컴퓨터 SW 시스템 개론
LAB 3 Report

20190650 김관호

1. 배운 것

86x64 구조에서 16개의 레지스터에 데이터를 저장하여 메모리, 하드디스크보다 빠른 속도로 데이터를 읽어들일 수 있다. 각 16개의 레지스터는 고유한 역할을 하며, 레지스터를 이용해 조건문, 반복문 또한 구현할 수 있다. 연산이 일어난 후에 자동으로 True, False로 값이 설정되는 ZF, SF, OF, CF의 네 가지 Flags를 이용해 분기를 나눌 수 있다. add, sub와 같은 값이 저장되는 연산 이외에 cmp, test와 같이 destination의 값을 바꾸지 않고 conditional flag를 설정하는 연산도 있다. 이렇게 conditional flags를 설정하는 연산과 더불어 jump instruction을 이용해 조건에 따라 다른 처리를 할 수 있게 해준다. 이렇게 if문을 구현할 수 있고, while문 또한 조건을 검사함으로써 작동되므로 조건이 맞다면 loop 시작 지점으로 jump함으로써 구현할 수 있게 된다. 또한, 분기처리로 switch문도 구현할 수 있다. jump table을 이용하면 switch문의 분기가 아주 많아져도 단 한 번의 연산으로 그에 맞는 위치로 이동할 수 있게 되어 효율적으로 처리할 수 있다.

함수의 호출과 반환, 스택, caller&callee saved register를 제어하는 프로시저(procedure)을 배웠다. 프로시저는 대표적으로 passing control, passing data, memory management를 수행한다. control을 passing함으로써 함수 속에서 다른 함수를 호출할 수 있게 되고 이 때 data를 passing함으로써 필요한 argument를 넘기거나, %rax에 반환 값을 저장할 수 있다. 만약, 함수 호출 전후로 바뀌면 안되는 변수가 있다면 해당 변수는 컴파일러가 callee saved register인 %rbp, %rbx 등에 저장하여 바뀌지 않게 컴파일 해준다. callee saved register의 뜻은 “해당 레지스터의 값을 바꾸고 싶다면 직접적으로 바꾸는게 아니라 callee 쪽에서 따로 다른 레지스터나 메모리에 그 값을 **저장**하고 사용해라”고 생각하면 되겠다. caller saved도 같은 맥락으로 이번엔 caller가 레지스터 값을 저장하고 사용해야 한다.

control과 data를 주고받는 일은 스택(Stack)에서 일어나는데, 스택은 직관과 달리 높은 주소에서 낮은 주소로 자란다. 즉, 넣으면 넣을수록 스택의 top을 가리키는 %rsp의 값은 작아지게 된다. 그래서 스택은 함수를 호출하는데 7번째 이상부터 필요한 인자, 리턴 주소, 지역 변수, caller&callee saved register을 포함한다. 함수가 호출될 때마다 스택은 낮은 주소로 **자라며**, 함수가 리턴되면 스택은 함수를 호출할 때 넣어놓은 리턴 주소로 복귀하면서 다시 높은 주소로 **줄어들게** 된다.

마지막으로 Single Type을 묶어놓은 Array와 Struct, Union 같은 데이터를 배웠다. 배열의 경우 배열의 크기만큼 메모리에 **연속적으로** 저장되는 것이 보장되어 있다. 그래서 배열의 가장 앞 주소만 알면 원하는 위치로 바로 이동할 수 있다. 2차원 배열같은 경우에도 정상적인

경우라면 순차적으로 메모리가 할당되어 있을 테지만, 강의자료 7장 19p와 같이 일부러 순서를 섞은 경우라면 원하는 인덱스로 접근하기 위해 **메모리 참조를 두 번**해야할 수도 있다. Struct는 여러 가지 자료형을 한데 모은 구조로, Struct 안에 **변수를 선언한 순서대로** 메모리가 할당된다. 이것이 중요한 이유는 Struct 내에서 가장 큰 변수의 크기의 배수로 Struct의 크기가 결정되고, 각 변수의 시작 주소는 해당 변수의 크기의 배수여야 한다. 그래서 변수 선언에 따라 같은 구성요소 Struct라도 그 크기가 달라질 수 있다. Union은 Struct와 동일한 구조로 선언할 수 있지만 Struct와 다르게 가장 **큰 변수의 크기가 곧 Union 전체의 크기**가 된다. 그리고 해당 공간을 Union 변수들이 **공유**한다. 그래서 **한 번에 한 변수만 표현**할 수 있다. 왼쪽, 오른쪽 자식을 하나만 가질 수 있는 이진 트리의 경우 Left, Right를 다 가지게 하는 Struct보다는 Union으로 구조를 잡는 것이 더 나을 수 있겠다.
마지막으로 floating point 계산을 수월하게 해주는 레지스터가 있고, 해당 레지스터를 포함한 floating point의 연산 명령을 배웠다. 병렬 연산을 통해 비약적으로 floating point 연산 시간을 단축시킬 수 있었다.

2. 문제 해결 과정

Setup

bomb31을 로컬에 풀고 chmod +x bomb로 실행 가능하게 만들어주고 **gdb bomb**로 디버깅을 시작한다. **disas** 커맨드로 각 phase와 원하는 함수의 assembly코드를 보며 reverse engineering을 통해 input으로 어떤 값이 들어가야 하는지를 파악했다.
문제에 들어가기 전에 main의 구조를 보자.
(gdb) disas main

```
0x0000000000400e3b <+126>:   mov    $0x402438,%edi
0x0000000000400e40 <+131>:   call   0x400b40 <puts@plt>
0x0000000000400e45 <+136>:   call   0x4015ec <read_line>
0x0000000000400e4a <+141>:   mov    %rax,%rdi
0x0000000000400e4d <+144>:   call   0x400ef0 <phase_1>
0x0000000000400e52 <+149>:   call   0x401712 <phase_defused>
0x0000000000400e57 <+154>:   mov    $0x402468,%edi
```

phase_1으로 들어가는 시작점이다. 코드를 보면 phase_1을 호출하기 전에 read_line을 호출하고 그 결과를 %rdi에 넣어주는 모습을 볼 수 있다. 즉, 입력값을 받고 그것을 인자로 phase_1에 넣어준다. 비단 phase_1뿐만 아니라 모든 단계에서 사용자 입력을 받고 이를 첫 번째 인자(%rdi)로 넘겨줌을 알 수 있다. 또한, 각 phase 속에서 스택 메모리를 할당하고 %rdx, %rcx 등에 나눠준다. 여기에 %rdi를 나눠줌으로써 **사용자 입력값을 지역변수로 이용함을 알 수 있다.**

Phase_1

요약

%rsi와 같은 문자열을 입력해야 한다. %rsi가 저장되어 있는 주소는 바로 알 수 있었고 **x/s 주소**로 정답을 알 수 있었다.

정답

And they have no disregard for human life.

상세 과정

binary로 되어있던 phase 1을 gdb로 disassemble한 결과는 아래와 같다.

(gdb) disas phase_1

```
Dump of assembler code for function phase_1:
   0x0000000000400ef0 <+0>:     sub    $0x8,%rsp
   0x0000000000400ef4 <+4>:     mov    $0x4024c0,%esi
   0x0000000000400ef9 <+9>:     call   0x40130e <strings_not_equal>
   0x0000000000400efe <+14>:    test   %eax,%eax
   0x0000000000400f00 <+16>:    je     0x400f07 <phase_1+23>
   0x0000000000400f02 <+18>:    call   0x401574 <explode_bomb>
   0x0000000000400f07 <+23>:    add    $0x8,%rsp
   0x0000000000400f0b <+27>:    ret
End of assembler dump.
```

$0x4024c0을 두 번째 인자(%esi)로 넣고 strings_not_equal을 호출한다.

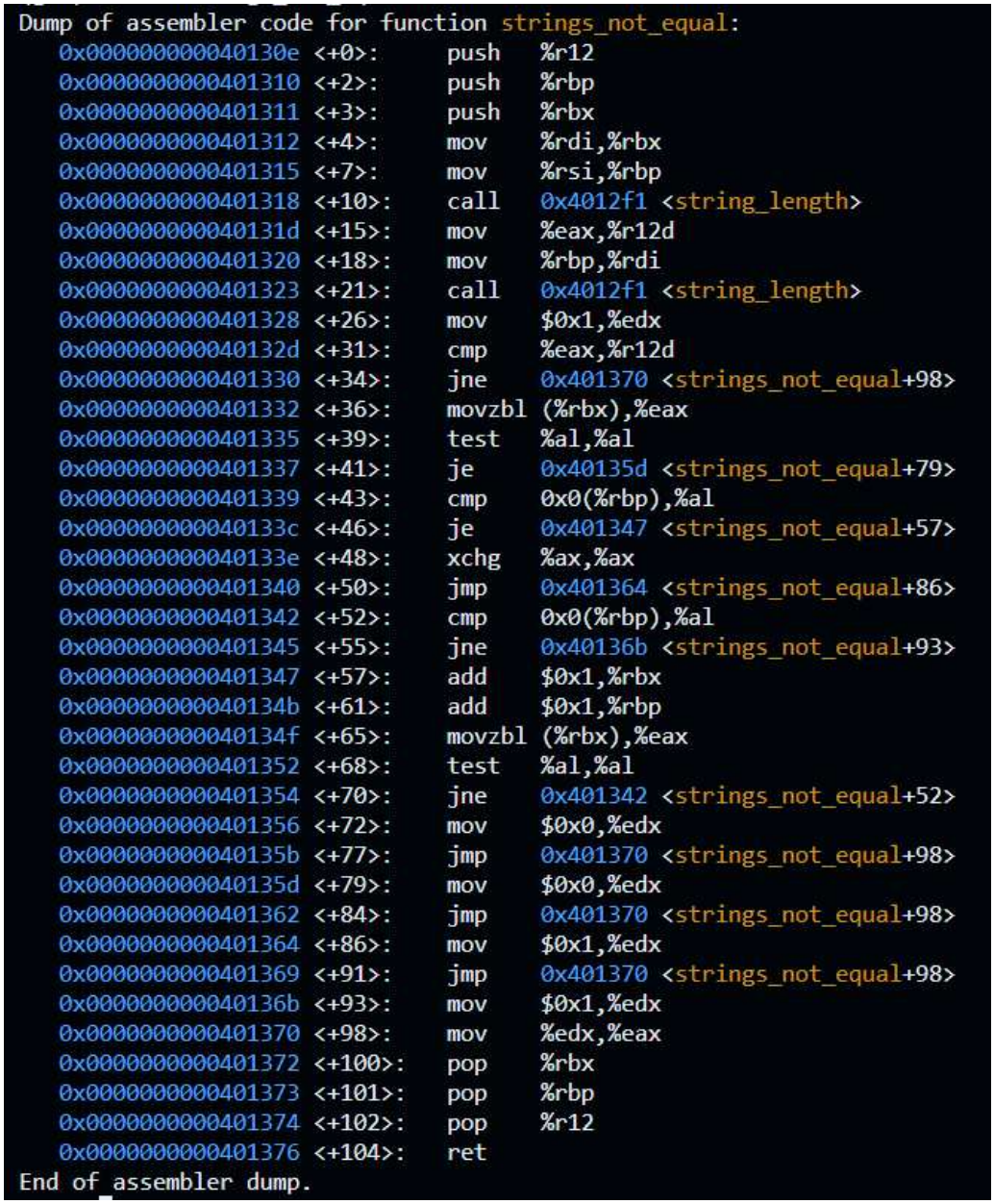
그 뒤 **test %eax, %eax**를 통해 결과값이 0이라면 통과, 1이라면 실패다.

