2018 시스템 프로그래밍
- Lab 05 -

제출일자	2018.11.01
분 반	00
이 름	노효근
학 번	201502049

Phase 1 [결과 화면 캡처]

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
The future will be better tomorrow.
```

Phase 1 [진행 과정 설명]

```
Breakpoint 1, 0x0000000000400f2d in phase_1 ()
(gdb) disas phase_1
Dump of assembler code for function phase_1:
=> 0x0000000000400f2d <+0>:     sub    $0x8,%rsp
   0x0000000000400f31 <+4>:     mov    $0x402610,%esi
   0x0000000000400f36 <+9>:     callq  0x40137c <strings_not_equal>
   0x0000000000400f3b <+14>:    test   %eax,%eax
   0x0000000000400f3d <+16>:    je     0x400f44 <phase_1+23>
   0x0000000000400f3f <+18>:    callq  0x401650 <explode_bomb>
   0x0000000000400f44 <+23>:    add    $0x8,%rsp
   0x0000000000400f48 <+27>:    retq
```

break_point를 phase_1과 explode_bomb 설정 후 disas 명령어를 통해 phase_1 단계의 어셈블리어를 해석해 보았다.

call 명령어를 통해 string_not_equal 함수를 호출하였고 그와 비교하여 맞다면 폭탄을 해체할 수 있게 설계가 된 구조임을 알수 있었다.

따라서 %esi의 값을 확인하기 위해 x/s 0x402690을 확인하였더니,

The future will be better tomorrow. 이라는 문자열이 나왔고, 이것이 phase_1의 정답이다.

Phase 1 [정답]

The future will be better tomorrow.

Phase 2 [결과 화면 캡처]

```
Continuing.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
```

Phase 2 [진행 과정 설명]

```
Dump of assembler code for function phase_2:
=> 0x0000000000400f49 <+0>:     push   %rbp
   0x0000000000400f4a <+1>:     push   %rbx
   0x0000000000400f4b <+2>:     sub    $0x28,%rsp
   0x0000000000400f4f <+6>:     mov    %fs:0x28,%rax
   0x0000000000400f58 <+15>:    mov    %rax,0x18(%rsp)
   0x0000000000400f5d <+20>:    xor    %eax,%eax
   0x0000000000400f5f <+22>:    mov    %rsp,%rsi
   0x0000000000400f62 <+25>:    callq  0x401686 <read_six_numbers>
   0x0000000000400f67 <+30>:    cmpl   $0x0,(%rsp)
   0x0000000000400f6b <+34>:    jns    0x400f72 <phase_2+41>
   0x0000000000400f6d <+36>:    callq  0x401650 <explode_bomb>
   0x0000000000400f72 <+41>:    mov    %rsp,%rbp
   0x0000000000400f75 <+44>:    mov    $0x1,%ebx
   0x0000000000400f7a <+49>:    mov    %ebx,%eax
   0x0000000000400f7c <+51>:    add    0x0(%rbp),%eax
   0x0000000000400f7f <+54>:    cmp    %eax,0x4(%rbp)
   0x0000000000400f82 <+57>:    je     0x400f89 <phase_2+64>
   0x0000000000400f84 <+59>:    callq  0x401650 <explode_bomb>
   0x0000000000400f89 <+64>:    add    $0x1,%ebx
   0x0000000000400f8c <+67>:    add    $0x4,%rbp
   0x0000000000400f90 <+71>:    cmp    $0x6,%ebx
   0x0000000000400f93 <+74>:    jne    0x400f7a <phase_2+49>
---Type <return> to continue, or q <return> to quit---
   0x0000000000400f95 <+76>:    mov    0x18(%rsp),%rax
   0x0000000000400f9a <+81>:    xor    %fs:0x28,%rax
   0x0000000000400fa3 <+90>:    je     0x400faa <phase_2+97>
   0x0000000000400fa5 <+92>:    callq  0x400b90 <__stack_chk_fail@plt>
   0x0000000000400faa <+97>:    add    $0x28,%rsp
   0x0000000000400fae <+101>:   pop    %rbx
   0x0000000000400faf <+102>:   pop    %rbp
   0x0000000000400fb0 <+103>:   retq
End of assembler dump.
```

break_point를 phase_2과 explode_bomb 설정 후 disas 명령어를 통해 phase_2 단계의 어셈블리어를 해석해 보았다.

call 명령어를 통해 read_six_numbers 함수를 호출하였고 이를 통해 phase_2의 답은 input값이 6개인걸 알 수 있다. 이로써, <+34>줄의 조건을 충족하여 첫 번째 폭탄을 넘기고, 다음을 확인해 보았다. 이후 <+54>줄을 통해 %eax와 내가 넣은 첫 번째 값을 비교하고 있다.

<+64~+67>을 통해 6개의 숫자 간 규칙성이 있는 것을 알 수 있었다.

이를 통해 6개 자리수의 규칙성은 등차가 1부터 5까지 증가하는 것이었고, 따라서

1 2 4 7 11 16 임을 알게 되었다. 이것이 phase_2의 정답이다.

Phase 2 [정답]

1 2 4 7 11 16

Phase 3 [결과 화면 캡처]

