컴퓨터 SW시스템 개론 Lab3 Report

20220923 이윤혁

1. 개요

Lab 3에서는 gdb를 이용해 bomb 파일의 assembly code를 분석한다. 분석 결과를 이용해 폭탄을 해체하도록 올바른 입력을 넣는다.

2. 구조

먼저 bomb에 선언된 모든 함수를 보기 위해, info function를 입력했다.

```
All defined functions:

File bomb.c:
int main(int, char **);

Non-debugging symbols:
0x0000000000400ad0  _init
0x0000000000400b00  getenv@plt
0x0000000000400b10  strcasecmp@plt
0x0000000000400b20  __errno_location@plt
0x0000000000400b30  strcpy@plt
0x0000000000400b40  puts@plt
0x0000000000400b50  write@plt
0x0000000000400b60  printf@plt
0x0000000000400b70  alarm@plt
0x0000000000400b80  close@plt
0x0000000000400b90  read@plt
0x0000000000400ba0  __libc_start_main@plt
0x0000000000400bb0  fgets@plt
0x0000000000400bc0  signal@plt
0x0000000000400bd0  gethostbyname@plt
0x0000000000400be0  fprintf@plt
0x0000000000400bf0  __gmon_start__@plt
0x0000000000400c00  strtol@plt
0x0000000000400c10  memcpy@plt
0x0000000000400c20  fflush@plt
0x0000000000400c30  __isoc99_sscanf@plt
0x0000000000400c40  bcopy@plt
0x0000000000400c50  fopen@plt
```

```
0x0000000000400c60   gethostname@plt
0x0000000000400c70   sprintf@plt
0x0000000000400c80   exit@plt
0x0000000000400c90   connect@plt
0x0000000000400ca0   sleep@plt
0x0000000000400cb0   __ctype_b_loc@plt
0x0000000000400cc0   socket@plt
0x0000000000400cd0   _start
0x0000000000400d00   deregister_tm_clones
0x0000000000400d30   register_tm_clones
0x0000000000400d70   __do_global_dtors_aux
0x0000000000400d90   frame_dummy
0x0000000000400ef0   phase_1
0x0000000000400f0c   phase_2
0x0000000000400f5b   phase_3
0x00000000004010a5   func4
0x00000000004010dd   phase_4
0x000000000040112e   phase_5
0x000000000040119b   phase_6
0x000000000040128c   fun7
0x00000000004012ca   secret_phase
0x0000000000401320   sig_handler
0x0000000000401371   invalid_phase
0x0000000000401391   string_length
0x00000000004013ae   strings_not_equal
0x0000000000401417   initialize_bomb
0x00000000004014d0   initialize_bomb_solve
0x00000000004014d2   blank_line
0x000000000040150f   skip
0x0000000000401550   send_msg
0x0000000000401614   explode_bomb
0x000000000040164a   read_six_numbers
0x000000000040168c   read_line
0x00000000004017b2   phase_defused
0x0000000000401840   sigalrm_handler
0x0000000000401869   rio_readlineb
0x0000000000401980   submitr
0x000000000040213a   init_timeout
0x0000000000402161   init_driver
```

```
        0x00000000004022ff   driver_post
        0x0000000000402380   __libc_csu_init
        0x00000000004023f0   __libc_csu_fini
        0x00000000004023f4   _fini
```
볼드처리한 함수들은 Lab3에서 분석이 필요했다.

3. main

bomb이 전체 프로세스는 main에서 일어날 것이다. main 함수의 구조를 알기 위해 disas main을 실행했다.

```
Dump of assembler code for function main:
   0x0000000000400dbd <+0>:     push   %rbx
   0x0000000000400dbe <+1>:     cmp    $0x1,%edi
   0x0000000000400dc1 <+4>:     jne    0x400dd3 <main+22>
   0x0000000000400dc3 <+6>:     mov    0x2039be(%rip),%rax        #
0x604788 <stdin@@GLIBC_2.2.5>
   0x0000000000400dca <+13>:    mov    %rax,0x2039cf(%rip)        #
0x6047a0 <infile>
   0x0000000000400dd1 <+20>:    jmp    0x400e2c <main+111>
   0x0000000000400dd3 <+22>:    mov    %rsi,%rbx
   0x0000000000400dd6 <+25>:    cmp    $0x2,%edi
   0x0000000000400dd9 <+28>:    jne    0x400e10 <main+83>
   0x0000000000400ddb <+30>:    mov    0x8(%rsi),%rdi
   0x0000000000400ddf <+34>:    mov    $0x402914,%esi
   0x0000000000400de4 <+39>:    callq  0x400c50 <fopen@plt>
   0x0000000000400de9 <+44>:    mov    %rax,0x2039b0(%rip)        #
0x6047a0 <infile>
   0x0000000000400df0 <+51>:    test   %rax,%rax
   0x0000000000400df3 <+54>:    jne    0x400e2c <main+111>
   0x0000000000400df5 <+56>:    mov    0x8(%rbx),%rdx
   0x0000000000400df9 <+60>:    mov    (%rbx),%rsi
   0x0000000000400dfc <+63>:    mov    $0x402410,%edi
   0x0000000000400e01 <+68>:    callq  0x400b60 <printf@plt>
   0x0000000000400e06 <+73>:    mov    $0x8,%edi
   0x0000000000400e0b <+78>:    callq  0x400c80 <exit@plt>
   0x0000000000400e10 <+83>:    mov    (%rsi),%rsi
   0x0000000000400e13 <+86>:    mov    $0x40242d,%edi
   0x0000000000400e18 <+91>:    mov    $0x0,%eax
   0x0000000000400e1d <+96>:    callq  0x400b60 <printf@plt>
```

```
0x0000000000400e22 <+101>:    mov     $0x8,%edi
0x0000000000400e27 <+106>:    callq   0x400c80 <exit@plt>
0x0000000000400e2c <+111>:    callq   0x401417 <initialize_bomb>
0x0000000000400e31 <+116>:    mov     $0x402498,%edi
0x0000000000400e36 <+121>:    callq   0x400b40 <puts@plt>
0x0000000000400e3b <+126>:    mov     $0x4024d8,%edi
0x0000000000400e40 <+131>:    callq   0x400b40 <puts@plt>
0x0000000000400e45 <+136>:    callq   0x40168c <read_line>
0x0000000000400e4a <+141>:    mov     %rax,%rdi
0x0000000000400e4d <+144>:    callq   0x400ef0 <phase_1>
0x0000000000400e52 <+149>:    callq   0x4017b2 <phase_defused>
0x0000000000400e57 <+154>:    mov     $0x402508,%edi
0x0000000000400e5c <+159>:    callq   0x400b40 <puts@plt>
0x0000000000400e61 <+164>:    callq   0x40168c <read_line>
0x0000000000400e66 <+169>:    mov     %rax,%rdi
0x0000000000400e69 <+172>:    callq   0x400f0c <phase_2>
0x0000000000400e6e <+177>:    callq   0x4017b2 <phase_defused>
0x0000000000400e73 <+182>:    mov     $0x402447,%edi
0x0000000000400e78 <+187>:    callq   0x400b40 <puts@plt>
0x0000000000400e7d <+192>:    callq   0x40168c <read_line>
0x0000000000400e82 <+197>:    mov     %rax,%rdi
0x0000000000400e85 <+200>:    callq   0x400f5b <phase_3>
0x0000000000400e8a <+205>:    callq   0x4017b2 <phase_defused>
0x0000000000400e8f <+210>:    mov     $0x402465,%edi
0x0000000000400e94 <+215>:    callq   0x400b40 <puts@plt>
0x0000000000400e99 <+220>:    callq   0x40168c <read_line>
0x0000000000400e9e <+225>:    mov     %rax,%rdi
0x0000000000400ea1 <+228>:    callq   0x4010dd <phase_4>
0x0000000000400ea6 <+233>:    callq   0x4017b2 <phase_defused>
0x0000000000400eab <+238>:    mov     $0x402538,%edi
0x0000000000400eb0 <+243>:    callq   0x400b40 <puts@plt>
0x0000000000400eb5 <+248>:    callq   0x40168c <read_line>
0x0000000000400eba <+253>:    mov     %rax,%rdi
0x0000000000400ebd <+256>:    callq   0x40112e <phase_5>
0x0000000000400ec2 <+261>:    callq   0x4017b2 <phase_defused>
0x0000000000400ec7 <+266>:    mov     $0x402474,%edi
0x0000000000400ecc <+271>:    callq   0x400b40 <puts@plt>
0x0000000000400ed1 <+276>:    callq   0x40168c <read_line>
0x0000000000400ed6 <+281>:    mov     %rax,%rdi
```

```
    0x0000000000400ed9 <+284>:    callq   0x40119b <phase_6>
    0x0000000000400ede <+289>:    callq   0x4017b2 <phase_defused>
    0x0000000000400ee3 <+294>:    mov     $0x0,%eax
    0x0000000000400ee8 <+299>:    pop     %rbx
    0x0000000000400ee9 <+300>:    retq
End of assembler dump.
```

