

컴퓨터시스템개론 보고서

2016310936 우승민

gdb를 이용해서 assembly 코드를 main함수를 분석한다.

```
400f18: e8 c2 09 00 00    callq 4018df <read_line>
400f1d: 48 89 c7          mov    %rax,%rdi
400f20: e8 98 00 00 00    callq 400fbd <phase_1>
400f25: e8 db 0a 00 00    callq 401a05 <phase_defused>
```

위 4코드는 read_line에서 내가 입력해준 한 줄에 있는 값을 받고 phase1을 실행하는 것을 알 수 있다. phase_1을 call한 후 phase_defused를 call하는 것을 보면 phase_1 함수안에서 실패했을 때 폭발하는게 있고 통과하면 그냥 나오는 것으로 유추할 수 있다.

Phase_defused를 disassemble 해보면

```
401a23: 83 3d a2 35 20 00 06 cmpl    $0x6,0x2035a2(%rip)
401a2a: 75 6d             jne    401a99 <phase_defused+0x94>
```

6단계인지를 확인하는 코드가 있다. 6단계가 맞으면 jmp를 하지 않는데 아래부분에 secret phase를 부르는 부분이 있다.

```
401a80: e8 1f f9 ff ff    callq 4013a4 <secret_phase>
```

그 사이의 코드를 보면

```
401a3b: be 08 2c 40 00    mov    $0x402c08,%esi
401a54: be 11 2c 40 00    mov    $0x402c11,%esi
```

(gdb) x/s 0x402c08

0x402c08: "%d %d %s"

(gdb) x/s 0x402c11

0x402c11: "DrEvil"

나중에 phase4를 보면 %d %d를 입력하는 부분이 있는데 여기서는 %s 하나를 더 확인하여 DrEvil이라는 글자를 확인한다. 이게 secret code를 의미하는 것 같다.

이제 하나씩 살펴보도록 한다.

Phase_1

```
400fc1: be 20 28 40 00    mov    $0x402820,%esi
400fc6: e8 c3 04 00 00    callq  40148e <strings_not_equal>
400fcb: 85 c0             test    %eax,%eax
400fcd: 74 05            je     400fd4 <phase_1+0x17>
400fcf: e8 96 08 00 00    callq  40186a <explode_bomb>
```

여기서 test와 je를 사용하여 bomb을 call할지 통과시킬지 정하는 것을 볼 수 있다.

Je는 값이 test값이 1일 때 jump를 한다. 우리는 bomb을 피해야 하므로 test값이 1이 나와야하고 test는 and 연산자이므로 즉 **strings_not_equal**이라는 함수의 return값이 1이 나와야 할 것이다. 여기서 **strings_not_equal** 함수로 넘어가기 전에 그 위에 코드에서 **\$0x402820**을 esi에 넣어주는데 이 값을 읽어보면

(gdb) x/s 0x402820

0x402820: "Why make trillions when we could make... billions?"

이 문장이 들어있다. **Strings_not_equal** 함수의 code를 보면

```
4014db: ba 00 00 00 00    mov    $0x0,%edx
4014e0: eb 0c            jmp     4014ee <strings_not_equal+0x60>
4014e2: ba 01 00 00 00    mov    $0x1,%edx
4014e7: eb 05            jmp     4014ee <strings_not_equal+0x60>
4014e9: ba 01 00 00 00    mov    $0x1,%edx
4014ee: 89 d0            mov    %edx,%eax
```

%edx값이 %eax으로 옮겨가고 retutn한다는 것을 알 수 있다. 여기서 1이 될지 0이 될지는 함수 내부에서 jump가 어디로 되냐에 따라 달라지고 아까 esi에 넣어준 문장과 내가 입력한 값을 비교하여 맞으면 1로 되도록 해준다. 따라서 1번의 정답은 0x402820에 들어 있던 **"Why make trillions when we could make... billions?"** 이 문장이 정답이다.

Phase_2

코드 초반부를 보면 아래와 같은 함수를 호출하는 것을 볼 수 있다.

```
400ff2: e8 a9 08 00 00    callq 4018a0 <read_six_numbers>
```

이 코드에서 **read_six_numbers** 라는 함수를 call 한다. 이 함수 안으로 들어가보면

```
4018c2: b8 00 00 00 00    mov    $0x0,%eax
4018c7: e8 04 f4 ff ff    callq 400cd0 <__isoc99_sscanf@plt>
4018cc: 48 83 c4 10       add    $0x10,%rsp
4018d0: 83 f8 05          cmp    $0x5,%eax
4018d3: 7f 05            jg     4018da <read_six_numbers+0x3a>
4018d5: e8 90 ff ff ff    callq 40186a <explode_bomb>
```