```
That's number 2.  Keep going!
3 840
```

Phase 3 [진행 과정 설명]

```
Dump of assembler code for function phase_3:
=> 0x0000000000400fbl <+0>:     sub    $0x18,%rsp
   0x0000000000400fb5 <+4>:     mov    %fs:0x28,%rax
   0x0000000000400fbe <+13>:    mov    %rax,0x8(%rsp)
   0x0000000000400fc3 <+18>:    xor    %eax,%eax
   0x0000000000400fc5 <+20>:    lea    0x4(%rsp),%rcx
   0x0000000000400fca <+25>:    mov    %rsp,%rdx
   0x0000000000400fcd <+28>:    mov    $0x40292d,%esi
   0x0000000000400fd2 <+33>:    callq  0x400c40 <__isoc99_sscanf@plt>
   0x0000000000400fd7 <+38>:    cmp    $0x1,%eax
   0x0000000000400fda <+41>:    jg     0x400fel <phase_3+48>
   0x0000000000400fdc <+43>:    callq  0x401650 <explode_bomb>
   0x0000000000400fel <+48>:    cmpl   $0x7,(%rsp)
   0x0000000000400fe5 <+52>:    ja     0x401022 <phase_3+113>
   0x0000000000400fe7 <+54>:    mov    (%rsp),%eax
   0x0000000000400fea <+57>:    jmpq   *0x402660(,%rax,8)
   0x0000000000400ffl <+64>:    mov    $0x1b7,%eax
   0x0000000000400ff6 <+69>:    jmp    0x401033 <phase_3+130>
   0x0000000000400ff8 <+71>:    mov    $0x272,%eax
   0x0000000000400ffd <+76>:    jmp    0x401033 <phase_3+130>
   0x0000000000400fff <+78>:    mov    $0x348,%eax
   0x0000000000401004 <+83>:    jmp    0x401033 <phase_3+130>
   0x0000000000401006 <+85>:    mov    $0x385,%eax
   0x000000000040100b <+90>:    jmp    0x401033 <phase_3+130>
   0x000000000040100d <+92>:    mov    $0x3c7,%eax
   0x0000000000401012 <+97>:    jmp    0x401033 <phase_3+130>
   0x0000000000401014 <+99>:    mov    $0x134,%eax
   0x0000000000401019 <+104>:   jmp    0x401033 <phase_3+130>
   0x000000000040101b <+106>:   mov    $0x2fc,%eax
   0x0000000000401020 <+111>:   jmp    0x401033 <phase_3+130>
   0x0000000000401022 <+113>:   callq  0x401650 <explode_bomb>
   0x0000000000401027 <+118>:   mov    $0x0,%eax
   0x000000000040102c <+123>:   jmp    0x401033 <phase_3+130>
   0x000000000040102e <+125>:   mov    $0x2a2,%eax
   0x0000000000401033 <+130>:   cmp    0x4(%rsp),%eax
   0x0000000000401037 <+134>:   je     0x40103e <phase_3+141>
   0x0000000000401039 <+136>:   callq  0x401650 <explode_bomb>
   0x000000000040103e <+141>:   mov    0x8(%rsp),%rax
   0x0000000000401043 <+146>:   xor    %fs:0x28,%rax
   0x000000000040104c <+155>:   je     0x401053 <phase_3+162>
   0x000000000040104e <+157>:   callq  0x400b90 <__stack_chk_fail@plt>
   0x0000000000401053 <+162>:   add    $0x18,%rsp
   0x0000000000401057 <+166>:   retq
---Type <return> to continue, or q <return> to quit---
End of assembler dump.
```

break_point를 phase_3과 explode_bomb 설정 후 disas 명령어를 통해 phase_3 단계의 어셈블리어를 해석해 보았다.

call 명령어를 통해 __isoc99_sscanf@plt 함수를 호출하였고 이를 통해 input값을 받는다는 것을 유추 할 수 있었다. __isoc99_sscanf@plt에 어떤 형태의 값이 들어가는지 확인하니, "%d , %d " 의 정수 값 2개를 입력하는 형태로 나타났다.

callq 이후 %eax 값과 1을 비교하여 2개의 값을 잘 받았는지 확인하고
(%rsp)값과 7를 비교하여 첫 번째 값이 7보다 작아야 함을 알 수 있었다.
또한 다수의 jmp 구문으로 보아 phase_3는 switch_case 문인걸 확인할 수있었고,
그에 따라 0x402660의 주소 값을 확인 해보았다.

