

리버싱 교육 3회차

201702075 조수환





INDEX



001/

phase 3

002/

phase 4

003/

QnA

리버싱 교육 3회차

phase 3

- 구문 분석
- 실습





```
(gdb) disas phase_3
Dump of assembler code for function phase_3:
0x000000000400f43 <+0>:      sub    rsp,0x18
0x000000000400f47 <+4>:      lea    rcx,[rsp+0xc]
0x000000000400f4c <+9>:      lea    rdx,[rsp+0x8]
0x000000000400f51 <+14>:     mov    esi,0x4025cf
0x000000000400f56 <+19>:     mov    eax,0x0
0x000000000400f5b <+24>:     call  0x400bf0 <_isoc99_sscanf@plt>
0x000000000400f60 <+29>:     cmp    eax,0x1
0x000000000400f63 <+32>:     jg     0x400f6a <phase_3+39>
0x000000000400f65 <+34>:     call  0x40143a <explode_bomb>
0x000000000400f6a <+39>:     cmp    DWORD PTR [rsp+0x8],0x7
0x000000000400f6f <+44>:     ja     0x400fad <phase_3+106>
0x000000000400f71 <+46>:     mov    eax,DWORD PTR [rsp+0x8]
0x000000000400f75 <+50>:     jmp    QWORD PTR [rax*8+0x402470]
0x000000000400f7c <+57>:     mov    eax,0xcf
0x000000000400f81 <+62>:     jmp    0x400fbe <phase_3+123>
0x000000000400f83 <+64>:     mov    eax,0x2c3
0x000000000400f88 <+69>:     jmp    0x400fbe <phase_3+123>
0x000000000400f8a <+71>:     mov    eax,0x100
0x000000000400f8f <+76>:     jmp    0x400fbe <phase_3+123>
0x000000000400f91 <+78>:     mov    eax,0x185
0x000000000400f96 <+83>:     jmp    0x400fbe <phase_3+123>
0x000000000400f98 <+85>:     mov    eax,0xce
0x000000000400f9d <+90>:     jmp    0x400fbe <phase_3+123>
0x000000000400f9f <+92>:     mov    eax,0x2aa
0x000000000400fa4 <+97>:     jmp    0x400fbe <phase_3+123>
0x000000000400fa6 <+99>:     mov    eax,0x147
0x000000000400fab <+104>:    jmp    0x400fbe <phase_3+123>
0x000000000400fad <+106>:    call  0x40143a <explode_bomb>
0x000000000400fb2 <+111>:    mov    eax,0x0
0x000000000400fb7 <+116>:    jmp    0x400fbe <phase_3+123>
0x000000000400fb9 <+118>:    mov    eax,0x137
0x000000000400fbe <+123>:    cmp    eax,DWORD PTR [rsp+0xc]
0x000000000400fc2 <+127>:    je     0x400fc9 <phase_3+134>
0x000000000400fc4 <+129>:    call  0x40143a <explode_bomb>
0x000000000400fc9 <+134>:    add    rsp,0x18
0x000000000400fcd <+138>:    ret
```

~~압도적인 바주알~~

굉장히 양이 많아 보이지만

사실 다 거품입니다.

폭탄을 피할 조건을 조목 조목 따져보면

꼭 분석해야하는 구문은 많지 않아요.



```
0x000000000400f56 <+19>:  mov    eax,0x0
0x000000000400f5b <+24>:  call   0x400bf0 <_isoc99_sscanf@plt>
0x000000000400f60 <+29>:  cmp    eax,0x1
0x000000000400f63 <+32>:  jg     0x400f6a <phase_3+39>
0x000000000400f65 <+34>:  call   0x40143a <explode_bomb>
```

폭탄을 피하기 위한 첫번째 조건, eax가 0x1보다 커야 한다.

eax는 주로 함수의 리턴값을 담는 레지스터로

함수를 call한 직후에 나오는 eax 레지스터는 해당 함수의 리턴 값을 의미한다.

참고로 위의 함수는 scanf()를 말하는 겁니다.

