한다.

- <n48> 노드의 수 또한 little endian의 방식으로 적혀 있으므로 해당하는 수는 '0x3e9'이고, 이는 10진수로 1001이다. 따라서 secret_phase의 입력 값은 1001임을 알았다.