```
(gdb) x /6gx 0x402660
0x402660:       0x000000000040102e      0x0000000000400ff1
0x402670:       0x0000000000400ff8      0x0000000000400fff
0x402680:       0x0000000000401006      0x000000000040100d
```

다음과 같은 값이 나오는걸 확인 하였고, 따라서 각 숫자에 대한 case문으로의 분기점을 알 수 있었다. 난 첫 숫자를 "3" 으로 입력 하였으므로 0x000000000040fff번지로 이동하여 보니, $0x348값이 있었고 이를 10진수로 나타내보니 "840"의 값이 나왔다.
따라서 6가지의 경우 중 하나인 " 3 840 " 이것이 phase_3의 정답이다.

Phase 3 [정답]

3 840

Phase 4 [결과 화면 캡처]

```
Continuing.
Halfway there!
10 5
```

Phase 4 [진행 과정 설명]

```
Dump of assembler code for function phase_4:
=> 0x0000000000401096 <+0>:     sub    $0x18,%rsp
   0x000000000040109a <+4>:     mov    %fs:0x28,%rax
   0x00000000004010a3 <+13>:    mov    %rax,0x8(%rsp)
   0x00000000004010a8 <+18>:    xor    %eax,%eax
   0x00000000004010aa <+20>:    lea    0x4(%rsp),%rcx
   0x00000000004010af <+25>:    mov    %rsp,%rdx
   0x00000000004010b2 <+28>:    mov    $0x40292d,%esi
   0x00000000004010b7 <+33>:    callq  0x400c40 <__isoc99_sscanf@plt>
   0x00000000004010bc <+38>:    cmp    $0x2,%eax
   0x00000000004010bf <+41>:    jne    0x4010c7 <phase_4+49>
   0x00000000004010c1 <+43>:    cmpl   $0xe,(%rsp)
   0x00000000004010c5 <+47>:    jbe    0x4010cc <phase_4+54>
   0x00000000004010c7 <+49>:    callq  0x401650 <explode_bomb>
   0x00000000004010cc <+54>:    mov    $0xe,%edx
   0x00000000004010d1 <+59>:    mov    $0x0,%esi
   0x00000000004010d6 <+64>:    mov    (%rsp),%edi
   0x00000000004010d9 <+67>:    callq  0x401058 <func4>
   0x00000000004010de <+72>:    cmp    $0x5,%eax
   0x00000000004010e1 <+75>:    jne    0x4010ea <phase_4+84>
   0x00000000004010e3 <+77>:    cmpl   $0x5,0x4(%rsp)
   0x00000000004010e8 <+82>:    je     0x4010ef <phase_4+89>
   0x00000000004010ea <+84>:    callq  0x401650 <explode_bomb>
   0x00000000004010ef <+89>:    mov    0x8(%rsp),%rax
   0x00000000004010f4 <+94>:    xor    %fs:0x28,%rax
   0x00000000004010fd <+103>:   je     0x401104 <phase_4+110>
   0x00000000004010ff <+105>:   callq  0x400b90 <__stack_chk_fail@plt>
   0x0000000000401104 <+110>:   add    $0x18,%rsp
   0x0000000000401108 <+114>:   retq
```

break_point를 phase_1과 explode_bomb 설정 후 disas 명령어를 통해 phase_4 단계의 어셈블리어를 해석해 보았다. 위와 마찬가지로 call 명령어를 통해 __isoc99_sscanf@plt 함수를 호출하였고 이를 통해 input값을 받는다는 것을 유추 할 수 있었다. __isoc99_sscanf@plt에 어떤 형태의 값이 들어가는지 확인하니, "%d , %d " 의 정수 값 2개를 입력하는 형태로 나타났다.

만약 scanf로 받아온 숫자가 2개가 아닐 경우는 폭탄으로 분기하고, 그렇지 않을 경우는 첫 번째 입력한 수와 0xe(=14)를 비교하여 14보다 더 크면 폭탄으로 분기하고, 14보다 작은 경우에는 <+54>으로 분기한다. 따라서 첫 번째 숫자는 14보다 작아야한다.

이후 값을 받아 <func 4>를 불러 함수를 실행한다.

<func4> 함수 구성은 다음과 같다.