위 코드에서 %eax값에 0을 넣어주고 input값을 받은 후 %eax를 5와 비교하여 커야 bomb을 피해가는 것을 볼 수 있다. 따라서 read_six_nubmers 함수는 **6개 이상의 숫자를 받는 함수임**을 알 수 있다.

```
400ff7: 83 3c 24 00       cmpl   $0x0,(%rsp)
400ffb: 75 07            jne    401004 <phase_2+0x2b>
400ffd: 83 7c 24 04 01    cmpl   $0x1,0x4(%rsp)
401002: 74 05            je     401009 <phase_2+0x30>
401004: e8 61 08 00 00    callq 40186a <explode_bomb>
```

이 부분을 보면 입력해준 값의 첫번째와 0을 비교하고 두번째와 1을 비교하는 것을 볼 수 있다. 여기서 **첫번째 값과 두번째 값은 0과 1을 넣어야 하는 것**을 알 수 있다.

```
401009: 48 89 e3          mov    %rsp,%rbx
40100c: 48 8d 6c 24 10    lea    0x10(%rsp),%rbp
401011: 8b 43 04          mov    0x4(%rbx),%eax
401014: 03 03            add    (%rbx),%eax
401016: 39 43 08          cmp    %eax,0x8(%rbx)
401019: 74 05            je     401020 <phase_2+0x47>
40101b: e8 4a 08 00 00    callq 40186a <explode_bomb>
```

%rbx에도 %rsp와 같은 주소를 옮겨주고 %eax에는 이에 따라 2번째 값이 들어간다. %rbp에는 마지막으로 입력해준 값이 들어간다. 그 후 **첫번째 값과 두번째 값의 합을 세번째 값과 비교한다**. 같아야 하므로 세번째 값은 0과 1을 더한 1임을 알 수 있다.

```

401020: 48 83 c3 04      add    $0x4,%rbx
401024: 48 39 eb         cmp    %rbp,%rbx
401027: 75 e8           jne    401011 <phase_2+0x38>

```

이 부분에서는 %rbx를 다음 값으로 넘기고 %rbp(마지막 입력 해준 값과 같을때까지 401011부터 반복한다. 즉 **피보나치 수열**이다. 그러므로 2번의 **답은 0 1 1 2 3 5**가 된다.

Phase_3

처음에 값을 입력받는 곳을 보면

```

401061: be be 2b 40 00    mov    $0x402bbe,%esi
401066: e8 65 fc ff ff    callq  400cd0 <__isoc99_sscanf@plt>

```

위 코드의 첫 줄을 보면 **\$0x402bbe**를 %esi에 옮겨주는데 안의 데이터 값을 읽어보면

(gdb) x/s 0x402bbe

0x402bbe: "%d %d"

위와 같이 나온다. 이를 통해 입력해 주어야 하는 값이 **정수 두개**임을 알 수 있다. 그 다음 코드를 보면

```

40106b: 83 f8 01         cmp    $0x1,%eax
40106e: 7f 05           jg     401075 <phase_3+0x30>
401070: e8 f5 07 00 00   callq  40186a <explode_bomb>

```

처음 입력해준 값과 %eax를 비교해주는데 **1보다 커야** bomb을 피할 수 있는 것을 알 수 있다.

```

401075: 83 3c 24 07      cmpl   $0x7,(%rsp)
401079: 77 65           ja     4010e0 <phase_3+0x9b>

```

```

4010ea: 83 3c 24 05      cmpl   $0x5,(%rsp)
4010ee: 7f 06           jg     4010f6 <phase_3+0xb1>

```

이 두 코드도 첫번째 값을 비교해주는 곳인데 각각 jump하는 부분이 explode_bomb이므로 첫번째 숫자 $x <= 7$ & $x <= 5$ 이므로 $x <= 5$ 이어야 하고 위에서 찾은 조건을 종합하면 **첫번째 입력값은 $1 < x <= 5$ 인 정수**이다.

```

4010f0: 3b 44 24 04      cmp    0x4(%rsp),%eax
4010f4: 74 05           je     4010fb <phase_3+0xb6>
4010f6: e8 6f 07 00 00   callq 40186a <explode_bomb>

```

0x4(%rsp)는 %rsp보다 4만큼 떨어진 위치의 주소에 있는 메모리값 즉 두번째 입력값을 %eax와 비교하는 것이다.

```

40107e: ff 24 c5 90 28 40 00 jmpq   *0x402890(%rax,8)

```

이 부분에서 첫번째 입력 값에 따라 jump하는 곳이 정해진다는 것을 알 수 있다.