strings_not_equal이 0을 반환하게 해야한다. x/s 커맨드로 수상해보이는 주소 $0x4024c0를 읽어보면,

```
(gdb) x/s 0x4024c0
0x4024c0:       "And they have no disregard for human life."
```

와 같다. **입력 문자열이 이것과 같으면 될 것**이라는 강한 생각이 든다. 하지만, 그렇게 될 수밖에 없는 이유를 알아봐야 한다. strings_not_equal을 보자. %rdi, %rsi를 받아 서로 같은지 비교해주는 함수 같다.

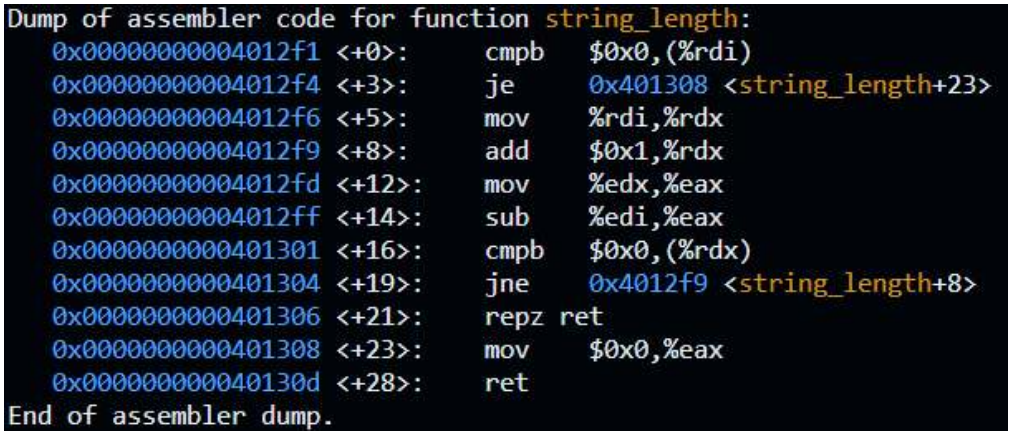
(gdb) disas string_not_equal



```
Dump of assembler code for function strings_not_equal:
   0x000000000040130e <+0>:     push   %r12
   0x0000000000401310 <+2>:     push   %rbp
   0x0000000000401311 <+3>:     push   %rbx
   0x0000000000401312 <+4>:     mov    %rdi,%rbx
   0x0000000000401315 <+7>:     mov    %rsi,%rbp
   0x0000000000401318 <+10>:    call   0x4012f1 <string_length>
   0x000000000040131d <+15>:    mov    %eax,%r12d
   0x0000000000401320 <+18>:    mov    %rbp,%rdi
   0x0000000000401323 <+21>:    call   0x4012f1 <string_length>
   0x0000000000401328 <+26>:    mov    $0x1,%edx
   0x000000000040132d <+31>:    cmp    %eax,%r12d
   0x0000000000401330 <+34>:    jne    0x401370 <strings_not_equal+98>
   0x0000000000401332 <+36>:    movzbl (%rbx),%eax
   0x0000000000401335 <+39>:    test   %al,%al
   0x0000000000401337 <+41>:    je     0x40135d <strings_not_equal+79>
   0x0000000000401339 <+43>:    cmp    0x0(%rbp),%al
   0x000000000040133c <+46>:    je     0x401347 <strings_not_equal+57>
   0x000000000040133e <+48>:    xchg   %ax,%ax
   0x0000000000401340 <+50>:    jmp    0x401364 <strings_not_equal+86>
   0x0000000000401342 <+52>:    cmp    0x0(%rbp),%al
   0x0000000000401345 <+55>:    jne    0x40136b <strings_not_equal+93>
   0x0000000000401347 <+57>:    add    $0x1,%rbx
   0x000000000040134b <+61>:    add    $0x1,%rbp
   0x000000000040134f <+65>:    movzbl (%rbx),%eax
   0x0000000000401352 <+68>:    test   %al,%al
   0x0000000000401354 <+70>:    jne    0x401342 <strings_not_equal+52>
   0x0000000000401356 <+72>:    mov    $0x0,%edx
   0x000000000040135b <+77>:    jmp    0x401370 <strings_not_equal+98>
   0x000000000040135d <+79>:    mov    $0x0,%edx
   0x0000000000401362 <+84>:    jmp    0x401370 <strings_not_equal+98>
   0x0000000000401364 <+86>:    mov    $0x1,%edx
   0x0000000000401369 <+91>:    jmp    0x401370 <strings_not_equal+98>
   0x000000000040136b <+93>:    mov    $0x1,%edx
   0x0000000000401370 <+98>:    mov    %edx,%eax
   0x0000000000401372 <+100>:   pop    %rbx
   0x0000000000401373 <+101>:   pop    %rbp
   0x0000000000401374 <+102>:   pop    %r12
   0x0000000000401376 <+104>:   ret
End of assembler dump.
```

string_length를 통해 %rdi와 %rsi 각각의 길이를 구하고 길이가 같은지 먼저 확인한다. 길이가 다르다면 1을 리턴하고, 길이가 같다면 아래로 내려가 각 자릿수를 1씩 늘리면서 각 자리의 문자가 서로 같은지 비교한다. 만약 같지 않은 것이 나왔다면 1을 리턴한다. 끝까지 같다면, 0을 리턴한다. string_length의 코드도 보자.

(gdb) disas string_length



입력 문자열 %rdi를 하나씩 따라가며 0이 나올 때까지의 문자열 길이를 반환한다. 이름 그대로 문자열의 길이를 반환한다.

즉, 처음 생각대로 입력값을 %rsi와 비교해서 같다면 통과한다. %rsi에 저장된 값인 **And they have no disregard for human life.**를 입력값으로 넣어주면 된다.

Phase_2

요약

int형 크기 6인 배열의 값을 규칙에 따라 채워줘야 한다. 첫 번째 값은 1이어야 하고, 다음 값은 이전 값의 2배여야 한다.

정답

1 2 4 8 16 32

상세 과정

```
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>:     call   0x4015aa <read_six_numbers>
   0x0000000000400f1a <+14>:    cmpl   $0x1,(%rsp)
   0x0000000000400f1e <+18>:    je     0x400f40 <phase_2+52>
   0x0000000000400f20 <+20>:    call   0x401574 <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>:    call   0x401574 <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>:    ret
End of assembler dump.
```

여섯 자리 숫자를 읽어서 조건문을 처리하는 것처럼 보인다. 1번과 마찬가지로 %rdi에 입력값을 넣으면 read_six_numbers로도 함께 넘어가서 처리되는 것 같다. 입력값이 레지스터에 저장되는지, 스택에 지역 변수로 저장되는지 알아보기 위해 read_six_numbers를 보자. **%rsp를 %rsi에 옮겨놓고 호출**한 점을 유의하자.

(gdb) read_six_numbers

```
Dump of assembler code for function read_six_numbers:
   0x00000000004015aa <+0>:     sub    $0x18,%rsp
   0x00000000004015ae <+4>:     mov    %rsi,%rdx
   0x00000000004015b1 <+7>:     lea    0x4(%rsi),%rcx
   0x00000000004015b5 <+11>:    lea    0x14(%rsi),%rax
   0x00000000004015b9 <+15>:    mov    %rax,0x8(%rsp)
   0x00000000004015be <+20>:    lea    0x10(%rsi),%rax
   0x00000000004015c2 <+24>:    mov    %rax,(%rsp)
   0x00000000004015c6 <+28>:    lea    0xc(%rsi),%r9
   0x00000000004015ca <+32>:    lea    0x8(%rsi),%r8
   0x00000000004015ce <+36>:    mov    $0x4027e1,%esi
   0x00000000004015d3 <+41>:    mov    $0x0,%eax
   0x00000000004015d8 <+46>:    call   0x400c30 <__isoc99_sscanf@plt>
   0x00000000004015dd <+51>:    cmp    $0x5,%eax
   0x00000000004015e0 <+54>:    jg     0x4015e7 <read_six_numbers+61>
   0x00000000004015e2 <+56>:    call   0x401574 <explode_bomb>
   0x00000000004015e7 <+61>:    add    $0x18,%rsp
   0x00000000004015eb <+65>:    ret
End of assembler dump.
```

요약하면 **%rsp, %rsp+4, ... , %rsp+20의 6군데에 4bytes로 input을 차례차례 채워넣는다.** 함수 호출 전에 %rsi에 %rsp를 집어넣어줬으므로 %rsi가 %rsp와 같은 역할을 한다. 스택 영역에 지역변수로 값이 들어감을 알 수 있다. 이제 돌아와서 나머지 코드를 보자.

```
0x0000000000400f15 <+9>:     call   0x4015aa <read_six_numbers>
0x0000000000400f1a <+14>:    cmpl   $0x1,(%rsp)
0x0000000000400f1e <+18>:    je     0x400f40 <phase_2+52>
0x0000000000400f20 <+20>:    call   0x401574 <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>:    call   0x401574 <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>:    ret
```

+14영역에선 첫 번째 값이 1인지 확인하고 있다. 1이 아니라면 폭발하므로 **첫 번째 값은 1이어야 함**을 알 수 있다. 이 부분을 넘어가면 +52로 가는데, 그 다음 값의 주소인 %rsp+4를 %rbx에 넣고 %rbp에는 %rsp+24를 넣었다. 마지막 입력값의 주소가 %rsp+20임을 생각하면 %rbp는 반복문의 종료 조건이 될 수 있을 것이다. 그리고 +27로 간다. 여기선 %rbx의 이전 값을 두 배하고 %rbx와 비교한다. 그리고 이 둘이 같아야 한다. 만약 같다면, %rbx는 다음 변수를 가리키고 %rbp와 비교하며 6번째 변수까지 다 봤는지 확인한다.