```
Dump of assembler code for function func4:
   0x0000000000401058 <+0>:     sub    $0x8,%rsp
   0x000000000040105c <+4>:     mov    %edx,%eax
   0x000000000040105e <+6>:     sub    %esi,%eax
   0x0000000000401060 <+8>:     mov    %eax,%ecx
   0x0000000000401062 <+10>:    shr    $0x1f,%ecx
   0x0000000000401065 <+13>:    add    %ecx,%eax
   0x0000000000401067 <+15>:    sar    %eax
   0x0000000000401069 <+17>:    lea    (%rax,%rsi,1),%ecx
   0x000000000040106c <+20>:    cmp    %edi,%ecx
   0x000000000040106e <+22>:    jle    0x40107c <func4+36>
   0x0000000000401070 <+24>:    lea    -0x1(%rcx),%edx
   0x0000000000401073 <+27>:    callq  0x401058 <func4>
   0x0000000000401078 <+32>:    add    %eax,%eax
   0x000000000040107a <+34>:    jmp    0x401091 <func4+57>
   0x000000000040107c <+36>:    mov    $0x0,%eax
   0x0000000000401081 <+41>:    cmp    %edi,%ecx
   0x0000000000401083 <+43>:    jge    0x401091 <func4+57>
   0x0000000000401085 <+45>:    lea    0x1(%rcx),%esi
   0x0000000000401088 <+48>:    callq  0x401058 <func4>
   0x000000000040108d <+53>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401091 <+57>:    add    $0x8,%rsp
   0x0000000000401095 <+61>:    retq
End of assembler dump.
```

구조를 보니 값이 맞지 않으면 func4 계속 호출하는 형태가 보이므로 해당 phase의 문제는 재귀함수 문제로 알수 있었다. 따라서 재귀함수인 func4를 c언어로 분석 하면 다음과 같다.

```
void func4(int x, int y, int z) {
    int t = z - y;
    int k = t >> 31;
    t = (t + k) >> 1;
    k = t + y;
    if(k <= x) {
        t = 0;
        if(k >= x) return;
        else { y = k + 1; func4(x, y, z); }
    } else {  z = k – 1;  func4(x, y, z); }
}
```

따라서 다음을 만족하는 값을 찾아보니 그 중 하나가 10이었다.
이후 다음 두 번째 숫자는 $0x5와 비교하여 같지 않다면 폭탄으로 분기하게 되므로 두 번째 숫자는 "5"가 됨을 알 수 있었다. 따라서 " 10 5 " 이 phase_4의 정답이다.

Phase 4 [정답]

10 5

Phase 5 [결과 화면 캡처]

```
Continuing.
So you got that one.  Try this one.
15:<>?
```

Phase 5 [진행 과정 설명]

