

2023 시스템 프로그래밍

- Bomb Lab -

제출일자	2023. 10. 22.
분 반	00
이 름	김재덕
학 번	202104340

Phase 0 [폭탄 해체 준비]

1. 폭탄의 어셈블리 코드 추출하기

```
$ objdump -D ./bomb16/bomb > bomb-voyage.obj
$ vim bomb-voyage.obj
```

2. .gdbinit 파일¹로 중단점 자동 설정하기

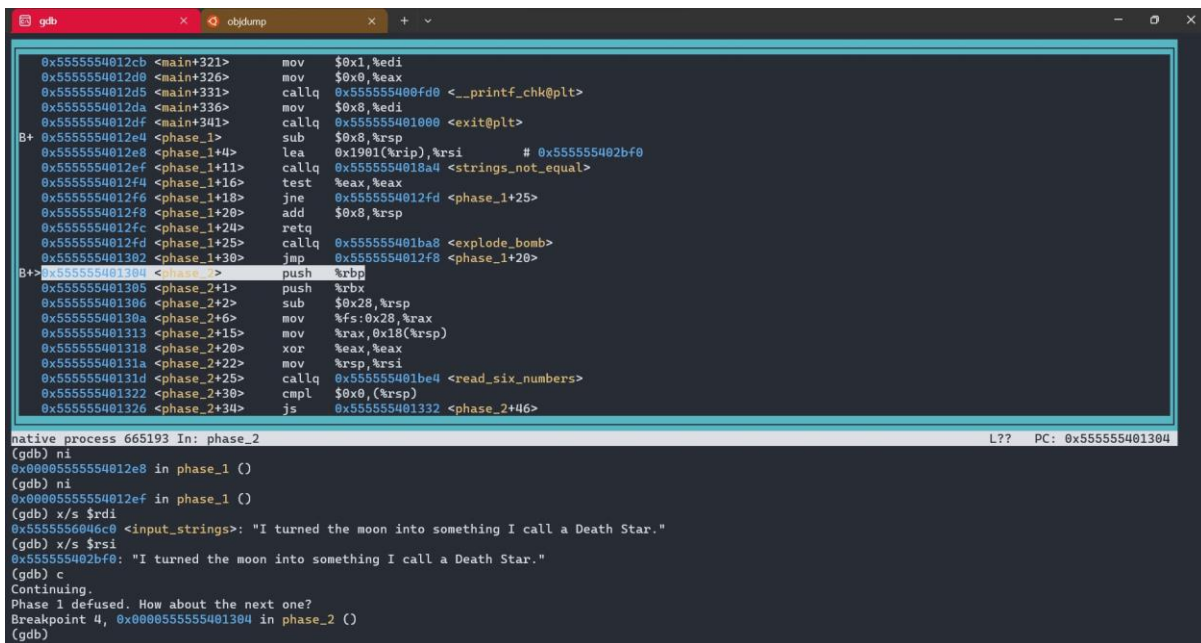
- *bomb-voyage.obj* 파일을 열어서 *main()* 함수의 정의부터 확인해보면, *read_line()*, *phase_1()*, *phase_defused()* 등의 함수들이 정의되어 있음을 알 수 있다. 그 다음으로 *phase_1()*의 함수 정의를 확인해보면, *explode_bomb()*이라는 함수가 존재한다는 것을 알 수 있다. 이제 *~/.gdbinit* 파일을 통해 *gdb*를 실행할 때마다 해당 함수들에 중단점을 자동으로 설정하게끔 만들어보자.

```
$ vim ~/.gdbinit
```

```
add-auto-load-safe-path ~
tui enable
layout asm
file ~/bomb16/bomb
b main
b explode_bomb
# b phase_defused
b phase_1
b phase_2
b phase_3
b phase_4
b phase_5
b phase_6
b secret_phase
```

¹ <https://manpages.ubuntu.com/manpages/jammy/man5/gdbinit.5.html>

Phase 1 [결과 화면 캡처]



```
0x5555554012cb <main+321>    mov     $0x1,%edi
0x5555554012d0 <main+326>    mov     $0x0,%eax
0x5555554012d5 <main+331>    callq   0x555555400fd0 <__printf_chk@plt>
0x5555554012da <main+336>    mov     $0x0,%edi
0x5555554012df <main+341>    callq   0x555555401000 <exit@plt>
0x5555554012e4 <phase_1>      sub     $0x8,%rsp
0x5555554012e8 <phase_1+4>    lea     0x1901(%rip),%rsi    # 0x555555402bf0
0x5555554012ef <phase_1+11>   callq   0x5555554018a4 <strings_not_equal>
0x5555554012f4 <phase_1+16>    test    %eax,%eax
0x5555554012f6 <phase_1+18>    jne     0x5555554012fd <phase_1+25>
0x5555554012f8 <phase_1+20>    add     $0x8,%rsp
0x5555554012fc <phase_1+24>    retq
0x5555554012fd <phase_1+25>   callq   0x555555401ba8 <explode_bomb>
0x555555401302 <phase_1+30>   jmp     0x5555554012f8 <phase_1+20>
0x555555401304 <phase_2>      push    %rbp
0x555555401305 <phase_2+1>    push    %rbx
0x555555401306 <phase_2+2>    sub     $0x28,%rsp
0x55555540130a <phase_2+6>    mov     %fs:0x28,%rax
0x555555401313 <phase_2+15>   mov     %rax,0x18(%rsp)
0x555555401319 <phase_2+20>    xor     %eax,%eax
0x55555540131a <phase_2+22>    mov     %rsp,%rsi
0x55555540131d <phase_2+25>   callq   0x555555401be4 <read_six_numbers>
0x555555401322 <phase_2+30>    cmpl    $0x0,(%rsp)
0x555555401326 <phase_2+34>    js      0x555555401332 <phase_2+46>

native process 665193 In: phase_2
(gdb) ni
0x00005555554012e8 in phase_1 ()
(gdb) ni
0x00005555554012ef in phase_1 ()
(gdb) x/s $rdi
0x5555556046c0 <input_strings>: "I turned the moon into something I call a Death Star."
(gdb) x/s $rsi
0x555555402bf0: "I turned the moon into something I call a Death Star."
(gdb) c
Continuing.
Phase 1 defused. How about the next one?
Breakpoint 4, 0x0000555555401304 in phase_2 ()
(gdb)
```

Phase 1 [진행 과정 설명]

- 어차피 각 페이지마다 중단점이 걸려 있으므로, 아무 문자열이나 한번 입력해보고 *c*를 입력해 프로그램이 *phase_1()*에서 실행을 멈추게 만든다.

```
(gdb) r input.txt
Starting program: /home/sys00/a202104340/bomb16/bomb input.txt

Breakpoint 1, main (argc=2, argv=0x7fffffff488) at bomb.c:37
(gdb) c
Continuing.

Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!

Breakpoint 3, 0x00005555554012e4 in phase_1 ()
(gdb)
```

- 그 다음에 `phase_1()`의 어셈블리 코드를 확인해보면 `phase_1()`이 `strings_not_equal()` 함수를 호출하는데, 함수의 두 번째 인자 (argument)를 저장할 때 주로 사용하는 `%rsi` 레지스터가 등장하고 `jne`로 함수의 반환값인 `%rax`가 0인지 아닌지를 검사하는 코드임을 알 수 있다.

```
00000000000012e4 <phase_1>:
  12e4:  sub    $0x8,%rsp
  12e8:  lea    0x1901(%rip),%rsi      # 2bf0 <_IO_stdin_used+0x150>
  12ef:  callq  18a4 <strings_not_equal>
  12f4:  test   %eax,%eax
  12f6:  jne    12fd <phase_1+0x19>
  12f8:  add    $0x8,%rsp
  12fc:  retq
  12fd:  callq  1ba8 <explode_bomb>
  1302:  jmp    12f8 <phase_1+0x14>
```

- 이러한 분석을 바탕으로, `strings_not_equal()`는 `strcmp(%rdi, %rsi) == 0`와 같은 기능을 하는 함수임을 알 수 있다.

```
(gdb) x/s %rsi
0x555555402bf0: "I turned the moon into something I call a Death Star."
```

Phase 1 [정답]

I turned the moon into something I call a Death Star.