요약하면 이렇다. **int a[6] 배열이 있다고 하자. 여기서 a[0] = 1이어야 하며, a[i] = 2*a[i-1]**

(i=1, 2, 3, 4, 5)이어야 한다. 이런 a[6]의 값을 입력해줘야 한다.
즉, 입력값 1 2 4 8 16 32가 된다.

Phase_3

요약

입력 값에 따른 Switch문 분기점을 x/h로 알아내고, 조건에 맞게 두 입력 값을 넣어주면 됐다.

정답

(2, 181) 또는 (4, 0). 더 존재할 수 있다.

상세 과정

```
Dump of assembler code for function phase_3:
   0x0000000000400f53 <+0>:     sub    $0x18,%rsp
   0x0000000000400f57 <+4>:     lea    0x8(%rsp),%rcx
   0x0000000000400f5c <+9>:     lea    0xc(%rsp),%rdx
   0x0000000000400f61 <+14>:    mov    $0x4027ed,%esi
   0x0000000000400f66 <+19>:    mov    $0x0,%eax
   0x0000000000400f6b <+24>:    call   0x400c30 <__isoc99_sscanf@plt>
   0x0000000000400f70 <+29>:    cmp    $0x1,%eax
   0x0000000000400f73 <+32>:    jg     0x400f7a <phase_3+39>
   0x0000000000400f75 <+34>:    call   0x401574 <explode_bomb>
   0x0000000000400f7a <+39>:    cmpl   $0x7,0xc(%rsp)
   0x0000000000400f7f <+44>:    ja     0x400fdd <phase_3+138>
   0x0000000000400f81 <+46>:    mov    0xc(%rsp),%eax
   0x0000000000400f85 <+50>:    jmp    *0x402520(,%rax,8)
   0x0000000000400f8c <+57>:    mov    $0x0,%eax
   0x0000000000400f91 <+62>:    jmp    0x400f98 <phase_3+69>
   0x0000000000400f93 <+64>:    mov    $0xaa,%eax
   0x0000000000400f98 <+69>:    sub    $0x3a1,%eax
   0x0000000000400f9d <+74>:    jmp    0x400fa4 <phase_3+81>
   0x0000000000400f9f <+76>:    mov    $0x0,%eax
   0x0000000000400fa4 <+81>:    add    $0x101,%eax
   0x0000000000400fa9 <+86>:    jmp    0x400fb0 <phase_3+93>
   0x0000000000400fab <+88>:    mov    $0x0,%eax
   0x0000000000400fb0 <+93>:    sub    $0x4c,%eax
   0x0000000000400fb3 <+96>:    jmp    0x400fba <phase_3+103>
   0x0000000000400fb5 <+98>:    mov    $0x0,%eax
   0x0000000000400fba <+103>:   add    $0x4c,%eax
   0x0000000000400fbd <+106>:   jmp    0x400fc4 <phase_3+113>
   0x0000000000400fbf <+108>:   mov    $0x0,%eax
   0x0000000000400fc4 <+113>:   sub    $0x4c,%eax
   0x0000000000400fc7 <+116>:   jmp    0x400fce <phase_3+123>
   0x0000000000400fc9 <+118>:   mov    $0x0,%eax
   0x0000000000400fce <+123>:   add    $0x4c,%eax
   0x0000000000400fd1 <+126>:   jmp    0x400fd8 <phase_3+133>
   0x0000000000400fd3 <+128>:   mov    $0x0,%eax
   0x0000000000400fd8 <+133>:   sub    $0x4c,%eax
   0x0000000000400fdb <+136>:   jmp    0x400fe7 <phase_3+148>
   0x0000000000400fdd <+138>:   call   0x401574 <explode_bomb>
   0x0000000000400fe2 <+143>:   mov    $0x0,%eax
   0x0000000000400fe7 <+148>:   cmpl   $0x5,0xc(%rsp)
   0x0000000000400fec <+153>:   jg     0x400ff4 <phase_3+161>
   0x0000000000400fee <+155>:   cmp    0x8(%rsp),%eax
   0x0000000000400ff2 <+159>:   je     0x400ff9 <phase_3+166>
```

```
 0x0000000000400ff4 <+161>:    call   0x401574 <explode_bomb>
-Type <RET> for more, q to quit, c to continue without paging--c
 0x0000000000400ff9 <+166>:    add    $0x18,%rsp
 0x0000000000400ffd <+170>:    ret
```

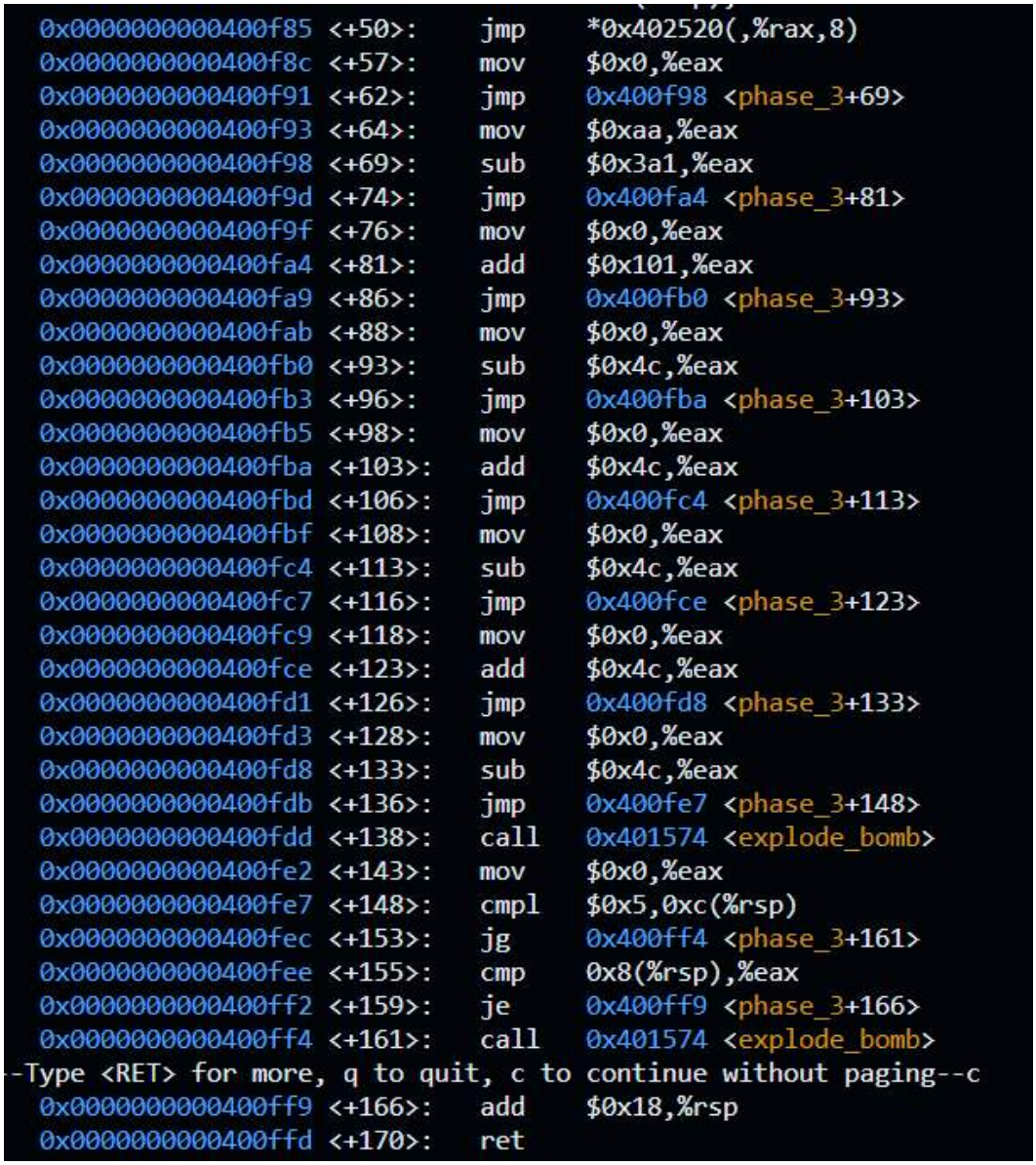
입력값 %rdi를 %rdx와 %rcx에 나눠 넣어준다. 즉, 입력값은 두 개의 정수가 되겠다. **첫 번째 값**은 %rdx인 **0xc(%rsp)**에 저장되어 있고, **두 번째 값**은 **0x8(%rsp)**에 저장되어 있다. **앞으로 첫 번째 값을 a로, 두 번째 값을 b로 부르겠다.**

다음으로 a와 7을 비교하고 a가 7보다 크면 폭탄이 터진다. 즉, a <= 7이다. 또한, jg가 아닌 ja를 사용했으므로 a는 unsigned임을 알 수 있다. 결과적으로 **0 <= a <= 7**이 된다.

그리고 *(0x402520+8*a)로 jump한다. a따라 다른 지점으로 jump하므로 jump table을 사용하는 switch문임을 알 수 있다. **x/h *(0x402520+8*a)**로 알아본 a의 값에 따른 jump 위치는 아래와 같다.

```
(gdb) x/h *0x402520
0x400f93 <phase_3+64>:  0xaab8
(gdb) x/h *(0x402520+8)
0x400f8c <phase_3+57>:  0x00b8
(gdb) x/h *(0x402520+16)
0x400f9f <phase_3+76>:  0x00b8
(gdb) x/h *(0x402520+24)
0x400fab <phase_3+88>:  0x00b8
(gdb) x/h *(0x402520+32)
0x400fb5 <phase_3+98>:  0x00b8
(gdb) x/h *(0x402520+40)
0x400fbf <phase_3+108>: 0x00b8
(gdb) x/h *(0x402520+48)
0x400fc9 <phase_3+118>: 0x00b8
(gdb) x/h *(0x402520+56)
0x400fd3 <phase_3+128>: 0x00b8
(gdb) x/h *(0x402520+64)
0xa:    Cannot access memory at address 0xa
```

a=8일 때는 접근할 수 없다. 앞서 확인한 것처럼 0~7까지 유효함을 알 수 있다. 다시 코드로 돌아가자.



fall down도 있고 복잡해 보이므로 통과되는 조건부터 역으로 올라가 보자. +155에서 %eax가 0x8(%rsp)인 b와 같은지 확인한다. 나머지 코드에서 0x8(%rsp)가 바뀐 부분은 없으므로 **b == %eax**여야 한다. 또 올라가보면 0xc(%rsp)와 5를 비교한다. 0xc(%rsp)또한 나머지 코드에서 바뀐 부분이 없으니 **a <= 5**여야 한다.