main의 assembly code를 C-style의 pseudo code로 변화시켜, 코드의 흐름을 제시할 것이다. 코드 전체를 pseudo code로 변환하는 것은 많은 시간을 소요하기에, 필요한 부분만 추출해서 볼 것이다. 또한 자료형의 명시도 엄밀하지 않을 수 있다. 작동이해에는 지장이 없으므로 pseudo code는 strict하게 명시하지 않았다. assembly code에 직접적으로 명시되어 있지 않은 string들은 주소를 참조해 pseudo code에 삽입했다.

```
int main(int, char **)
{
  ... 중략 ...
  puts("Welcome to my fiendish little bomb. You have 6 phases with");
  puts("which to blow yourself up. Have a nice day!");
  line = read_line();
  phase_1(line);
  phase_defused();
  puts("Phase 1 defused. How about the next one?");
  line = read_line();
  phase_2(line);
  phase_defused();
  puts("That's number 2.  Keep going!");
  line = read_line();
  phase_3(line);
  phase_defused();
  puts("Halfway there!");
  line = read_line();
  phase_4(line);
  phase_defused();
  puts("So you got that one.  Try this one.");
  line = read_line();
  phase_5(line);
  phase_defused();
  puts("Good work!  On to the next...");
  line = read_line();
```

```
        phase_6(line);
        phase_defused();
        return 0;
}
```

전체적인 흐름을 요약하면 다음과 같다.

1. line을 읽는다(사용자의 입력을 받는다)
2. phase_{n}을 실행한다.
3. phase_{n}의 실행결과가 올바른 경우, phase_defused()를 실행한다.
4. phase_{n}의 실행결과가 올바르지 않은 경우, explode_bomb을 실행한다. (후술)
5. n을 1~6까지 반복한다.

read_line의 assembly code는 다음과 같다.

```
Dump of assembler code for function read_line:
   0x000000000040168c <+0>:     sub    $0x8,%rsp
   0x0000000000401690 <+4>:     mov    $0x0,%eax
   0x0000000000401695 <+9>:     callq  0x40150f <skip>
   0x000000000040169a <+14>:    test   %rax,%rax
   0x000000000040169d <+17>:    jne    0x40170d <read_line+129>
   0x000000000040169f <+19>:    mov    0x2030e2(%rip),%rax        #
0x604788 <stdin@@GLIBC_2.2.5>
   0x00000000004016a6 <+26>:    cmp    %rax,0x2030f3(%rip)        #
0x6047a0 <infile>
   0x00000000004016ad <+33>:    jne    0x4016c3 <read_line+55>
   0x00000000004016af <+35>:    mov    $0x402893,%edi
   0x00000000004016b4 <+40>:    callq  0x400b40 <puts@plt>
   0x00000000004016b9 <+45>:    mov    $0x8,%edi
   0x00000000004016be <+50>:    callq  0x400c80 <exit@plt>
   0x00000000004016c3 <+55>:    mov    $0x4028b1,%edi
   0x00000000004016c8 <+60>:    callq  0x400b00 <getenv@plt>
   0x00000000004016cd <+65>:    test   %rax,%rax
   0x00000000004016d0 <+68>:    je     0x4016dc <read_line+80>
   0x00000000004016d2 <+70>:    mov    $0x0,%edi
   0x00000000004016d7 <+75>:    callq  0x400c80 <exit@plt>
   0x00000000004016dc <+80>:    mov    0x2030a5(%rip),%rax        #
0x604788 <stdin@@GLIBC_2.2.5>
   0x00000000004016e3 <+87>:    mov    %rax,0x2030b6(%rip)        #
```

```
0x6047a0 <infile>
   0x00000000004016ea <+94>:    mov     $0x0,%eax
   0x00000000004016ef <+99>:    callq   0x40150f <skip>
   0x00000000004016f4 <+104>:   test    %rax,%rax
   0x00000000004016f7 <+107>:   jne     0x40170d <read_line+129>
   0x00000000004016f9 <+109>:   mov     $0x402893,%edi
   0x00000000004016fe <+114>:   callq   0x400b40 <puts@plt>
   0x0000000000401703 <+119>:   mov     $0x0,%edi
   0x0000000000401708 <+124>:   callq   0x400c80 <exit@plt>
   0x000000000040170d <+129>:   mov     0x203089(%rip),%edx        #
0x60479c <num_input_strings>
   0x0000000000401713 <+135>:   movslq %edx,%rax
   0x0000000000401716 <+138>:   lea     (%rax,%rax,4),%rsi
   0x000000000040171a <+142>:   shl     $0x4,%rsi
   0x000000000040171e <+146>:   add     $0x6047c0,%rsi
   0x0000000000401725 <+153>:   mov     %rsi,%rdi
   0x0000000000401728 <+156>:   mov     $0x0,%eax
   0x000000000040172d <+161>:   mov     $0xffffffffffffffff,%rcx
   0x0000000000401734 <+168>:   repnz scas %es:(%rdi),%al
   0x0000000000401736 <+170>:   not     %rcx
   0x0000000000401739 <+173>:   sub     $0x1,%rcx
   0x000000000040173d <+177>:   cmp     $0x4e,%ecx
   0x0000000000401740 <+180>:   jle     0x401788 <read_line+252>
   0x0000000000401742 <+182>:   mov     $0x4028bc,%edi
   0x0000000000401747 <+187>:   callq   0x400b40 <puts@plt>
   0x000000000040174c <+192>:   mov     0x20304a(%rip),%eax        #
0x60479c <num_input_strings>
   0x0000000000401752 <+198>:   lea     0x1(%rax),%edx
   0x0000000000401755 <+201>:   mov     %edx,0x203041(%rip)        #
0x60479c <num_input_strings>
   0x000000000040175b <+207>:   cltq
   0x000000000040175d <+209>:   imul    $0x50,%rax,%rax
   0x0000000000401761 <+213>:   movabs $0x636e7572742a2a2a,%rdi
   0x000000000040176b <+223>:   mov     %rdi,0x6047c0(%rax)
   0x0000000000401772 <+230>:   movabs $0x2a2a2a64657461,%rdi
   0x000000000040177c <+240>:   mov     %rdi,0x6047c8(%rax)
   0x0000000000401783 <+247>:   callq   0x401614 <explode_bomb>
   0x0000000000401788 <+252>:   sub     $0x1,%ecx
   0x000000000040178b <+255>:   movslq %ecx,%rcx
```

```
        0x000000000040178e <+258>:    movslq %edx,%rax
        0x0000000000401791 <+261>:    lea     (%rax,%rax,4),%rax
        0x0000000000401795 <+265>:    shl     $0x4,%rax
        0x0000000000401799 <+269>:    movb    $0x0,0x6047c0(%rcx,%rax,1)
        0x00000000004017a1 <+277>:    add     $0x1,%edx
        0x00000000004017a4 <+280>:    mov     %edx,0x202ff2(%rip)          #
    0x60479c <num_input_strings>
        0x00000000004017aa <+286>:    mov     %rsi,%rax
        0x00000000004017ad <+289>:    add     $0x8,%rsp
        0x00000000004017b1 <+293>:    retq
    End of assembler dump.
```