```
Breakpoint 5, 0x0000000000401109 in phase_5 ()
(gdb) disas phase_5
Dump of assembler code for function phase_5:
=> 0x0000000000401109 <+0>:     push   %rbx
   0x000000000040110a <+1>:     mov    %rdi,%rbx
   0x000000000040110d <+4>:     callq  0x40135e <string_length>
   0x0000000000401112 <+9>:     cmp    $0x6,%eax
   0x0000000000401115 <+12>:    je     0x40111c <phase_5+19>
   0x0000000000401117 <+14>:    callq  0x401650 <explode_bomb>
   0x000000000040111c <+19>:    mov    %rbx,%rax
   0x000000000040111f <+22>:    lea    0x6(%rbx),%rdi
   0x0000000000401123 <+26>:    mov    $0x0,%ecx
   0x0000000000401128 <+31>:    movzbl (%rax),%edx
   0x000000000040112b <+34>:    and    $0xf,%edx
   0x000000000040112e <+37>:    add    0x4026a0(,%rdx,4),%ecx
   0x0000000000401135 <+44>:    add    $0x1,%rax
   0x0000000000401139 <+48>:    cmp    %rdi,%rax
   0x000000000040113c <+51>:    jne    0x401128 <phase_5+31>
   0x000000000040113e <+53>:    cmp    $0x4f,%ecx
   0x0000000000401141 <+56>:    je     0x401148 <phase_5+63>
   0x0000000000401143 <+58>:    callq  0x401650 <explode_bomb>
   0x0000000000401148 <+63>:    pop    %rbx
   0x0000000000401149 <+64>:    retq
End of assembler dump.
```

break_point를 phase_5과 explode_bomb 설정 후 disas 명령어를 통해 phase_5 단계의 어셈블리어를 해석해 보았다.

callq함수가 string_length함수를 불러오는 것을 보아 어떠한 문자열을 입력해야 한다는 것을 알았다. 그리고 %eax와 6을 비교하여 같지 않을 경우 폭탄으로 가는 것을 보아 6개의 string을 입력해야 한다는 사실도 유추할 수 있었다.

입력한 string이 6개이면 <+19>로 분기하고 %rax와 %rdi에 값을 지정하고 %ecx에 0을 저장한다. 그 다음 %edx값과 0xf(= 0000 0000 0000 0000 0000 0000 0000 1111)를 AND연산하여 %ecx의 마지막 4비트만 남겨준다. 그런 다음 0x402720+(4*%rcx)값을 %edx로 옮겨준다. 이 과정을 보며 x/16wd 명령어를 통하여 0x4026a0에 어떤 값이 있는지 확인하였다.

```
(gdb) x /16wd 0x4026a0
0x4026a0 <array.3601>:  2       10      6       1
0x4026b0 <array.3601+16>:       12      16      9       3
0x4026c0 <array.3601+32>:       4       7       14      5
0x4026d0 <array.3601+48>:       11      8       15      13
```

다음과 같은 배열이 나오는 것을 확인하였다.

이후 %ecx값과 $0x4f의 값을 비교하여 pop하고 함수가 종료됨을 알 수 있다.

$0x4f = 79 이고, 따라서 배열의 합이 79가 되어야 함을 알 수 있었다.

0	1	2	3	4	5	6	7
2	10	6	1	12	16	9	3
8	9	10(a)	11(b)	12(c)	13(d)	14(e)	15(f)
4	7	14	5	11	8	15	13

10+16+14+11+15+13 = 79임을 찾아냈고, 따라서 해당 index의 값은 1,5,10,12,14,15였다

0110000	0
0110001	1
0110010	2
0110011	3
0110100	4
0110101	5
0110110	6
0110111	7
0111000	8
0111001	9
0111010	:
0111011	;
0111100	<
0111101	=
0111110	>
0111111	?

이때, 10,12,14,15는 아스키코드를 참고하여 각각 : < > ? 임을 알수 있었고, 따라서 1 5 : < > ? 이 phase_5의 정답이다.

Phase 5 [정답]

1 5 : < > ?

Phase 6 [결과 화면 캡처]

```
Continuing.
Good work!  On to the next...
3 1 2 5 4 6
```

Phase 6 [진행 과정 설명]