Phase 2 [결과 화면 캡처]

```

B+ 0x5555554012e4 <phase_1>      sub    $0x8,%rsp
> 0x5555554012f2 <phase_1>      sub    $0x28,%rsp),%rsi    # 0x555555402bf0
0x555555401376 <phase_3+11>     mov    %fs:0x28,%rax    trings_n
0x55555540137a <phase_3+13>     mov    %rax,0x18(%rsp)
0x555555401386 <phase_3+18>     xor    0x5555554012fd <phase_1+25>
0x555555401388 <phase_3+20>     lea    0xf(%rsp),%rcx
0x55555540138b <phase_3+25>     lea    0x10(%rsp),%rdx
0x555555401398 <phase_3+30>     lea    0x14(%rsp),%r8 <explode_bomb>
0x555555401395 <phase_3+30>     lea    0x18b2(%rip),%rsi    # 0x555555402c4e
B+> 0x55555540139c <phase_3+42>     callq  %rbp    0fc0 <__isoc99_sscanf@plt>
0x5555554013a1 <phase_3+47>     cmp    $0x2,%eax
0x5555554013a4 <phase_3+50>     jle    0x5555554013c5 <phase_3+83>
0x5555554013aa <phase_3+52>     cmpl   %fs:7,0x10(%rsp)
0x5555554013ab <phase_3+57>     ja     0x5555554014bd <phase_3+331>
0x5555554013b1 <phase_3+63>     xor    0x10(%rsp),%eax
0x5555554013b5 <phase_3+67>     lea    0x18a4(%rip),%rdx    # 0x555555402c60
0x5555554013bc <phase_3+74>     calslq (%rdx,%rax,4),%raxad_six_numbers>
0x5555554013c0 <phase_3+78>     add    %rdx,%rax
0x5555554013c3 <phase_3+81>     jmpq   *%rax    332 <phase_2+46>
0x5555554013c5 <phase_3+83>     callq  $0x1,%ebx    ba8 <explode_bomb>
0x5555554013ca <phase_3+88>     jmp    0x5555554013a6 <phase_3+52>
0x5555554013cc <phase_2+90>     jmp    $0x71,%eax    1343 <phase_2+63>
0x5555554013d1 <phase_3+95>     cmpl   $0x154,0x14(%rsp)    b>
0x5555554013d9 <phase_3+103>     je     0x5555554014c7 <phase_3+341>
0x5555554013de <phase_3+109>     callq  %rax    ba8 <explode_bomb>
native process 679582 In: phase_2
Continuing.
Breakpoint 3, 0x00005555554012e4 in phase_1 ()
(gdb) c
Continuing.
Breakpoint 4, 0x0000555555401304 in phase_2 ()
(gdb) c
Continuing.
That's number 2. Keep going!
Breakpoint 5, 0x0000555555401372 in phase_3 ()
(gdb)

```

Phase 2 [진행 과정 설명]

- 페이지 2부터는 코드가 조금씩 길어지기 시작한다. 하지만 쫓지 말자! *phase_2()*에서 맨 처음으로 주목해야 할 곳은 바로 *read_six_numbers()*를 호출하는 부분이다.

```

0000000000001304 <phase_2>:
# ===== ... =====
131d:  callq  1be4 <read_six_numbers>    # int rsp[6] = { ... };
1322:  cmpl    $0x0,(%rsp)
1326:  js      1332 <phase_2+0x2e>        # if (rsp[0] < 0)
                                   #     explode_bomb();
# ===== ... =====

```

- *x/?d*를 이용해 스택에 저장된 값을 확인해보면, 내가 입력한 6개의 숫자가 들어있는 것을 알 수 있다.

```

(gdb) x/6d $rsp
0x7fffffffef380: 1      2      3      4
0x7fffffffef390: 5      6

```

- 1328번지와 1339번지 주소를 보면, `%ebx`는 1로 초기화되고 6이 될 때까지 계속 1씩 증가하는데, 이를 통해 `for (int i = 1; i < 6; i++)` 패턴의 코드일 수도 있겠다는 생각이 들었다.

```
0000000000001304 <phase_2>:
# ===== ... =====
1328:  mov    $0x1,%ebx          # `%ebx` = 1;
132d:  mov    %rsp,%rbp          # `%ebp` = `%esp`;
1330:  jmp     1343 <phase_2+0x3f> # goto <phase_2+0x3f>;
1332:  callq   1ba8 <explode_bomb>
1337:  jmp     1328 <phase_2+0x24>
1339:  add     $0x1,%rbx
133d:  cmp     $0x6,%rbx          # for (; `%ebx` < 6; `%ebx`++) { ... }
1341:  je      1356 <phase_2+0x52>
1343:  mov     %ebx,%eax          # `%eax` = `%ebx`
1345:  add     -0x4(%rbp,%rbx,4),%eax # + `-0x4(%rbp,%rbx,4)`;
1349:  cmp     %eax,0x0(%rbp,%rbx,4)
134d:  je      1339 <phase_2+0x35> # if (`%eax` != `0x0(%rbp,%rbx,4)`)
134f:  callq   1ba8 <explode_bomb> # explode_bomb();
1354:  jmp     1339 <phase_2+0x35>
1356:  mov     0x18(%rsp),%rax
# ===== ... =====
```

- `phase_2()`의 어셈블리 코드를 참고하여 C 코드를 작성해보았다.

```
int ebp[6] = { /* 내가 입력한 6개의 숫자 */ };

for (int ebx = 1; ebx < 6; ebx++) {
    int eax = i;

    // NOTE: "pointer arithmetic"
    eax += ebp[i - 1];

    // assert(eax == ebp[i]);
    if (eax != ebp[i]) explode_bomb();
}
```

- 따라서 내가 입력한 숫자가 1로 시작한다면, 그 다음 숫자들은 $(1 + \text{ebp}[0])$, $(2 + \text{ebp}[1])$, $(3 + \text{ebp}[2])$, ...가 되고, 이 규칙에 맞는 6개의 숫자가 바로 문제의 정답이 된다.

Phase 2 [정답]

1 2 4 7 11 16

Phase 3 [결과 화면 캡처]

```
0x555555401372 <phase_3> sub $0x28,%rsp
0x555555401373 <phase_3+4> mov %fs:0x28,%rax
0x555555401374 <phase_3+13> mov %rax,0x18(%rsp)
0x555555401375 <phase_3+18> xor %eax,%eax
0x555555401376 <phase_3+20> lea 0xf(%rsp),%rcx
0x555555401377 <phase_3+25> lea 0x40(%rsp),%rdx
0x555555401378 <phase_3+30> mov %rsp,%rdx
0x555555401379 <phase_3+38> lea 0x19ca(%rip),%rsi # 0x555555402f0d
0x55555540137a <phase_3+42> callq 0x555555400fc0 <__isoc99_sscanf@plt>
0x55555540137b <phase_3+40> cmp $0x2,%eax
0x55555540137c <phase_3+43> jne 0x555555401553 <phase_3+51>
0x55555540137d <phase_3+45> cmpl $0xe,(%rsp)
0x55555540137e <phase_3+49> jbe 0x555555401558 <phase_3+56>
0x55555540137f <phase_3+51> callq 0x10(%rsp),ba8 <explode_bomb>
0x555555401380 <phase_3+56> lea $0xe,%edx # 0x555555402c60
0x555555401381 <phase_3+61> mov $0x0,%esi
0x555555401382 <phase_3+66> add (%rsp),%edi
0x555555401383 <phase_3+69> callq 0x5555554014ec <func4>
0x555555401384 <phase_3+74> cmp $0x1f,%eax1ba8 <explode_bomb>
0x555555401385 <phase_3+77> jne 0x555555401576 <phase_3+86>
0x555555401386 <phase_3+79> cmpl $0x1f,0x0(%rsp)
0x555555401387 <phase_3+84> je 0x55555540157b <phase_3+91>
0x555555401388 <phase_3+86> jallq 0x555555401ba8 <explode_bomb>
0x555555401389 <phase_3+91> mov 0x8(%rsp),%rax
0x55555540138a <phase_3+96> xor %fs:0x28,%rax
```

Phase 3 [진행 과정 설명]

- 가장 먼저 주목해야 할 부분은 139c번지에서 `sscanf()` 함수를 호출하는 부분이다.

```
0000000000001372 <phase_3>:
# ===== ... =====
1395: lea    0x18b2(%rip),%rsi      # 2c4e <_IO_stdin_used+0x1ae>
139c: callq  fc0 <__isoc99_sscanf@plt> # sscanf(...);
# ===== ... =====
```

- `sscanf()`의 인자로 어떤 형식 지정자 (format specifier)²가 넘겨지는지 확인해보자!

```
(gdb) x/s $rsi
0x555555402c4e: "%d %c %d"

(gdb) x/s $rdi
0x555555604760 <input_strings+160>: "살려주세요"
```