phase_defused의 assembly code는 다음과 같다.

```
    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_strings>
        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     $0x4028d7,%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     $0x4028e0,%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     $0x402738,%edi
        0x0000000000401809 <+87>:    callq   0x400b40 <puts@plt>
        0x000000000040180e <+92>:    mov     $0x402760,%edi
        0x0000000000401813 <+97>:    callq   0x400b40 <puts@plt>
    0x0000000000401818 <+102>:    mov     $0x0,%eax
    0x000000000040181d <+107>:    callq   0x4012ca <secret_phase>
    0x0000000000401822 <+112>:    mov     $0x402798,%edi
    0x0000000000401827 <+117>:    callq   0x400b40 <puts@plt>
    0x000000000040182c <+122>:    mov     $0x4027c8,%edi
    0x0000000000401831 <+127>:    callq   0x400b40 <puts@plt>
    0x0000000000401836 <+132>:    add     $0x68,%rsp
    0x000000000040183a <+136>:    retq
End of assembler dump.
```

secret_phase라는 함수가 호출되는 것을 보아 phase_defused에서 호출하는 phase가 있는 것으로 추정된다. 후에 secret_phase의 진입조건을 찾아볼 것이다.

4. phase_1

disas phase_1을 통해 phase_1의 assembly code를 추출하면 다음과 같다.

```
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.
```

<phase_1+4> 줄에서 rdi 말고 esi가 사용되는 것으로 보아, phase_1 함수는 argument를 한 개 받아 rdi에 저장함을 알 수 있다. esi에는 0x402560주소에 있는 string을 대입하고, strings_not_equal로 string이 edi와 esi가 같은지 검사한다. 같은 경우 <phase_1+23> 줄로 넘어가 리턴하는 것을 보아, 사용자의 입력과 $0x402560에 담긴 string이 같아야 폭탄이 터지지 않는다.

$0x402560에 담긴 string의 값은 다음과 같고, 이것이 phase_1의 답이다.

```
For NASA, space is still a high priority.
```

5. phase_2

disas phase_2를 통해 phase_2의 assembly code를 추출하면 다음과 같다.

```
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   $0x0,(%rsp)
   0x0000000000400f1e <+18>:    jne    0x400f27 <phase_2+27>
   0x0000000000400f20 <+20>:    cmpl   $0x1,0x4(%rsp)
   0x0000000000400f25 <+25>:    je     0x400f48 <phase_2+60>
   0x0000000000400f27 <+27>:    callq  0x401614 <explode_bomb>
   0x0000000000400f2c <+32>:    jmp    0x400f48 <phase_2+60>
   0x0000000000400f2e <+34>:    mov    -0x8(%rbx),%eax
   0x0000000000400f31 <+37>:    add    -0x4(%rbx),%eax
   0x0000000000400f34 <+40>:    cmp    %eax,(%rbx)
   0x0000000000400f36 <+42>:    je     0x400f3d <phase_2+49>
   0x0000000000400f38 <+44>:    callq  0x401614 <explode_bomb>
   0x0000000000400f3d <+49>:    add    $0x4,%rbx
   0x0000000000400f41 <+53>:    cmp    %rbp,%rbx
   0x0000000000400f44 <+56>:    jne    0x400f2e <phase_2+34>
   0x0000000000400f46 <+58>:    jmp    0x400f54 <phase_2+72>
   0x0000000000400f48 <+60>:    lea    0x8(%rsp),%rbx
   0x0000000000400f4d <+65>:    lea    0x18(%rsp),%rbp
   0x0000000000400f52 <+70>:    jmp    0x400f2e <phase_2+34>
   0x0000000000400f54 <+72>:    add    $0x28,%rsp
   0x0000000000400f58 <+76>:    pop    %rbx
   0x0000000000400f59 <+77>:    pop    %rbp
   0x0000000000400f5a <+78>:    retq
End of assembler dump.
```

phase_2의 assembly code를 C-style의 pseudo code로 변화시키면 다음과 같다.

```
int phase_2(int arg)
{
int *next;
int result;
int array[6];
```

```
read_six_numbers(arg, array);
if (array[0] != 0 || array[1] != 1)
  explode_bomb();
next = array[2]
do
{
  result = *(array - 1) + *(array -2);
  if (result != *array)
    explode_bomb();
  next++;
} while(next != array + 5)
return;
```

array라는 6자리 배열이고, read_six_numbers로 값을 읽어 array에 값을 대입한다.

따라서 입력값으로 6개의 정수를 넣어줘야 한다.

array[0] != 0 || array[1] != 1이 참일 때 explode_bomb이 호출되므로, array[0]는 0, array[1]은 1이어야 한다.

do-while 문의 내용은 현재 array pointer에 들어있는 값이 전 pointer와 전전 pointer에 들어있는 값들의 합과 같은지 검사한다. 이것을 array의 3번째 원소부터 6번째 원소까지 계속한다. 따라서 do-while문은 array에 담긴 값들이 피보나치 수열의 관계를 만족하고 있는지 검사한다고 판단할 수 있다.