```
Dump of assembler code for function phase_6:
=> 0x000000000040114a <+0>:     push   %r13
   0x000000000040114c <+2>:     push   %r12
   0x000000000040114e <+4>:     push   %rbp
   0x000000000040114f <+5>:     push   %rbx
   0x0000000000401150 <+6>:     sub    $0x68,%rsp
   0x0000000000401154 <+10>:    mov    %fs:0x28,%rax
   0x000000000040115d <+19>:    mov    %rax,0x58(%rsp)
   0x0000000000401162 <+24>:    xor    %eax,%eax
   0x0000000000401164 <+26>:    mov    %rsp,%rsi
   0x0000000000401167 <+29>:    callq  0x401686 <read_six_numbers>
   0x000000000040116c <+34>:    mov    %rsp,%r12
   0x000000000040116f <+37>:    mov    $0x0,%r13d
   0x0000000000401175 <+43>:    mov    %r12,%rbp
   0x0000000000401178 <+46>:    mov    (%r12),%eax
   0x000000000040117c <+50>:    sub    $0x1,%eax
   0x000000000040117f <+53>:    cmp    $0x5,%eax
   0x0000000000401182 <+56>:    jbe    0x401189 <phase_6+63>
   0x0000000000401184 <+58>:    callq  0x401650 <explode_bomb>
   0x0000000000401189 <+63>:    add    $0x1,%r13d
   0x000000000040118d <+67>:    cmp    $0x6,%r13d
   0x0000000000401191 <+71>:    je     0x4011d0 <phase_6+134>
   0x0000000000401193 <+73>:    mov    %r13d,%ebx
   0x0000000000401196 <+76>:    movslq %ebx,%rax
   0x0000000000401199 <+79>:    mov    (%rsp,%rax,4),%eax
   0x000000000040119c <+82>:    cmp    %eax,0x0(%rbp)
   0x000000000040119f <+85>:    jne    0x4011a6 <phase_6+92>
   0x00000000004011a1 <+87>:    callq  0x401650 <explode_bomb>
   0x00000000004011a6 <+92>:    add    $0x1,%ebx
   0x00000000004011a9 <+95>:    cmp    $0x5,%ebx
   0x00000000004011ac <+98>:    jle    0x401196 <phase_6+76>
   0x00000000004011ae <+100>:   add    $0x4,%r12
   0x00000000004011b2 <+104>:   jmp    0x401175 <phase_6+43>
   0x00000000004011b4 <+106>:   mov    0x8(%rdx),%rdx
   0x00000000004011b8 <+110>:   add    $0x1,%eax
   0x00000000004011bb <+113>:   cmp    %ecx,%eax
   0x00000000004011bd <+115>:   jne    0x4011b4 <phase_6+106>
   0x00000000004011bf <+117>:   mov    %rdx,0x20(%rsp,%rsi,2)
   0x00000000004011c4 <+122>:   add    $0x4,%rsi
   0x00000000004011c8 <+126>:   cmp    $0x18,%rsi
   0x00000000004011cc <+130>:   jne    0x4011d5 <phase_6+139>
   0x00000000004011ce <+132>:   jmp    0x4011e9 <phase_6+159>
```

```
   0x00000000004011d0 <+134>:   mov    $0x0,%esi
   0x00000000004011d5 <+139>:   mov    (%rsp,%rsi,1),%ecx
   0x00000000004011d8 <+142>:   mov    $0x1,%eax
   0x00000000004011dd <+147>:   mov    $0x6042f0,%edx
   0x00000000004011e2 <+152>:   cmp    $0x1,%ecx
   0x00000000004011e5 <+155>:   jg     0x4011b4 <phase_6+106>
   0x00000000004011e7 <+157>:   jmp    0x4011bf <phase_6+117>
   0x00000000004011e9 <+159>:   mov    0x20(%rsp),%rbx
   0x00000000004011ee <+164>:   lea    0x20(%rsp),%rax
   0x00000000004011f3 <+169>:   lea    0x48(%rsp),%rsi
   0x00000000004011f8 <+174>:   mov    %rbx,%rcx
   0x00000000004011fb <+177>:   mov    0x8(%rax),%rdx
   0x00000000004011ff <+181>:   mov    %rdx,0x8(%rcx)
   0x0000000000401203 <+185>:   add    $0x8,%rax
   0x0000000000401207 <+189>:   mov    %rdx,%rcx
   0x000000000040120a <+192>:   cmp    %rsi,%rax
   0x000000000040120d <+195>:   jne    0x4011fb <phase_6+177>
   0x000000000040120f <+197>:   movq   $0x0,0x8(%rdx)
   0x0000000000401217 <+205>:   mov    $0x5,%ebp
   0x000000000040121c <+210>:   mov    0x8(%rbx),%rax
   0x0000000000401220 <+214>:   mov    (%rax),%eax
   0x0000000000401222 <+216>:   cmp    %eax,(%rbx)
   0x0000000000401224 <+218>:   jge    0x40122b <phase_6+225>
   0x0000000000401226 <+220>:   callq  0x401650 <explode_bomb>
   0x000000000040122b <+225>:   mov    0x8(%rbx),%rbx
   0x000000000040122f <+229>:   sub    $0x1,%ebp
   0x0000000000401232 <+232>:   jne    0x40121c <phase_6+210>
   0x0000000000401234 <+234>:   mov    0x58(%rsp),%rax
   0x0000000000401239 <+239>:   xor    %fs:0x28,%rax
   0x0000000000401242 <+248>:   je     0x401249 <phase_6+255>
   0x0000000000401244 <+250>:   callq  0x400b90 <__stack_chk_fail@plt>
   0x0000000000401249 <+255>:   add    $0x68,%rsp
   0x000000000040124d <+259>:   pop    %rbx
   0x000000000040124e <+260>:   pop    %rbp
   0x000000000040124f <+261>:   pop    %r12
   0x0000000000401251 <+263>:   pop    %r13
   0x0000000000401253 <+265>:   retq
End of assembler dump.
```

break_point를 phase_6과 explode_bomb 설정 후 disas 명령어를 통해 phase_6 단계의 어셈블리어를 해석해 보았다.

call 명령어를 통해 read_six_numbers 함수를 호출하였고 이를 통해 phase_6의 답은 input값이 6개인걸 알 수 있다.

<+50~53> 줄을 통해 첫 번째 입력 값이 양수 이며 또한 1~6사이 이어야 한다는 걸 알 수 있다. <+63~67>줄을 통해 내가 넣은 값 6개가 맞는지에 대한 반복문을 볼 수 있다.

<+92~95>줄을 보면 내가 넣은 6개의 각 값 비교를 5번 하는 것을 확인하였다.

그리하여 어떤 값을 비교하는지 알고 싶어 다음과 같이 확인하였다.