(gdb) x/w 0x402890

0x402890: 0x00401085

따라서 0x401085+8*%rax 의 위치로 jump 하는 것을 볼 수 있다.

```

401085: b8 f2 01 00 00   mov    $0x1f2,%eax           //$0x241=498
40108a: eb 05           jmp     401091 <phase_3+0x4c>
40108c: b8 00 00 00 00   mov    $0x0,%eax
401091: 2d 41 02 00 00   sub    $0x241,%eax           //$0x241= 577
401096: eb 05           jmp     40109d <phase_3+0x58>
401098: b8 00 00 00 00   mov    $0x0,%eax
40109d: 05 41 02 00 00   add    $0x241,%eax
4010a2: eb 05           jmp     4010a9 <phase_3+0x64>
4010a4: b8 00 00 00 00   mov    $0x0,%eax
4010a9: 2d ad 02 00 00   sub    $0x2ad,%eax           //$0x2ad=685
4010ae: eb 05           jmp     4010b5 <phase_3+0x70>
4010b0: b8 00 00 00 00   mov    $0x0,%eax
4010b5: 05 ad 02 00 00   add    $0x2ad,%eax
4010ba: eb 05           jmp     4010c1 <phase_3+0x7c>
4010bc: b8 00 00 00 00   mov    $0x0,%eax
4010c1: 2d ad 02 00 00   sub    $0x2ad,%eax
4010c6: eb 05           jmp     4010cd <phase_3+0x88>
4010c8: b8 00 00 00 00   mov    $0x0,%eax
4010cd: 05 ad 02 00 00   add    $0x2ad,%eax
4010d2: eb 05           jmp     4010d9 <phase_3+0x94>
4010d4: b8 00 00 00 00   mov    $0x0,%eax
4010d9: 2d ad 02 00 00   sub    $0x2ad,%eax
4010de: eb 0a           jmp     4010ea <phase_3+0xa5>

```

첫 번째 입력값이 2 3 4 5 중 하나라는 것을 알기 때문에 2를 넣어주면 0x40109d로 이동하고 그 이후부터 계산해주면 %eax = 577-685=-108이 되는 것을 알 수 있다. 같은 방법

으로 구하면 3번의 정답은 2/-108 or 3/-685 or 4/0 or 5/-685 중 하나를 입력하면 된다.

Phase_4

이 문제 또한 위 문제처럼 처음 코드에서 넣어 주어야하는 값을 찾을 수 있는데,

```
401164: be be 2b 40 00      mov    $0x402bbe,%esi
401169: e8 62 fb ff ff      callq  400cd0 <__isoc99_sscanf@plt>
40116e: 83 f8 02             cmp     $0x2,%eax
401171: 75 06               jne     401179 <phase_4+0x31>
401173: 83 3c 24 0e          cmpl    $0xe,(%rsp)
401177: 76 05               jbe     40117e <phase_4+0x36>
401179: e8 ec 06 00 00      callq  40186a <explode_bomb>
```

%esi에 넣어주는 **\$0x402bbe** 주소 안의 메모리 값을 확인해 보면

(gdb) x/s 0x402bbe

0x402bbe: "%d %d"

이 문제 또한 정수 두개의 입력을 받는다는 것을 알 수 있다. 바로 이어지는 코드를 보면 %eax는 2가 되어야하고 %rsp 메모리 안의 값 즉 **첫번째 값이 14보다 작거나 같아야 함**을 알 수 있다.

이후 코드에서 **func4**라는 함수를 호출한다.

```
40118b: e8 85 ff ff ff      callq  401115 <func4>
401190: 83 f8 2d             cmp     $0x2d,%eax
401193: 75 07               jne     40119c <phase_4+0x54>
401195: 83 7c 24 04 2d        cmpl    $0x2d,0x4(%rsp)
40119a: 74 05               je      4011a1 <phase_4+0x59>
40119c: e8 c9 06 00 00      callq  40186a <explode_bomb>
```

함수 호출 이후 %eax값이 45가 되야하고 %rsp+4주소 안의 메모리 값 즉 **두번째 입력값이 45가 되어야 함**을 알 수 있다.

```
40117e: ba 0e 00 00 00      mov     $0xe,%edx
401183: be 00 00 00 00      mov     $0x0,%esi
401188: 8b 3c 24             mov     (%rsp),%edi
```