따라서 phase_2의 답은 다음과 같다.

```
0 1 1 2 3 5
```

6. phase_3

disas phase_3를 통해 phase_3의 assembly code를 얻으면 다음과 같다.

```
Dump of assembler code for function phase_3:
   0x0000000000400f5b <+0>:     sub    $0x18,%rsp
   0x0000000000400f5f <+4>:     lea    0x8(%rsp),%r8
   0x0000000000400f64 <+9>:     lea    0x7(%rsp),%rcx
   0x0000000000400f69 <+14>:    lea    0xc(%rsp),%rdx
   0x0000000000400f6e <+19>:    mov    $0x4025b6,%esi
```

```
0x0000000000400f73 <+24>:    mov    $0x0,%eax
0x0000000000400f78 <+29>:    callq  0x400c30 <__isoc99_sscanf@plt>
0x0000000000400f7d <+34>:    cmp    $0x2,%eax
0x0000000000400f80 <+37>:    jg     0x400f87 <phase_3+44>
0x0000000000400f82 <+39>:    callq  0x401614 <explode_bomb>
0x0000000000400f87 <+44>:    cmpl   $0x7,0xc(%rsp)
0x0000000000400f8c <+49>:    ja     0x40108b <phase_3+304>
0x0000000000400f92 <+55>:    mov    0xc(%rsp),%eax
0x0000000000400f96 <+59>:    jmpq   *0x4025c0(,%rax,8)
0x0000000000400f9d <+66>:    mov    $0x76,%eax
0x0000000000400fa2 <+71>:    cmpl   $0x279,0x8(%rsp)
0x0000000000400faa <+79>:    je     0x401095 <phase_3+314>
0x0000000000400fb0 <+85>:    callq  0x401614 <explode_bomb>
0x0000000000400fb5 <+90>:    mov    $0x76,%eax
0x0000000000400fba <+95>:    jmpq   0x401095 <phase_3+314>
0x0000000000400fbf <+100>:   mov    $0x77,%eax
0x0000000000400fc4 <+105>:   cmpl   $0x12b,0x8(%rsp)
0x0000000000400fcc <+113>:   je     0x401095 <phase_3+314>
0x0000000000400fd2 <+119>:   callq  0x401614 <explode_bomb>
0x0000000000400fd7 <+124>:   mov    $0x77,%eax
0x0000000000400fdc <+129>:   jmpq   0x401095 <phase_3+314>
0x0000000000400fe1 <+134>:   mov    $0x68,%eax
0x0000000000400fe6 <+139>:   cmpl   $0x61,0x8(%rsp)
0x0000000000400feb <+144>:   je     0x401095 <phase_3+314>
0x0000000000400ff1 <+150>:   callq  0x401614 <explode_bomb>
0x0000000000400ff6 <+155>:   mov    $0x68,%eax
0x0000000000400ffb <+160>:   jmpq   0x401095 <phase_3+314>
0x0000000000401000 <+165>:   mov    $0x77,%eax
0x0000000000401005 <+170>:   cmpl   $0x1e3,0x8(%rsp)
0x000000000040100d <+178>:   je     0x401095 <phase_3+314>
0x0000000000401013 <+184>:   callq  0x401614 <explode_bomb>
0x0000000000401018 <+189>:   mov    $0x77,%eax
0x000000000040101d <+194>:   jmp    0x401095 <phase_3+314>
0x000000000040101f <+196>:   mov    $0x63,%eax
0x0000000000401024 <+201>:   cmpl   $0x2b7,0x8(%rsp)
0x000000000040102c <+209>:   je     0x401095 <phase_3+314>
0x000000000040102e <+211>:   callq  0x401614 <explode_bomb>
0x0000000000401033 <+216>:   mov    $0x63,%eax
0x0000000000401038 <+221>:   jmp    0x401095 <phase_3+314>
```

```
        0x000000000040103a <+223>:    mov     $0x78,%eax
        0x000000000040103f <+228>:    cmpl    $0x2aa,0x8(%rsp)
        0x0000000000401047 <+236>:    je      0x401095 <phase_3+314>
        0x0000000000401049 <+238>:    callq   0x401614 <explode_bomb>
        0x000000000040104e <+243>:    mov     $0x78,%eax
        0x0000000000401053 <+248>:    jmp     0x401095 <phase_3+314>
        0x0000000000401055 <+250>:    mov     $0x7a,%eax
        0x000000000040105a <+255>:    cmpl    $0x1b0,0x8(%rsp)
        0x0000000000401062 <+263>:    je      0x401095 <phase_3+314>
        0x0000000000401064 <+265>:    callq   0x401614 <explode_bomb>
        0x0000000000401069 <+270>:    mov     $0x7a,%eax
        0x000000000040106e <+275>:    jmp     0x401095 <phase_3+314>
        0x0000000000401070 <+277>:    mov     $0x79,%eax
        0x0000000000401075 <+282>:    cmpl    $0x133,0x8(%rsp)
        0x000000000040107d <+290>:    je      0x401095 <phase_3+314>
        0x000000000040107f <+292>:    callq   0x401614 <explode_bomb>
        0x0000000000401084 <+297>:    mov     $0x79,%eax
        0x0000000000401089 <+302>:    jmp     0x401095 <phase_3+314>
        0x000000000040108b <+304>:    callq   0x401614 <explode_bomb>
        0x0000000000401090 <+309>:    mov     $0x61,%eax
        0x0000000000401095 <+314>:    cmp     0x7(%rsp),%al
        0x0000000000401099 <+318>:    je      0x4010a0 <phase_3+325>
        0x000000000040109b <+320>:    callq   0x401614 <explode_bomb>
        0x00000000004010a0 <+325>:    add     $0x18,%rsp
        0x00000000004010a4 <+329>:    retq
End of assembler dump.
```

phase_3의 assembly code를 C-style의 pseudo code로 변화시키면 다음과 같다.

```
      int phase_3(int arg)
      {
        int result;
        char key;
        int value;
        int mod;

        if ( (int)__isoc99_sscanf(arg, "%d %c %d", &mod, &key, &value) <= 2 )
          explode_bomb();
        switch ( mod )
        {
```

```
case 0:
  result = 118;
  if ( value != 633 )
    explode_bomb();
  return result;
case 1:
  result = 119;
  if ( value != 299 )
    explode_bomb();
  return result;
case 2:
  result = 104;
  if ( value != 97 )
    explode_bomb();
  return result;
case 3:
  result = 119;
  if ( value != 483 )
    explode_bomb();
  return result;
case 4:
  result = 99;
  if ( value != 695 )
    explode_bomb();
  return result;
case 5:
  result = 120;
  if ( value != 682 )
    explode_bomb();
  return result;
case 6:
  result = 122;
  if ( value != 432 )
    explode_bomb();
  return result;
case 7:
  result = 121;
  if ( value != 307 )
    explode_bomb();
```

```
                return result;
            default:
                explode_bomb();
        }
        if ( result != key )
            explode_bomb();
        return result;
}
```

isco_99_scanf의 인자로 mod, key, value가 들어온다. 입력값이 2개 이하이면 explode_bomb이 실행되므로, mod, key, value의 값이 다 들어와야 한다.

switch-case문은 다음과 같은 구조를 가진다.

```
해당하는 case에 대해서:
매칭되는 result 값 할당
case에 매핑된 value와 input value가 다르면 explode_bomb 실행
```

switch-case문을 벗어나면 result와 key를 비교하여 두 값이 같지 않으면 explode_bomb이 실행된다.

따라서 mod, key, value은 다음과 같은 규칙을 따라야 한다.

```
mod: 0 ~ 7 사이 정수 중 아무 값
key: 선택한 mod에 대응되는 result 값
value: 선택한 mod에 대응되는 value 값
```

mod로 5를 선택하면, result는 120 (ascii로 변환시 x), value는 682가 되어야 한다.