² <https://en.cppreference.com/w/cpp/io/c/fscanf>

- 이제 다시 *phase_3()*의 맨 처음으로 돌아와서, 형식 지정자에 맞는 숫자 2개와 문자 1개를 입력해보고 코드의 흐름을 관찰해보자. *x/d \$rsp+0x10*를 통해 *0x10(%rsp)*에는 내가 입력한 첫 번째 숫자가 저장된다는 것, 그리고 13a6번지 주소의 코드를 통해 첫 번째 숫자는 무조건 7보다 작거나 같아야 한다는 것을 확인할 수 있다.

```
0000000000001372 <phase_3>:
# ===== ... =====
13a1:  cmp    $0x2,%eax          # if (sscanf("%d %c %d", ...) <= 2)
13a4:  jle     13c5 <phase_3+0x53> # explode_bomb();

13a6:  cmpl    $0x7,0x10(%rsp)    # if (`0x10(%rsp)` > 7)
13ab:  ja      14bd <phase_3+0x14b> # explode_bomb();
# ===== ... =====
13c5:  callq   1ba8 <explode_bomb>
# ===== ... =====
```

- 또한, 13b1번지부터 13c3번지까지의 주소에 해당하는 코드를 보면 내가 첫 번째로 입력한 숫자를 가지고 주소를 만들어서 *%rax*에 저장한 다음 이 주소로 점프하는 코드가 나오는데, 이 코드의 아래 부분에 *mov ?, %rax, cmpl \$0x154, 0x14(%rsp)*라는 똑같은 패턴이 계속 반복해서 등장하는 것으로 보아 여기서부터는 *%rax*의 값에 따라 실행할 코드가 결정되는 switch-case 문인 것 같다는 생각이 들었다. 난 첫 번째 숫자로 1을 입력했기 때문에 일단 *phase_3()*를 계속 실행해보고 첫 번째 숫자에 따라 정답이 어떻게 달라질지 관찰해보기로 했다.

```
0000000000001372 <phase_3>:
# ===== ... =====
13b1:  mov     0x10(%rsp),%eax
13b5:  lea     0x18a4(%rip),%rdx
13bc:  movslq  (%rdx,%rax,4),%rax
13c0:  add     %rdx,%rax
13c3:  jmpq    *%rax
# ===== ... =====
```

- *ni*로 함수를 계속 실행시키다 보면, 13ee번지 주소로 점프한 후에 *0x14(%rsp)*에 저장된 값을 *0x26f*라는 값과 비교하는 것을 확인할 수 있다. 그런데 *0x14(\$rsp)*에는 내가 입력한 세 번째 숫자가 저장되어 있기 때문에, 세 번째 숫자의 값은 무조건 *0x26f*, 즉 623이 되어야 함을 알 수 있다.

```
0000000000001372 <phase_3>:
# ===== ... =====
13ee:  mov    $0x79,%eax          # eax = 0x79;
13f3:  cmpl   $0x26f,0x14(%rsp)
13fa:
13fb:  je     14c7 <phase_3+0x155> # if (`0x14(%rsp)` == 0x26f) { ... }
1401:  callq  1ba8 <explode_bomb>
1406:  mov    $0x79,%eax
140b:  jmpq   14c7 <phase_3+0x155>
1410:  mov    $0x64,%eax
1415:  cmpl   $0x3de,0x14(%rsp)
141c:
141d:  je     14c7 <phase_3+0x155>
1423:  callq  1ba8 <explode_bomb>
1428:  mov    $0x64,%eax
142d:  jmpq   14c7 <phase_3+0x155>
1432:  mov    $0x72,%eax
1437:  cmpl   $0x23b,0x14(%rsp)
# ===== ... =====
```

- 마지막으로, 14c7번지로 점프하고 나서는 `0xf(%rsp)`에 저장된 값을 `%al`, 즉 `%rax`의 하위 8비트와 비교한다. 그런데 `%rax`에는 `0x79`가 저장되어 있다는 것을 위에서 이미 확인했고, 또한 `x/d $rsp+0xf`를 통해 `0xf(%rsp)`는 내가 입력한 두 번째 문자의 ASCII 코드에 대응함을 알 수 있다. 따라서 두 번째 문자의 값은 `0x79`, 즉 'y'가 되어야 한다.

```
00000000000001372 <phase_3>:
# ===== ... =====
14c7:  cmp    %al,0xf(%rsp)
14cb:  je     14d2 <phase_3+0x160>
14cd:  callq  1ba8 <explode_bomb>
14d2:  mov    0x18(%rsp),%rax
14d7:  xor    %fs:0x28,%rax
14de:
14e0:  jne    14e7 <phase_3+0x175>
14e2:  add    $0x28,%rsp
14e6:  retq
# ===== ... =====
```

Phase 3 [정답]

1 y 623

Phase 4 [결과 화면 캡처]

```
0x55555401595 <phase_5> sub $0x18,%rsp
0x55555401599 <phase_5+4> mov %fs:0x28,%rax
0x555554015a2 <phase_5+13> mov %rax,0x8(%rsp)
0x555554015a7 <phase_5+18> xor %eax,%eax
0x555554015a9 <phase_5+20> lea 0x4(%rsp),%rcx
0x555554015ae <phase_5+25> mov %rsp,%rdx
0x555554015b1 <phase_5+28> lea 0x1955(%rip),%rsi # 0x55555402f0d
0x555554015b8 <phase_5+35> callq 0x55555400fc0 <__isoc99_sscanf@plt>
0x555554015bd <phase_5+40> cmp $0x1,%eax
0x555554015c0 <phase_5+43> jle 0x5555540161c <phase_5+135>
0x555554015c2 <phase_5+45> mov (%rsp),%eax
0x555554015c5 <phase_5+48> and $0xf,%eax
0x555554015c8 <phase_5+51> mov %eax,(%rsp)
0x555554015cb <phase_5+54> cmp $0xf,%eax
0x555554015ce <phase_5+57> je 0x55555401602 <phase_5+109>
0x555554015d0 <phase_5+59> mov $0x0,%ecx
0x555554015d5 <phase_5+64> mov $0x0,%edx
0x555554015da <phase_5+69> lea 0x169f(%rip),%rsi # 0x55555402c80 <array.3418>
0x555554015e1 <phase_5+76> add $0x1,%edx
0x555554015e4 <phase_5+79> cllq
0x555554015e6 <phase_5+81> mov (%rsi,%rax,4),%eax
0x555554015e9 <phase_5+84> add %eax,%ecx
0x555554015eb <phase_5+86> cmp $0xf,%eax
0x555554015ee <phase_5+89> jne 0x555554015e1 <phase_5+76>
```

Phase 4 [진행 과정 설명]

- 페이지 4의 코드를 보니 여기도 `sscanf()`를 호출하는 부분이 보인다.

```
0000000000001520 <phase_4>:
# =====
1520: sub    $0x18,%rsp
1524: mov    %fs:0x28,%rax
152b:
152d: mov    %rax,0x8(%rsp)
1532: xor    %eax,%eax
1534: lea    0x4(%rsp),%rcx
1539: mov    %rsp,%rdx
153c: lea    0x19ca(%rip),%rsi
1543: callq  fc0 <__isoc99_sscanf@plt>
1548: cmp    $0x2,%eax # if (sscanf("%d %d", ...) != 2)
154b: jne    1553 <phase_4+0x33> # explode_bomb();
# =====
```

- 페이지 3에서 했던 것처럼, *sscanf()*의 인자로 들어가는 형식 지정자 문자열을 확인해보자.

```
(gdb) x/s $rdi
0x5555556047b0 <input_strings+240>: "12 34"

(gdb) x/s $rsi
0x555555402f0d: "%d %d"
```

- 154d번지부터 1553번지까지의 주소를 보면, *%rsp*의 메모리 주소에 저장된 값, 즉 첫 번째로 입력한 숫자가 0xe, 즉 14보다 작거나 같아야 폭탄이 터지지 않는다는 것을 알 수 있다.

```
0000000000001520 <phase_4>:
# ===== ... =====
154d:  cml    $0xe, (%rsp)          # if (`(%rsp)` <= 0xe)
1551:  jbe    1558 <phase_4+0x38> #    goto <phase_4+0x38>;
1553:  callq  1ba8 <explode_bomb>
# ===== ... =====
```

- 다음으로 주목해야 할 곳은 1565번지 주소에서 *func4()*를 호출하는 부분인데, 156a번지부터 156d번지 주소까지는 *func4()*의 반환값이 0x1f, 즉 31인지를 확인하고 있다. 아직은 *func4()*가 무슨 연산을 수행하는 함수인지 알 수 없지만, 일단 이 함수가 31을 반환해야 한다는 것은 알 수 있었다.

```
0000000000001520 <phase_4>:
# ===== ... =====
1558:  mov    $0xe,%edx
155d:  mov    $0x0,%esi
1562:  mov    (%rsp),%edi
1565:  callq  14ec <func4>
156a:  cmp    $0x1f,%eax          # if (func4(...) != 0x1f)
156d:  jne    1576 <phase_4+0x56> #    explode_bomb();
# ===== ... =====
```

