CSED211 Lab 3 Report

Bomb Lab: Defusing a Binary Bomb

20220041 김유진

I. Method

Bomb Server에 접속하여 실습에 사용할 bomb 파일을 다운로드하여 사용하였다. Port forwarding을 하여 교내 프로그래밍 계정으로 서버에 접속하였고, bomb67으로 이번 실습을 진행하였다. Bomb lab은 어셈블리 코드를 분석하여 그 안의 폭탄 코드를 찾아 해체시키는 컨셉의 과제이다. phase_1부터 시작하여 올바른 입력값을 주었을 때 다음 phase로 진입이 가능하다. 총 6개의 phase와 추가적으로 숨겨진 secret_phase가 존재하는데, 폭탄이 터지지 않게 올바른 답을 입력함으로써 폭탄을 해체할 수 있다. 어셈블리어에 대한 이해를 높이는 것에 의의를 둔 실습 과제이다.

II. Solution

시작하기에 앞서 다음과 같이 break point들을 설정해주었다. 1부터 6까지의 phase에 break point를 설정하여 각 phase에서 멈추어 유용하게 디버깅하도록 하였고, explode_bomb에 break point를 설정함으로써 bomb이 터지는 것을 방지하였다.

```
Reading symbols from /home/std/kimyujin1224/bomb67/bomb...done.
(gdb) b explode_bomb
Breakpoint 1 at 0x401614
(gdb) b phase_1
Breakpoint 2 at 0x400ef0
(gdb) b phase_2
Breakpoint 3 at 0x400f0c
(gdb) b phase_3
Breakpoint 4 at 0x400f53
(gdb) b phase_4
Breakpoint 5 at 0x4010c9
(gdb) b phase_5
Breakpoint 6 at 0x401120
(gdb) b phase_6
Breakpoint 7 at 0x40117f
```

먼저 터미널을 열어 gdb bomb을 입력하여 bomb에 대한 gdb를 실행하고, bomb 전반에 걸친 main 함수를 disas 명령어를 사용해 disassemble하면 다음과 같은 어셈블리어 코드를 확인할 수 있다.

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000000400dbd <+0>:     push   %rbx
   0x00000000000400dbe <+1>:     cmp    $0x1,%edi
   0x00000000000400dc1 <+4>:     jne    0x400dd3 <main+22>
   0x00000000000400dc3 <+6>:     mov    0x2039be(%rip),%rax        # 0x604788 <stdin@@GLIBC_2.2.5>
   0x00000000000400dca <+13>:    mov    %rax,0x2039cf(%rip)        # 0x6047a0 <infile>
   0x00000000000400dd1 <+20>:    jmp    0x400e2c <main+111>
   0x00000000000400dd3 <+22>:    mov    %rsi,%rbx
   0x00000000000400dd6 <+25>:    cmp    $0x2,%edi
   0x00000000000400dd9 <+28>:    jne    0x400e10 <main+83>
   0x00000000000400ddb <+30>:    mov    0x8(%rsi),%rdi
   0x00000000000400ddf <+34>:    mov    $0x4028f4,%esi
   0x00000000000400de4 <+39>:    callq  0x400c50 <fopen@plt>
   0x00000000000400de9 <+44>:    mov    %rax,0x2039b0(%rip)        # 0x6047a0 <infile>
   0x00000000000400df0 <+51>:    test   %rax,%rax
   0x00000000000400df3 <+54>:    jne    0x400e2c <main+111>
   0x00000000000400df5 <+56>:    mov    0x8(%rbx),%rdx
   0x00000000000400df9 <+60>:    mov    (%rbx),%rsi
   0x00000000000400dfc <+63>:    mov    $0x402410,%edi
   0x00000000000400e01 <+68>:    callq  0x400b60 <printf@plt>
   0x00000000000400e06 <+73>:    mov    $0x8,%edi
   0x00000000000400e0b <+78>:    callq  0x400c80 <exit@plt>
   0x00000000000400e10 <+83>:    mov    (%rsi),%rsi
   0x00000000000400e13 <+86>:    mov    $0x40242d,%edi
   0x00000000000400e18 <+91>:    mov    $0x0,%eax
   0x00000000000400e1d <+96>:    callq  0x400b60 <printf@plt>
   0x00000000000400e22 <+101>:   mov    $0x8,%edi
   0x00000000000400e27 <+106>:   callq  0x400c80 <exit@plt>
   0x00000000000400e2c <+111>:   callq  0x401417 <initialize_bomb>
   0x00000000000400e31 <+116>:   mov    $0x402498,%edi
   0x00000000000400e36 <+121>:   callq  0x400b40 <puts@plt>
   0x00000000000400e3b <+126>:   mov    $0x4024d8,%edi
   0x00000000000400e40 <+131>:   callq  0x400b40 <puts@plt>
   0x00000000000400e45 <+136>:   callq  0x40168c <read_line>
   0x00000000000400e4a <+141>:   mov    %rax,%rdi
   0x00000000000400e4d <+144>:   callq  0x400ef0 <phase_1>
   0x00000000000400e52 <+149>:   callq  0x4017b2 <phase_defused>
   0x00000000000400e57 <+154>:   mov    $0x402508,%edi
   0x00000000000400e5c <+159>:   callq  0x400b40 <puts@plt>
   0x00000000000400e61 <+164>:   callq  0x40168c <read_line>
   0x00000000000400e66 <+169>:   mov    %rax,%rdi
   0x00000000000400e69 <+172>:   callq  0x400f0c <phase_2>
   0x00000000000400e6e <+177>:   callq  0x4017b2 <phase_defused>
   0x00000000000400e73 <+182>:   mov    $0x402447,%edi
```

```
   0x00000000000400e78 <+187>:   callq  0x400b40 <puts@plt>
   0x00000000000400e7d <+192>:   callq  0x40168c <read_line>
   0x00000000000400e82 <+197>:   mov    %rax,%rdi
   0x00000000000400e85 <+200>:   callq  0x400f53 <phase_3>
   0x00000000000400e8a <+205>:   callq  0x4017b2 <phase_defused>
   0x00000000000400e8f <+210>:   mov    $0x402465,%edi
   0x00000000000400e94 <+215>:   callq  0x400b40 <puts@plt>
   0x00000000000400e99 <+220>:   callq  0x40168c <read_line>
   0x00000000000400e9e <+225>:   mov    %rax,%rdi
   0x00000000000400ea1 <+228>:   callq  0x4010c9 <phase_4>
   0x00000000000400ea6 <+233>:   callq  0x4017b2 <phase_defused>
   0x00000000000400eab <+238>:   mov    $0x402538,%edi
   0x00000000000400eb0 <+243>:   callq  0x400b40 <puts@plt>
   0x00000000000400eb5 <+248>:   callq  0x40168c <read_line>
   0x00000000000400eba <+253>:   mov    %rax,%rdi
   0x00000000000400ebd <+256>:   callq  0x401120 <phase_5>
   0x00000000000400ec2 <+261>:   callq  0x4017b2 <phase_defused>
   0x00000000000400ec7 <+266>:   mov    $0x402474,%edi
   0x00000000000400ecc <+271>:   callq  0x400b40 <puts@plt>
   0x00000000000400ed1 <+276>:   callq  0x40168c <read_line>
   0x00000000000400ed6 <+281>:   mov    %rax,%rdi
   0x00000000000400ed9 <+284>:   callq  0x40117f <phase_6>
   0x00000000000400ede <+289>:   callq  0x4017b2 <phase_defused>
   0x00000000000400ee3 <+294>:   mov    $0x0,%eax
   0x00000000000400ee8 <+299>:   pop    %rbx
   0x00000000000400ee9 <+300>:   retq
End of assembler dump.
```

각 phase에 대한 함수를 호출하고, phase가 defused되면 다음 phase로 넘어가도록 하고 있음을 알 수 있다. 명령 창에 run을 입력하여 본격적으로 bomb을 실행한다.

- phase_1

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

Breakpoint 2, 0x0000000000400ef0 in phase_1 ()
(gdb) disas phase_1
Dump of assembler code for function phase_1:
=> 0x0000000000400ef0 <+0>:     sub    $0x8,%rsp
   0x0000000000400ef4 <+4>:     mov    $0x402560,%esi
   0x0000000000400ef9 <+9>:     callq  0x4013ae <strings_not_equal>
   0x0000000000400efe <+14>:    test   %eax,%eax
   0x0000000000400f00 <+16>:    je     0x400f07 <phase_1+23>
   0x0000000000400f02 <+18>:    callq  0x401614 <explode_bomb>
   0x0000000000400f07 <+23>:    add    $0x8,%rsp
   0x0000000000400f0b <+27>:    retq
End of assembler dump.
```

bomb을 실행했을 때 처음 마주하게 되는 phase이다. 먼저 아무 키워드나 입력하는데 이때 breakpoint를 걸어두었기 때문에 bomb이 터지지 않는다. disas 키워드를 통해 phase_1을 disassemble해주면 다음과 같은 코드가 나오는 것을 확인할 수 있다. 첫번째 줄은 스택 포인터로 공간을 할당해주는 것이고 두번째 줄은 %esi 레지스터에 $0x402560 값을 넣어주고 있다.