따라서 phase_3의 답안은 다음과 같다.

```
5 x 682
```

7. phase_4

disas phase_4를 통해 phase_4의 assembly code를 얻으면 다음과 같다.

```
Dump of assembler code for function phase_4:
   0x00000000004010dd <+0>:     sub    $0x18,%rsp
   0x00000000004010e1 <+4>:     lea    0xc(%rsp),%rcx
   0x00000000004010e6 <+9>:     lea    0x8(%rsp),%rdx
   0x00000000004010eb <+14>:    mov    $0x40288d,%esi
   0x00000000004010f0 <+19>:     mov     $0x0,%eax
   0x00000000004010f5 <+24>:     callq   0x400c30 <__isoc99_sscanf@plt>
   0x00000000004010fa <+29>:     cmp     $0x2,%eax
   0x00000000004010fd <+32>:     jne     0x40110b <phase_4+46>
   0x00000000004010ff <+34>:     mov     0xc(%rsp),%eax
   0x0000000000401103 <+38>:     sub     $0x2,%eax
   0x0000000000401106 <+41>:     cmp     $0x2,%eax
   0x0000000000401109 <+44>:     jbe     0x401110 <phase_4+51>
   0x000000000040110b <+46>:     callq   0x401614 <explode_bomb>
   0x0000000000401110 <+51>:     mov     0xc(%rsp),%esi
   0x0000000000401114 <+55>:     mov     $0x7,%edi
   0x0000000000401119 <+60>:     callq   0x4010a5 <func4>
   0x000000000040111e <+65>:     cmp     0x8(%rsp),%eax
   0x0000000000401122 <+69>:     je      0x401129 <phase_4+76>
   0x0000000000401124 <+71>:     callq   0x401614 <explode_bomb>
   0x0000000000401129 <+76>:     add     $0x18,%rsp
   0x000000000040112d <+80>:     retq
End of assembler dump.
```

phase_4의 assembly code를 C-style의 pseudo code로 변화시키면 다음과 같다.

```
int phase_4(int arg)
{
  int result;
  int key;
  int bias;

  if ( __isoc99_sscanf(a1, "%d %d", &key, &bias) != 2 || bias - 2 > 2 )
    explode_bomb();
  result = func4(7, bias);
  if ( result != key )
    explode_bomb();
  return result;
}
```

input으로 key와 bias를 받는다. (__isoc99_sscanf(a1, "%d %d", &key, &bias) != 2 조건에 따라 key와 bias 모두 받고, bias - 2 > 2 조건에 따라 bias가 4 이하인 값이어야 폭탄이 터지지 않는다.

그 다음 func4를 호출하여, fucn4의 반환값이 key와 같아야 폭탄이 터지지 않는다.

disas func4를 통해 func_4의 assembly code를 얻으면 다음과 같다.

```
Dump of assembler code for function func4:
   0x00000000004010a5 <+0>:     push   %r12
   0x00000000004010a7 <+2>:     push   %rbp
   0x00000000004010a8 <+3>:     push   %rbx
   0x00000000004010a9 <+4>:     mov    %edi,%ebx
   0x00000000004010ab <+6>:     test   %edi,%edi
   0x00000000004010ad <+8>:     jle    0x4010d3 <func4+46>
   0x00000000004010af <+10>:    mov    %esi,%ebp
   0x00000000004010b1 <+12>:    mov    %esi,%eax
   0x00000000004010b3 <+14>:    cmp    $0x1,%edi
   0x00000000004010b6 <+17>:    je     0x4010d8 <func4+51>
   0x00000000004010b8 <+19>:    lea    -0x1(%rdi),%edi
   0x00000000004010bb <+22>:    callq  0x4010a5 <func4>
   0x00000000004010c0 <+27>:    lea    (%rax,%rbp,1),%r12d
   0x00000000004010c4 <+31>:    lea    -0x2(%rbx),%edi
   0x00000000004010c7 <+34>:    mov    %ebp,%esi
   0x00000000004010c9 <+36>:    callq  0x4010a5 <func4>
   0x00000000004010ce <+41>:    add    %r12d,%eax
   0x00000000004010d1 <+44>:    jmp    0x4010d8 <func4+51>
   0x00000000004010d3 <+46>:    mov    $0x0,%eax
   0x00000000004010d8 <+51>:    pop    %rbx
   0x00000000004010d9 <+52>:    pop    %rbp
   0x00000000004010da <+53>:    pop    %r12
   0x00000000004010dc <+55>:    retq
End of assembler dump.
```

func_4의 assembly code를 C-style의 pseudo code로 변화시키면 다음과 같다.

```
int func4(int count, int value)
{
  int result;
  int left_term;

  if ( count <= 0 )
    return 0;
  result = value;
  if ( count != 1 )
  {
    left_term = func4((count - 1), value) + value;
    return left_term + func4((count - 2), value);
    }
    return result;
}
```

재귀함수의 형태를 가짐을 알 수 있다. bias 값이 4 이하이면 되므로, bias를 3으로 지정하고 func4(7, 3)을 계산하면 **99**를 얻을 수 있다.

key에 99가 들어가야 explode_bomb이 실행되지 않고 phase_4를 통과할 수 있다. 따라서 phase_4의 답은 다음과 같다.

```
99 3
```

8. phase_5

disas phase_5를 통해 phase_5의 assembly code를 추출하면 다음과 같다.

```
Dump of assembler code for function phase_5:
   0x000000000040112e <+0>:     sub    $0x18,%rsp
   0x0000000000401132 <+4>:     lea    0x8(%rsp),%rcx
   0x0000000000401137 <+9>:     lea    0xc(%rsp),%rdx
   0x000000000040113c <+14>:    mov    $0x40288d,%esi
   0x0000000000401141 <+19>:    mov    $0x0,%eax
   0x0000000000401146 <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x000000000040114b <+29>:    cmp    $0x1,%eax
   0x000000000040114e <+32>:    jg     0x401155 <phase_5+39>
   0x0000000000401150 <+34>:    callq  0x401614 <explode_bomb>
   0x0000000000401155 <+39>:    mov    0xc(%rsp),%eax
   0x0000000000401159 <+43>:    and    $0xf,%eax
   0x000000000040115c <+46>:    mov    %eax,0xc(%rsp)
   0x0000000000401160 <+50>:    cmp    $0xf,%eax
   0x0000000000401163 <+53>:    je     0x401191 <phase_5+99>
   0x0000000000401165 <+55>:    mov    $0x0,%ecx
   0x000000000040116a <+60>:    mov    $0x0,%edx
   0x000000000040116f <+65>:    add    $0x1,%edx
   0x0000000000401172 <+68>:    cltq
   0x0000000000401174 <+70>:    mov    0x402600(,%rax,4),%eax
   0x000000000040117b <+77>:    add    %eax,%ecx
   0x000000000040117d <+79>:    cmp    $0xf,%eax
   0x0000000000401180 <+82>:    jne    0x40116f <phase_5+65>
   0x0000000000401182 <+84>:    mov    %eax,0xc(%rsp)
   0x0000000000401186 <+88>:    cmp    $0xf,%edx
```

```
        0x0000000000401189 <+91>:     jne     0x401191 <phase_5+99>
        0x000000000040118b <+93>:     cmp     0x8(%rsp),%ecx
        0x000000000040118f <+97>:     je      0x401196 <phase_5+104>
        0x0000000000401191 <+99>:     callq   0x401614 <explode_bomb>
        0x0000000000401196 <+104>:    add     $0x18,%rsp
        0x000000000040119a <+108>:    retq
      End of assembler dump.
```