- 마지막 부분은 `0x4(%rsp)`, 즉 내가 입력한 두 번째 숫자가 `0x1f`이어야 폭탄을 해체할 수 있다는 것을 뜻한다. 이제 `si`를 이용해서 `func4()` 함수 내부를 조사해보자.

```
0000000000001520 <phase_4>:
# ===== ... =====
156f:  cmpl  $0x1f,0x4(%rsp)      # if (`0x4(%rsp)` == 0x1f)
1574:  je    157b <phase_4+0x5b>   #      goto <phase_4+0x5b>;
1576:  callq 1ba8 <explode_bomb>
157b:  mov    0x8(%rsp),%rax
1580:  xor    %fs:0x28,%rax
# ===== ... =====
```

- 일단 `func4()` 함수 정의에서 150b번지, 1517번지 주소의 `callq 14ec`를 통해 이 함수가 재귀 호출을 수행하는 함수라는 것을 파악하였다. 또한, `%edx`의 초깃값은 14, `%esi`의 초깃값은 0이고, `%edi`에는 내가 첫 번째로 입력한 숫자가 저장되어 있다는 것도 알고 있다.

```
00000000000014ec <func4>:
14ec:  push  %rbx
14ed:  mov    %edx,%eax           # `%eax` = `%edx`
14ef:  sub    %esi,%eax           #      - `%esi`;
14f1:  mov    %eax,%ebx           # `%ebx` = `%eax`
14f3:  shr    $0x1f,%ebx          #      >> 31;
14f6:  add    %eax,%ebx           # `%ebx` += `%eax`;
14f8:  sar    %ebx                # `%ebx` (arithmetic) >>= 1;
14fa:  add    %esi,%ebx           # `%ebx` += `%esi`;
14fc:  cmp    %edi,%ebx           # if (`%ebx` > `%edi`)
14fe:  jg     1508 <func4+0x1c>    #      goto <func4+0x1c>;
1500:  cmp    %edi,%ebx           # if (`%ebx` < `%edi`)
1502:  jl     1514 <func4+0x28>    #      goto <func4+0x28>;
1504:  mov    %ebx,%eax           # `%eax` = `%ebx`;
1506:  pop    %rbx
1507:  retq
# ===== ... =====
```

```
00000000000014ec <func4>:
```

```
# ===== ... =====  
1508: lea    -0x1(%rbx),%edx    # `%edx` = `%rbx` - 1;  
150b: callq  14ec <func4>      # func4(...);  
1510: add    %eax,%ebx          # `%ebx` += `%eax`  
1512: jmp     1504 <func4+0x18>   # goto <func4+0x18>;  
1514: lea     0x1(%rbx),%esi     # `%esi` = `%rbx` + 1;  
1517: callq  14ec <func4>      # func4(...);  
151c: add    %eax,%ebx          # `%ebx` += `%eax`  
151e: jmp     1504 <func4+0x18>   # goto <func4+0x18>;
```

- *func4()*의 어셈블리 코드를 바탕으로 C 코드를 아래와 같이 짜보았는데, 만약 내가 이 함수를 제대로 구현했다면 이 함수의 결과값이 문제 정답의 두 번째 숫자가 될 것이다.

```
#include <stdio.h>
```

```
/*
```

```
int func4(int edi, int esi, int edx) {  
    int eax = (edx - esi), ebx = eax;
```

```
    ebx >>= 31;
```

```
    ebx += eax;
```

```
    ebx >>= 1;
```

```
    ebx += esi;
```

```
    if (ebx > edi) {  
        edx = ebx - 1;
```

```
        ebx += func4(edi, esi, edx);
```

```
    } else if (ebx < edi) {  
        esi = ebx + 1;
```

```
        ebx += func4(edi, esi, edx);
```

```
    }
```

```
    eax = ebx;
```

```

        return eax;
    }
*/

int func4(int edi, int esi, int edx) {
    int eax = (edx - esi), ebx = (((eax >> 31) + eax) >> 1) + esi;

    if (ebx > edi) ebx += func4(edi, esi, ebx - 1);
    else if (ebx < edi) ebx += func4(edi, ebx + 1, edx);

    return ebx;
}

int main(void) {
    printf("%d\n", func4(13, 0, 14));

    return 0;
}

```

```

$ make && ./src/main.out
31

```

Phase 4 [정답]

13 31

Phase 5 [결과 화면 캡처]

```
gdb objdump
B> 0x555555401595 <phase_5> sub $0x18,%rsp
0x555555401628 <phase_5+4> push %r14, %rax
0x55555540162a <phase_5+23> push %r13
0x55555540162c <phase_5+44> push %r12
0x55555540162e <phase_5+64> push %rbp
0x55555540162f <phase_5+7> push %rbx, %rcx
0x555555401630 <phase_5+8> sub $0x60,%rsp, %rsi # 0x555555402f0d
0x555555401634 <phase_5+12> mov %fs:0x28,%rax __isoc99_
0x55555540163d <phase_5+21> mov %rax,0x58(%rsp)
0x555555401642 <phase_5+26> xor %eax,0x555540161c <phase_5+135>
0x555555401644 <phase_5+28> mov %rsp,%r13
0x555555401647 <phase_5+31> and %r13,%rsi
0x55555540164a <phase_5+34> callq 0x555555401be4 <read_six_numbers>
0x55555540164f <phase_5+39> cmp $r13,%r12
0x555555401652 <phase_5+42> mov 0x0,%r14d1602 <phase_5+109>
0x555555401658 <phase_5+48> mmp $0x0,%ecx 7f 6+87>
0x55555540165a <phase_5+50> callq 0x555555401ba8 <explode_bomb>
0x55555540165f <phase_5+55> jmp 0x55555540168e <phase_5+102>x555555402c80 <array.3418>
0x555555401661 <phase_5+57> add $0x1,%ebx
0x555555401664 <phase_5+60> cmpq 5,%ebx
0x555555401667 <phase_5+63> jg 0x55555540167b <phase_5+83>
0x555555401669 <phase_5+65> addslq %ebx,%rax
0x55555540166c <phase_5+68> mov (%rsp,%rax,4),%eax
0x55555540166f <phase_5+71> jne %eax,0x0(%rbp) <phase_5+76>
672 6+74 66 6+57
native process 679582 In: phase_5 L?? PC: 0x555555401595
(gdb) c 628
Continuing.
Breakpoint 6, 0x0000555555401520 in phase_4 ()
(gdb) c
Continuing.
Breakpoint 7, 0x0000555555401595 in phase_5 ()
(gdb) c
Continuing.
Good work! On to the next...
Breakpoint 8, 0x0000555555401628 in phase_6 ()
(gdb)
```

Phase 5 [진행 과정 설명]

- 페이지 5에서도 보이는 `sscanf()` 함수...

```
0000000000001595 <phase_5>:
1595: sub    $0x18,%rsp
1599: mov    %fs:0x28,%rax
15a0:
15a2: mov    %rax,0x8(%rsp)
15a7: xor    %eax,%eax
15a9: lea    0x4(%rsp),%rcx
15ae: mov    %rsp,%rdx
15b1: lea    0x1955(%rip),%rsi          # 2f0d <array.3418+0x28d>
15b8: callq  fc0 <__isoc99_sscanf@plt>
15bd: cmp    $0x1,%eax                # if (sscanf("%d %d", ...) <= 1)
15c0: jle    161c <phase_5+0x87>        # explode_bomb();
# ===== ... =====
```

- 일단 `%rdi`와 `%rsi`부터 확인해본다.

```
(gdb) x/s $rdi
0x555555604800 <input_strings+320>: "허허허허허허허허허허허허허허허허허허"

(gdb) x/s $rsi
0x555555402f0d: "%d %d"
```

- 15c2번지부터 15ce번지까지의 주소에 해당하는 코드를 보니, $((\%rsp) \& 0xf) == 0xf$ 라는 조건을 만족하면 폭탄이 터진다는 것을 알 수 있는데, 이것은 하위 4비트가 모두 1인 15, 31, 63, 127, ... 등의 숫자는 첫 번째 숫자로 입력할 수 없다는 것을 뜻한다.

```
0000000000001595 <phase_5>:
# ===== ... =====
    15c2:  mov    (%rsp),%eax      # `eax` = `(rsp)`
    15c5:  and    $0xf,%eax        # `eax` &= 0xf;
    15c8:  mov    %eax,(%rsp)      # `(rsp)` = `eax`;
    15cb:  cmp    $0xf,%eax        # if (`eax` == 0xf)
    15ce:  je     1602 <phase_5+0x6d> #    explode_bomb();
# ===== ... =====
```