함수 호출 전에 있는 코드에서 %edx에는 14 %esi에 0을 넣어주고 %edi에는 첫번째 input을 넣어주는 것을 볼 수있다.

```
401115: 53          push  %rbx
401116: 89 d0       mov   %edx,%eax
401118: 29 f0       sub   %esi,%eax
40111a: 89 c3       mov   %eax,%ebx
40111c: c1 eb 1f    shr   $0x1f,%ebx
40111f: 01 d8       add   %ebx,%eax
401121: d1 f8       sar   %eax
401123: 8d 1c 30    lea   (%rax,%rsi,1),%ebx
401126: 39 fb       cmp   %edi,%ebx
401128: 7e 0c       jle   401136 <func4+0x21>
40112a: 8d 53 ff    lea   -0x1(%rbx),%edx
40112d: e8 e3 ff ff callq 401115 <func4>
401132: 01 d8       add   %ebx,%eax
401134: eb 10       jmp   401146 <func4+0x31>
401136: 89 d8       mov   %ebx,%eax
401138: 39 fb       cmp   %edi,%ebx
40113a: 7d 0a       jge   401146 <func4+0x31>
40113c: 8d 73 01    lea   0x1(%rbx),%esi
40113f: e8 d1 ff ff callq 401115 <func4>
401144: 01 d8       add   %ebx,%eax
401146: 5b         pop   %rbx
401147: c3         retq
```

Func4는 재귀함수인 것을 볼 수 있는데 위에 과정을 정리하자면, 평균값을 구하는 것이라 볼 수 있다. 처음에는 함수 호출 전에 0과 14를 넣어주어 7이 된다. 그 후 평균값과 input을 비교하여 같으면 input값이 return 된다. 만약 input이 작으면 (mid-1,0)의 평균과 평균의 합을 return하고 반대로 input이 더 크면 (mid+1,14)의 평균과 평균의 합을 return해준다. return해준 %eax값이

```
40118b: e8 85 ff ff callq 401115 <func4>
401190: 83 f8 2d    cmp   $0x2d,%eax
401193: 75 07       jne   40119c <phase_4+0x54>
```

이 코드에서 봤듯이 45가 나와야 하는데 input이 14보다 작거나 같아야 하는 점을 알고 첫번째 평균이 7이라는 것을 안다.

14를 넣어주면 $7 \rightarrow (8+14)/2 \rightarrow (12+14) \rightarrow (13+14)/2$ 까지 하면 mid가 14가 되기 때문에 input값이 그대로 return되고 $7+11+13+14=45$ 가 된다. 따라서 **첫번째 input값은 14**가 되어야한다. 따라서 4번의 **정답은 14와 45**이다.

Phase_5

```
4011d3: e8 98 02 00 00    callq 401470 <string_length>
```

Phase_5에 들어가면 처음 input을 입력받을 때 **string_length**라는 함수를 호출한다. 의미는 추측이 되지만 확실히 보기위해 함수를 들어가보면

```
401470: 80 3f 00          cmpb  $0x0,(%rdi)
401473: 74 13             je    401488 <string_length+0x18>
401475: b8 00 00 00 00    mov   $0x0,%eax
40147a: 48 83 c7 01       add   $0x1,%rdi
40147e: 83 c0 01          add   $0x1,%eax
401481: 80 3f 00          cmpb  $0x0,(%rdi)
401484: 75 f4             jne   40147a <string_length+0xa>
401486: f3 c3            repz retq
401488: b8 00 00 00 00    mov   $0x0,%eax
40148d: c3              retq
```

여기서 `cmpb $0x0,(%rdi)` 코드는 지금 확인하는 값이 NULL값인지를 보는 것이다. 두 군데 있는데 위에 있는 jump는 NULL값에 도착하면 return해주는 것이고 밑에 있는 jump는 NULL값이 아니면 다시 다음 자리를 확인하면서 글자 수(%eax)를 늘려 주는 것이다. 따라서 %eax에는 내가 넣어준 글자의 길이가 들어가게 될 것이다.

```
4011d8: 83 f8 06          cmp   $0x6,%eax
4011db: 74 05             je    4011e2 <phase_5+0x27>
4011dd: e8 88 06 00 00    callq 40186a <explode_bomb>
```

함수 밖으로 나오면 %eax를 6과 비교하여 맞아야 bomb을 피할 수 있다. 따라서 **6자리**의 **문자**를 입력해야 한다.

```
4011e2: b8 00 00 00 00    mov   $0x0,%eax
4011e7: 0f b6 14 03       movzbl (%rbx,%rax,1),%edx
4011eb: 83 e2 0f          and   $0xf,%edx
4011ee: 0f b6 92 d0 28 40 movzbl 0x4028d0(%rdx),%edx
4011f5: 88 14 04          mov   %dl,(%rsp,%rax,1)
```



```

4011f8: 48 83 c0 01      add    $0x1,%rax
4011fc: 48 83 f8 06      cmp    $0x6,%rax
401200: 75 e5           jne    4011e7 <phase_5+0x2c>

```

그 후 코드를 살펴보면 %eax에 다시 0을 집어넣고 %rbx에는 입력해준 문자열이 들어가 있기 때문에 %edx에는 **%rax*1번째의 단어**가 들어간다.