phase_5의 assembly code를 C-style의 pseudo code로 변화시키면 다음과 같다.

```
      int phase_5(int arg)
      {
        int result;
        int sum;
        int count;
        int input_sum;
        int index;

        if ( __isoc99_sscanf(arg, "%d %d", &index, &input_sum) <= 1 )
          explode_bomb();
        LODWORD(result) = index & 0xF;
        index = result;
        if (result == 15 )
          explode_bomb();
        sum = 0;
        count = 0;
        do
        {
          count++;
          index = array_3160[index];
          sum += index;
        }
        while ( index != 15 );
        if ( count != 15 || sum != input_sum )
          explode_bomb();
        return result;
}
```

array_3160(0x402600)의 값은 다음과 같다.

```
0xA, 0x2, 0xE, 0x7, 0x8, 0xC, 0xF, 0xB, 0x0, 0x4, 0x1, 0xD, 0x3, 0x9, 0x6, 0x5
```

phase5는 index와 input_sum을 입력받는다. (__isoc99_sscanf의 반환값이 1 이하이면

explode_bomb()이 실행되므로, index와 input_sum 모두 입력해야한다.

index에 0xF를 마스킹해주어 index가 15이하의 값만 가지도록 만들어준다.

index의 초기값이 15이면 explode_bomb이 실행되므로, index의 초기값은 15가 아니다.

array_3160을 총 16번 읽는데, index는 각 array_3160 값으로 설정되고, array_3160은 0~15의 값을 저장하므로 index에 array_3160의 index가 다시 들어감을 알 수 있다.

그리고 sum에 index들의 합을 저장한다.

do-while문 종료 후 count가 15, input_sum이 sum과 같으면 phase_5를 통과할 수 있다.

count가 15가 될 때 do-while문이 종료되는 index를 찾아보면 그 값은 5이고, index가 5일 때 sum은 115이므로 input_sum은 115이어야한다.

따라서 phase_5의 답은 다음과 같다.

```
5 115
```

9. phase_6

disas phase_6를 통해 phase_6의 assembly code를 얻으면 다음과 같다.

```
Dump of assembler code for function phase_6:
   0x000000000040119b <+0>:     push   %r13
   0x000000000040119d <+2>:     push   %r12
   0x000000000040119f <+4>:     push   %rbp
   0x00000000004011a0 <+5>:     push   %rbx
   0x00000000004011a1 <+6>:     sub    $0x58,%rsp
   0x00000000004011a5 <+10>:    lea    0x30(%rsp),%rsi
   0x00000000004011aa <+15>:    callq  0x40164a <read_six_numbers>
   0x00000000004011af <+20>:    lea    0x30(%rsp),%r13
   0x00000000004011b4 <+25>:    mov    $0x0,%r12d
   0x00000000004011ba <+31>:    mov    %r13,%rbp
   0x00000000004011bd <+34>:    mov    0x0(%r13),%eax
   0x00000000004011c1 <+38>:    sub    $0x1,%eax
   0x00000000004011c4 <+41>:    cmp    $0x5,%eax
   0x00000000004011c7 <+44>:    jbe    0x4011ce <phase_6+51>
   0x00000000004011c9 <+46>:    callq  0x401614 <explode_bomb>
   0x00000000004011ce <+51>:    add    $0x1,%r12d
   0x00000000004011d2 <+55>:    cmp    $0x6,%r12d
   0x00000000004011d6 <+59>:    jne    0x4011df <phase_6+68>
```

```
0x00000000004011d8 <+61>:    mov    $0x0,%esi
0x00000000004011dd <+66>:    jmp    0x401221 <phase_6+134>
0x00000000004011df <+68>:    mov    %r12d,%ebx
0x00000000004011e2 <+71>:    movslq %ebx,%rax
0x00000000004011e5 <+74>:    mov    0x30(%rsp,%rax,4),%eax
0x00000000004011e9 <+78>:    cmp    %eax,0x0(%rbp)
0x00000000004011ec <+81>:    jne    0x4011f3 <phase_6+88>
0x00000000004011ee <+83>:    callq  0x401614 <explode_bomb>
0x00000000004011f3 <+88>:    add    $0x1,%ebx
0x00000000004011f6 <+91>:    cmp    $0x5,%ebx
0x00000000004011f9 <+94>:    jle    0x4011e2 <phase_6+71>
0x00000000004011fb <+96>:    add    $0x4,%r13
0x00000000004011ff <+100>:   jmp    0x4011ba <phase_6+31>
0x0000000000401201 <+102>:   mov    0x8(%rdx),%rdx
0x0000000000401205 <+106>:   add    $0x1,%eax
0x0000000000401208 <+109>:   cmp    %ecx,%eax
0x000000000040120a <+111>:   jne    0x401201 <phase_6+102>
0x000000000040120c <+113>:   jmp    0x401213 <phase_6+120>
0x000000000040120e <+115>:   mov    $0x6042f0,%edx
0x0000000000401213 <+120>:   mov    %rdx,(%rsp,%rsi,2)
0x0000000000401217 <+124>:   add    $0x4,%rsi
0x000000000040121b <+128>:   cmp    $0x18,%rsi
0x000000000040121f <+132>:   je     0x401236 <phase_6+155>
0x0000000000401221 <+134>:   mov    0x30(%rsp,%rsi,1),%ecx
0x0000000000401225 <+138>:   cmp    $0x1,%ecx
0x0000000000401228 <+141>:   jle    0x40120e <phase_6+115>
0x000000000040122a <+143>:   mov    $0x1,%eax
0x000000000040122f <+148>:   mov    $0x6042f0,%edx
0x0000000000401234 <+153>:   jmp    0x401201 <phase_6+102>
0x0000000000401236 <+155>:   mov    (%rsp),%rbx
0x000000000040123a <+159>:   lea    0x8(%rsp),%rax
0x000000000040123f <+164>:   lea    0x30(%rsp),%rsi
0x0000000000401244 <+169>:   mov    %rbx,%rcx
0x0000000000401247 <+172>:   mov    (%rax),%rdx
0x000000000040124a <+175>:   mov    %rdx,0x8(%rcx)
0x000000000040124e <+179>:   add    $0x8,%rax
0x0000000000401252 <+183>:   cmp    %rsi,%rax
0x0000000000401255 <+186>:   je     0x40125c <phase_6+193>
0x0000000000401257 <+188>:   mov    %rdx,%rcx
```

```
            0x000000000040125a <+191>:    jmp      0x401247 <phase_6+172>
            0x000000000040125c <+193>:    movq     $0x0,0x8(%rdx)
            0x0000000000401264 <+201>:    mov      $0x5,%ebp
            0x0000000000401269 <+206>:    mov      0x8(%rbx),%rax
            0x000000000040126d <+210>:    mov      (%rax),%eax
            0x000000000040126f <+212>:    cmp      %eax,(%rbx)
            0x0000000000401271 <+214>:    jge      0x401278 <phase_6+221>
            0x0000000000401273 <+216>:    callq    0x401614 <explode_bomb>
            0x0000000000401278 <+221>:    mov      0x8(%rbx),%rbx
            0x000000000040127c <+225>:    sub      $0x1,%ebp
            0x000000000040127f <+228>:    jne      0x401269 <phase_6+206>
            0x0000000000401281 <+230>:    add      $0x58,%rsp
            0x0000000000401285 <+234>:    pop      %rbx
            0x0000000000401286 <+235>:    pop      %rbp
            0x0000000000401287 <+236>:    pop      %r12
            0x0000000000401289 <+238>:    pop      %r13
            0x000000000040128b <+240>:    retq
End of assembler dump.
```