- 그 아래 15d0번지 주소의 코드부터는 %edx가 0으로 초기화되고 계속 1씩 증가하는 것, 그리고 15da번지 주소의 'array'라는 키워드와 15e6번지 주소에서 값을 대입하는 부분 등을 종합해 보았을 때, 이 코드는 %rsi가 가리키는 배열의 각 원소를 반복문을 통해 접근하는 코드인 것 같다는 생각이 들었다.

```
0000000000001595 <phase_5>:
# ===== ... =====
    15d0:  mov    $0x0,%ecx          # `%ecx` = 0;
    15d5:  mov    $0x0,%edx          # `%edx` = 0;
    15da:  lea    0x169f(%rip),%rsi   # 2c80 <array.3418>
    15e1:  add    $0x1,%edx          # `%edx`++;
    15e4:  cltq
    15e6:  mov    (%rsi,%rax,4),%eax  # `%eax` = `(%esi + 4 * %eax)`;
    15e9:  add    %eax,%ecx          # `%ecx` += `%eax`;
    15eb:  cmp    $0xf,%eax          # if (`%eax` != 0xf)
    15ee:  jne    15e1 <phase_5+0x4c> # goto <phase_5+0x4c>;
    15f0:  movl   $0xf,(%rsp)        # `(%rsp)` = 0xf;
    15f7:  cmp    $0xf,%edx          # if (`%edx` != 0xf)
    15fa:  jne    1602 <phase_5+0x6d> # explode_bomb();
    15fc:  cmp    %ecx,0x4(%rsp)      # if (`0x4(%rsp)` == `%ecx`)
    1600:  je     1607 <phase_5+0x72> # goto <phase_5+0x72>;
# ===== ... =====
```

- 일단 %rsi가 가리키는 배열의 각 원소를 한번 출력해보자.

```
(gdb) x/32wd $rsi
0x555555402c80 <array.3418>:  10      2      14      7
0x555555402c90 <array.3418+16>:  8       12     15     11
0x555555402ca0 <array.3418+32>:  0        4      1      13
0x555555402cb0 <array.3418+48>:  3        9      6       5
0x555555402cc0: 2032168787    1948284271    1802398056    1970239776
0x555555402cd0: 1851876128    1869902624    1752440944    1868701797
0x555555402ce0: 1998611053    543716457     1819440227    539779885
0x555555402cf0: 2032168804    4158831 1953066569    1768710505
```

- `phase_5()`의 어셈블리 코드와 `%rsi`가 가리키는 배열의 내용을 참고하여 C 코드를 작성해보자.

```
#include <assert.h>
#include <stdio.h>

static int array_3418[] = {
    10, 2, 14, 7,
    8, 12, 15, 11,
    0, 4, 1, 13,
    3, 9, 6, 5
};

int main(void) {
    // NOTE: x, y는 각각 내가 입력한 첫 번째와 두 번째 숫자!

    int x = 12, y = 34;

    int eax = x, *rsi = array_3418, ecx = 0, edx = 0;

    for (;;) {
        edx++;

        eax = rsi[eax]; // `mov (%rsi,%rax,4),%eax`
        ecx += eax;

        if (eax == 0x0f) break; // `cmp $0xf,%eax`
    }

    x = 0x0f;

    assert(edx == 0x0f); // `cmp $0xf,%edx`
    assert(ecx == y);    // `cmp %ecx,0x4(%rsp)`

    return 0;
}
```

```
$ ./src/main.out
main.out: src/main.c:34: main: Assertion `edx == 0x0f' failed.
Aborted
```

- *edx*가 15일 때 반복문이 종료되려면, *eax*의 값은 반드시 *rsi[6]*이 되어야 한다. 이때, *edx*가 1일 때의 *eax*의 값을 알아낸다면, 첫 번째 숫자로 무엇을 입력해야 할지 알 수 있을 것 같아 아래와 같이 표를 통해 *edx*와 *eax* 값의 변화를 관찰해보았다.

edx	eax
15	<i>rsi</i> [6] = 15
14	<i>rsi</i> [14] = 6
13	<i>rsi</i> [2] = 14
12	<i>rsi</i> [1] = 2
11	<i>rsi</i> [10] = 1
10	<i>rsi</i> [0] = 10
9	<i>rsi</i> [8] = 0
8	<i>rsi</i> [4] = 8
7	<i>rsi</i> [9] = 4
6	<i>rsi</i> [13] = 9
5	<i>rsi</i> [11] = 13
4	<i>rsi</i> [7] = 11
3	<i>rsi</i> [3] = 7
2	<i>rsi</i> [12] = 3
1	<i>rsi</i> [5] = 12

- 표를 통해, 하위 8비트 (15c5번지 주소에서 `%eax &= 0xf`가 실행되었으므로)가 `0x05`가 되는 5, 21, 37, 53, 69, 133 등이 바로 첫 번째로 입력해야 하는 수임을 알 수 있다. 그렇다면 두 번째로 입력해야 하는 숫자는 무엇일까? 바로 `%ecx`에 들어있는 값이다.

```
#include <assert.h>
#include <stdio.h>

static int array_3418[] = {
    10, 2, 14, 7,
    8, 12, 15, 11,
    0, 4, 1, 13,
    3, 9, 6, 5
};

int main(void) {
    // NOTE: x, y는 각각 내가 입력한 첫 번째와 두 번째 숫자!

    int x = 5, y = 34;

    /* 코드 생략... */

    assert(edx == 0x0f); // `cmp $0xf,%edx`
    // assert(ecx == y); // `cmp %ecx,0x4(%rsp)`

    printf("y (ecx): %d\n", ecx);

    return 0;
}
```

```
$ ./src/main.out
y (ecx): 115
```

5 115

Phase 6 [결과 화면 캡처]

```

0x55555540162a <phase_6+2>      push    %r13
0x55555540162b <main>          push    %rbx
0x55555540162c <main+1>       cmp     $0x1,%edi
0x55555540162d <main+4>       je      0x55555540128c <main+258>
0x55555540162e <main+10>      mov     %rsi,%rbp
0x55555540162f <main+13>      cmp     $0x2,%edi,%rax
0x555555401630 <main+16>      jne     0x5555554012c1 <main+311>
0x555555401631 <main+22>      mov     0x8(%rsi),%rdi
0x555555401632 <main+26>      lea     0x1e4b(%rip),%rsi    # 0x555555402ff6
0x555555401633 <main+33>      callq   0x555555400fe0 <fopen@plt>
0x555555401634 <main+38>      mov     %rax,0x2034f9(%rip,%rsi,%rax) 0x5555554046b0 <infile>
0x555555401635 <main+45>      test    %rax,%rax
0x555555401636 <main+48>      je      0x55555540129f <main+277>
0x555555401637 <main+54>      callq   0x55555540190b <initialize_bomb>
0x555555401638 <main+59>      lea     0x195c(%rip),%rdi,%rsi 0x555555402b28
0x555555401639 <main+66>      callq   0x555555400f00 <puts@plt>
0x55555540163a <main+71>      lea     0x1990(%rip),%rdi    # 0x555555402b68
0x55555540163b <main+78>      callq   0x555555400f00 <puts@plt>
0x55555540163c <main+83>      callq   0x555555401c25 <read_line>
0x55555540163d <main+88>      mov     %rax,%rdi
0x55555540163e <main+91>      callq   0x5555554012e4 <phase_1>
0x55555540163f <main+96>      callq   0x555555401d69 <phase_defused>
0x555555401640 <main+101>     lea     0x19a2(%rip),%rdi,%rsi 0x555555402b90
0x555555401641 <main+108>     callq   0x555555400f00 <puts@plt>
0x555555401642 <main+113>     callq   0x555555401c25 <read_line>

```

Breakpoint 7, 0x0000555555401595 in phase_5 ()
(gdb) c
Continuing.

Breakpoint 8, 0x0000555555401628 in phase_6 ()
(gdb) c
Continuing.

Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
[Inferior 1 (process 620555) exited normally]
(gdb)

Phase 6 [진행 과정 설명]

- 오랜만에 보이는... `read_six_numbers()` 함수를 통해 페이지 6을 해체하기 위해서는 6개의 숫자를 입력해야 한다는 것을 알 수 있다.

```

0000000000001628 <phase_6>:
# ===== ... =====

164a:    callq 1be4 <read_six_numbers>
164f:    mov     %r13,%r12
1652:    mov     $0x0,%r14d
1658:    jmp     167f <phase_6+0x57>
# ===== ... =====

```

- `read_six_numbers()`로 읽어들이는 6개의 숫자는 스택에 저장된다.

```

(gdb) x/6wd $rsp
0x7fffffffef330: 1      2      3      4
0x7fffffffef340: 5      6

```


- *display \$eax, display \$ebx, display \$r14d*를 실행하고 코드의 흐름을 관찰해보았더니, 이 코드는 *%r14d*가 제어하는 외부 *for* 문, *%ebx*가 제어하는 내부 *for* 문, 그리고 스택에 저장되어 있는 *%ebx+1*번째 숫자가 저장되는 *%eax*로 이루어진 것 같아 보였다.

```
0000000000001628 <phase_6>:
# ===== ... =====
    1661:  add    $0x1,%ebx          # `%ebx`++;
    1664:  cmp    $0x5,%ebx          # if (`%ebx` > 0x5)
    1667:  jg     167b <phase_6+0x53> # goto <phase_6+0x53>;
    1669:  movslq %ebx,%rax          # `%rax` = `%ebx`;
    166c:  mov    (%rsp,%rax,4),%eax  # `%eax` = `rsp`[%rax];
    166f:  cmp    %eax,0x0(%rbp)      # if (*( `%rbp` ) != `%eax` )
    1672:  jne    1661 <phase_6+0x39> # goto <phase_6+0x39>;
    1674:  callq  1ba8 <explode_bomb>
    1679:  jmp    1661 <phase_6+0x39> # goto <phase_6+0x39>;
    167b:  add    $0x4,%r13          # `%r13` += 0x4;
    167f:  mov    %r13,%rbp          # `%rbp` = `%r13`;
    1682:  mov    0x0(%r13),%eax      # `%eax` = *`%r13`;
    1686:  sub    $0x1,%eax
    1689:  cmp    $0x5,%eax          # if ((`%eax` - 1) > 0x5)
    168c:  ja     165a <phase_6+0x32> # explode_bomb();
    168e:  add    $0x1,%r14d         # `%r14d`++;
    1692:  cmp    $0x6,%r14d         # if (`%r14d` == 0x6)
    1696:  je     169d <phase_6+0x75> # goto <phase_6+0x75>;
    1698:  mov    %r14d,%ebx         # `%ebx` = `%r14d`;
    169b:  jmp    1669 <phase_6+0x41> # goto <phase_6+0x41>;
# ===== ... =====
```

- 그래서 169b번지 주소까지의 코드와 같은 기능을 수행하는 C 코드를 한번 짜봤더니, 이 코드가 내가 입력한 6개의 숫자가 모두 6보다 작거나 같은 숫자인지, 그리고 중복된 숫자 없이 모두 다른 숫자인지를 검사하는 코드라는 것을 알 수 있었다.

```
int main(void) {
    int rsp[] = { 1, 2, 3, 4, 5, 6 };

    int *r13 = rsp; // NOTE: `mov %rsp,%r13`
    int *r12 = r13; // NOTE: `mov %r13,%r12`

    int *rbp, rbx;

    for (int r14d = 0; r14d < 6; r14d++) {
        rbp = r13; // NOTE: `mov %r13,%rbp`

        int rax = *r13; // NOTE: `mov 0x0(%r13),%eax`

        if (rax - 1 > 5) {
            explode_bomb(); // NOTE: `ja 165a <phase_6+0x32>`

            return 0;
        }

        rbx = r14d;

        if (r14d > 0) {
            for (; rbx <= 5; rbx++) {
                rax = rsp[rax]; // NOTE: `mov (%rsp,%rax,4),%eax`

                if (rax == *rbp) {
                    explode_bomb();

                    return 0;
                }
            }
        }

        r13++; // NOTE: `add $0x4,%r13`
    }
}
```

```

    }
}

// TODO: `je <phase_6+117>`

return 0;
}

```

- 그 다음으로 살펴볼 곳은 이중 반복문을 벗어난 후에 실행되는 코드인데, $b * phase_6 + 117$ 과 `display *$rsp@6`으로 스택에 저장된 배열을 관찰해보면 배열의 모든 원소가 $\{ a, b, c, d, e, f \}$ 형태에서 $\{ 7 - a, 7 - b, 7 - c, 7 - d, 7 - e, 7 - f \}$ 형태로 변경되는 것을 확인할 수 있다.

```

00000000000001628 <phase_6>:
# ===== ... =====
169d:  lea    0x18(%r12),%rcx    # `%rcx` = `%r12 + 0x18`
16a2:  mov    $0x7,%edx         # `%edx` = 0x7;
16a7:  mov    %edx,%eax         # `%eax` = `%edx`
16a9:  sub    (%r12),%eax       #      - *(`%r12`);
16ad:  mov    %eax,(%r12)       # *(`%r12`) = `%eax`;
16b1:  add    $0x4,%r12         # `%r12`++;
16b5:  cmp    %r12,%rcx        # if (`%rcx` != `%r12`)
16b8:  jne    16a7 <phase_6+0x7f> #      goto <phase_6+0x7f>;
16ba:  mov    $0x0,%esi        # `%esi` = 0;
16bf:  jmp    16db <phase_6+0xb3> # goto <phase_6+0xb3>;
# ===== ... =====

```

- %rsi가 1씩 증가하고, %rsi의 값을 0x6과 비교하는 것으로 볼 때 이 부분도 내가 입력한 6개의 숫자를 가지고 반복문을 돌리는 코드일 것이라는 생각이 들었다.

```
0000000000001628 <phase_6>:
# ===== ... =====
16c1:  mov    0x8(%rdx),%rdx      # `%rdx` += 0x8;
16c5:  add    $0x1,%eax           # `%eax`++;
16c8:  cmp    %ecx,%eax           # if (`%eax` != `%ecx`)
16ca:  jne    16c1 <phase_6+0x99>   # goto <phase_6+0x99>;
16cc:  mov    %rdx,0x20(%rsp,%rsi,8) # *(`0x20(%rsp,%rsi,8)`) = `%rdx`;
16d1:  add    $0x1,%rsi           # `%rsi` = 0x1;
16d5:  cmp    $0x6,%rsi           # if (`%rsi` == 0x6)
16d9:  je     16f1 <phase_6+0xc9>   # goto <phase_6+0xc9>;
16db:  mov    (%rsp,%rsi,4),%ecx   # `%ecx` = `%rsp`[%rsi];
16de:  mov    $0x1,%eax           # `%eax` = 1;
16e3:  lea    0x202b46(%rip),%rdx  # 204230 <node1>
16ea:  cmp    $0x1,%ecx           # if (`%ecx` > 1)
16ed:  jg     16c1 <phase_6+0x99>   # goto <phase_6+0x99>;
16ef:  jmp    16cc <phase_6+0xa4>   # goto <phase_6+0xa4>;
# ===== ... =====
```

- “그런데... 저 *node1*이라는 주석은 왜 붙어있는 것일까...?”라는 생각이 들어 %rdx의 값을 출력해 보니 *node2*, *node3*와 *node4*가 연속적으로 저장되어 있는 것을 확인할 수 있었다. 그런데 세 번째 4바이트의 0x55604240, 0x55604250, 0x55604260, 0x55604270이라는 값을 유심히 보니... “어...? 이거 설마 다음 *node*의 메모리 주소인가...?”라는 생각이 들었다.

```
(gdb) x/16wx $rdx
0x555555604230 <node1>: 0x00000082      0x00000001      0x55604240      0x00005555
0x555555604240 <node2>: 0x00000128      0x00000002      0x55604250      0x00005555
0x555555604250 <node3>: 0x0000029f      0x00000003      0x55604260      0x00005555
0x555555604260 <node4>: 0x0000023b      0x00000004      0x55604270      0x00005555
```

- 이렇게 해서 *node1*부터 *node6*까지 모두 찾았다.

```
(gdb) x/20wx $rdx
0x555555604230 <node1>: 0x00000082      0x00000001      0x55604240      0x00005555
0x555555604240 <node2>: 0x00000128      0x00000002      0x55604250      0x00005555
0x555555604250 <node3>: 0x0000029f      0x00000003      0x55604260      0x00005555
0x555555604260 <node4>: 0x0000023b      0x00000004      0x55604270      0x00005555
0x555555604270 <node5>: 0x00000170      0x00000005      0x55604110      0x00005555
(gdb) x/4wx 0x555555604110
0x555555604110 <node6>: 0x00000235      0x00000006      0x00000000      0x00000000
```