C에서 보편적으로 사용하는 입력함수, scanf()에도 return값이 존재하나?



scanf()의 리턴 값은 형식에 맞는 입력의 개수이다.

```
int main(void) {  
    int a, b, c;  
    int d = scanf("%d %d %d", &a, &b, &c);  
    printf("%d\n", d);  
  
    return 0;  
}
```

```
sh2358@edu-argos:~/2021-argos-bomblab-edu$ ./test  
1234 543 234  
3
```

즉 scanf()의 리턴값이 1보다 커야하므로
입력을 두개 이상 받는 것이 첫번째 조건이다.



```
0x0000000000400f6a <+39>:    cmp     DWORD PTR [rsp+0x8],0x7
0x0000000000400f6f <+44>:    ja      0x400fad <phase_3+106>
0x0000000000400fad <+106>:    call    0x40143a <explode_bomb>
```

폭탄을 피하기 위한 두번째 조건

rsp+0x8에 저장된 값이 0x7보다 크다면 폭탄이 터진다.

이런 구문이 나오면 일단 값을 입력 해본다음에

명령어를 통해서 레지스터를 까보는게 좋습니다.

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

Breakpoint 1, 0x0000000000400f43 in phase_3 ()
```



```
=> 0x0000000000400f6a <+39>:    cmp     DWORD PTR [rsp+0x8],0x7
      0x0000000000400f6f <+44>:    ja      0x400fad <phase_3+106>
```

```
(gdb) x/wx $rsp+0x8
0x7fffffffef398: 0x00000001
```

rsp+0x8에는 1이 저장되어 있다.

우리가 입력한 두개의 수 중 첫번째가 저장 되어 있는 것이다.

한마디로 폭탄을 피하기 위한 두번째 조건은

첫번째 입력한 수가 7보다 크면 안된다는 것.

이제 마지막 조건을 보자.



```
0x000000000400fbe <+123>:  cmp    eax,DWORD PTR [rsp+0xc]  
0x000000000400fc2 <+127>:  je      0x400fc9 <phase_3+134>  
0x000000000400fc4 <+129>:  call   0x40143a <explode_bomb>
```

eax와 rsp+0xc 안의 값이 같아야 폭탄을 피할 수 있다.

```
(gdb) x/wx $rsp+0xc  
0x7fffffffef39c: 0x00000002
```

rsp+0xc에 저장된 값은 2, 우리가 두번째로 입력한 숫자이다.

즉, 두번째로 입력된 수가 비교 시점의 eax와 같으면

phase 3을 통과할 수 있는 것이다.



```
0x0000000000400f71 <+46>: mov     eax,DWORD PTR [rsp+0x8]
0x0000000000400f75 <+50>: jmp     QWORD PTR [rax*8+0x402470]
0x0000000000400f7c <+57>: mov     eax,0xcf
0x0000000000400f81 <+62>: jmp     0x400fbe <phase_3+123>
0x0000000000400f83 <+64>: mov     eax,0x2c3
0x0000000000400f88 <+69>: jmp     0x400fbe <phase_3+123>
0x0000000000400f8a <+71>: mov     eax,0x100
0x0000000000400f8f <+76>: jmp     0x400fbe <phase_3+123>
0x0000000000400f91 <+78>: mov     eax,0x185
0x0000000000400f96 <+83>: jmp     0x400fbe <phase_3+123>
0x0000000000400f98 <+85>: mov     eax,0xce
0x0000000000400f9d <+90>: jmp     0x400fbe <phase_3+123>
0x0000000000400f9f <+92>: mov     eax,0x2aa
0x0000000000400fa4 <+97>: jmp     0x400fbe <phase_3+123>
0x0000000000400fa6 <+99>: mov     eax,0x147
0x0000000000400fab <+104>: jmp     0x400fbe <phase_3+123>
```

하지만 비교 명령어 이전에 eax의 값을 변경하는 구문이 엄청 많
아 보이지만 사실 eax의 값은 한번만 변경된다.

mov 명령 이후에 바로 비교 구문으로 무조건 jump하기 때문이다.



그렇다면 저 많은 mov명령어 중 무엇을 실행하며,
eax에는 어떤 값이 들어가 있을까

```
0x0000000000400f71 <+46>:  mov    eax,DWORD PTR [rsp+0x8]  
0x0000000000400f75 <+50>:  jmp    QWORD PTR [rax*8+0x402470]
```

핵심은 위의 구문에 있다. phase_3+50의 jump명령어에 따라서
실행하는 mov 명령이 달라지고, eax의 값이 결정된다. 즉

1. 두 수를 입력받아야 한다.
2. 첫번째 숫자는 7보다 크면 안된다.
3. 두번째로 입력받은 수와 어떤 값이 같아야 통과한다.



이제 phase 3을 풀어봅시다!

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