phase_6의 assembly code를 C-style의 pseudo code로 변화시키면 다음과 같다.

```
int phase_6(int arg)
{
    int *array_ptr;
    int count;
    int i;
    int index;
    node* *node_ptr;
    int node_count
    int node_index
    node* head;
    node* node_link;
    node* j;
    node* temp;
    int node_index2;
    int result;
    node *node_array;
    int array[6];

    read_six_numbers(arg, array);
```

```
array_ptr = array;
count = 0;
while ( 1 )
{
  if (*array_ptr - 1 > 5 )
    explode_bomb();
  count++;
  if ( count == 6 )
    break;
  index = count;
  do
  {
    if ( *array_ptr == array[index] )
      explode_bomb();
    index++;
  }
  while ( index <= 5 );
  array_ptr++;
}
for ( i = 0; i != 6; i++ )
{
  node_index = array[i];
  if ( node_index <= 1 )
    node_ptr = &node1;
  else
  {
    node_count = 1;
    node_ptr = &node1;
    do
    {
      node_ptr = (_QWORD *)node_ptr[1];
      node_count++;
    }
    while ( node_count != node_index );
  }
  * (&node_array + i) = node_ptr;
}
head = node_array;
node_link = &(node_array[1]);
```

```
          for ( j = node_array; ; j = temp )
          {
             temp = node_link;
             j->next = node_link;
             node_link = node_link + 1;
             if ( node_link == (end_of_node_array) )
                break;
          }
          temp->next = 0;
          node_index2 = 5;
          do
          {
             result = (head->next)->value;
             if ( head->value < result )
                explode_bomb();
             head = head->next;
             node_index2--;
          }
          while (node_index2 != 0);
          return result;
}
```

node1은 linked list의 node의 구조를 가지고 있다. 위의 pseudo code는 node의 구조가 다음과 같은 구조라고 가정하고 구성하였다.

```
struct node
{
   int value;         // <+0>
   node* next;        // <+8>
}
```

node는 총 node1, node2, node3, node4, node5, node6가 있다. 각각 value는 다음과 같다.

```
node1.value = 0x0CA
node2.value = 0x3C0
node3.value = 0x05B
node4.value = 0x209
node5.value = 0x132
```

```
node6.value = 0x1E7
```

pseudo code를 해석하면 다음과 같다.

1. array에 6자리 정수를 담는다.
2. array에 있는 정수 중 모든 정수가 서로 다르다면 explode_bomb을 실행시키지 않고, 그렇지 않다면 실행한다.
3. node_array[i]에 각각 node_{array[i]}의 주소를 넣어준다.
4. node_array의 순서에 따라 node들의 next들을 다음 node로 연결해준다. (곧, linked list를 구성한다.)
5. linked list를 처음부터 끝까지 순서대로 탐색하는데, 다음 노드의 value가 현재 노드의 value보다 크면 explode_bomb을 실행시킨다.
6. 5번에서 explode_bomb이 실행되지 않았다면 phase_6이 해결된다.

node의 value가 내림차순이 되도록 node의 index를 지정해주면 phase_6를 풀 수 있다. 따라서 답은 다음과 같다.

```
2 4 6 5 1 3
```

10. secret_phase

먼저, secret_phase를 실행하기 위한 진입 조건을 알아야한다.

phase_defused의 pseudo code를 구성하면 다음과 같다.

```
int phase_defused()
{
  int result;
  char b;
  char a;
  char key[88];

  result = send_msg(1);
  if ( num_input_strings == 6 )
  {
    if ( __isoc99_sscanf(&6048B0, "%d %d %s", &a, &b, key) == 3
      && ! strings_not_equal(key, "DrEvil") )
    {
      puts("Curses, you've found the secret phase!");
      puts("But finding it and solving it are quite different...");
      secret_phase();
```

```
            }
            puts("Congratulations! You've defused the bomb!");
            return puts("Your instructor has been notified and will verify your
        solution.");
          }
          return result;
}
```

num_input_strings가 6일 때, &6048B0을 “%d %d %s” format으로 파싱했을 때 a, b, key 세개의 변수를 얻을 수 있어야하고, key 값이 “DrEvil”이면 secret_phase로 진입할 수 있다.

num_input_strings는 read_line 실행시 계속 1씩 증가하므로, phase_6 종료 후 secret_phase가 실행된다.

&unk_6048B0에 저장된 값을 알기 위해 gdb로 값을 찍어보면, phase 4의 답이 들어있음을 알 수 있다. 따라서 phase_4에서 답으로 99 3 대신 99 3 DrEvil을 입력하면 phase_6 종료후 secret_phase를 실행할 수 있다.

disas secret_phase로 assembly code를 얻으면 다음과 같다.

```
Dump of assembler code for function secret_phase:
   0x00000000004012ca <+0>:     push   %rbx
   0x00000000004012cb <+1>:     callq  0x40168c <read_line>
   0x00000000004012d0 <+6>:     mov    $0xa,%edx
   0x00000000004012d5 <+11>:    mov    $0x0,%esi
   0x00000000004012da <+16>:    mov    %rax,%rdi
   0x00000000004012dd <+19>:    callq  0x400c00 <strtol@plt>
   0x00000000004012e2 <+24>:    mov    %rax,%rbx
   0x00000000004012e5 <+27>:    lea    -0x1(%rax),%eax
   0x00000000004012e8 <+30>:    cmp    $0x3e8,%eax
   0x00000000004012ed <+35>:    jbe    0x4012f4 <secret_phase+42>
   0x00000000004012ef <+37>:    callq  0x401614 <explode_bomb>
   0x00000000004012f4 <+42>:    mov    %ebx,%esi
   0x00000000004012f6 <+44>:    mov    $0x604110,%edi
   0x00000000004012fb <+49>:    callq  0x40128c <fun7>
   0x0000000000401300 <+54>:    cmp    $0x7,%eax
   0x0000000000401303 <+57>:    je     0x40130a <secret_phase+64>
   0x0000000000401305 <+59>:    callq  0x401614 <explode_bomb>
   0x000000000040130a <+64>:    mov    $0x402590,%edi
   0x000000000040130f <+69>:    callq  0x400b40 <puts@plt>
```

```
            0x0000000000401314 <+74>:     callq   0x4017b2 <phase_defused>
            0x0000000000401319 <+79>:     pop      %rbx
            0x000000000040131a <+80>:     retq
End of assembler dump.
```

secret_phase의 pseudo code는 다음과 같다.

```
         int secret_phase()
         {
           const char *line; // rdi
           int key; // ebx

           line = read_line();
           key = strtol(line, 0, 10);
           if ( key - 1 > 0x3E8 )
             explode_bomb(line, 0LL);
           if (fun7(&n1, key) != 7 )
             explode_bomb();
           puts("Wow! You've defused the secret stage!");
           return phase_defused();
}
```

key값이 1001이하이고, fun7(&n1, key) 값이 7이어야 secret_phase를 해결할 수 있다.