이제 0 <= a <= 5에서 앞서 알아본 a에 따른 분기점에서 내려오며 %eax를 계산한다. 그 %eax와 같은 b를 입력으로 넣어주면 통과한다. 주의해야 할 점은 a = 0일 때 큰 수를 빼므로 만약 b가 unsigned라면 overflow가 날 수도 있다. 그러니 안전하게 overflow가 안 날만한 a를 선택해주자. 대표적으로 a = 2일 때 b = 181, a = 4일 때 b = 0이 있겠다.

Phase_4

요약

재귀 함수의 구조를 파악해야 했다. callee saved register는 다른 함수를 불러도 그 값이 변하지 않는다. 위에서부터 아래로 내려가며 재귀함수의 값을 찾기보다 아래에서 위로 올라가며 원하는 값을 조립해 나가는 것이 더 쉬웠다.

정답

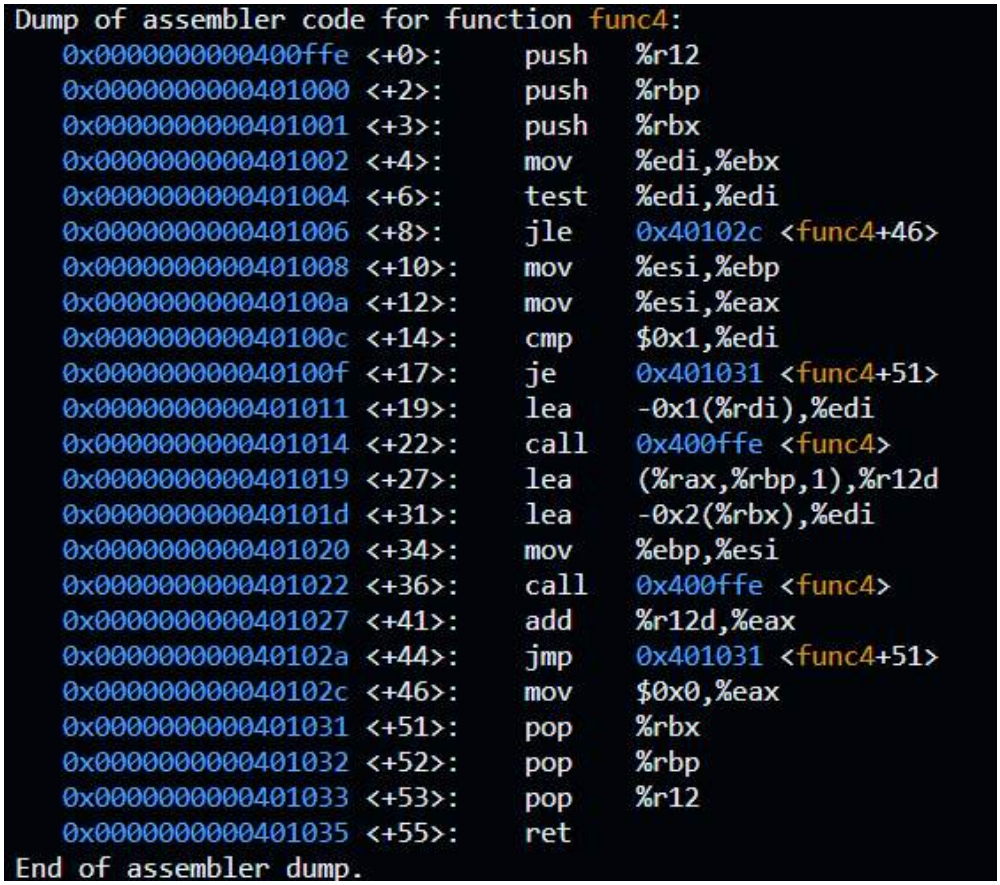
(24, 2), (36, 3), (48, 4)

상세 과정

```
Dump of assembler code for function phase_4:
   0x0000000000401036 <+0>:     sub    $0x18,%rsp
   0x000000000040103a <+4>:     lea    0xc(%rsp),%rcx
   0x000000000040103f <+9>:     lea    0x8(%rsp),%rdx
   0x0000000000401044 <+14>:    mov    $0x4027ed,%esi
   0x0000000000401049 <+19>:    mov    $0x0,%eax
   0x000000000040104e <+24>:    call   0x400c30 <__isoc99_sscanf@plt>
   0x0000000000401053 <+29>:    cmp    $0x2,%eax
   0x0000000000401056 <+32>:    jne    0x401064 <phase_4+46>
   0x0000000000401058 <+34>:    mov    0xc(%rsp),%eax
   0x000000000040105c <+38>:    sub    $0x2,%eax
   0x000000000040105f <+41>:    cmp    $0x2,%eax
   0x0000000000401062 <+44>:    jbe    0x401069 <phase_4+51>
   0x0000000000401064 <+46>:    call   0x401574 <explode_bomb>
   0x0000000000401069 <+51>:    mov    0xc(%rsp),%esi
   0x000000000040106d <+55>:    mov    $0x5,%edi
   0x0000000000401072 <+60>:    call   0x400ffe <func4>
   0x0000000000401077 <+65>:    cmp    0x8(%rsp),%eax
   0x000000000040107b <+69>:    je     0x401082 <phase_4+76>
   0x000000000040107d <+71>:    call   0x401574 <explode_bomb>
   0x0000000000401082 <+76>:    add    $0x18,%rsp
   0x0000000000401086 <+80>:    ret
End of assembler dump.
```

func4를 부른다. 그 전후를 분석해보자. 3번과 마찬가지로 입력값 두 개를 받고 첫 번째 값은 %rdx인 0x8(%rsp)에, 두 번째 값은 %rcx인 0xc(%rsp)에 넣어준다. **여기서도 첫 번째 입력값을 a로, 두 번째 입력값을 b라고 하겠다.** b-2 <= 2를 만족해야 한다. 여기서 **jbe**를 사용했으므로 b는 unsigned이다. 따라서 b가 2보다 작게되면 underflow가 일어나므로 **2 <= b <= 4**여야 한다. 그리고 **a == func4(5, b)**를 만족하는지 확인한다. 이제 func4(5, b)가 어떤 값을 가지는지 확인만 하면 된다.

(gdb) disas func4



재귀함수 형태이다. func4(x, y)라고 할 때 +0부터 +17까지를 해석해보면, x <= 0일 때 0을 리턴하고, x == 1일 때는 y를 리턴한다. 즉, **func4(0, y) = 0, func4(1, y) = y**이다.
x > 1일 때 나머지 부분으로 내려가게 되고 이 때 x = x-1을 해주고 func4(x, y)를 부른다. 이 결과는 %rax에 저장된다. 즉, **1씩 줄이면서 x <= 1인 base condition으로 향하는 재귀 구조이다.**
+27부터 +36까지 보자. %rbp는 처음에 y로 %rbx는 x로 초기화 되어 있다. 이를 고려하며 +27, +31, +34를 해석해보면 다음과 같다.
%r12d = %rax + y
x = x-2
다음으로 func4(x, y)를 또 호출한다. 그리고 그 리턴 값에 %r12d를 더하고 리턴하게 된다.

꽤 복잡해 보이는 구조지만 1씩 줄이면서 x <= 1인 base condition으로 향하는 재귀 구조임을 알고 있고, 우리가 원하는 결과는 func4(5, b)이므로 x = 1에서부터 5까지 차근차근 올라가면 된다. func4(x, y)는 func4(x-1, y), y, func4(x-2, y) 세 가지의 합으로 이루어진다. 이를 근거로 func4(5, y)까지 구해보면 아래와 같다.
func4(2, y) = func4(1, y) + y + func4(0, y) = 2y
func4(3, y) = func4(2, y) + y + func4(1, y) = 4y
func4(4, y) = func4(3, y) + y + func4(2, y) = 7y
func4(5, y) = func4(4, y) + y + func4(3, y) = **12y**
이제 모든 정보를 조합해보면, **2 <= b <= 4이고 a = 12b여야 한다. 가능한 쌍 (a, b) = (24,**

2), (36, 3), (48, 4)가 된다.

Phase_5

요약

배열이 존재함을 체크하고 어떻게 생겼는지 확인해야 한다. 확인한 배열과 관찰한 코드 조건을 바탕으로 어떤 인덱스를 선택해야 하는지, 어떤 값들이 참조되었는지 확인하면 되었다.

정답

5 115

상세 과정

```
Dump of assembler code for function phase_5:
   0x0000000000401087 <+0>:     sub    $0x18,%rsp
   0x000000000040108b <+4>:     lea    0x8(%rsp),%rcx
   0x0000000000401090 <+9>:     lea    0xc(%rsp),%rdx
   0x0000000000401095 <+14>:    mov    $0x4027ed,%esi
   0x000000000040109a <+19>:    mov    $0x0,%eax
   0x000000000040109f <+24>:    call   0x400c30 <__isoc99_sscanf@plt>
   0x00000000004010a4 <+29>:    cmp    $0x1,%eax
   0x00000000004010a7 <+32>:    jg     0x4010ae <phase_5+39>
   0x00000000004010a9 <+34>:    call   0x401574 <explode_bomb>
   0x00000000004010ae <+39>:    mov    0xc(%rsp),%eax
   0x00000000004010b2 <+43>:    and    $0xf,%eax
   0x00000000004010b5 <+46>:    mov    %eax,0xc(%rsp)
   0x00000000004010b9 <+50>:    cmp    $0xf,%eax
   0x00000000004010bc <+53>:    je     0x4010ea <phase_5+99>
   0x00000000004010be <+55>:    mov    $0x0,%ecx
   0x00000000004010c3 <+60>:    mov    $0x0,%edx
   0x00000000004010c8 <+65>:    add    $0x1,%edx
   0x00000000004010cb <+68>:    cltq
   0x00000000004010cd <+70>:    mov    0x402560(,%rax,4),%eax
   0x00000000004010d4 <+77>:    add    %eax,%ecx
   0x00000000004010d6 <+79>:    cmp    $0xf,%eax
   0x00000000004010d9 <+82>:    jne    0x4010c8 <phase_5+65>
   0x00000000004010db <+84>:    mov    %eax,0xc(%rsp)
   0x00000000004010df <+88>:    cmp    $0xf,%edx
   0x00000000004010e2 <+91>:    jne    0x4010ea <phase_5+99>
   0x00000000004010e4 <+93>:    cmp    0x8(%rsp),%ecx
   0x00000000004010e8 <+97>:    je     0x4010ef <phase_5+104>
   0x00000000004010ea <+99>:    call   0x401574 <explode_bomb>
   0x00000000004010ef <+104>:   add    $0x18,%rsp
   0x00000000004010f3 <+108>:   ret
End of assembler dump.
```