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

strings_not_equal 함수를 disassemble해서 살펴보면 이름에서 추측할 수 있듯이 두 string이 같은지를 비교하고 같다면 0, 다르다면 1을 반환하는 함수임을 알 수 있다. 따라서 phase_1에서는 입력하는 문자열이 일치해야만 bomb이 터지지 않고 다음 phase로 넘어간다. strings_not_equal 함수에게 전해지는 첫번째 인자에 뭐가 들어가는지를 확인하기 위해서 <+4> 줄을 확인한다.

```
(gdb) x/s 0x402560
0x402560:       "Border relations with Canada have never been better."
```

x/s 명령어를 사용하여 0x402560에 어떤 문자열이 들어있는지를 확인할 수 있고, 이것이 phase_1의 답임을 알 수 있다.

```
Starting program: /home/std/kimyujin1224/bomb67/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.

Breakpoint 2, 0x0000000000400ef0 in phase_1 ()
(gdb) continue
Continuing.
Phase 1 defused. How about the next one?
```

다시 실행시켰을 때 앞서 구한 답을 입력하면 phase_1이 해결되어 다음 phase로 넘어간다.

phase_1 답: Border relations with Canada have never been better.

- phase_2

마찬가지로 답을 입력하는 부분에 아무 값을 입력하여 breakpoint에 걸리게 만든 후, phase_2를 disassemble한다.

```
Breakpoint 3, 0x0000000000400f0c in phase_2 ()
(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  0x40164a <read_six_numbers>
   0x0000000000400f1a <+14>:    cmpl   $0x1,(%rsp)
   0x0000000000400f1e <+18>:    je     0x400f40 <phase_2+52>
   0x0000000000400f20 <+20>:    callq  0x401614 <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  0x401614 <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.
```

6개의 숫자를 입력받아 이를 stack에 저장하고 있다. <+14>과 <+18>에서 (%rsp)가 0x1이 같은지 아닌지를 비교하여 같다면 <+52>로 이동하고, 같지 않다면 explode_bomb을 호출하고 있음을 알 수 있다. 따라서 첫번째 값은 1이다.

```
0x0000000000400f12 <+6>:     mov    %rsp,%rsi
0x0000000000400f15 <+9>:     callq  0x40164a <read_six_numbers>
0x0000000000400f1a <+14>:    cmpl   $0x1,(%rsp)
0x0000000000400f1e <+18>:    je     0x400f40 <phase_2+52>
0x0000000000400f20 <+20>:    callq  0x401614 <explode_bomb>
```

<+52>부터 한줄씩 살펴보면 %rbx에 %rsp+0x4, %rbp에 %rsp+0x18을 넣어준다. 그리고 다시 위에 있는 <+27> 줄로 돌아간다.

```
0x0000000000400f40 <+52>:    lea    0x4(%rsp),%rbx
0x0000000000400f45 <+57>:    lea    0x18(%rsp),%rbp
0x0000000000400f4a <+62>:    jmp    0x400f27 <phase_2+27>
```

<+27>을 살펴보면 %eax에 %rbx-0x4를 해주고, 다음 줄에서 %eax에 %eax를 다시 더해준다. (%rbx)와 %eax를 비교하여 같은 경우에만 explode_bomb 호출하는 줄을 건너뛰고 <+41>로 뛴다. 앞서 %rbx에 먼저 %rsp+0x4를 넣어주었기 때문에 <+27>에서 %eax에는 %rbx-0x4, 즉 첫번째 값인 1이 들어가는 것을 알 수 있고 그에 따라 add 인스트럭션을 거치고 나서는 2가 저장되는 것을 알 수 있다. 따라서 1 다음으로 입력되는 값은 2이다.

```
   0x0000000000400f35 <+41>:    add    $0x4,%rbx
   0x0000000000400f39 <+45>:    cmp    %rbp,%rbx
   0x0000000000400f3c <+48>:    jne    0x400f27 <phase_2+27>
   0x0000000000400f3e <+50>:    jmp    0x400f4c <phase_2+64>
```

그리고 나서는 %rbx에 0x4를 더해주고 %rbx가 %rbp보다 작거나 같으면 다시 <+27>로 돌아가게 하는 것으로 보아 이는 반복문을 실행하는 코드임을 알 수 있다. rbp와 같아질 때까지 반복하는데, 이는 결국 첫번째 값이 1이고, i번째 값의 2배가 i+1번째의 값이라는 것을 가리킨다. 따라서 답은 1 2 4 8 16 32이다.

```
Starting program: /home/std/kimyujin1224/bomb67/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.

Breakpoint 2, 0x0000000000400ef0 in phase_1 ()
(gdb) continue
Continuing.
Phase 1 defused. How about the next one?
1 2 4 8 16 32

Breakpoint 3, 0x0000000000400f0c in phase_2 ()
(gdb) continue
Continuing.
That's number 2.  Keep going!
```

phase_2 답: 1 2 4 8 16 32

- phase_3

phase_3을 disassemble하면 다음과 같은 긴 어셈블리어코드를 확인할 수 있다.

```
Breakpoint 4, 0x0000000000400f53 in phase_3 ()
(gdb) disas phase_3
Dump of assembler code for function phase_3:
=> 0x0000000000400f53 <+0>:     sub    $0x18,%rsp
   0x0000000000400f57 <+4>:     lea    0x8(%rsp),%r8
   0x0000000000400f5c <+9>:     lea    0x7(%rsp),%rcx
   0x0000000000400f61 <+14>:    lea    0xc(%rsp),%rdx
   0x0000000000400f66 <+19>:    mov    $0x4025be,%esi
   0x0000000000400f6b <+24>:    mov    $0x0,%eax
   0x0000000000400f70 <+29>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x0000000000400f75 <+34>:    cmp    $0x2,%eax
   0x0000000000400f78 <+37>:    jg     0x400f7f <phase_3+44>
   0x0000000000400f7a <+39>:    callq  0x401614 <explode_bomb>
   0x0000000000400f7f <+44>:    cmpl   $0x7,0xc(%rsp)
   0x0000000000400f84 <+49>:    ja     0x40107c <phase_3+297>
   0x0000000000400f8a <+55>:    mov    0xc(%rsp),%eax
   0x0000000000400f8e <+59>:    jmpq   *0x4025d0(,%rax,8)
   0x0000000000400f95 <+66>:    mov    $0x6c,%eax
   0x0000000000400f9a <+71>:    cmpl   $0x36b,0x8(%rsp)
   0x0000000000400fa2 <+79>:    je     0x401086 <phase_3+307>
   0x0000000000400fa8 <+85>:    callq  0x401614 <explode_bomb>
   0x0000000000400fad <+90>:    mov    $0x6c,%eax
   0x0000000000400fb2 <+95>:    jmpq   0x401086 <phase_3+307>
   0x0000000000400fb7 <+100>:   mov    $0x77,%eax
   0x0000000000400fbc <+105>:   cmpl   $0x286,0x8(%rsp)
   0x0000000000400fc4 <+113>:   je     0x401086 <phase_3+307>
   0x0000000000400fca <+119>:   callq  0x401614 <explode_bomb>
   0x0000000000400fcf <+124>:   mov    $0x77,%eax
   0x0000000000400fd4 <+129>:   jmpq   0x401086 <phase_3+307>
   0x0000000000400fd9 <+134>:   mov    $0x77,%eax
   0x0000000000400fde <+139>:   cmpl   $0x279,0x8(%rsp)
---Type <return> to continue, or q <return> to quit---continue
```

```
   0x0000000000400fe6 <+147>:   je     0x401086 <phase_3+307>
   0x0000000000400fec <+153>:   callq  0x401614 <explode_bomb>
   0x0000000000400ff1 <+158>:   mov    $0x77,%eax
   0x0000000000400ff6 <+163>:   jmpq   0x401086 <phase_3+307>
   0x0000000000400ffb <+168>:   mov    $0x64,%eax
   0x0000000000401000 <+173>:   cmpl   $0x2b9,0x8(%rsp)
   0x0000000000401008 <+181>:   je     0x401086 <phase_3+307>
   0x000000000040100a <+183>:   callq  0x401614 <explode_bomb>
   0x000000000040100f <+188>:   mov    $0x64,%eax
   0x0000000000401014 <+193>:   jmp    0x401086 <phase_3+307>
   0x0000000000401016 <+195>:   mov    $0x72,%eax
   0x000000000040101b <+200>:   cmpl   $0xb9,0x8(%rsp)
   0x0000000000401023 <+208>:   je     0x401086 <phase_3+307>
   0x0000000000401025 <+210>:   callq  0x401614 <explode_bomb>
   0x000000000040102a <+215>:   mov    $0x72,%eax
   0x000000000040102f <+220>:   jmp    0x401086 <phase_3+307>
   0x0000000000401031 <+222>:   mov    $0x70,%eax
   0x0000000000401036 <+227>:   cmpl   $0x78,0x8(%rsp)
   0x000000000040103b <+232>:   je     0x401086 <phase_3+307>
   0x000000000040103d <+234>:   callq  0x401614 <explode_bomb>
   0x0000000000401042 <+239>:   mov    $0x70,%eax
   0x0000000000401047 <+244>:   jmp    0x401086 <phase_3+307>
   0x0000000000401049 <+246>:   mov    $0x71,%eax
   0x000000000040104e <+251>:   cmpl   $0x39d,0x8(%rsp)
   0x0000000000401056 <+259>:   je     0x401086 <phase_3+307>
   0x0000000000401058 <+261>:   callq  0x401614 <explode_bomb>
   0x000000000040105d <+266>:   mov    $0x71,%eax
   0x0000000000401062 <+271>:   jmp    0x401086 <phase_3+307>
   0x0000000000401064 <+273>:   mov    $0x6a,%eax
---Type <return> to continue, or q <return> to quit---continue
   0x0000000000401069 <+278>:   cmpl   $0x50,0x8(%rsp)
   0x000000000040106e <+283>:   je     0x401086 <phase_3+307>
   0x0000000000401070 <+285>:   callq  0x401614 <explode_bomb>
   0x0000000000401075 <+290>:   mov    $0x6a,%eax
   0x000000000040107a <+295>:   jmp    0x401086 <phase_3+307>
   0x000000000040107c <+297>:   callq  0x401614 <explode_bomb>
   0x0000000000401081 <+302>:   mov    $0x74,%eax
   0x0000000000401086 <+307>:   cmp    0x7(%rsp),%al
   0x000000000040108a <+311>:   je     0x401091 <phase_3+318>
   0x000000000040108c <+313>:   callq  0x401614 <explode_bomb>
   0x0000000000401091 <+318>:   add    $0x18,%rsp
   0x0000000000401095 <+322>:   retq
End of assembler dump.
```