```
Breakpoint 1, 0x000000000400f43 in phase_3 ()
```

```
(gdb) c  
Continuing.  
Halfway there!
```

힌트를 드리자면

```
0x000000000400f71 <+46>:  mov    eax,DWORD PTR [rsp+0x8]  
0x000000000400f75 <+50>:  jmp     QWORD PTR [rax*8+0x402470]
```

위의 구문을 자세히 분석하시면 알겠지만
입력받은 첫번째 수에 따라서 답이 달라집니다.

리버싱 교육 3회차

phase 4

- 구문 분석
- 실습





```
(gdb) disas phase_4
Dump of assembler code for function phase_4:
0x000000000040100c <+0>:      sub     rsp,0x18
0x0000000000401010 <+4>:      lea     rcx,[rsp+0xc]
0x0000000000401015 <+9>:      lea     rdx,[rsp+0x8]
0x000000000040101a <+14>:     mov     esi,0x4025cf
0x000000000040101f <+19>:     mov     eax,0x0
0x0000000000401024 <+24>:     call   0x400bf0 <__isoc99_sscanf@plt>
0x0000000000401029 <+29>:     cmp     eax,0x2
0x000000000040102c <+32>:     jne     0x401035 <phase_4+41>
0x000000000040102e <+34>:     cmp     DWORD PTR [rsp+0x8],0xe
0x0000000000401033 <+39>:     jbe     0x40103a <phase_4+46>
0x0000000000401035 <+41>:     call   0x40143a <explode_bomb>
0x000000000040103a <+46>:     mov     edx,0xe
0x000000000040103f <+51>:     mov     esi,0x0
0x0000000000401044 <+56>:     mov     edi,DWORD PTR [rsp+0x8]
0x0000000000401048 <+60>:     call   0x400fce <func4>
0x000000000040104d <+65>:     test    eax,eax
0x000000000040104f <+67>:     jne     0x401058 <phase_4+76>
0x0000000000401051 <+69>:     cmp     DWORD PTR [rsp+0xc],0x0
0x0000000000401056 <+74>:     je      0x40105d <phase_4+81>
0x0000000000401058 <+76>:     call   0x40143a <explode_bomb>
0x000000000040105d <+81>:     add     rsp,0x18
0x0000000000401061 <+85>:     ret
```

phase 4의 전체 구조

전체적인 길이가 짧고

조건이 복잡하지 않지만

func4를 까봐야 한다..



```
0x0000000000401024 <+24>:  call    0x400bf0 <__isoc99_sscanf@plt>
0x0000000000401029 <+29>:  cmp     eax,0x2
0x000000000040102c <+32>:  jne     0x401035 <phase_4+41>

0x0000000000401035 <+41>:  call    0x40143a <explode_bomb>
```

어디서 본듯한 조건, scanf()의 리턴 값이 0x2와 같아야 폭탄을 피할 수 있다.

즉 phase 4에서의 입력은 두개다.

```
0x000000000040102e <+34>:  cmp     DWORD PTR [rsp+0x8],0xe
0x0000000000401033 <+39>:  jbe     0x40103a <phase_4+46>
0x0000000000401035 <+41>:  call    0x40143a <explode_bomb>
```

아무 값이나 입력해서 rsp+0x8의 값을 까보면 알겠지만 이는 첫번째 입력이다.

즉, 첫번째로 입력받은 수는 0xe(14)보다 같거나 작아야 한다.



그러므로 폭탄을 피하려면 func4의 리턴값이 0이어야 한다.

이름만 봐서는 뭐 하는 함수인지 알 수가 없으니 까보도록 하자

```
0x0000000000400fce <+0>:  sub    rsp,0x8
0x0000000000400fd2 <+4>:  mov    eax,edx
0x0000000000400fd4 <+6>:  sub    eax,esi
0x0000000000400fd6 <+8>:  mov    ecx,eax
0x0000000000400fd8 <+10>: shr    ecx,0x1f
0x000000000040fdb <+13>:  add    eax,ecx
0x000000000040fdd <+15>:  sar    eax,1
0x000000000040fdf <+17>:  lea    ecx,[rax+rsi*1]
0x000000000040fe2 <+20>:  cmp    ecx,edi
0x000000000040fe4 <+22>:  jle    0x400ff2 <func4+36>
0x000000000040fe6 <+24>:  lea    edx,[rcx-0x1]
0x000000000040fe9 <+27>:  call   0x400fce <func4>
0x000000000040fee <+32>:  add    eax,eax
0x000000000040ff0 <+34>:  jmp    0x401007 <func4+57>
0x000000000040ff2 <+36>:  mov    eax,0x0
0x000000000040ff7 <+41>:  cmp    ecx,edi
0x000000000040ff9 <+43>:  jge    0x401007 <func4+57>
0x000000000040ffb <+45>:  lea    esi,[rcx+0x1]
0x000000000040ffe <+48>:  call   0x400fce <func4>
0x0000000000401003 <+53>:  lea    eax,[rax+rax*1+0x1]
0x0000000000401007 <+57>:  add    rsp,0x8
0x000000000040100b <+61>:  ret
```

= 재귀 함수



재귀 함수 func4의 리턴값은 3가지로 나뉩니다.