5번 또한 두 개의 정수형 입력을 %rdx와 %rcx에 저장하고 위치는 각각 0xc(%rsp), 0x8(%rsp)가 된다. 중간에 반복문이 있고 0x402560이라는 수상한 주소가 있다. +39부터 차례대로 살펴보자. 여기서도 **첫 번째 입력값을 a, 두 번째 입력값을 b라 하겠다.**
0xc(%rsp)에 저장된 첫 번째 입력값을 검사한다. 만약 **a의 하위 4비트가 1111이라면**

폭발한다.
다음으로 +55부터 살펴보면, %rcx와 %rdx 모두 0으로 초기화하고, %rdx에 1을 더한다. 반복문의 시작 지점이다. 반복할수록 %rdx는 1씩 커진다. long 형인 %edx의 상위 4bytes를 sign extension하고, 0x402560+4*%rax의 메모리 값을 다시 %eax에 넣는다. 이 값은 반복문 속에서 다시 다른 값을 참조하고 %eax에 들어가게 된다. 그 값을 0으로 초기화해 둔 %ecx에 더한다. 마지막으로 %eax가 0xf가 되면 반복문을 빠져나간다. 만약 같지 않다면 %edx에 1을 더하는 것으로 똑같은 반복을 시작한다.
%eax == 0xf여서 반복문을 빠져나온 후엔 %edx == 0xf인지를 확인한다. 같지 않다면 폭발한다. 마지막으로, 두 번째 인자인 b와 반복문 속에서 %eax의 값을 모두 더한 %ecx가 같은지 비교한다. 같지 않다면 폭발한다.
해석을 해 본 결과, +70의 메모리 주소에서 %rax만큼 진행하고 **참조한 값을 다시 %rax에 넣는 것을 반복**한다. **반복하면서 나온 값들을 %ecx에 더한다**. 참조한 값이 0xf일 때 반복문은 종료된다. +65와 +88에 따르면 **반복문은 총 0xf번 반복해야 한다**. 지금까지 봤을 때 **0x402560**은 4bytes인 데이터로 이루어진 배열이고 15를 포함해야 하고 해당 배열 안에서 돌고 도는 구조여야 함을 예상할 수 있다. 이제 0x402560이 우리 예상과 맞는지 확인해보자.

```
(gdb) print/x *0x402560@16
$3 = {0xa, 0x2, 0xe, 0x7, 0x8, 0xc, 0xf, 0xb, 0x0, 0x4, 0x1, 0xd, 0x3, 0x9, 0x6, 0x5}
```

print/x *0x402560@16의 의미는 0x402560주소로부터 4bytes(기본값)만큼 16번 이동하면서 그때마다 그 주소의 메모리 값을 16진수로 출력하라는 뜻이다.
해당 배열을 A라 하자. A는 0~15까지가 한 번씩만 나타나는 순환하는 배열임을 알 수 있다. 이제 우리가 찾아야 하는 부분은 **어떤 인덱스를 선택해야 반복문을 빠져나왔을 때 %edx를 0xf로 맞출 수 있는지이다.** 예를 들어 2번째 인덱스를 선택했을 때 %edx의 값을 확인해보자. A[2] = 0x6 != 0xf이므로 %edx는 2가 되고 다시 반복한다. A[0x6] = 0xf == 0xf 이므로 %edx는 2가 된다. **이런 규칙 속에서 %edx를 0xf로 맞추려면 5번째 인덱스를 선택하면 된다. 즉, a = 5가 된다.** 그러면 %eax가 0xf가 되어 반복문을 빠져나왔을 때 %edx 또한 0xf가 된다. 이제 %ecx의 값만 구해주면 된다. %ecx는 반복문 속에 나왔던 모든 배열 값의 합으로, **우리는 A[0xf]를 제외한 모든 값을 참조했다.** 그렇기 때문에 **%ecx = 1+2+...+15 – A[0xf] = 115가 되고 이 값은 곧 0x8(%rsp)인 b가 된다.** 따라서, a = 5, b = 115가 된다.

Phase_6

요약

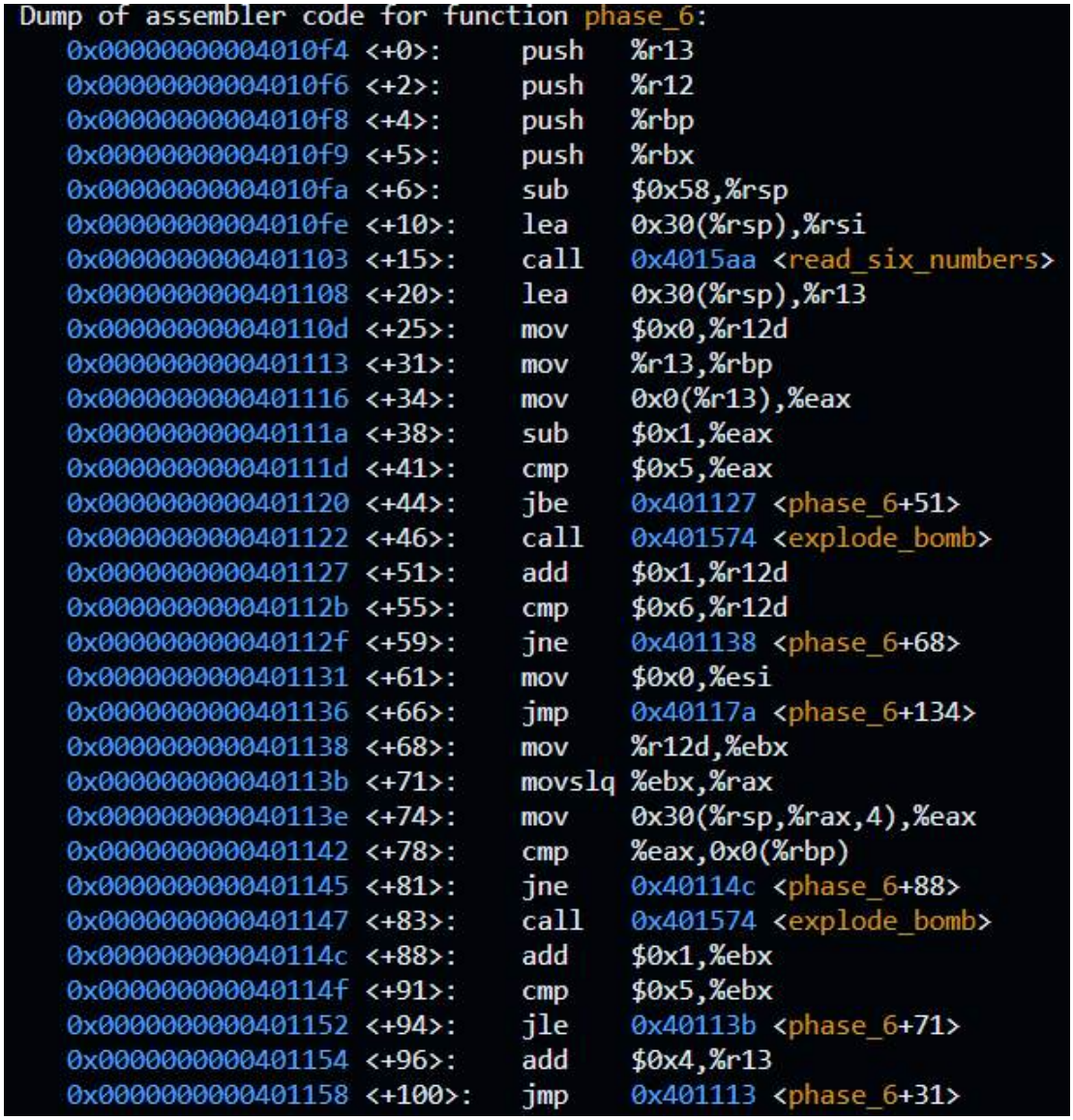
1~6을 각 하나씩만 사용해서 숫자 여섯 개를 입력 받고, 그 수를 인덱스로 가지는 구조체를 재배열한다. 재배열된 구조체의 첫 번째에 저장된 요소 값들은 단조증가해야 한다. 예를 들어 5 3 2 1 4 6으로 입력했다고 하자. 이 때 구조체의 요소 값들은 기존 737, 484, 419, 881, 61, 937에서 우리가 입력한 순서대로 재배열 될 것이다. 즉, 61, 419, 484, 737, 881, 937로 재배열되고 이는 조건에 맞게 단조증가함을 알 수 있다. 따라서 정답은 5 3 2 1 4 6이 된다.