<+29>에서 호출하는 함수의 이름에 scanf가 있는 것으로 보아 입력을 받고 있는 함수라는 것을 알 수 있다. 매개변수 esi에 어떤 것이 들어가있는지를 확인해주고자 x/s 명령어를 사용한다.

```
(gdb) x/s 0x4025be
0x4025be:       "%d %c %d"
```

이를 통해 입력받고자 하는 것은 정수, 문자, 정수 이렇게 세가지라는 것을 알 수 있다.

```
   0x0000000000400f7f <+44>:    cmpl   $0x7,0xc(%rsp)
   0x0000000000400f84 <+49>:    ja     0x40107c <phase_3+297>
   0x0000000000400f8a <+55>:    mov    0xc(%rsp),%eax
   0x0000000000400f8e <+59>:    jmpq   *0x4025d0(,%rax,8)
```

<+44>에서 %rsp+0xc와 0x7 비교했을 때 전자의 값이 더 크면 explode_bomb을 호출하므로 %rsp+0xc는 0x7(7)보다 작거나 같음, 즉 첫번째 입력값은 7 이하라는 것을 알 수 있다. 그다음에는 첫번째 입력값인 %rsp+0xc를 %eax에 넣어두고 0x4025d0+8*%rax로 jump하고 있다.

그 이후 코드를 보면 비슷한 구조가 되풀이되고 있는 것을 알 수 있는데, 이들은 첫번째 입력값에 따라 jump하여 도착하는 곳으로 이 phase에 대한 여러 개의 답이 존재할 수 있을 짐작할 수 있게 한다. 66-95, 100-129, 134-163, 168-193, 195-220, 222-244, 246-271, 273-295 줄들을 확인해보면 동일한 구조임을 알 수 있다. 그 중 한가지 예시를 통해 어셈블리어 코드를 분석한다.

```
   0x0000000000400f95 <+66>:    mov    $0x6c,%eax
   0x0000000000400f9a <+71>:    cmpl   $0x36b,0x8(%rsp)
   0x0000000000400fa2 <+79>:    je     0x401086 <phase_3+307>
   0x0000000000400fa8 <+85>:    callq  0x401614 <explode_bomb>
   0x0000000000400fad <+90>:    mov    $0x6c,%eax
   0x0000000000400fb2 <+95>:    jmpq   0x401086 <phase_3+307>
```

이 부분은 %rax, 즉 첫번째 값이 0일 때 jump되어 오게 된다. %eax에 어떤 상수를 넣어주고 %rsp+0x8의 값, 즉 세번째 입력값을 0x36b와 비교하여서 두 값이 일치해야지만 explode_bomb을 호출하지 않고 정상적으로 진행되어 <+307>로 이동하도록 코드가 작성되어있다. 따라서 이 경우에 첫번째 입력값은 0, 세번째 입력값은 875(0x36b)이다.

```
   0x0000000000401086 <+307>:   cmp    0x7(%rsp),%al
   0x000000000040108a <+311>:   je     0x401091 <phase_3+318>
   0x000000000040108c <+313>:   callq  0x401614 <explode_bomb>
   0x0000000000401091 <+318>:   add    $0x18,%rsp
   0x0000000000401095 <+322>:   retq
End of assembler dump.
```

jump하여 오게 되는 <+307> 줄을 보면 %al을 %rsp+0x7, 즉 두번째 입력값과 비교하고 있다. 이것이 일치해야만 explode_bomb을 건너뛰는데, %al은 %eax의 하위 1바이트와 같다. 따라서 첫번째 값이 0인 위의 경우, 두번째 입력값은 1바이트 0x6c에 해당하는 ASCII문자, 'l'이다.

따라서 답이 될 수 있는 한 가지 예로 0 l 875가 있다.

```
Starting program: /home/std/kimyujin1224/bomb67/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.

Breakpoint 2, 0x0000000000400ef0 in phase_1 ()
(gdb) continue
Continuing.
Phase 1 defused. How about the next one?
1 2 4 8 16 32

Breakpoint 3, 0x0000000000400f0c in phase_2 ()
(gdb) continue
Continuing.
That's number 2.  Keep going!
0 l 875

Breakpoint 4, 0x0000000000400f53 in phase_3 ()
(gdb) continue
Continuing.
Halfway there!
```

0 l 875 입력했더니 phase를 통과한 것을 확인할 수 있다.

```
0x0000000000400fb7 <+100>:   mov    $0x77,%eax
0x0000000000400fbc <+105>:   cmpl   $0x286,0x8(%rsp)
0x0000000000400fc4 <+113>:   je     0x401086 <phase_3+307>
0x0000000000400fca <+119>:   callq  0x401614 <explode_bomb>
0x0000000000400fcf <+124>:   mov    $0x77,%eax
0x0000000000400fd4 <+129>:   jmpq   0x401086 <phase_3+307>
```

아스키코드(0x77): w, 0x286(646)

이 경우, 세 입력값은 각각 1 w 646으로, 입력값으로 넣어주면 마찬가지로 defuse에 성공한다.

```
That's number 2.  Keep going!
1 w 646

Breakpoint 2, 0x0000000000400f53 in phase_3 ()
Missing separate debuginfos, use: debuginfo-install glibc-2.17-326.el7_9.x86_64
(gdb) continue
Continuing.
Halfway there!
```

phase_3 답 : 0 l 875, 1 w 646 등

- phase_4

phase_4를 disassemble한 모습은 다음과 같다.

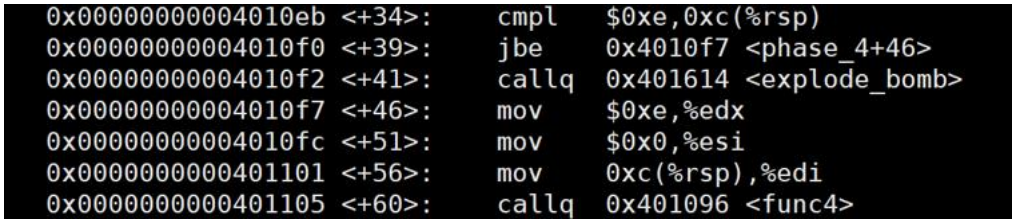
```
Breakpoint 5, 0x00000000004010c9 in phase_4 ()
(gdb) disas phase_4
Dump of assembler code for function phase_4:
=> 0x00000000004010c9 <+0>:     sub    $0x18,%rsp
   0x00000000004010cd <+4>:     lea    0x8(%rsp),%rcx
   0x00000000004010d2 <+9>:     lea    0xc(%rsp),%rdx
   0x00000000004010d7 <+14>:    mov    $0x40286d,%esi
   0x00000000004010dc <+19>:    mov    $0x0,%eax
   0x00000000004010e1 <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x00000000004010e6 <+29>:    cmp    $0x2,%eax
   0x00000000004010e9 <+32>:    jne    0x4010f2 <phase_4+41>
   0x00000000004010eb <+34>:    cmpl   $0xe,0xc(%rsp)
   0x00000000004010f0 <+39>:    jbe    0x4010f7 <phase_4+46>
   0x00000000004010f2 <+41>:    callq  0x401614 <explode_bomb>
   0x00000000004010f7 <+46>:    mov    $0xe,%edx
   0x00000000004010fc <+51>:    mov    $0x0,%esi
   0x0000000000401101 <+56>:    mov    0xc(%rsp),%edi
   0x0000000000401105 <+60>:    callq  0x401096 <func4>
   0x000000000040110a <+65>:    cmp    $0x13,%eax
   0x000000000040110d <+68>:    jne    0x401116 <phase_4+77>
   0x000000000040110f <+70>:    cmpl   $0x13,0x8(%rsp)
   0x0000000000401114 <+75>:    je     0x40111b <phase_4+82>
   0x0000000000401116 <+77>:    callq  0x401614 <explode_bomb>
   0x000000000040111b <+82>:    add    $0x18,%rsp
   0x000000000040111f <+86>:    retq
End of assembler dump.
```

```
   0x00000000004010cd <+4>:     lea    0x8(%rsp),%rcx
   0x00000000004010d2 <+9>:     lea    0xc(%rsp),%rdx
   0x00000000004010d7 <+14>:    mov    $0x40286d,%esi
   0x00000000004010dc <+19>:    mov    $0x0,%eax
   0x00000000004010e1 <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
```

%rsp+0x8을 %rxc에 넣어주고 %rsp+0xc를 %rdx에 넣어준다. phase_4에서도 scanf와 비슷한 이름의 함수를 호출하고 있으니 입력을 받는 함수임을 추측할 수 있다. 앞서 했던 것처럼 어떤 값을 입력받아야하는지를 x/s 명령어를 사용하여 조사한다.