마지막 세줄을 보면 %rax에 1을 더하고 6과 비교하여 아니면 다시 반복하는데 내가 입력해준 모든 단어를 하나씩 넣어주는 **반복문**임을 알 수 있다.

그 후 \$0xf와 and 연산을 해준다. 내가 입력해주는 것이 문자열이기 때문에 알파벳순으로 ascii code에서 97~부터 and를 해줄 것이다. %edx에 0x4028d0(%rdx)를 movzbl해주는 데 %edx에는 0x4028d0+%edx라는 주소안의 메모리 값이 들어가게 된다. 이 주소안의 메모리 값을 보면

(gdb) x/s 0x4028d0

0x4028d0 <array.3601>: "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"

앞에서부터 순서대로 0x4028d0 + 0 1 2 3 4 5 6 순으로 주소가 쓰여있을 것이다. 따라서 %edx에는 input해준 문자의 알파벳 순서에 맞추어 0x4028d0 주소 안의 순서에 맞추어 단어가 변환될 것이다.

예를 들어 a b c d e f -> a d u i e r

```

4011f5: 88 14 04          mov    %dl,(%rsp,%rax,1)

```

코드 중간의 이 부분은 %rsp+%rax*1에 내가 변환해 준 단어를 넣어주는 것이다. (%dl은 %edx안에 있는 레지스터)

반복문 이후 코드를 보면

```

401202: c6 44 24 06 00    movb   $0x0,0x6(%rsp)
401207: be 7e 28 40 00    mov    $0x40287e,%esi
40120c: 48 89 e7          mov    %rsp,%rdi
40120f: e8 7a 02 00 00    callq  40148e <strings_not_equal>

```

%rsp+6의 주소에 NULL값을 넣어주고 %esi에 \$0x40287e 주소를 넣어 %rsp와 %rdi를

비교해 준다. 여기서 %rsp에는 +6에 NULL을 넣어주었음으로 변환된 문자열의 주소가 있을 것이고 %rsi는

(gdb) x/s 0x40287e

0x40287e: "sabres"

이 단어가 들어가 있다. 따라서 5번은 **"sabres"**로 변환이 되는 문자열을 입력하는 것이 답이다.

"maduiersnfotvbyl" 이 단어에서 s a b r e s 는 7 1 13 6 5 7 순으로 들어있다. 알파벳 순으로 다시 변환해주면 **g a m f e g**이다.

따라서 5번의 정답은 **gamfeg**이다.

Phase_6

Phase_6도 앞에 있던 phase_2와 같이

401257: e8 44 06 00 00 callq 4018a0 <read_six_numbers>

여섯개의 숫자를 입력받는 것을 알 수 있다.

그 이후에는 반복문이 진행된다.

```
40125c: 49 89 e4            mov    %rsp,%r12
40125f: 49 89 e5            mov    %rsp,%r13
401262: 41 be 00 00 00 00    mov    $0x0,%r14d
401268: 4c 89 ed            mov    %r13,%rbp
40126b: 41 8b 45 00          mov    0x0(%r13),%eax
40126f: 83 e8 01            sub    $0x1,%eax
401272: 83 f8 05            cmp    $0x5,%eax
401275: 76 05                jbe    40127c <phase_6+0x44>
401277: e8 ee 05 00 00       callq 40186a <explode_bomb>
```

이 부분은 %eax에 첫번째 숫자를 넣어주고 1을 뺀 후 5와 비교하는 것이다. 따라서 첫번째 숫자는 **6보다 작거나 같아야** bomb을 피한다.

```

40127c: 41 83 c6 01      add    $0x1,%r14d
401280: 41 83 fe 06      cmp    $0x6,%r14d
401284: 74 21           je     4012a7 <phase_6+0x6f>

```

이 부분은 %r14d에 1을 더하면서 시작하는데 6이 될 때까지 반복한다는 것이다.

```

401286: 44 89 f3      mov    %r14d,%ebx
401289: 48 63 c3      movslq %ebx,%rax
40128c: 8b 04 84      mov    (%rsp,%rax,4),%eax

```

여기서는 %eax에 $\text{\%rsp} + 4 * \text{\%rax} = \text{\%rsp} + 4 * \text{\%r14d}$ 의 주소를 넣어준다. %rsp에는 첫번째 입력값의 주소가 들어가 있으므로 %eax는 %r14d 번째 숫자가 입력될 것이다.

```

40128f: 39 45 00      cmp    %eax,0x0(%rbp)
401292: 75 05      jne    401299 <phase_6+0x61>
401294: e8 d1 05 00 00 callq  40186a <explode_bomb>