정답

5 3 2 1 4 6

상세 과정

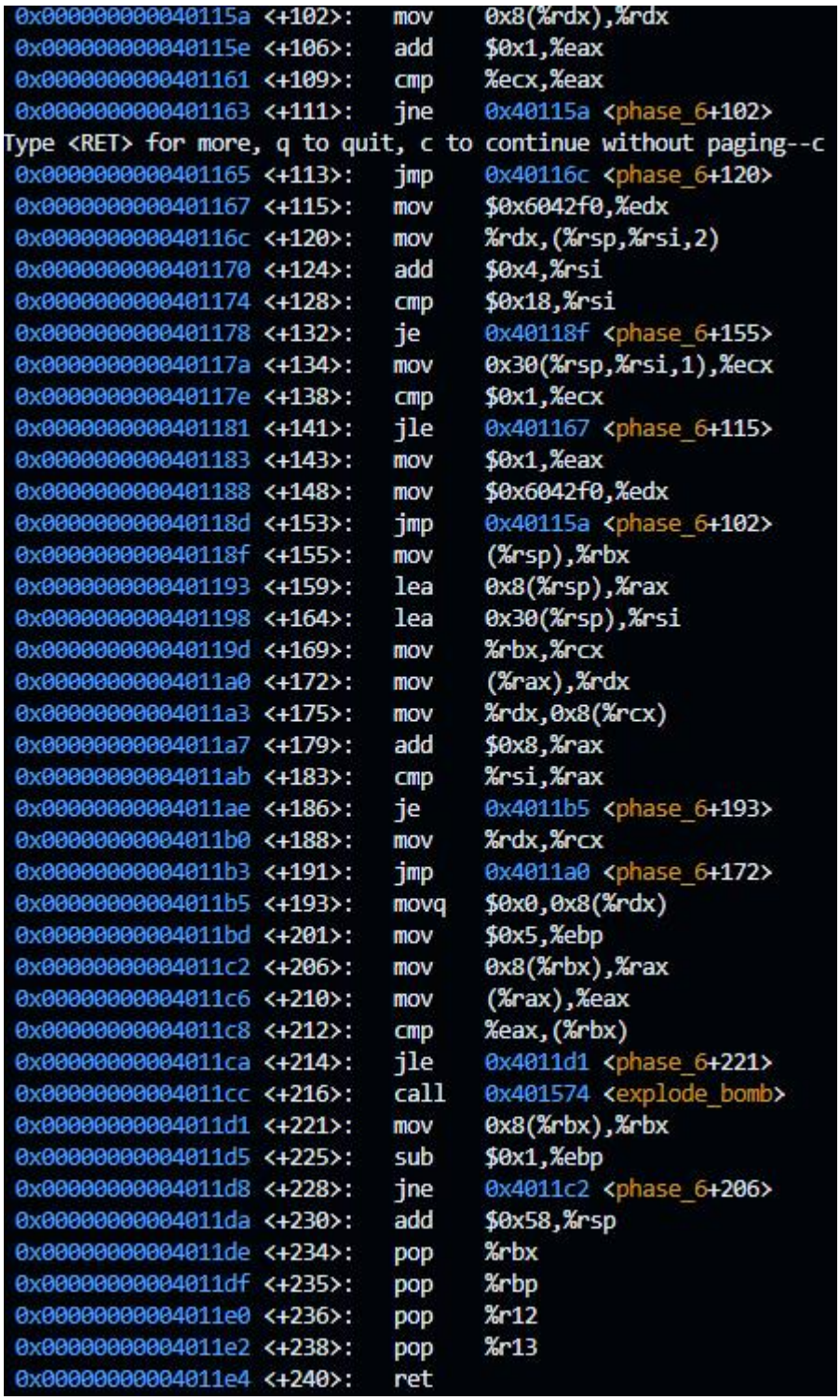
phase_6은 상당히 복잡하므로 먼저 구조와 흐름을 짚고 넘어가겠다. +0~+100까지와 그 이후로 나눌 수 있다. 첫 번째 부분은 여섯 가지 입력 값의 범위와 조건을 알려주고, 두 번째 부분은 입력받은 여섯 가지 수 순서대로 연결 리스트의 순서를 바꾸는데, 이 때, 오름차순으로 정렬되어야 한다. 두 번째 부분도 세 부분으로 나뉘게 되는데, 이는 그 때 가서 말하도록 하고 먼저 첫 번째 부분을 들여다보자.



```
Dump of assembler code for function phase_6:
   0x00000000004010f4 <+0>:     push   %r13
   0x00000000004010f6 <+2>:     push   %r12
   0x00000000004010f8 <+4>:     push   %rbp
   0x00000000004010f9 <+5>:     push   %rbx
   0x00000000004010fa <+6>:     sub    $0x58,%rsp
   0x00000000004010fe <+10>:    lea    0x30(%rsp),%rsi
   0x0000000000401103 <+15>:    call   0x4015aa <read_six_numbers>
   0x0000000000401108 <+20>:    lea    0x30(%rsp),%r13
   0x000000000040110d <+25>:    mov    $0x0,%r12d
   0x0000000000401113 <+31>:    mov    %r13,%rbp
   0x0000000000401116 <+34>:    mov    0x0(%r13),%eax
   0x000000000040111a <+38>:    sub    $0x1,%eax
   0x000000000040111d <+41>:    cmp    $0x5,%eax
   0x0000000000401120 <+44>:    jbe    0x401127 <phase_6+51>
   0x0000000000401122 <+46>:    call   0x401574 <explode_bomb>
   0x0000000000401127 <+51>:    add    $0x1,%r12d
   0x000000000040112b <+55>:    cmp    $0x6,%r12d
   0x000000000040112f <+59>:    jne    0x401138 <phase_6+68>
   0x0000000000401131 <+61>:    mov    $0x0,%esi
   0x0000000000401136 <+66>:    jmp    0x40117a <phase_6+134>
   0x0000000000401138 <+68>:    mov    %r12d,%ebx
   0x000000000040113b <+71>:    movslq %ebx,%rax
   0x000000000040113e <+74>:    mov    0x30(%rsp,%rax,4),%eax
   0x0000000000401142 <+78>:    cmp    %eax,0x0(%rbp)
   0x0000000000401145 <+81>:    jne    0x40114c <phase_6+88>
   0x0000000000401147 <+83>:    call   0x401574 <explode_bomb>
   0x000000000040114c <+88>:    add    $0x1,%ebx
   0x000000000040114f <+91>:    cmp    $0x5,%ebx
   0x0000000000401152 <+94>:    jle    0x40113b <phase_6+71>
   0x0000000000401154 <+96>:    add    $0x4,%r13
   0x0000000000401158 <+100>:   jmp    0x401113 <phase_6+31>
```

0x30(%rsp)부터 4bytes씩 6자리를 입력받는다. 첫 번째 입력값은 0x30(%rsp)에 두 번째는 0x34(%rsp), ..., 여섯 번째 값은 0x44(%rsp)에 저장된다. **이는 4bytes 데이터의 배열로 볼 수 있다. 이를 A[6]이라 하자.** +31에서 +100까지 반복문을 이룬다. 그리고 +71부터 +94까지도 반복문을 이룬다. 이 두 반복문에서 입력값의 조건을 알아낼 수 있다. 바깥 반복문에서는 %r13이 배열 A를 돌면서 배열의 값 A[i]-1 <= 5임을 알려준다. 이 때 jbe가 사용되었으므로 A[i]는 unsigned이고 따라서 **1 <= A[i] <= 6**이다.

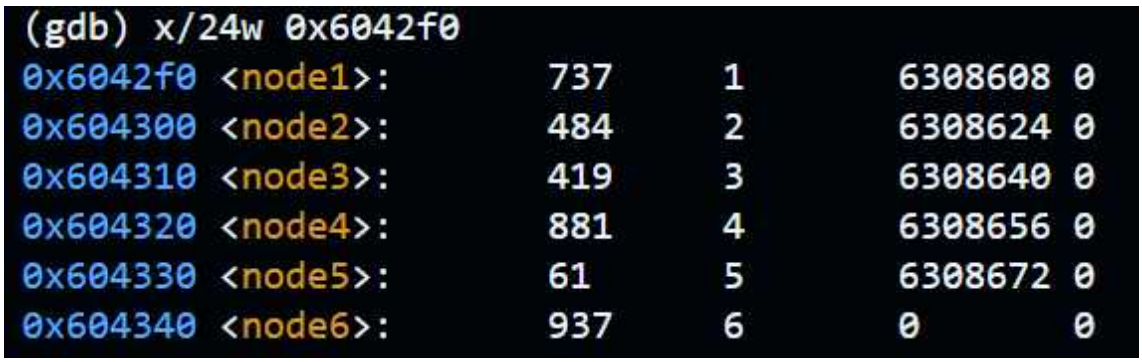
이제 안쪽 반복문으로 들어가보자. 여기선 %r12d와 %ebx가 반복을 담당한다. 바깥 반복문과 마찬가지로 배열의 원소를 참조하는데 2중 반복문 형식이다. 바깥 반복문의 %rbp는 0에서 1씩 증가하여 5까지 진행하고 이를 i라고 하자. 안쪽 반복문의 %ebx는 %rbp+1부터 5까지 진행하고 이를 j라고 하자. +81을 풀어 써보면 조건 식은 A[i] != A[j]가 된다. 즉, A[0]은 A[1], A[2], ..., A[5]와 달라야 한다. 또한, A[1]도 A[2], ..., A[5]와 달라야 한다. 결과적으로 **A[i]는 유일해야 한다.** 앞선 조건인 1 <= A[i] <= 6과 조합해보면, **A[i]는 1~6까지 값을 하나씩만 갖는 배열이다.** 이렇게 %r12d가 배열의 끝에 도달하면 %esi를 0으로 바꾸고 +134로 jump하게 된다. 이제 두 번째 부분을 들여다보자.



두 번째 부분도 반복문에 따라 **세 부분으로 나눌 수 있다.** +102 ~ +153, +155 ~ +191, **그리고 이외 부분이다.** 먼저 +102 ~ +153을 보자.

해당 부분은 스택 메모리 공간에 구조체 연결 리스트의 포인터를 낮은 주소부터 높은 주소로 6개를 배정한다. 배정하는 순서는 우리가 입력한 배열에 따른다. +148의 수상한 주소와 +102에서 %rdx+8을 참조하는 이상한 연산에서 0x6042f0을 여러 크기로 조사해봤다.

4bytes씩 끊어 조사해본 결과, 다음과 같은 정보를 알 수 있었다.