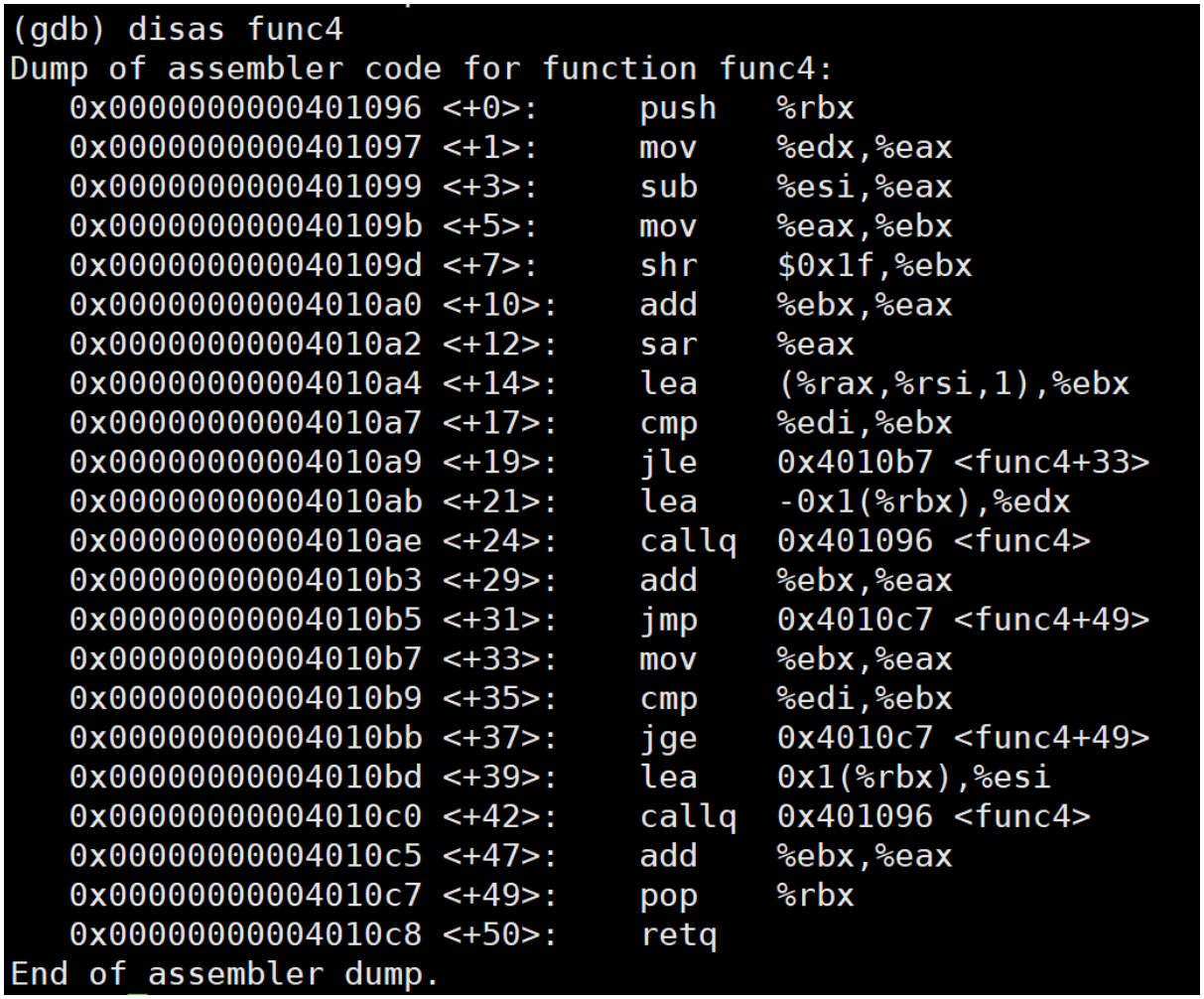
```
(gdb) x/s 0x40286d
0x40286d:       "%d %d"
```

두 정수를 입력받고자 하는 것을 알 수 있다.



%rsp+0xc를 0xe와 비교했을 때의 조건부 jump 인스트럭션에 따라 %rsp+0xc가 14보다 작은 수라는 것을 알 수 있다. 그 다음 %edx에 0xe를 넣고, %esi에 0x0를 넣고, %edi에 %rsp+0xc를 넣고, 이들을 매개변수로 하여 func4를 호출한다.

내부에서 콜 인스트럭션에 의해 호출되는 함수 func4를 disassemble한 모습은 다음과 같다. 앞서 언급했듯이 %edx의 값으로 0xe, %esi에는 0x0, %edi에는 %rsp+0xc, 즉 첫번째 입력값이 들어간다.



%rbx를 push, %edx를 %eax에 넣은 다음 %esi를 빼주고, %eax를 %ebx에 넣는다. <+7>을 통해 %ebx를 31 bit만큼 logical right shift한 값을 %ebx에 넣어주는데, 이는 결국 %ebx의 sign bit를 가리킨다. %eax에 %ebx를 더해주고, %eax를 1만큼 arithmetic right shift를 해준다. %ebx에 %rax+%rsi를 넣어준 다음, %ebx와 %edi를 비교하여 만약 %ebx가 %edi보다 작거나 같으면 <+33>으로 점프한다. 반대로 큰 경우에는 %rbx에 0x1을 뺀 값을 %edx에 다시 넣어주고 다시

재귀적으로 func4를 호출한다. 여기서 값을 얻었을 때 %ebx+%eax를 %eax 해주고 <+49>로 jump하여 리턴된다. 만약 %ebx가 %edi보다 작거나 같으면 <+33>으로 점프해서 %ebx를 %eax에 넣어주고 %edi와 %ebx를 비교하여 비슷한 작업을 해주는데 이를 통해 func4는 재귀함수의 구조라는 것을 알 수 있다. 이를 간단히 C언어 함수의 형태로 나타내면 다음과 같다.

```
int func4(edx, esi, edi) {
        ...
        if (ebx<edi) {
                rbx+1 -> new_esi
                return func4(edx, new_esi, edi) + ebx
        }
        else if (ebx>edi) {
                rbx-1 -> new_edx
                return func4(new_edx, esi, edi) + ebx
        }
        else {
                return eax
        }
}
```

```
0x0000000000401105 <+60>:    callq  0x401096 <func4>
0x000000000040110a <+65>:    cmp    $0x13,%eax
0x000000000040110d <+68>:    jne    0x401116 <phase_4+77>
0x000000000040110f <+70>:    cmpl   $0x13,0x8(%rsp)
0x0000000000401114 <+75>:    je     0x40111b <phase_4+82>
0x0000000000401116 <+77>:    callq  0x401614 <explode_bomb>
0x000000000040111b <+82>:    add    $0x18,%rsp
0x000000000040111f <+86>:    retq
```

다시 phase_4 함수를 관찰하면, func4의 결과값이 0x13(19)여야 하고, 아니면 explode_bomb이 호출됨을 알 수 있고 또한 0x8+%rsp의 값 역시 0x13(19)여야 함을 알 수 있다. 어떤 첫번째 입력값에 의해 func4를 호출했을 때의 결과가 19인지를 계산해보면 4라는 것을 알 수 있다. 따라서 첫번째 입력값은 4가 되어야 한다.