```

이후 0x0(%rbp)와 비교하는데 첫번째 숫자와 비교하는 것을 의미한다. Bomb을 피하기 위해서는 첫번째 숫자와 달라야 한다.

```

401299: 83 c3 01      add    $0x1,%ebx
40129c: 83 fb 05      cmp    $0x5,%ebx
40129f: 7e e8      jle    401289 <phase_6+0x51>

```

이후

```

401289: 48 63 c3      movslq %ebx,%rax

```

로 이동하여 다시 반복하는데 정리하자면 2번째 3번째 4번째 5번째 6번째 숫자가 1과 달라야 함을 의미한다.

```

4012a1: 49 83 c5 04      add    $0x4,%r13
4012a5: eb c1      jmp    401268 <phase_6+0x30>

```

다음에는 %r13에 4를 더한 후

```

401268: 4c 89 ed      mov    %r13,%rbp
40126b: 41 8b 45 00      mov    0x0(%r13),%eax
40126f: 83 e8 01      sub    $0x1,%eax
401272: 83 f8 05      cmp    $0x5,%eax

```

이 부분으로 이동하는데 여기서 %r13이 4만큼 증가되었기 때문에 %eax에는 2번째 숫자(next number)가 입력될 것이다. 이번에는 2번째 숫자(next number)가 6보다 작거나 같은지 확인하고 그 후 다시 반복한다.

결국 위에 코드 부분은 내가 입력해 준 모든 숫자가 다르고 전부 6보다 작거나 같은지 확인하는 반복문이다.

```
4012a7: 48 8d 4c 24 18    lea    0x18(%rsp),%rcx
4012ac: ba 07 00 00 00    mov    $0x7,%edx
4012b1: 89 d0             mov    %edx,%eax
4012b3: 41 2b 04 24       sub    (%r12),%eax
4012b7: 41 89 04 24       mov    %eax,(%r12)
4012bb: 49 83 c4 04       add    $0x4,%r12
4012bf: 4c 39 e1          cmp    %r12,%rcx
4012c2: 75 ed            jne    4012b1 <phase_6+0x79>
```

다음에는 %rsp+24인 주소 즉 마지막 숫자를 %rcx에 옮겨준다. %eax에 7을 넣어주고 %r12에 들어있는 숫자를 빼준 후 그 값을 다시 %r12 주소안에 넣어준다. %r12의 주소를 4만큼 키운 후 %rsp+24 주소인 %rcx와 비교하여 같을 때까지 반복한다.

간단히 정리하면 내가 입력해준 숫자를 순서대로 7에서부터 빼주는 것이다.

Ex) 1 2 3 4 5 6 -> 6 5 4 3 2 1

```
4012cb: 48 8b 52 08       mov    0x8(%rdx),%rdx
4012cf: 83 c0 01          add    $0x1,%eax
4012d2: 39 c8             cmp    %ecx,%eax
4012d4: 75 f5            jne    4012cb <phase_6+0x93>
4012d6: 48 89 54 74 20    mov    %rdx,0x20(%rsp,%rsi,2)
4012db: 48 83 c6 04       add    $0x4,%rsi
4012df: 48 83 fe 18       cmp    $0x18,%rsi
4012e3: 74 14            je     4012f9 <phase_6+0xc1>
4012e5: 8b 0c 34          mov    (%rsp,%rsi,1),%ecx
4012e8: b8 01 00 00 00    mov    $0x1,%eax
4012ed: ba 10 43 60 00    mov    $0x604310,%edx
4012f2: 83 f9 01          cmp    $0x1,%ecx
4012f5: 7f d4            jg     4012cb <phase_6+0x93>
4012f7: eb dd            jmp    4012d6 <phase_6+0x9e>
```

위 코드에서는 내가 넣어주는 숫자를 **Linked list**로 만드는 것을 의미한다. 예를 들어 5 3 6 2 4 1순으로 입력하면 node에 5 -> 3 -> 6 -> 2 -> 4 -> 1가 각각 연결되도록 해주는 것이다.

```
40132c: 48 8b 43 08      mov     0x8(%rbx),%rax
401330: 8b 00            mov     (%rax),%eax
401332: 39 03           cmp     %eax,(%rbx)
401334: 7d 05           jge     40133b <phase_6+0x103>
401336: e8 2f 05 00 00   callq   40186a <explode_bomb>
40133b: 48 8b 5b 08      mov     0x8(%rbx),%rbx
40133f: 83 ed 01        sub     $0x1,%ebp
401342: 75 e8           jne     40132c <phase_6+0xf4>
```

이 부분은 (7-입력해준 숫자)가 가르키는 node안의 숫자와 node안의 node 안의 숫자를 비교하여 모든 경우가 커야 끝나는 것을 알 수 있다. 이 코드 위에서 node의 Linked list를 만든 부분을 생각하면 node5 >= node3 >= node6 >= node2 >= node4 >= node1이 되어야한다.