```
(gdb) x /24x 0x6042f0
0x6042f0 <node1>:       0x000002f9      0x00000001      0x00604300      0x00000000
0x604300 <node2>:       0x000001e3      0x00000002      0x00604310      0x00000000
0x604310 <node3>:       0x000003aa      0x00000003      0x00604320      0x00000000
0x604320 <node4>:       0x00000170      0x00000004      0x00604330      0x00000000
0x604330 <node5>:       0x000001d5      0x00000005      0x00604340      0x00000000
0x604340 <node6>:       0x000000c5      0x00000006      0x00000000      0x00000000
```

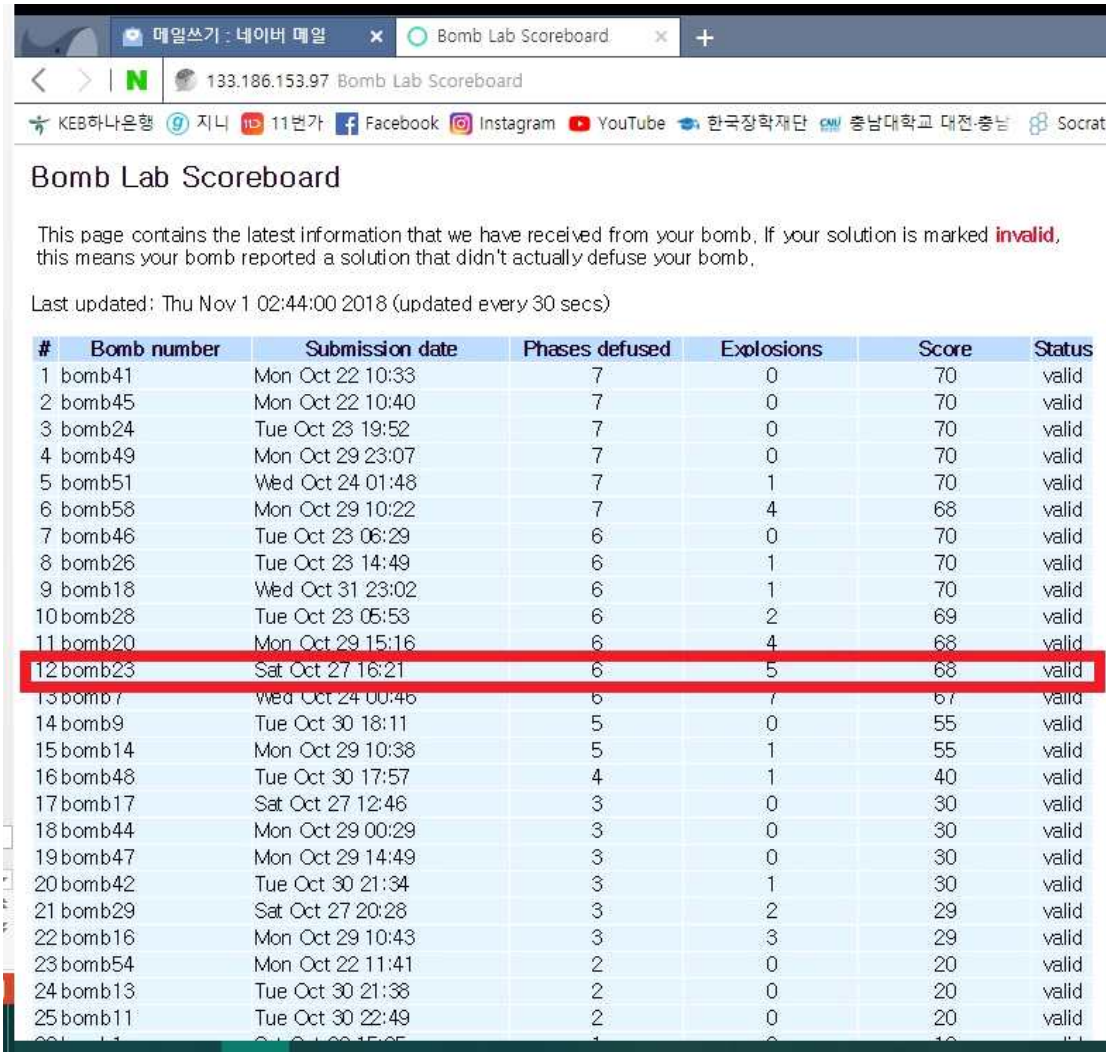
따라서 비교 값은 node 값이 였으며, 이를 통해 각 노드가 어떤 값을 갖고 있는지 확인해 보았다. node1 : (761) / node2 : (483) / node3 : (938) / node4 : (368) / node5 :(469) / node6 : (197) 또한 노드의 연결상태도 확인 할수 있었다.

<+216~218>줄을 통해 각 노드 값을 비교하는 조건이 (%rbx)의 주소값이 가리키는 값이

%eax값 보다 커야함을 알 수 있었다. 따라서 말을 정리해보면 노드 값을 내림차순으로 적으면 phase_6의 답이 되는 것이다. " 3 1 2 5 4 6 " 이 phase_6의 답이다.

Phase 6 [정답]

3 1 2 5 4 6



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: Thu Nov 1 02:44:00 2018 (updated every 30 secs)