phase_4 답: 4 19

- phase_5

phase_5를 disassemble한 모습이다.

```
Breakpoint 3, 0x0000000000401120 in phase_5 ()
(gdb) disas phase_5
Dump of assembler code for function phase_5:
=> 0x0000000000401120 <+0>:     push   %rbx
   0x0000000000401121 <+1>:     sub    $0x10,%rsp
   0x0000000000401125 <+5>:     mov    %rdi,%rbx
   0x0000000000401128 <+8>:     callq  0x401391 <string_length>
   0x000000000040112d <+13>:    cmp    $0x6,%eax
   0x0000000000401130 <+16>:    je     0x401172 <phase_5+82>
   0x0000000000401132 <+18>:    callq  0x401614 <explode_bomb>
   0x0000000000401137 <+23>:    jmp    0x401172 <phase_5+82>
   0x0000000000401139 <+25>:    movzbl (%rbx,%rax,1),%edx
   0x000000000040113d <+29>:    and    $0xf,%edx
   0x0000000000401140 <+32>:    movzbl 0x402610(%rdx),%edx
   0x0000000000401147 <+39>:    mov    %dl,(%rsp,%rax,1)
   0x000000000040114a <+42>:    add    $0x1,%rax
   0x000000000040114e <+46>:    cmp    $0x6,%rax
   0x0000000000401152 <+50>:    jne    0x401139 <phase_5+25>
   0x0000000000401154 <+52>:    movb   $0x0,0x6(%rsp)
   0x0000000000401159 <+57>:    mov    $0x4025c7,%esi
   0x000000000040115e <+62>:    mov    %rsp,%rdi
   0x0000000000401161 <+65>:    callq  0x4013ae <strings_not_equal>
   0x0000000000401166 <+70>:    test   %eax,%eax
   0x0000000000401168 <+72>:    je     0x401179 <phase_5+89>
   0x000000000040116a <+74>:    callq  0x401614 <explode_bomb>
   0x000000000040116f <+79>:    nop
   0x0000000000401170 <+80>:    jmp    0x401179 <phase_5+89>
   0x0000000000401172 <+82>:    mov    $0x0,%eax
   0x0000000000401177 <+87>:    jmp    0x401139 <phase_5+25>
   0x0000000000401179 <+89>:    add    $0x10,%rsp
   0x000000000040117d <+93>:    pop    %rbx
   0x000000000040117e <+94>:    retq
End of assembler dump.
```

가장 먼저 호출되는 함수인 string_length 함수를 disassemble하면 다음과 같다.

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

%rdi가 0이면 0을 return하고 %rdi가 0이 아니면 %rdx가 %rdi

어셈블리어 코드를 통해 이는 반복문을 돌면서 string의 길이를 얻는 함수임을 파악할 수 있다.

다시 phase_5를 관찰한다.

```
0x0000000000401128 <+8>:     callq  0x401391 <string_length>
0x000000000040112d <+13>:    cmp    $0x6,%eax
0x0000000000401130 <+16>:    je     0x401172 <phase_5+82>
0x0000000000401132 <+18>:    callq  0x401614 <explode_bomb>
0x0000000000401137 <+23>:    jmp    0x401172 <phase_5+82>
```

string_length 함수의 리턴 값이 %rax에 들어있을 것이고 %rax의 하위 4byte인 %eax와 0x6을 비교했을 때 같지 않으면 explode_bomb이 호출된다. 따라서 string의 길이는 6임을 알 수 있다. %eax와 0x6이 같으면 <+82>로 점프한다.

```
0x0000000000401172 <+82>:    mov    $0x0,%eax
0x0000000000401177 <+87>:    jmp    0x401139 <phase_5+25>
0x0000000000401179 <+89>:    add    $0x10,%rsp
```

%eax에는 0x0를 넣어주고 다시 <+25>로 향한다.

```
0x0000000000401139 <+25>:    movzbl (%rbx,%rax,1),%edx
0x000000000040113d <+29>:    and    $0xf,%edx
0x0000000000401140 <+32>:    movzbl 0x402610(%rdx),%edx
0x0000000000401147 <+39>:    mov    %dl,(%rsp,%rax,1)
0x000000000040114a <+42>:    add    $0x1,%rax
0x000000000040114e <+46>:    cmp    $0x6,%rax
0x0000000000401152 <+50>:    jne    0x401139 <phase_5+25>
```

```
(gdb) x/s 0x402610
0x402610 <array.3162>:  "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
```

<+25>를 보면 %edx에 %rbx+%rax를 넣어주고 %edx와 0xf(1111)을 AND 연산해준다. 즉 edx의 하위 4bit만 추출한다는 뜻으로 0~15까지의 값을 갖게 된다. 0x402610에 어떤 string이 있는지를 확인하면 위와 같은 문자열이 들어있음을 확인할 수 있는데, 인스트럭션의 내용에 의하면 %rdx+0x402610을 %edx에 넣어주는데 이는 0x402610+rdx 위치에 있는 값을 edx에 저장하라는 뜻이다.

0	1	2	3	4	5	6	7	8	9	10A	11B	12C	13D	14E	15F
m	a	d	u	i	e	r	s	n	f	o	t	v	b	y	l

%dl은 %rsp+%rax에 넣어주는데, 그 밑에 <+42>~<+50>을 보면 반복문의 형태를 띠고 있음을 알 수 있다. 즉, 이 부분은 입력값을 배율로 하여 0x402610의 문자열에서 6개의 값을 추출하여 새로운 배열에 저장을 한다는 것을 알 수 있다.

```
0x0000000000401154 <+52>:    movb   $0x0,0x6(%rsp)
0x0000000000401159 <+57>:    mov    $0x4025c7,%esi
0x000000000040115e <+62>:    mov    %rsp,%rdi
0x0000000000401161 <+65>:    callq  0x4013ae <strings_not_equal>
0x0000000000401166 <+70>:    test   %eax,%eax
0x0000000000401168 <+72>:    je     0x401179 <phase_5+89>
0x000000000040116a <+74>:    callq  0x401614 <explode_bomb>
0x000000000040116f <+79>:    nop
0x0000000000401170 <+80>:    jmp    0x401179 <phase_5+89>
```

글자를 추출하고 나서는 <+57>에서 %esi에 어떤 string이 담겨있는지를 확인한다. 마찬가지로 x/s 명령어를 사용해 0x4025c7에 담겨있는 string을 확인한다.

```
(gdb) x/s 0x4025c7
0x4025c7:       "oilers"
```

그러면 다음과 같이 "oilers"라는 string이 담겨있음을 확인할 수 있다.

0	1	2	3	**4**	**5**	**6**	**7**	8	9	**10A**	11B	12C	13D	14E	**15F**
m	a	d	u	**i**	**e**	**r**	**s**	n	f	**o**	t	v	b	y	**l**

oilers를 추출하기 위해서는 입력한 문자의 하위 4bit가 <a 4 f 5 6 7>인 ascii코드 조합을 입력해야 한다. 이 조건에 부합하는 문자들의 조합으로는 jdoefg, zt?u&' 등이 있다.

```
So you got that one.  Try this one.
jdoefg

Breakpoint 2, 0x0000000000401120 in phase_5 ()
(gdb) continue
Continuing.
Good work!  On to the next...
```