결국 크기 순으로 입력해 주면되는데 node 안의 숫자가

(gdb) x/24w 0x604310

0x604310 <node1>:	342	1	6308640 0
0x604320 <node2>:	252	2	6308656 0
0x604330 <node3>:	622	3	6308672 0
0x604340 <node4>:	944	4	6308688 0
0x604350 <node5>:	842	5	6308704 0
0x604360 <node6>:	264	6	0 0

아래와 같으므로 **4 5 3 1 6 2** 순인데 7에서 빼준값으로 나와야 하므로 정답은 **3 2 4 6 1 5**이다.

Secret_phase

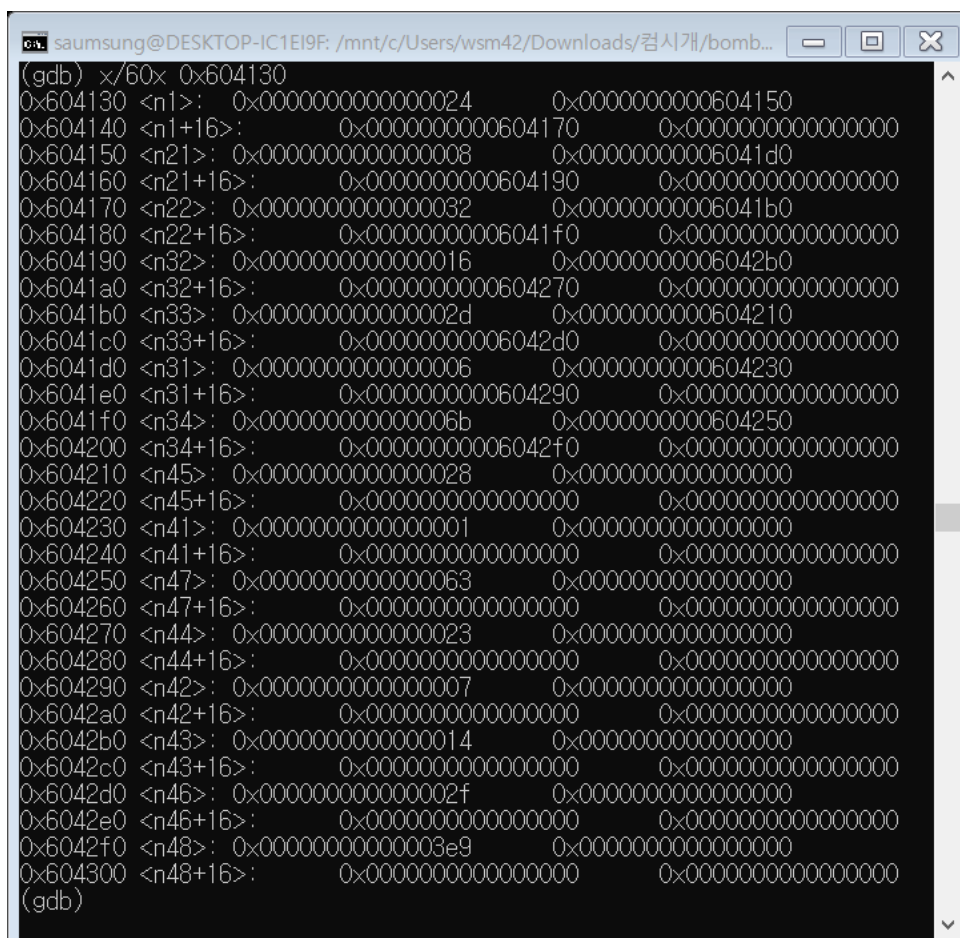
Secret_phase는 phase_defused에서 봤듯이 4번 정답에 DrEvil이라는 단어를 추가해서 입력해 주면 들어갈 수 있다.

```
4013c2: 3d e8 03 00 00      cmp    $0x3e8,%eax
4013c7: 76 05              jbe    4013ce <secret_phase+0x2a>
4013c9: e8 9c 04 00 00      callq 40186a <explode_bomb>
```