- `p/x $rax`와 `x/4wx $rbx`로 `%rax`와 `%rbx`에 저장된 값을 확인해보니, `%rbx`에는 아까 정렬된 연결 리스트의 *k*번째 *node*의 메모리 주소가 저장되고 `%rax`에는 `%rbx`의 다음 *node*의 첫 번째 4바이트가 저장되는 것을 알 수 있었다. 그런데 1741번지 주소에서 `cmp`로 현재 *node*와 그 다음 *node*의 값을 비교하고 있으므로, 결국 페이지 6의 해답은 연결 리스트가 내림차 순으로 정렬되도록 하는 6개의 숫자가 된다.

```
0000000000001628 <phase_6>:
# ===== ... =====
1732:  mov    0x8(%rbx),%rbx      # ` %rbx ` += 0x8;
1736:  sub    $0x1,%ebp          # ` %ebp ` --;
1739:  je     174c <phase_6+0x124> # goto <phase_6+0x124>;
173b:  mov    0x8(%rbx),%rax      # ` %rax ` = ` %rbx ` + 0x8;
173f:  mov    (%rax),%eax         # ` %eax ` = *( ` %rax ` );
1741:  cmp    %eax, (%rbx)        # if ( *( ` %rbx ` ) >= ` %eax ` )
1743:  jge    1732 <phase_6+0x10a> #      goto <phase_6+0x10a>;
1745:  callq  1ba8 <explode_bomb>
174a:  jmp    1732 <phase_6+0x10a>
174c:  mov    0x58(%rsp),%rax
# ===== ... =====
```

Phase 6 [정답]

4 3 1 2 5 6

Phase 7 [결과 화면 캡처]

```

0x5555540118a <main>      push    %rbx
0x5555540118b <main+1>      cmp     $0x1,%edi
0x5555540118c <main+4>      je      0x5555540128c <main+258>
0x5555540118d <main+10>     mov     %rsi,%rbx
0x5555540118e <main+12>     cmp     $0x2,%edi
0x5555540118f <main+16>     jne     0x555554012c1 <main+311>
0x55555401190 <main+22>     mov     0x8(%rsi),%rdi
0x55555401191 <main+26>     lea     0x1e4b(%rip),%rsi    # 0x55555402ff6
0x55555401192 <main+33>     callq   0x55555400fe0 <fopen@plt>
0x55555401193 <main+38>     mov     %rax,0x2034f9(%rip)  # 0x555554046b0 <infile>
0x55555401194 <main+45>     test    %rax,%rax
0x55555401195 <main+48>     je      0x5555540129f <main+277>
0x55555401196 <main+54>     callq   0x5555540190b <initialize_bomb>
0x55555401197 <main+59>     lea     0x195c(%rip),%rdi    # 0x55555402b28
0x55555401198 <main+66>     callq   0x55555400f00 <puts@plt>
0x55555401199 <main+71>     lea     0x1990(%rip),%rdi    # 0x55555402b68
0x5555540119a <main+78>     callq   0x55555400f00 <puts@plt>
0x5555540119b <main+83>     callq   0x55555401c25 <read_line>
0x5555540119c <main+88>     mov     %rax,%rdi
0x5555540119d <main+91>     callq   0x555554012e4 <phase_1>
0x5555540119e <main+96>     callq   0x55555401d69 <phase_defused>
0x5555540119f <main+101>    lea     0x19a2(%rip),%rdi    # 0x55555402b98
0x555554011a0 <main+108>    callq   0x55555400f00 <puts@plt>
0x555554011a1 <main+113>    callq   0x55555401c25 <read_line>
  
```

native No process in: L?? PC: ??
 (gdb) c
 Continuing.
 Curses, you've found the secret phase!
 (gdb) nding it and solving it are quite different...
 Wow! You've defused the secret stage!
 Congratulations! You've defused the bomb!
 Your instructor has been notified and will verify your solution.
 [Inferior 1 (process 679582) exited normally]

Phase 7 [진행 과정 설명]

- 페이지 7의 진입에 필요한 단서는 `phase_defused()`에 있다. 먼저, `b *phase_defused+30`으로 `phase_defused()` 함수에 중단점을 걸고 `1da9`로 점프하기 위한 조건이 무엇인지 확인해본다.

```

0000000000001d69 <phase_defused>:
# ===== ... =====
1d82:  callq  1a84 <send_msg>
1d87:  cmpl   $0x6,0x20291e(%rip)    # 2046ac <num_input_strings>
1d8e:  je     1da9 <phase_defused+0x40>
# ===== ... =====
  
```

- `display *0x5555556046ac`를 통해 `num_input_strings`의 값이 어떻게 변하는지를 관찰해보았는데, 이 값은 각 페이지가 해체될 때마다 1씩 증가한다. 따라서, 페이지 7의 진입에 필요한 첫 번째 조건은 페이지 6을 해체하는 것이다.

```

(gdb) ni
0x000055555401dae in phase_defused ()
1: *0x5555556046ac = 6
  
```

- 두 번째 조건은 1dcb번지 주소의 `sscanf()`에 숨겨져 있는데, 해당 주소에 중단점을 걸고 `x/s $rdi`와 `x/s $rsi`를 통해 `%rdi`와 `%rsi`의 값을 확인해보면 `%rdi`에는 페이지 4의 정답이 되는 두 개의 숫자, 그리고 형식 문자열 `“%d %d %s”`이 저장되어 있는 것을 확인할 수 있다. 이것은 곧 페이지 7을 찾기 위해서는 페이지 4의 정답 뒤에 추가적인 문자열을 입력해야 한다는 것을 뜻한다.

```
0000000000001d69 <phase_defused>:
# ===== ... =====
    1da9: lea    0xc(%rsp),%rcx
    1dae: lea    0x8(%rsp),%rdx
    1db3: lea    0x10(%rsp),%r8
    1db8: lea    0x1198(%rip),%rsi      # 2f57 <array.3418+0x2d7>
    1dbf: lea    0x2029ea(%rip),%rdi    # 2047b0 <input_strings+0xf0>
    1dc6: mov    $0x0,%eax
    1dcb: callq  fc0 <__isoc99_sscanf@plt>
    1dd0: cmp    $0x3,%eax
    1dd3: je     1def <phase_defused+0x86>
# ===== ... =====
    1def: lea    0x10(%rsp),%rdi
    1df4: lea    0x1165(%rip),%rsi      # 2f60 <array.3418+0x2e0>
    1dfb: callq  18a4 <strings_not_equal>
    1e00: test   %eax,%eax
    1e02: jne    1dd5 <phase_defused+0x6c>
    1e04: lea    0xfad(%rip),%rdi       # 2db8 <array.3418+0x138>
    1e0b: callq  f00 <puts@plt>
    1e10: lea    0xfc9(%rip),%rdi       # 2de0 <array.3418+0x160>
    1e17: callq  f00 <puts@plt>
    1e1c: mov    $0x0,%eax
    1e21: callq  17ad <secret_phase>
    1e26: jmp    1dd5 <phase_defused+0x6c>
# ===== ... =====
```

- 그렇다면 마지막에 입력해야 하는 문자열은 과연 무엇일까? 마지막에 입력해야 하는 문자열이 무엇인지 확인하기 위해 `b *phase_defused+134`로 중단점을 걸고 `$rdi`와 `$rsi`의 값을 출력해본다.

```
(gdb) x/s $rsi
0x555555402f60: "DrEvil"
```

- 이제 프로그램을 다시 실행하고, 페이지 4의 정답에 “DrEvil”이라는 문자열을 넣어주면, 페이지 7로 진입할 수 있다.

```
(gdb) c
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

- 일단 여기서 확실하게 알 수 있는 것은, 주어진 문자열을 *long* 자료형의 숫자로 변환하는 `strtol()`³ 함수가 사용되기 때문에 입력 값은 무조건 숫자여야 한다는 것이다. 그 다음으로 알 수 있는 사실은, 입력 값이 `0x3e8 + 1`, 즉 1001보다 크면 안된다는 것이다.

```
0000000000017ad <secret_phase>:
# ===== ... =====
17ad: push    %rbx
17ae: callq   1c25 <read_line>
17b3: mov     $0xa,%edx
17b8: mov     $0x0,%esi
17bd: mov     %rax,%rdi
17c0: callq   fa0 <strtol@plt>      # strtol(`%rdi`, NULL, 10);
17c5: mov     %rax,%rbx            # `%rbx` = `%rax`;
17c8: lea     -0x1(%rax),%eax       # `%rax`--;
17cb: cmp     $0x3e8,%eax          # if (`%eax` > 0x3e8)
17d0: ja      17fd <secret_phase+0x50> # explode_bomb();
# ===== ... =====
```

³ <https://en.cppreference.com/w/c/string/byte/strtol>

- 이제 `%rdi`에 무엇이 들어있는지와 `fun7()`이 어떤 기능을 수행하는 함수인지 한번 분석해보자!