phase_6 답: (예) jdoefg, zt?u&', 등

- phase_6

disassemble phase_6을 하면 다음과 같다.

```
(gdb) disas phase_6
Dump of assembler code for function phase_6:
=> 0x000000000040117f <+0>:     push   %r14
   0x0000000000401181 <+2>:     push   %r13
   0x0000000000401183 <+4>:     push   %r12
   0x0000000000401185 <+6>:     push   %rbp
   0x0000000000401186 <+7>:     push   %rbx
   0x0000000000401187 <+8>:     sub    $0x50,%rsp
   0x000000000040118b <+12>:    lea    0x30(%rsp),%r13
   0x0000000000401190 <+17>:    mov    %r13,%rsi
   0x0000000000401193 <+20>:    callq  0x40164a <read_six_numbers>
   0x0000000000401198 <+25>:    mov    %r13,%r14
   0x000000000040119b <+28>:    mov    $0x0,%r12d
   0x00000000004011a1 <+34>:    mov    %r13,%rbp
   0x00000000004011a4 <+37>:    mov    0x0(%r13),%eax
   0x00000000004011a8 <+41>:    sub    $0x1,%eax
   0x00000000004011ab <+44>:    cmp    $0x5,%eax
   0x00000000004011ae <+47>:    jbe    0x4011b5 <phase_6+54>
   0x00000000004011b0 <+49>:    callq  0x401614 <explode_bomb>
   0x00000000004011b5 <+54>:    add    $0x1,%r12d
   0x00000000004011b9 <+58>:    cmp    $0x6,%r12d
   0x00000000004011bd <+62>:    je     0x4011e1 <phase_6+98>
   0x00000000004011bf <+64>:    mov    %r12d,%ebx
   0x00000000004011c2 <+67>:    movslq %ebx,%rax
   0x00000000004011c5 <+70>:    mov    0x30(%rsp,%rax,4),%eax
   0x00000000004011c9 <+74>:    cmp    %eax,0x0(%rbp)
   0x00000000004011cc <+77>:    jne    0x4011d3 <phase_6+84>
   0x00000000004011ce <+79>:    callq  0x401614 <explode_bomb>
   0x00000000004011d3 <+84>:    add    $0x1,%ebx
   0x00000000004011d6 <+87>:    cmp    $0x5,%ebx
   0x00000000004011d9 <+90>:    jle    0x4011c2 <phase_6+67>
   0x00000000004011db <+92>:    add    $0x4,%r13
   0x00000000004011df <+96>:    jmp    0x4011a1 <phase_6+34>
   0x00000000004011e1 <+98>:    lea    0x48(%rsp),%rsi
   0x00000000004011e6 <+103>:   mov    %r14,%rax
---Type <return> to continue, or q <return> to quit---continue
```

```
   0x00000000004011e9 <+106>:   mov    $0x7,%ecx
   0x00000000004011ee <+111>:   mov    %ecx,%edx
   0x00000000004011f0 <+113>:   sub    (%rax),%edx
   0x00000000004011f2 <+115>:   mov    %edx,(%rax)
   0x00000000004011f4 <+117>:   add    $0x4,%rax
   0x00000000004011f8 <+121>:   cmp    %rsi,%rax
   0x00000000004011fb <+124>:   jne    0x4011ee <phase_6+111>
   0x00000000004011fd <+126>:   mov    $0x0,%esi
   0x0000000000401202 <+131>:   jmp    0x401224 <phase_6+165>
   0x0000000000401204 <+133>:   mov    0x8(%rdx),%rdx
   0x0000000000401208 <+137>:   add    $0x1,%eax
   0x000000000040120b <+140>:   cmp    %ecx,%eax
   0x000000000040120d <+142>:   jne    0x401204 <phase_6+133>
   0x000000000040120f <+144>:   jmp    0x401216 <phase_6+151>
   0x0000000000401211 <+146>:   mov    $0x6042f0,%edx
   0x0000000000401216 <+151>:   mov    %rdx,(%rsp,%rsi,2)
   0x000000000040121a <+155>:   add    $0x4,%rsi
   0x000000000040121e <+159>:   cmp    $0x18,%rsi
   0x0000000000401222 <+163>:   je     0x401239 <phase_6+186>
   0x0000000000401224 <+165>:   mov    0x30(%rsp,%rsi,1),%ecx
   0x0000000000401228 <+169>:   cmp    $0x1,%ecx
   0x000000000040122b <+172>:   jle    0x401211 <phase_6+146>
   0x000000000040122d <+174>:   mov    $0x1,%eax
   0x0000000000401232 <+179>:   mov    $0x6042f0,%edx
   0x0000000000401237 <+184>:   jmp    0x401204 <phase_6+133>
   0x0000000000401239 <+186>:   mov    (%rsp),%rbx
   0x000000000040123d <+190>:   lea    0x8(%rsp),%rax
   0x0000000000401242 <+195>:   lea    0x30(%rsp),%rsi
   0x0000000000401247 <+200>:   mov    %rbx,%rcx
   0x000000000040124a <+203>:   mov    (%rax),%rdx
   0x000000000040124d <+206>:   mov    %rdx,0x8(%rcx)
   0x0000000000401251 <+210>:   add    $0x8,%rax
   0x0000000000401255 <+214>:   cmp    %rsi,%rax
   0x0000000000401258 <+217>:   je     0x40125f <phase_6+224>
---Type <return> to continue, or q <return> to quit---continue
```

```
   0x000000000040125a <+219>:   mov    %rdx,%rcx
   0x000000000040125d <+222>:   jmp    0x40124a <phase_6+203>
   0x000000000040125f <+224>:   movq   $0x0,0x8(%rdx)
   0x0000000000401267 <+232>:   mov    $0x5,%ebp
   0x000000000040126c <+237>:   mov    0x8(%rbx),%rax
   0x0000000000401270 <+241>:   mov    (%rax),%eax
   0x0000000000401272 <+243>:   cmp    %eax,(%rbx)
   0x0000000000401274 <+245>:   jge    0x40127b <phase_6+252>
   0x0000000000401276 <+247>:   callq  0x401614 <explode_bomb>
   0x000000000040127b <+252>:   mov    0x8(%rbx),%rbx
   0x000000000040127f <+256>:   sub    $0x1,%ebp
   0x0000000000401282 <+259>:   jne    0x40126c <phase_6+237>
   0x0000000000401284 <+261>:   add    $0x50,%rsp
   0x0000000000401288 <+265>:   pop    %rbx
   0x0000000000401289 <+266>:   pop    %rbp
   0x000000000040128a <+267>:   pop    %r12
   0x000000000040128c <+269>:   pop    %r13
   0x000000000040128e <+271>:   pop    %r14
   0x0000000000401290 <+273>:   retq
End of assembler dump.
```

```
  0x0000000000401198 <+25>:     mov    %r13,%r14
  0x000000000040119b <+28>:     mov    $0x0,%r12d
  0x00000000004011a1 <+34>:     mov    %r13,%rbp
  0x00000000004011a4 <+37>:     mov    0x0(%r13),%eax
  0x00000000004011a8 <+41>:     sub    $0x1,%eax
  0x00000000004011ab <+44>:     cmp    $0x5,%eax
  0x00000000004011ae <+47>:     jbe    0x4011b5 <phase_6+54>
  0x00000000004011b0 <+49>:     callq  0x401614 <explode_bomb>
```

%eax-$0x1을 %eax에 넣어준다. %eax가 5보다 작거나 같아야함으로 입력값은 6 이하의 수여야 한다.

```
  0x00000000004011b5 <+54>:     add    $0x1,%r12d
  0x00000000004011b9 <+58>:     cmp    $0x6,%r12d
  0x00000000004011bd <+62>:     je     0x4011e1 <phase_6+98>
  0x00000000004011bf <+64>:     mov    %r12d,%ebx
  0x00000000004011c2 <+67>:     movslq %ebx,%rax
  0x00000000004011c5 <+70>:     mov    0x30(%rsp,%rax,4),%eax
  0x00000000004011c9 <+74>:     cmp    %eax,0x0(%rbp)
  0x00000000004011cc <+77>:     jne    0x4011d3 <phase_6+84>
  0x00000000004011ce <+79>:     callq  0x401614 <explode_bomb>
  0x00000000004011d3 <+84>:     add    $0x1,%ebx
  0x00000000004011d6 <+87>:     cmp    $0x5,%ebx
  0x00000000004011d9 <+90>:     jle    0x4011c2 <phase_6+67>
  0x00000000004011db <+92>:     add    $0x4,%r13
  0x00000000004011df <+96>:     jmp    0x4011a1 <phase_6+34>
```

%r12d에 1을 더하고 이를 6과 비교하여 맞으면 <+98>로 점프한다. 이를 통해 %r12d는 인자 수를 세는 레지스터라는 것을 알 수 있다. %r12d를 %ebx에 넣어주고 %ebx를 %rax에 sign extend해서 넣어준다. <+70> 에서는 원소의 값을 이동시켜가며 <+74>~<+87>에서는 반복문을 구현하여 중복을 확인하는 작업을 한다. 중복되는 원소가 존재한다면 explode_bomb을 호출하여 폭탄이 터지게 된다. 즉 정리하자면, 입력값의 조건은 1에서 6까지의 숫자이며 중복이 없는 숫자들이라는 것이다.

```
  0x00000000004011e1 <+98>:    lea    0x48(%rsp),%rsi
  0x00000000004011e6 <+103>:   mov    %r14,%rax
--Type <return> to continue, or q <return> to quit---continue
```

```
  0x00000000004011e9 <+106>:   mov    $0x7,%ecx
  0x00000000004011ee <+111>:   mov    %ecx,%edx
  0x00000000004011f0 <+113>:   sub    (%rax),%edx
  0x00000000004011f2 <+115>:   mov    %edx,(%rax)
  0x00000000004011f4 <+117>:   add    $0x4,%rax
  0x00000000004011f8 <+121>:   cmp    %rsi,%rax
  0x00000000004011fb <+124>:   jne    0x4011ee <phase_6+111>
```

<+98>부터 <+124>까지의 코드를 살펴보면 0x48+%rsp를 %rsi에 넣어주는데, 이는 배열 요소르 가리키는 포인터이다. 그다음 %r14를 %rax에 넣어주는데 이 %r14는 이전에 %r13으로부터 가져온 값이다. 상수 0x7을 %ecx에 넣어서 %ecx를 loop counter로 사용한다. %ecx를 %edx에 넣어서 %edx를 루프 반복 횟수를 추적하는데 사용한다. %edx-(%rax)를 %edx에 넣어주고 %edx를 %rax에 넘겨주는데 %rax는 이전 스택 포인터가 가리키는 배열을 가리킨다. 즉 각각의 모든 입력값을 7에서 뺀 값을 배열의 index로 하겠다는 뜻이다.

```
  0x00000000004011fd <+126>:   mov    $0x0,%esi
  0x0000000000401202 <+131>:   jmp    0x401224 <phase_6+165>
  0x0000000000401204 <+133>:   mov    0x8(%rdx),%rdx
  0x0000000000401208 <+137>:   add    $0x1,%eax
  0x000000000040120b <+140>:   cmp    %ecx,%eax
  0x000000000040120d <+142>:   jne    0x401204 <phase_6+133>
  0x000000000040120f <+144>:   jmp    0x401216 <phase_6+151>
  0x0000000000401211 <+146>:   mov    $0x6042f0,%edx
  0x0000000000401216 <+151>:   mov    %rdx,(%rsp,%rsi,2)
  0x000000000040121a <+155>:   add    $0x4,%rsi
  0x000000000040121e <+159>:   cmp    $0x18,%rsi
  0x0000000000401222 <+163>:   je     0x401239 <phase_6+186>
  0x0000000000401224 <+165>:   mov    0x30(%rsp,%rsi,1),%ecx
```