즉, 이것은 node라는 16bytes 구조체 6개를 연결하는 연결 리스트이다. 구**조체 정보는 첫 4bytes는 구조체 요소 값, 다음 4bytes는 구조체의 인덱스, 마지막 8bytes는 다음 구조체를 가리키는 next 포인터라고 생각한다.** 그리고, **0x6042f0는 이들 중 첫 번째 구조체의 시작을 가리키는 주소**임을 알 수 있었다.
다시 살펴보면 이전 부분에서 jump해 온 +134은 배열 A의 값이 1과 같으면 첫 번째 구조체의 주소를 갖고, 아니라면 +102의 반복문으로 가서 A[i]번째 구조체를 주소로 갖는다. 그리고 +120에서는 앞서 말한대로 **%rsp에서 8bytes만큼 뛰면서 A[i]번째 구조체의 주소를 넣는다.** 총 여섯 번 반복하면 %rsi는 0x18을 가리키게 되고 +155로 jump한다.
이제 +155 ~ +191을 보자. 해당 부분은 %rsp에 연결된 구조체들의 next 포인터를 %rsp의 순서에 맞게 바꿔주는 역할을 수행한다. 스택 메모리에는 구조체를 가리키는 **포인터**가 저장되어 있다. 이를 이용해 구조체 속의 next 포인터를 바꿔주게 된다. 자세한 과정은 +159에서 %rax에 다음 구조체의 주소를 저장한 주소인 %rsp+8을 넣어줌으로써 시작한다. 다음으로 +175에서 현재 구조체의 다음 구조체의 주소를 담은 공간에 *(%rcx+8)로 접근해 우리가 연결한 다음 구조체의 주소인 *rax를 넣어준다. 이를 구조체 개수만큼 반복시킴으로써 우리가 연결한대로 구조체도 똑같이 연결시킬 수 있다.
마지막 단계인 +193부터는 최종적으로 스택 메모리에 연결시킨 구조체의 요소값이 오름차순으로 정렬되어있는지 확인한다. 그 방법은 +206부터 시작한다. %rbx는 +155에서 (%rsp)로 할당된 후 바뀌지 않았다. 0x8(%rbx)를 통해 %rax에 다음 구조체의 시작 주소를 넘긴다. +210에서는 시작 주소를 메모리 참조한다. 구조체의 요소는 구조체 가장 앞부분에 있으므로 +210에서는 다음 구조체의 요소 값을 얻는다고 볼 수 있다. +212에서는 현재 구조체의 요소 값인 (%rbx)와 다음 구조체의 요소 값인 %rax를 비교하고 만약 다음 값이 현재 값보다 작다면 폭발한다. 만약 같거나 크다면, %rbx는 다음 구조체의 시작 주소를 저장하고 이를 구조체 끝까지 수행한다. 즉, **재배열한 구조체의 값들은 단조증가해야 한다.**

모두 정리하면, 1~6을 각 하나씩만 사용해서 숫자 여섯 개를 입력 받고, 그 수를 인덱스로 가지는 구조체를 재배열한다. 재배열된 구조체의 첫 번째에 저장된 요소 값들은 단조증가해야 한다. 예를 들어 5 3 2 1 4 6으로 입력했다고 하자. 이 때 구조체의 요소 값들은 기존 737, 484, 419, 881, 61, 937에서 우리가 입력한 순서대로 재배열 될 것이다. 즉, 61, 419, 484, 737, 881, 937로 재배열되고 이는 조건에 맞게 단조증가함을 알 수 있다. 따라서 정답은 5 3 2 1 4 6이 된다.

Secret_Phase

요약

각 노드 당 요소 값 하나를 가지는 이진 탐색 트리이다. n=0에서 시작하고 입력 값이 노드의 값보다 크면 오른쪽, 작으면 왼쪽 자식으로 재귀적으로 leaf 노드까지 내려간다. Leaf 노드에 이르고 난 후 내려왔던 길 그대로 다시 root 노드로 올라간다. 이 때, 오른쪽 자식이었다면 n = 2*n+1을, 왼쪽 자식이었다면 n = 2*n을 수행하며 root노드에 도달했을 때 n이 4가 되게끔 하는 값을 입력해야 한다.

정답

7

상세 과정

disas secret_phase를 통해서 secret_phase란 단계가 있음을 알 수 있었다. 코드를 다 뜯어본 결과, secret_phase는 아래와 같이 phase_defused의 +107에 숨어있음을 알게 됐다.

```
Dump of assembler code for function phase_defused:
   0x0000000000401712 <+0>:     sub    $0x68,%rsp
   0x0000000000401716 <+4>:     mov    $0x1,%edi
   0x000000000040171b <+9>:     call   0x4014b0 <send_msg>
   0x0000000000401720 <+14>:    cmpl   $0x6,0x203075(%rip)        # 0x60479c <num_input_strings>
   0x0000000000401727 <+21>:    jne    0x401796 <phase_defused+132>
   0x0000000000401729 <+23>:    lea    0x10(%rsp),%r8
   0x000000000040172e <+28>:    lea    0x8(%rsp),%rcx
   0x0000000000401733 <+33>:    lea    0xc(%rsp),%rdx
   0x0000000000401738 <+38>:    mov    $0x402837,%esi
   0x000000000040173d <+43>:    mov    $0x6048b0,%edi
   0x0000000000401742 <+48>:    mov    $0x0,%eax
   0x0000000000401747 <+53>:    call   0x400c30 <__isoc99_sscanf@plt>
   0x000000000040174c <+58>:    cmp    $0x3,%eax
   0x000000000040174f <+61>:    jne    0x401782 <phase_defused+112>
   0x0000000000401751 <+63>:    mov    $0x402840,%esi
   0x0000000000401756 <+68>:    lea    0x10(%rsp),%rdi
   0x000000000040175b <+73>:    call   0x40130e <strings_not_equal>
   0x0000000000401760 <+78>:    test   %eax,%eax
   0x0000000000401762 <+80>:    jne    0x401782 <phase_defused+112>
   0x0000000000401764 <+82>:    mov    $0x402698,%edi
   0x0000000000401769 <+87>:    call   0x400b40 <puts@plt>
   0x000000000040176e <+92>:    mov    $0x4026c0,%edi
   0x0000000000401773 <+97>:    call   0x400b40 <puts@plt>
   0x0000000000401778 <+102>:   mov    $0x0,%eax
   0x000000000040177d <+107>:   call   0x401223 <secret_phase>
   0x0000000000401782 <+112>:   mov    $0x4026f8,%edi
   0x0000000000401787 <+117>:   call   0x400b40 <puts@plt>
   0x000000000040178c <+122>:   mov    $0x402728,%edi
   0x0000000000401791 <+127>:   call   0x400b40 <puts@plt>
   0x0000000000401796 <+132>:   add    $0x68,%rsp
   0x000000000040179a <+136>:   ret
```

이제 +107로 진입하는 방법을 알아야 한다. 차근차근 보면, +14에서 %rip의 무엇이 6과 같은지 비교한다. 오른쪽의 num_input_string에서 입력 문자열의 개수임을 추론할 수 있고, 입력 문자열이 6개란 말은 6단계까지 입력을 다 했다는 뜻이다. 그 다음에 스택 포인터에 지역 변수 세 개를 할당하고 +38의 주소를 확인하면 숫자 두 개와 문자열 하나를 읽는 것을 알 수 있다. 세 개를 제대로 읽으면 +63에서 어떤 문자열과 세 번째로 입력받은 문자열을

비교하고, 두 개가 같다면 secret_phase로 간다. +63에 있는 문자열은 **"DrEvil"**이고 이것을 어느 부분에서 입력해줘야 한다. 어느 부분인지 알기 위해 비교되는 대상인 %rdi에 저장되는 주소인 0x6048b0을 조사하기로 한다. 그냥 조사해보면 빈 값만 뜬다. 프로그램을 돌리면서 해당 부분에 어떤 값이 저장될 것이라 생각하고 +53에 break point를 걸고 bomb 시작 후 정답을 모두 입력한다. 그러면 프로세스는 break point에 걸리게 되는데, 이 때 0x6048b0을 조사하면 아래와 같이 나온다.

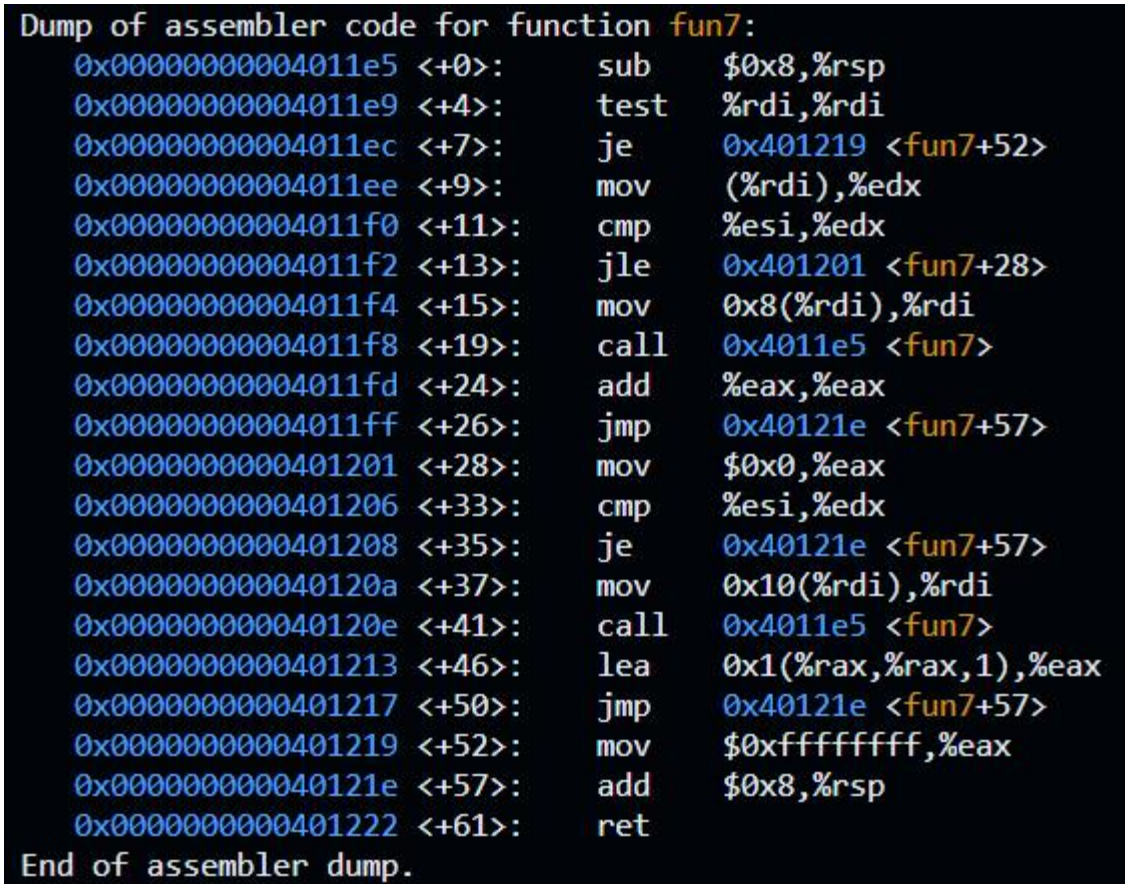
```
Breakpoint 1, 0x0000000000401747 in phase_defused ()
(gdb) x/s 0x6048b0
0x6048b0 <input_strings+240>:    "24 2"
```

즉, 네 번째 문제에서 추가로 문자열을 하나 더 읽는 것이다. 이제 네 번째 문제의 답인 24 2 뒤에 DrEvil을 입력하면 secret_phase로 갈 수 있게 된다.

이제 secret_phase를 보자.