입력한 숫자가 1000보다 작거나 같아야 함을 알 수 있다. 그 후 fun7함수를 호출한다.

```
4013d0: bf 30 41 60 00      mov    $0x604130,%edi
4013d5: e8 8c ff ff ff      callq 401366 <fun7>
4013da: 83 f8 05            cmp    $0x5,%eax
4013dd: 74 05              je     4013e4 <secret_phase+0x40>
4013df: e8 86 04 00 00      callq 40186a <explode_bomb>
```

fun7의 return값이 5가 되어야 secret phase를 풀 수 있다. %edi 옮긴 주소를 살펴보면



```
saumsung@DESKTOP-IC1E19F: /mnt/c/Users/wsm42/Downloads/컴시개/bomb...
(gdb) x/60x 0x604130
0x604130 <n1>: 0x0000000000000024      0x0000000000000040
0x604140 <n1+16>: 0x0000000000000000      0x0000000000000000
0x604150 <n21>: 0x0000000000000008      0x00000000000000d0
0x604160 <n21+16>: 0x0000000000000090      0x0000000000000000
0x604170 <n22>: 0x0000000000000032      0x00000000000000b0
0x604180 <n22+16>: 0x00000000000000f0      0x0000000000000000
0x604190 <n32>: 0x0000000000000016      0x000000000000002b
0x6041a0 <n32+16>: 0x0000000000000027      0x0000000000000000
0x6041b0 <n33>: 0x000000000000000d      0x0000000000000021
0x6041c0 <n33+16>: 0x00000000000000d0      0x0000000000000000
0x6041d0 <n31>: 0x0000000000000006      0x0000000000000023
0x6041e0 <n31+16>: 0x0000000000000029      0x0000000000000000
0x6041f0 <n34>: 0x000000000000000b      0x0000000000000025
0x604200 <n34+16>: 0x000000000000002f      0x0000000000000000
0x604210 <n45>: 0x0000000000000028      0x0000000000000000
0x604220 <n45+16>: 0x0000000000000000      0x0000000000000000
0x604230 <n41>: 0x0000000000000001      0x0000000000000000
0x604240 <n41+16>: 0x0000000000000000      0x0000000000000000
0x604250 <n47>: 0x0000000000000063      0x0000000000000000
0x604260 <n47+16>: 0x0000000000000000      0x0000000000000000
0x604270 <n44>: 0x0000000000000023      0x0000000000000000
0x604280 <n44+16>: 0x0000000000000000      0x0000000000000000
0x604290 <n42>: 0x0000000000000007      0x0000000000000000
0x6042a0 <n42+16>: 0x0000000000000000      0x0000000000000000
0x6042b0 <n43>: 0x0000000000000014      0x0000000000000000
0x6042c0 <n43+16>: 0x0000000000000000      0x0000000000000000
0x6042d0 <n46>: 0x000000000000002f      0x0000000000000000
0x6042e0 <n46+16>: 0x0000000000000000      0x0000000000000000
0x6042f0 <n48>: 0x000000000000003e      0x0000000000000000
0x604300 <n48+16>: 0x0000000000000000      0x0000000000000000
(gdb)
```

이진트리 모양을 갖는 다는 것을 알 수 있다. Fun7은 내가 넣어준 input값이 가르키는 이진트리의 위치를 return 해준 다는 것을 예상할 수 있다.

```

401366: 48 83 ec 08      sub    $0x8,%rsp
40136a: 48 85 ff         test   %rdi,%rdi
40136d: 74 2b           je     40139a <fun7+0x34>
40136f: 8b 17           mov    (%rdi),%edx

401371: 39 f2           cmp    %esi,%edx
401373: 7e 0d           jle    401382 <fun7+0x1c>
401375: 48 8b 7f 08      mov    0x8(%rdi),%rdi
401379: e8 e8 ff ff     callq 401366 <fun7>

40137e: 01 c0           add    %eax,%eax
401380: eb 1d           jmp    40139f <fun7+0x39>
401382: b8 00 00 00 00   mov    $0x0,%eax

401387: 39 f2           cmp    %esi,%edx
401389: 74 14           je     40139f <fun7+0x39>
40138b: 48 8b 7f 10      mov    0x10(%rdi),%rdi
40138f: e8 d2 ff ff     callq 401366 <fun7>

401394: 8d 44 00 01      lea    0x1(%rax,%rax,1),%eax
401398: eb 05           jmp    40139f <fun7+0x39>
40139a: b8 ff ff ff ff   mov    $0xffffffff,%eax
40139f: 48 83 c4 08      add    $0x8,%rsp
4013a3: c3             retq

```

Fun7의 return이 일어나는 상황을 보면 %esi(input)가 %edx와 같으면 위에 jump문과 아래 jump문을 통해 %eax에 0을 return한다. Input이 더 작으면 %rdi+ 0x8 왼쪽 아래 이진트리로 이동하고 input이 더 크면 %rdi+16 오른쪽 아래 이진트리로 이동한다. 내가 원하는 return 값은 5이므로 $5=(1*2+0)*2+1$ **오른쪽 왼쪽 오른쪽으로 이동하여 도착하는 0x2f= 47**이 정답이다.