<+142>, <+144>를 통해 eax가 첫번째 인자가 1이어야 다음으로 넘어간다는 것을 알 수 있다. 그러므로 첫번째 입력 값은 1이다. <+151>~<+163>을 보면 이중 반복문을 돌면서 배열 요소를 다 확인하고 있음을 알 수 있다. 0x6042f0을 시작 주소로 하는 %rdx를 x/96x 명령어로 확인해보면 다음과 같이 6개의 노드를 추적할 수 있다.

```
(gdb) x/96x 0x6042f0
0x6042f0 <node1>:       0xc5    0x01    0x00    0x00    0x01    0x00    0x00    0x00
0x6042f8 <node1+8>:     0x00    0x43    0x60    0x00    0x00    0x00    0x00    0x00
0x604300 <node2>:       0x99    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x604308 <node2+8>:     0x10    0x43    0x60    0x00    0x00    0x00    0x00    0x00
0x604310 <node3>:       0x5f    0x01    0x00    0x00    0x03    0x00    0x00    0x00
0x604318 <node3+8>:     0x20    0x43    0x60    0x00    0x00    0x00    0x00    0x00
0x604320 <node4>:       0x9c    0x00    0x00    0x00    0x04    0x00    0x00    0x00
0x604328 <node4+8>:     0x30    0x43    0x60    0x00    0x00    0x00    0x00    0x00
0x604330 <node5>:       0x0f    0x02    0x00    0x00    0x05    0x00    0x00    0x00
0x604338 <node5+8>:     0x40    0x43    0x60    0x00    0x00    0x00    0x00    0x00
0x604340 <node6>:       0x5d    0x01    0x00    0x00    0x06    0x00    0x00    0x00
0x604348 <node6+8>:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

<node1>:0x1c5 <node2>::0x99 <node3>:0x15f <node4>:0x9c <node5>: 0x20f <node6>:0x15d

→ 각 노드에는 다음과 같은 값이 들어있음을 알 수 있으며 <node+8>에는 다음 노드의 주소를 담고 있음을 파악할 수 있다. 따라서 이는 노드들이 연결되어 있는 linked list 형태라는 것을 짐작할 수 있다.

<+243>을 보면 (%rbx)와 %eax를 비교했을 때 (%rbx)가 %eax보다 크거나 같아야한다는 것을 알 수 있다. 이때 앞선 코드를 보면 %rbx는 어떤 노드를 가리키고 있는 것이고 %rax는 0x8만큼 이동하여 해당 노드에 다음 노드를 가리키는 포인터를 담게 된다. 이 포인터를 참조하여 다음 노드를 얻어 %eax에 담는 것이다. 이를 통해 %rbx가 항상 %eax보다 크거나 같아야하므로 노드들이 내림차순으로 정렬되어야함을 알 수 있다. 노드들의 값이 큰 순서대로 다시 정렬하면 다음과 같다.

node5 > node1 > node3 > node6 > node4 > node2

```
0x00000000004011e9 <+106>:   mov    $0x7,%ecx
0x00000000004011ee <+111>:   mov    %ecx,%edx
0x00000000004011f0 <+113>:   sub    (%rax),%edx
```

앞선 코드에 다음과 같은 부분이 있었으므로 실제 입력 값은 각 노드의 인덱스를 7에서 빼준 값이라는 것을 적용하면 답은 2 6 4 1 3 5 이다.

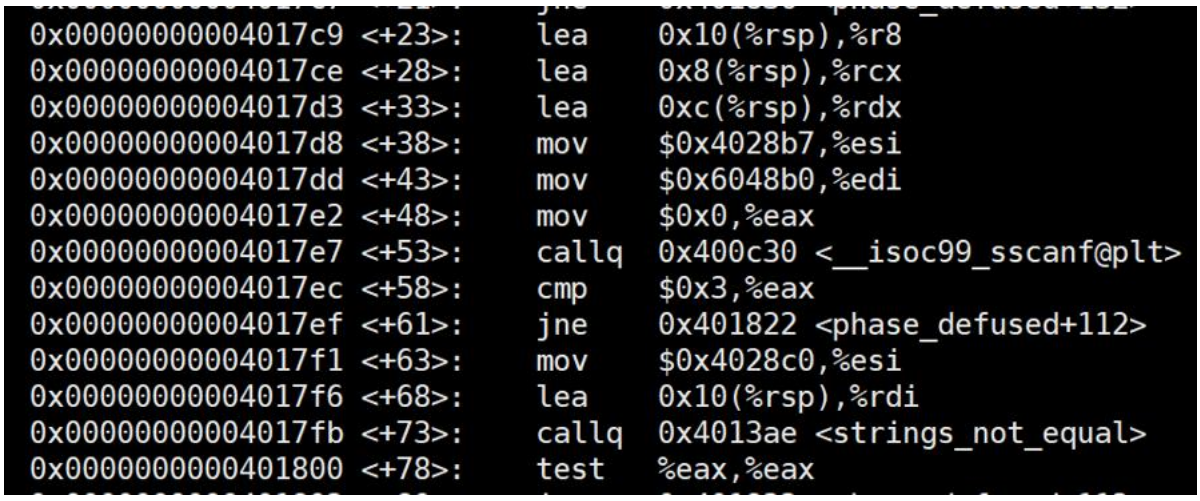
```
Good work!  On to the next...
2 6 4 1 3 5

Breakpoint 2, 0x000000000040117f in phase_6 ()
(gdb) continue
Continuing.
Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
```

phase_6 답: 2 6 4 1 3 5

- secret_phase

phase_defused를 disassemble하면 다음과 같다.

```
(gdb) disas phase_defused
Dump of assembler code for function phase_defused:
   0x00000000004017b2 <+0>:     sub    $0x68,%rsp
   0x00000000004017b6 <+4>:     mov    $0x1,%edi
   0x00000000004017bb <+9>:     callq  0x401550 <send_msg>
   0x00000000004017c0 <+14>:    cmpl   $0x6,0x202fd5(%rip)        # 0x60479c <num_input_strin
gs>
   0x00000000004017c7 <+21>:    jne    0x401836 <phase_defused+132>
   0x00000000004017c9 <+23>:    lea    0x10(%rsp),%r8
   0x00000000004017ce <+28>:    lea    0x8(%rsp),%rcx
   0x00000000004017d3 <+33>:    lea    0xc(%rsp),%rdx
   0x00000000004017d8 <+38>:    mov    $0x4028b7,%esi
   0x00000000004017dd <+43>:    mov    $0x6048b0,%edi
   0x00000000004017e2 <+48>:    mov    $0x0,%eax
   0x00000000004017e7 <+53>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x00000000004017ec <+58>:    cmp    $0x3,%eax
   0x00000000004017ef <+61>:    jne    0x401822 <phase_defused+112>
   0x00000000004017f1 <+63>:    mov    $0x4028c0,%esi
   0x00000000004017f6 <+68>:    lea    0x10(%rsp),%rdi
   0x00000000004017fb <+73>:    callq  0x4013ae <strings_not_equal>
   0x0000000000401800 <+78>:    test   %eax,%eax
   0x0000000000401802 <+80>:    jne    0x401822 <phase_defused+112>
   0x0000000000401804 <+82>:    mov    $0x402718,%edi
   0x0000000000401809 <+87>:    callq  0x400b40 <puts@plt>
---Type <return> to continue, or q <return> to quit---c
   0x000000000040180e <+92>:    mov    $0x402740,%edi
   0x0000000000401813 <+97>:    callq  0x400b40 <puts@plt>
   0x0000000000401818 <+102>:   mov    $0x0,%eax
   0x000000000040181d <+107>:   callq  0x4012cf <secret_phase>
   0x0000000000401822 <+112>:   mov    $0x402778,%edi
   0x0000000000401827 <+117>:   callq  0x400b40 <puts@plt>
   0x000000000040182c <+122>:   mov    $0x4027a8,%edi
   0x0000000000401831 <+127>:   callq  0x400b40 <puts@plt>
   0x0000000000401836 <+132>:   add    $0x68,%rsp
   0x000000000040183a <+136>:   retq
End of assembler dump.
```



이 부분에서 x 명령어로 0x4028b7을 확인해보면 "%d %d %s"가 들어있음을 확인할 수 있다. 따라서 정수 2개와 문자열 1개를 입력받아야 한다. 또한 <+58>에서 scanf 함수의 리턴값이 3이어야만 secret_phase에 진입할 수 있다는 것을 알 수 있다. <+63>으로부터 x 명령어를 사용하여 0x4028c0에 저장된 string을 확인해보면 "DrEvil"이 들어있음을 알 수 있다. 따라서 이는 phase_4

에서 두 개의 정수를 입력 받고 추가로 "DrEvil"이라는 문자열을 입력해줄 때 secret phase를 열 수 있다는 것을 의미한다. phase_4에 대한 답으로 기존의 답인 4 19에 추가로 문자열을 입력하여 4 19 DrEvil을 입력한다.

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

secret phase에 진입해서 disassemble하면 다음과 같다.