disas fun7을 통해 fun7의 assembly code를 얻으면 다음과 같다.

```
         Dump of assembler code for function fun7:
            0x000000000040128c <+0>:      sub      $0x8,%rsp
            0x0000000000401290 <+4>:      test     %rdi,%rdi
            0x0000000000401293 <+7>:      je       0x4012c0 <fun7+52>
            0x0000000000401295 <+9>:      mov      (%rdi),%edx
            0x0000000000401297 <+11>:     cmp      %esi,%edx
            0x0000000000401299 <+13>:     jle      0x4012a8 <fun7+28>
            0x000000000040129b <+15>:     mov      0x8(%rdi),%rdi
            0x000000000040129f <+19>:     callq    0x40128c <fun7>
            0x00000000004012a4 <+24>:     add      %eax,%eax
            0x00000000004012a6 <+26>:     jmp      0x4012c5 <fun7+57>
            0x00000000004012a8 <+28>:     mov      $0x0,%eax
            0x00000000004012ad <+33>:     cmp      %esi,%edx
            0x00000000004012af <+35>:     je       0x4012c5 <fun7+57>
            0x00000000004012b1 <+37>:     mov      0x10(%rdi),%rdi
            0x00000000004012b5 <+41>:     callq    0x40128c <fun7>
            0x00000000004012ba <+46>:    lea     0x1(%rax,%rax,1),%eax
            0x00000000004012be <+50>:    jmp      0x4012c5 <fun7+57>
            0x00000000004012c0 <+52>:    mov      $0xffffffff,%eax
            0x00000000004012c5 <+57>:    add      $0x8,%rsp
            0x00000000004012c9 <+61>:    retq
End of assembler dump.
```

fun7의 pseudo code는 다음과 같다.

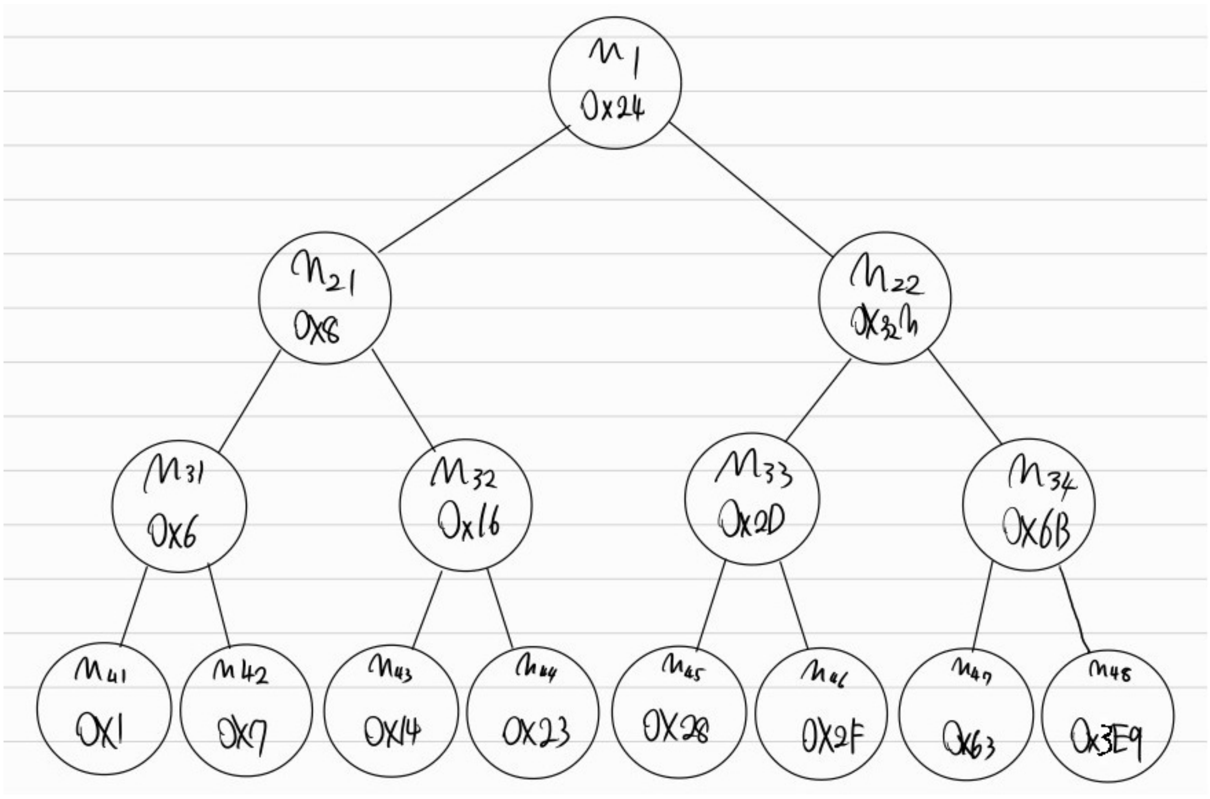
```
         int fun7(tree_node* a1, int key)
         {
           int result; // rax

           if ( a1 == NULL)
             return 0xFFFFFFFFLL;
           if (a1->value > key )
             return 2 * fun7( a1->left, key);
           result = 0;
           if (a1->value != key )
             return 2 * fun7(a1->right, key) + 1;
           return result;
}
```

tree_node는 다음과 같은 구조이다.

```
struct tree_node
{
  int value;             // <+0>
  tree_node* left;       // <+8>
  tree_node* right       // <+16>
}
```

n1의 데이터를 읽어보면, tree_node들의 연결이 binary tree 구조를 가짐을 알 수 있다.

n1을 따라서 데이터를 파싱해 tree를 구축하면 다음과 같다.



fun7(&n1, key) 값이 7이 되려면, 재귀함수의 반환값은 다음과 같은 구조를 가져야한다.

```
7 = 2 ( 2 (2 (0) + 1) + 1) + 1
```

따라서, n1 -> n22 -> n34 -> n48의 순서대로 fun7가 호출되어야한다. 그렇게 호출되기 위해선 key 값은 다음 조건을 만족해야한다.

```
key > 0x24
key > 0x32
key > 0x6B
key = 3E9h
```

이 모든 조건을 만족하는 key는 1001이다. 따라서 secret phase의 답은 다음과 같다.

```
1001
```

11. 결과

최종 답안은 다음과 같다.

```
Phase 1: For NASA, space is still a high priority.
Phase 2: 0 1 1 2 3 5
Phase 3: 5 x 682
Phase 4: 99 3 DrEvil
Phase 5: 5 115
Phase 6: 2 4 6 5 1 3
Secret Phase: 1001
```

실제로 폭탄이 잘 해체되었고, 다음과 같은 메시지를 얻었다.

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