#	Bomb number	Submission date	Phases defused	Explosions	Score	Status
1	bomb41	Mon Oct 22 10:33	7	0	70	valid
2	bomb45	Mon Oct 22 10:40	7	0	70	valid
3	bomb24	Tue Oct 23 19:52	7	0	70	valid
4	bomb49	Mon Oct 29 23:07	7	0	70	valid
5	bomb51	Wed Oct 24 01:48	7	1	70	valid
6	bomb58	Mon Oct 29 10:22	7	4	68	valid
7	bomb46	Tue Oct 23 06:29	6	0	70	valid
8	bomb26	Tue Oct 23 14:49	6	1	70	valid
9	bomb18	Wed Oct 31 23:02	6	1	70	valid
10	bomb28	Tue Oct 23 05:53	6	2	69	valid
11	bomb20	Mon Oct 29 15:16	6	4	68	valid
12	bomb23	Sat Oct 27 16:21	6	5	68	valid
13	bomb7	Wed Oct 24 00:46	6	7	67	valid
14	bomb9	Tue Oct 30 18:11	5	0	55	valid
15	bomb14	Mon Oct 29 10:38	5	1	55	valid
16	bomb48	Tue Oct 30 17:57	4	1	40	valid
17	bomb17	Sat Oct 27 12:46	3	0	30	valid
18	bomb44	Mon Oct 29 00:29	3	0	30	valid
19	bomb47	Mon Oct 29 14:49	3	0	30	valid
20	bomb42	Tue Oct 30 21:34	3	1	30	valid
21	bomb29	Sat Oct 27 20:28	3	2	29	valid
22	bomb16	Mon Oct 29 10:43	3	3	29	valid
23	bomb54	Mon Oct 22 11:41	2	0	20	valid
24	bomb13	Tue Oct 30 21:38	2	0	20	valid
25	bomb11	Tue Oct 30 22:49	2	0	20	valid