```
(gdb) disas secret_phase
Dump of assembler code for function secret_phase:
=> 0x00000000004012cf <+0>:     push   %rbx
   0x00000000004012d0 <+1>:     callq  0x40168c <read_line>
   0x00000000004012d5 <+6>:     mov    $0xa,%edx
   0x00000000004012da <+11>:    mov    $0x0,%esi
   0x00000000004012df <+16>:    mov    %rax,%rdi
   0x00000000004012e2 <+19>:    callq  0x400c00 <strtol@plt>
   0x00000000004012e7 <+24>:    mov    %rax,%rbx
   0x00000000004012ea <+27>:    lea    -0x1(%rax),%eax
   0x00000000004012ed <+30>:    cmp    $0x3e8,%eax
   0x00000000004012f2 <+35>:    jbe    0x4012f9 <secret_phase+42>
   0x00000000004012f4 <+37>:    callq  0x401614 <explode_bomb>
   0x00000000004012f9 <+42>:    mov    %ebx,%esi
   0x00000000004012fb <+44>:    mov    $0x604110,%edi
   0x0000000000401300 <+49>:    callq  0x401291 <fun7>
   0x0000000000401305 <+54>:    cmp    $0x7,%eax
   0x0000000000401308 <+57>:    je     0x40130f <secret_phase+64>
   0x000000000040130a <+59>:    callq  0x401614 <explode_bomb>
   0x000000000040130f <+64>:    mov    $0x402598,%edi
   0x0000000000401314 <+69>:    callq  0x400b40 <puts@plt>
   0x0000000000401319 <+74>:    callq  0x4017b2 <phase_defused>
---Type <return> to continue, or q <return> to quit---c
   0x000000000040131e <+79>:    pop    %rbx
   0x000000000040131f <+80>:    retq
End of assembler dump.
```

```
 0x00000000004012e7 <+24>:    mov    %rax,%rbx
 0x00000000004012ea <+27>:    lea    -0x1(%rax),%eax
 0x00000000004012ed <+30>:    cmp    $0x3e8,%eax
 0x00000000004012f2 <+35>:    jbe    0x4012f9 <secret_phase+42>
 0x00000000004012f4 <+37>:    callq  0x401614 <explode_bomb>
 0x00000000004012f9 <+42>:    mov    %ebx,%esi
 0x00000000004012fb <+44>:    mov    $0x604110,%edi
 0x0000000000401300 <+49>:    callq  0x401291 <fun7>
 0x0000000000401305 <+54>:    cmp    $0x7,%eax
```

초반의 코드를 보면 strtol 함수의 리턴값에서 -1한 값이 0x328(1000)과 비교하였을 때 작거나 같

아야 통과하므로 정답은 1001 이하의 숫자여야 한다는 것을 알 수 있다. 폭탄이 터지지 않고 통과하면 %esi에 %ebx를 넘겨주고, %edi에 0x604110을 넘겨준다. 앞선 문제들처럼 x 명령어를 사용해 0x604110을 가보니 다음과 같다.

```
(gdb) x/d 0x604110
0x604110 <n1>:  36
```

노드에 의해 구성되었을 것이라고 예측할 수 있다.

또한 <+57>에서 비교구문을 통해 fun7의 리턴값이 0x7일 때 secret_phase를 통과할 수 있다는 것을 알 수 있다. 따라서 구해야하는 것은 어떤 값을 fun7에 넘겨주었을 때 리턴값이 7인지이다.

인자로 값을 넘겨주어 fun7을 호출하므로 이 fun7을 살펴본다. disassemble하면 다음과 같다.

```
(gdb) disas fun7
Dump of assembler code for function fun7:
   0x0000000000401291 <+0>:     sub    $0x8,%rsp
   0x0000000000401295 <+4>:     test   %rdi,%rdi
   0x0000000000401298 <+7>:     je     0x4012c5 <fun7+52>
   0x000000000040129a <+9>:     mov    (%rdi),%edx
   0x000000000040129c <+11>:    cmp    %esi,%edx
   0x000000000040129e <+13>:    jle    0x4012ad <fun7+28>
   0x00000000004012a0 <+15>:    mov    0x8(%rdi),%rdi
   0x00000000004012a4 <+19>:    callq  0x401291 <fun7>
   0x00000000004012a9 <+24>:    add    %eax,%eax
   0x00000000004012ab <+26>:    jmp    0x4012ca <fun7+57>
   0x00000000004012ad <+28>:    mov    $0x0,%eax
   0x00000000004012b2 <+33>:    cmp    %esi,%edx
   0x00000000004012b4 <+35>:    je     0x4012ca <fun7+57>
   0x00000000004012b6 <+37>:    mov    0x10(%rdi),%rdi
   0x00000000004012ba <+41>:    callq  0x401291 <fun7>
   0x00000000004012bf <+46>:    lea    0x1(%rax,%rax,1),%eax
   0x00000000004012c3 <+50>:    jmp    0x4012ca <fun7+57>
   0x00000000004012c5 <+52>:    mov    $0xffffffff,%eax
   0x00000000004012ca <+57>:    add    $0x8,%rsp
   0x00000000004012ce <+61>:    retq
---Type <return> to continue, or q <return> to quit---c
End of assembler dump.
```

<+7>의 비교구문에서는 null인지를 확인하고 null이라면 0xffffffff을 리턴하여 종료한다. null이 아니라 비교구문을 통과하면 %edx에 (%rdi)를 넘겨주고 %edx와 %rdi를 비교한다. 이때 %edx가 %esi보다 작거나 같으면, <+28>으로 점프하고 %esi보다 크면 이 구문을 통과하여 계속해서 진행된다. 각각의 경우에 대해서 정리를 해보고, 노드라는 점을 참고하여 트리구조라고 예측할 수 있다. 앞서서 트리를 구성하는 노드 하나를 보았으니 계속해서 다른 노드들을 찾아본다.

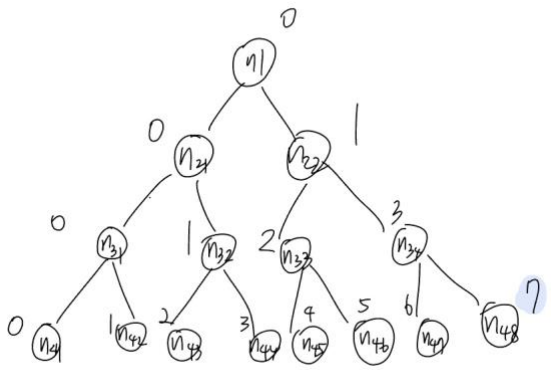
```
(gdb) x/24x 0x604110
0x604110 <n1>:  0x24    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x604118 <n1+8>:        0x30    0x41    0x60    0x00    0x00    0x00    0x00    0x00
0x604120 <n1+16>:       0x50    0x41    0x60    0x00    0x00    0x00    0x00    0x00
```

```
(gdb) x/24x 0x604130
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
```

같은 방식으로 다른 노드들을 찾아보면 다음과 같다.

```
(gdb) x/24x 0x604150
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
(gdb) x/24x 0x604170
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
(gdb) x/24x 0x604190
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
(gdb) x/24x 0x6041b0
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
(gdb) x/24x 0x6041d0
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
(gdb) x/24x 0x6041f0
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
```

<node+8>에는 left child의 주소를, <node+10>에는 right child의 주소를 나타내고 있다. 위의 값들과 주소 값을 참고하여 트리가 어떻게 생겼는지를 추측해보면 다음과 같다.



노드를 나타내는 원 위에 적힌 숫자들은 그 노드에 해당하는 입력 값을 넣었을 때의 fun7 리턴값을 나타낸다. 우리가 구하고자 하는 것은 리턴값이 7일 때 그 노드의 값이니까 그림을 보다시피 n48에 위치할 값을 찾으면 된다. fun7의 첫번째 인자가 36이니까 n1의 값이 36이라고 본다. n48은 n1->n22->n34->n48을 따르므로 <n34+16>을 참고하여 0x604270이 n48의 주소임을 알 수 있다. 이를 x 명령어로 찾아보면 다음과 같다.

```
(gdb) x/4x 0x6042d0
0x6042d0 <n48>: 0xe9    0x03    0x00    0x00
```

따라서 0x3e9, 10진수로 계산하면 1001임을 알 수 있다. 이는 앞서 코드에서 명시된 조건들을 만족하며 fun7에 인자로 넣었을 때 7을 리턴하는 값이 된다. 따라서 secret phase의 답은 1001이다.

```
1001
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
```

secret_phase 답: 1001

secret phase까지 포함한 phase들을 모두 해결하여 최종적으로 defused된 모습이다.

```
Starting program: /home/std/kimyujin1224/bomb67/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
1 2 4 8 16 32
0 l 875
4 19 DrEvil
jdoefg
2 6 4 1 3 5
1001
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
Curses, you've found the secret phase!
But finding 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.
```