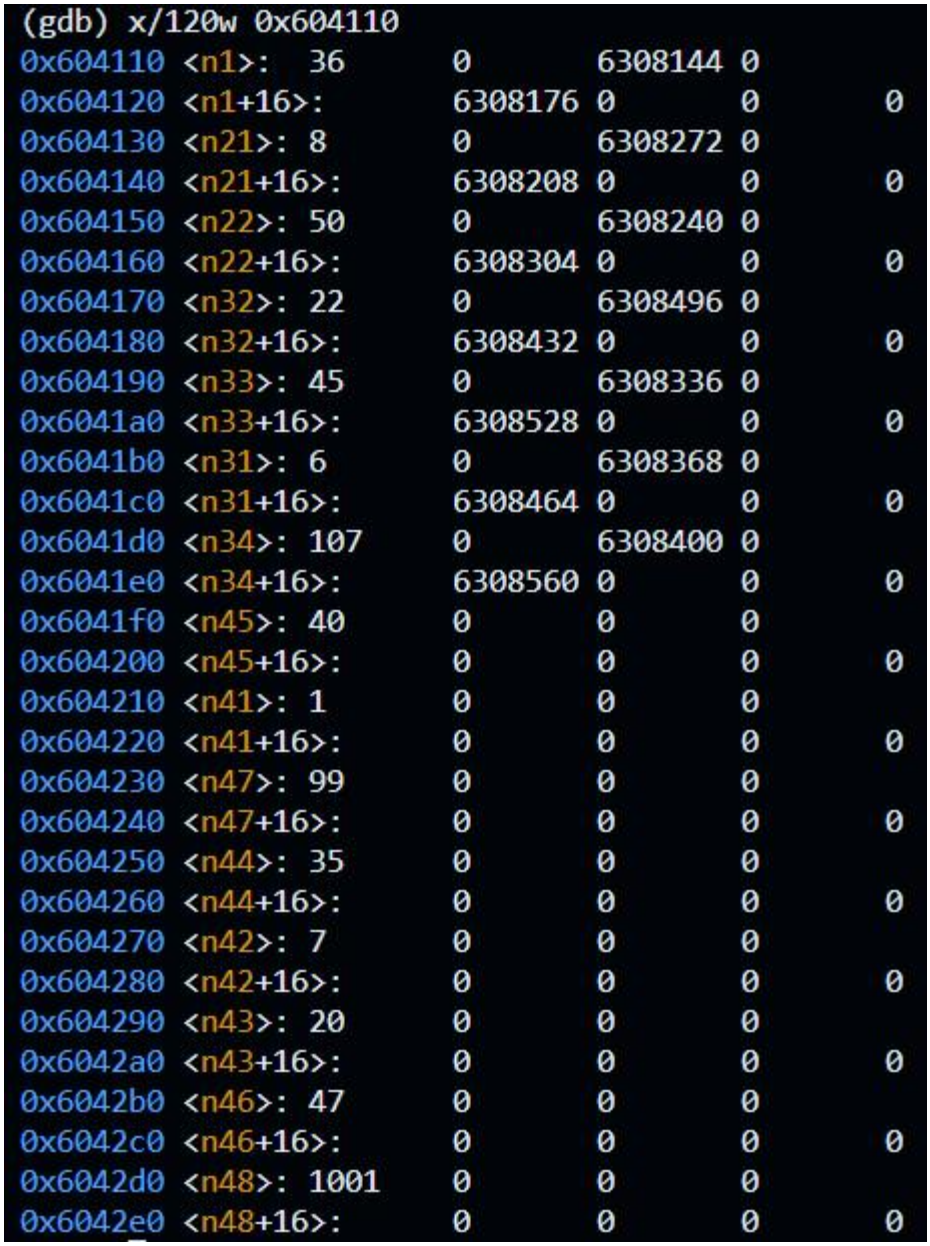
```
Dump of assembler code for function secret_phase:
   0x0000000000401223 <+0>:     push   %rbx
   0x0000000000401224 <+1>:     call   0x4015ec <read_line>
   0x0000000000401229 <+6>:     mov    $0xa,%edx
   0x000000000040122e <+11>:    mov    $0x0,%esi
   0x0000000000401233 <+16>:    mov    %rax,%rdi
   0x0000000000401236 <+19>:    call   0x400c00 <strtol@plt>
   0x000000000040123b <+24>:    mov    %rax,%rbx
   0x000000000040123e <+27>:    lea    -0x1(%rax),%eax
   0x0000000000401241 <+30>:    cmp    $0x3e8,%eax
   0x0000000000401246 <+35>:    jbe    0x40124d <secret_phase+42>
   0x0000000000401248 <+37>:    call   0x401574 <explode_bomb>
   0x000000000040124d <+42>:    mov    %ebx,%esi
   0x000000000040124f <+44>:    mov    $0x604110,%edi
   0x0000000000401254 <+49>:    call   0x4011e5 <fun7>
   0x0000000000401259 <+54>:    cmp    $0x4,%eax
   0x000000000040125c <+57>:    je     0x401263 <secret_phase+64>
   0x000000000040125e <+59>:    call   0x401574 <explode_bomb>
   0x0000000000401263 <+64>:    mov    $0x4024f0,%edi
   0x0000000000401268 <+69>:    call   0x400b40 <puts@plt>
   0x000000000040126d <+74>:    call   0x401712 <phase_defused>
   0x0000000000401272 <+79>:    pop    %rbx
   0x0000000000401273 <+80>:    ret
End of assembler dump.
```

분석해보면, %rsi에 입력값이 하나 들어가고 fun7을 호출한 결과값이 4와 같아야 한다. %rdi에 수상한 주소가 들어가 있다. fun7을 확인하자.



```
Dump of assembler code for function fun7:
   0x00000000004011e5 <+0>:     sub    $0x8,%rsp
   0x00000000004011e9 <+4>:     test   %rdi,%rdi
   0x00000000004011ec <+7>:     je     0x401219 <fun7+52>
   0x00000000004011ee <+9>:     mov    (%rdi),%edx
   0x00000000004011f0 <+11>:    cmp    %esi,%edx
   0x00000000004011f2 <+13>:    jle    0x401201 <fun7+28>
   0x00000000004011f4 <+15>:    mov    0x8(%rdi),%rdi
   0x00000000004011f8 <+19>:    call   0x4011e5 <fun7>
   0x00000000004011fd <+24>:    add    %eax,%eax
   0x00000000004011ff <+26>:    jmp    0x40121e <fun7+57>
   0x0000000000401201 <+28>:    mov    $0x0,%eax
   0x0000000000401206 <+33>:    cmp    %esi,%edx
   0x0000000000401208 <+35>:    je     0x40121e <fun7+57>
   0x000000000040120a <+37>:    mov    0x10(%rdi),%rdi
   0x000000000040120e <+41>:    call   0x4011e5 <fun7>
   0x0000000000401213 <+46>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401217 <+50>:    jmp    0x40121e <fun7+57>
   0x0000000000401219 <+52>:    mov    $0xffffffff,%eax
   0x000000000040121e <+57>:    add    $0x8,%rsp
   0x0000000000401222 <+61>:    ret
End of assembler dump.
```

(%rdi)와 %rsi의 대소에 따라 %rdi는 0x8(%rdi) 또는 0x10(%rdi)로 값이 바뀐채로 재귀호출된다. (%rdi+8)이나 (%rdi+16)은 또 다른 주소를 가리킬 것이라 생각할 수 있다. 아까 봤던 출력 타입을 바꿔가며 수상한 주소를 살펴본 결과는 아래와 같다.



```
(gdb) x/120w 0x604110
0x604110 <n1>:  36      0       6308144 0
0x604120 <n1+16>:       6308176 0       0       0
0x604130 <n21>: 8       0       6308272 0
0x604140 <n21+16>:      6308208 0       0       0
0x604150 <n22>: 50      0       6308240 0
0x604160 <n22+16>:      6308304 0       0       0
0x604170 <n32>: 22      0       6308496 0
0x604180 <n32+16>:      6308432 0       0       0
0x604190 <n33>: 45      0       6308336 0
0x6041a0 <n33+16>:      6308528 0       0       0
0x6041b0 <n31>: 6       0       6308368 0
0x6041c0 <n31+16>:      6308464 0       0       0
0x6041d0 <n34>: 107     0       6308400 0
0x6041e0 <n34+16>:      6308560 0       0       0
0x6041f0 <n45>: 40      0       0       0
0x604200 <n45+16>:      0       0       0       0
0x604210 <n41>: 1       0       0       0
0x604220 <n41+16>:      0       0       0       0
0x604230 <n47>: 99      0       0       0
0x604240 <n47+16>:      0       0       0       0
0x604250 <n44>: 35      0       0       0
0x604260 <n44+16>:      0       0       0       0
0x604270 <n42>: 7       0       0       0
0x604280 <n42+16>:      0       0       0       0
0x604290 <n43>: 20      0       0       0
0x6042a0 <n43+16>:      0       0       0       0
0x6042b0 <n46>: 47      0       0       0
0x6042c0 <n46+16>:      0       0       0       0
0x6042d0 <n48>: 1001    0       0       0
0x6042e0 <n48+16>:      0       0       0       0
```

n1을 root node로 가지는 이진 탐색 트리임을 알 수 있다. 각 노드는 값을 세 개 가지고 있는데 차례대로 노드의 값, 왼쪽 자식의 주소, 오른쪽 자식의 주소임을 알 수 있다. n21에서 왼쪽 자식으로 가면 n31이 되고 오른쪽 자식으로 가면 n32가 된다. 이를 인지한 채로 fun7을 다시 보자. root node인 n1부터 시작해 입력값과 같아지는 지점 혹은 leaf node에 도달할 때까지 자식들로 이동하며 트리 탐색을 한다. 조건에 맞는 node에 도달하고 난 후에는 왔던 경로로 다시 되돌아 가는데, 이 때 자신이 오른쪽 자식이었다면 2*%rax+1을 리턴하고, 왼쪽 자식이었다면 2*%rax를 리턴한다. 만약 leaf node까지 갔는데도 같은 값이 없다면 0xffffffff를 리턴하게 되므로 원하는 값인 4를 맞추지 못하게 된다. **%rax는 0에서 시작하며, 한 노드에서 root node까지 갔을 때 결과가 4가 나오게 하는 노드의 값을 찾으면 된다. 이는 root node 기준으로 왼쪽, 왼쪽, 오른쪽으로 내려가면 되고, 이 때의 값은 7이 된다.**

3. 느낀 점

어셈블리어를 제대로 공부할 수 있었다. 어셈블리어를 봤을 때 대략 지역변수, 반복문 위치와 역할, 조건 등을 알 수 있게 됐다. 또한, GDB 디버거에는 생각보다 다양하고 편리한 기능이 많음을 느꼈다.