```

0x0000000000400fe2 <+20>:  cmp    ecx,edi
0x0000000000400fe4 <+22>:  jle     0x400ff2 <func4+36>
0x0000000000400fe6 <+24>:  lea     edx,[rcx-0x1]
0x0000000000400fe9 <+27>:  call   0x400fce <func4>
0x0000000000400fee <+32>:  add     eax,eax
0x0000000000400ff0 <+34>:  jmp     0x401007 <func4+57>
0x0000000000400ff2 <+36>:  mov     eax,0x0
0x0000000000400ff7 <+41>:  cmp     ecx,edi
0x0000000000400ff9 <+43>:  jge     0x401007 <func4+57>
0x0000000000400ffb <+45>:  lea     esi,[rcx+0x1]
0x0000000000400ffe <+48>:  call   0x400fce <func4>
0x0000000000401003 <+53>:  lea     eax,[rax+rax*1+0x1]
  
```

Exit case

함수의 구조 상 exit case는 무조건 0이 되고,

eax가 0이라면 add eax,eax 또한 0이므로

'lea eax,[rax+rax*1+0x1]' 만 실행하지 않으면 되는거죠.



```
0x0000000000400fe2 <+20>: cmp    ecx,edi
0x0000000000400fe4 <+22>: jle    0x400ff2 <func4+36>
0x0000000000400fe6 <+24>: lea    edx,[rcx-0x1]
0x0000000000400fe9 <+27>: call   0x400fce <func4>
0x0000000000400fee <+32>: add    eax,eax
0x0000000000400ff0 <+34>: jmp    0x401007 <func4+57>
0x0000000000400ff2 <+36>: mov    eax,0x0
0x0000000000400ff7 <+41>: cmp    ecx,edi
0x0000000000400ff9 <+43>: jge    0x401007 <func4+57>
0x0000000000400ffb <+45>: lea    esi,[rcx+0x1]
0x0000000000400ffe <+48>: call   0x400fce <func4>
0x0000000000401003 <+53>: lea    eax,[rax+rax*1+0x1]
```

이를 통과하려면 jge <func4+57>을 실행해야 합니다.

ecx가 edi보다 같거나 커야 한다는 뜻이죠,

그런데 해당 비교 구문에 진입하려면 ecx가 edi보다 작거나 같아야 합니다.

즉, ecx와 edi가 같아야 func4의 리턴값이 0이 된다는 뜻이죠.



```
0x0000000000400fd2 <+4>:    mov     eax,edx
0x0000000000400fd4 <+6>:    sub     eax,esi
0x0000000000400fd6 <+8>:    mov     ecx,eax
0x0000000000400fd8 <+10>:   shr     ecx,0x1f
0x0000000000400fdb <+13>:   add     eax,ecx
0x0000000000400fdd <+15>:   sar     eax,1
0x0000000000400fdf <+17>:   lea     ecx,[rax+rsi*1]
0x0000000000400fe2 <+20>:   cmp     ecx,edi
```

```
0x000000000040103a <+46>:    mov     edx,0xe
0x000000000040103f <+51>:    mov     esi,0x0
```

위의 구문들은 ecx의 값을 결정하는 구문들 입니다.

여러 복잡한 연산들이 적용되어 있지만

사실 아무것도 볼 필요가 없습니다.

연산 과정에 들어가는 레지스터들은 전부 상수를 담고 있기 때문에

우리가 입력한 값과 무관하게 항상 고정된 결과값이 나오게 됩니다.



이제 phase 4를 풀어봅시다!

```
Halfway there!  
[REDACTED]  
Breakpoint 1, 0x000000000040100c in phase_4 ()
```

```
(gdb) c  
Continuing.  
So you got that one. Try this one.  
[REDACTED]
```

특정 레지스터의 값을 알아내는게 핵심입니다.

참고로 레지스터의 값을 보고 싶으면 p/[출력형식] \$[레지스터]

Ex) p/x \$edx

두번째 입력은 조건이 없는 수준이라서.. 보시면 그냥 압니다.

Q & A

Thank You for Listening