```
00000000000017ad <secret_phase>:
# ===== ... =====
17d2:  mov    %ebx,%esi
17d4:  lea     0x202975(%rip),%rdi      # 204150 <n1>
17db:  callq   176e <fun7>
17e0:  cmp     $0x2,%eax
17e3:  je      17ea <secret_phase+0x3d>
17e5:  callq   1ba8 <explode_bomb>
17ea:  lea     0x1437(%rip),%rdi      # 2c28 <_IO_stdin_used+0x188>
17f1:  callq   f00 <puts@plt>
17f6:  callq   1d69 <phase_defused>
17fb:  pop     %rbx
17fc:  retq
17fd:  callq   1ba8 <explode_bomb>
1802:  jmp     17d2 <secret_phase+0x25>
# ===== ... =====
```

- 아무래도 `%rdi`는 구조체나 배열을 가리킬 것 같아 보여서 17db번지 주소에 중단점을 설정해놓고 `%rdi`가 가리키는 메모리 공간을 조사해보니... 헉... 이게 다 뭐지...

```
(gdb) x/32wx $rdi
0x555555604150 <n1>:  0x00000024      0x00000000      0x55604170      0x00005555
0x555555604160 <n1+16>: 0x55604190      0x00005555      0x00000000      0x00000000
0x555555604170 <n21>:  0x00000008      0x00000000      0x556041f0      0x00005555
0x555555604180 <n21+16>: 0x556041b0      0x00005555      0x00000000      0x00000000
0x555555604190 <n22>:  0x00000032      0x00000000      0x556041d0      0x00005555
0x5555556041a0 <n22+16>: 0x55604210      0x00005555      0x00000000      0x00000000
0x5555556041b0 <n32>:  0x00000016      0x00000000      0x556040b0      0x00005555
0x5555556041c0 <n32+16>: 0x55604070      0x00005555      0x00000000      0x00000000
```

- 일단 생긴 거로 보았을 때는 페이지 6의 *node*처럼 첫 번째 4바이트는 특정 값을 나타내고, 세 번째 ~ 네 번째 8바이트 (0x0000555555604010, 0x0000555555604030, ...)는 메모리 주소를 나타내는 것 같다는 생각이 들었다. 그렇다면 첫 번째 4바이트 다음에 보이는 빈 공간은... 그냥 구조체 패딩 바이트 (struct padding)⁴라고 생각하면 되려나...?

```
(gdb) x/32wx $rdi+128
0x5555556041d0 <n33>:  0x0000002d      0x00000000      0x55604010      0x00005555
0x5555556041e0 <n33+16>: 0x556040d0      0x00005555      0x00000000      0x00000000
0x5555556041f0 <n31>:  0x00000006      0x00000000      0x55604030      0x00005555
0x555555604200 <n31+16>: 0x55604090      0x00005555      0x00000000      0x00000000
0x555555604210 <n34>:  0x0000006b      0x00000000      0x55604050      0x00005555
0x555555604220 <n34+16>: 0x556040f0      0x00005555      0x00000000      0x00000000
0x555555604230 <node1>: 0x00000082      0x00000001      0x00000000      0x00000000
0x555555604240 <node2>: 0x00000128      0x00000002      0x55604230      0x00005555
```

- 이번에는 *fun7()* 함수를 살펴보니, 페이지 4에서 봤던 것과 비슷하게 재귀 호출을 하는 함수라는 것을 알 수 있었다.

```
00000000000176e <fun7>:
    176e: test    %rdi,%rdi          # if (`%rdi` == NULL)
    1771: je      17a7 <fun7+0x39>    # goto <fun7+0x39>;
    1773: sub     $0x8,%rsp          # `%rsp` -= 0x8;
    1777: mov     (%rdi),%edx        # `%edx` = *(`%rdi`);
    1779: cmp     %esi,%edx          # if (`%edx` > `%esi`)
    177b: jg      178b <fun7+0x1d>    # goto <fun7+0x1d>;
    177d: mov     $0x0,%eax          # `%eax` = 0;
    1782: cmp     %esi,%edx          # if (`%edx` != `%esi`)
    1784: jne     1798 <fun7+0x2a>    # goto <fun7+0x2a>;
# ===== ... =====
    178b: mov     0x8(%rdi),%rdi      # `%rdi` += 0x8;
    178f: callq   176e <fun7>        # fun7();
    1794: add     %eax,%eax          # `%eax` += `%eax`;
    1796: jmp     1786 <fun7+0x18>    # goto <fun7+0x18>;
    1798: mov     0x10(%rdi),%rdi    # `%rdi` += 0x10;
```

⁴ <https://en.cppreference.com/w/c/language/object>

```

179c:  callq  176e <fun7>          # fun7();
17a1:  lea     0x1(%rax,%rax,1),%eax # `%eax` = `0x1(%rax,%rax,1)`;
17a5:  jmp     1786 <fun7+0x18>      # goto <fun7+0x18>;
17a7:  mov     $0xffffffff,%eax     # `%eax` = 0xffffffff;
17ac:  retq

```

- *fun7()*와 같은 기능을 수행하는 함수를 C 코드로 짜보면 아래와 같은 형태가 되는데, 이진 트리를 탐색하는 코드와 매우 유사함을 알 수 있었다.

```

typedef struct _n {
    int x, *y, *z;
} n;

int fun7(n *rdi, int rsi) {
    // NOTE: `test %rdi,%rdi`
    if (rdi == NULL) return 0xFFFFFFFF;

    int rax, rdx = rdi->x; // NOTE: `mov (%rdi),%edx`

    if (rdx > rsi) {
        rdi = rdi->y; // NOTE: `mov 0x8(%rdi),%rdi`

        rax = fun7(rdi, rsi);

        rax += rax; // NOTE: `add %eax,%eax`

        return rax;
    }

    rax = 0;

    if (rdx != rsi) {
        rdi = rdi->z; // NOTE: `mov 0x10(%rdi),%rdi`

        rax = fun7(rdi, rsi);
    }
}

```

```
    rax = 1 + rax + rax;  // NOTE: `lea 0x1(%rax,%rax,1),%eax`  
  
    return rax;  
} else {  
    return rax;  
}  
}
```

Phase 7 [정답]

- 페이지 6: 스코어보드

← ↻ 🏠 ⚠️ 안전하지 않음 | 219.254.89.47:10000/scoreboard

Google NAVER YouTube

Bomb Lab Scoreboard

This page contains the latest information that we have received from your bomb. If your solution is marked **invalid**, this means your bomb reported a solution that didn't actually defuse your bomb.

Last updated: Tue Oct 17 16:41:05 2023 (updated every 30 secs)

#	Bomb number	Submission date	Phases defused	Explosions	Score	Status
1	bomb15	Tue Oct 17 13:36	7	0	70	valid
2	bomb16	Tue Oct 17 16:40	6	2	69	valid
3	bomb6	Tue Oct 17 15:56	4	0	40	valid
4	bomb33	Tue Oct 17 16:15	4	3	39	valid
5	bomb35	Tue Oct 17 16:21	3	1	30	valid
6	bomb48	Tue Oct 17 13:30	2	0	20	valid
7	bomb30	Tue Oct 17 13:32	2	0	20	valid
8	bomb23	Tue Oct 17 13:33	2	0	20	valid
9	bomb22	Tue Oct 17 14:34	2	0	20	valid
10	bomb50	Tue Oct 17 12:28	1	0	10	valid
11	bomb14	Tue Oct 17 12:29	1	0	10	valid
12	bomb19	Tue Oct 17 12:29	1	0	10	valid
13	bomb38	Tue Oct 17 12:29	1	0	10	valid
14	bomb12	Tue Oct 17 12:33	1	0	10	valid
15	bomb36	Tue Oct 17 12:33	1	0	10	valid

- 페이지 7: 스코어보드

Bomb Lab Scoreboard

This page contains the latest information that we have received from your bomb. If your solution is marked **invalid**, this means your bomb reported a solution that didn't actually defuse your bomb.

Last updated: Tue Oct 17 18:16:25 2023 (updated every 30 secs)

#	Bomb number	Submission date	Phases defused	Explosions	Score	Status
1	bomb15	Tue Oct 17 13:36	7	0	70	valid
2	bomb16	Tue Oct 17 18:16	7	2	69	valid
3	bomb6	Tue Oct 17 17:19	5	0	55	valid
4	bomb35	Tue Oct 17 17:49	5	1	55	valid
5	bomb33	Tue Oct 17 17:44	5	3	54	valid
6	bomb48	Tue Oct 17 13:30	2	0	20	valid
7	bomb30	Tue Oct 17 13:32	2	0	20	valid
8	bomb23	Tue Oct 17 13:33	2	0	20	valid
9	bomb22	Tue Oct 17 14:34	2	0	20	valid
10	bomb8	Tue Oct 17 17:21	2	0	20	valid
11	bomb50	Tue Oct 17 12:28	1	0	10	valid
12	bomb14	Tue Oct 17 12:29	1	0	10	valid
13	bomb19	Tue Oct 17 12:29	1	0	10	valid