Bomb Lab: Defusing a Binary Bomb

Report

20220127 임유진

이번 Lab 에서는 Binary Bomb 을 해체하는 것을 목표로 하며, Phase 별 풀이는 다음과 같다.

Phase 1)

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x0000000000400ef0 <+0>:     sub    $0x8,%rsp
   0x0000000000400ef4 <+4>:     mov    $0x4024fc,%esi
   0x0000000000400ef9 <+9>:     callq  0x40133e <strings_not_equal>
   0x0000000000400efe <+14>:    test   %eax,%eax
   0x0000000000400f00 <+16>:    je     0x400f07 <phase_1+23>
   0x0000000000400f02 <+18>:    callq  0x4015a4 <explode_bomb>
   0x0000000000400f07 <+23>:    add    $0x8,%rsp
   0x0000000000400f0b <+27>:    retq
End of assembler dump.
```

phase_1 함수를 disassemble 해본 결과는 위와 같다.

```
0x0000000000400e45 <+136>:   callq  0x40161c <read_line>
0x0000000000400e4a <+141>:   mov    %rax,%rdi
0x0000000000400e4d <+144>:   callq  0x400ef0 <phase_1>
```

main 함수를 disassemble 해보면 phase 1 함수가 호출되기 직전 read_line 함수가 호출되며, 이후 함수의 반환값을 저장하는 %rax 레지스터의 값을 %rdi 레지스터로 복사한 후 phase_1 함수의 호출이 일어나는 것을 확인할 수 있다.

```
Dump of assembler code for function strings_not_equal:
   0x000000000040133e <+0>:     push   %r12
   0x0000000000401340 <+2>:     push   %rbp
   0x0000000000401341 <+3>:     push   %rbx
   0x0000000000401342 <+4>:     mov    %rdi,%rbx
   0x0000000000401345 <+7>:     mov    %rsi,%rbp
```

phase_1 에서 호출되는 strings_not_equal 함수를 disassemble 해보면 각각 함수의 첫번째, 두번째 인자를 저장하는 데 사용되는 %rdi, %rsi 레지스터의 값을 각각 %rbx, %rbp 레지스터로 복사하는 것을 볼 수 있는데, 이를 통해 이 함수가 두 개의 인자를 받음을 유추할 수 있다. phase 1 에서 “abcdefg”라는 아무 문자열을 입력하고 r command 를 통해 실행시킨 후 strings_not_equal 의 호출 직전 %rdi 와 %rsi 레지스터의 값을 확인해보자.

```
(gdb) x/s 0x6047c0
0x6047c0 <input_strings>:       "abcdefg"
(gdb)
```

```
(gdb) x/s 0x4024fc
0x4024fc:       "Wow! Brazil is big."
(gdb)
```

%rdi 레지스터는 0x6047c0 의 값을 저장하고 있음을 볼 수 있고, 해당 주소의 memory content 를 x/s command 를 이용해 확인해보면 입력한 문자열이 저장되어 있음을 확인할 수 있다.

phase_1 함수에서 %esi 레지스터에 $0x4024fc 를 복사한 후 strings_not_equal 함수의 호출이 일어난다. 메모리의 0x4024fc 주소에는 “Wow! Brazil is big.”이 저장되어 있음을 확인할 수 있다. 따라서 입력한 문자열의 주소와 “Wow! Brazil is big.” 문자열의 주소가 strings_not_equal 의 인자가 된다.

phase_1 함수에서는 strings_not_equal 의 호출 이후 test %eax, %eax 의 연산 후 je, 즉 zero flag 가 0 일 경우 explode_bomb 함수의 호출을 건너뛰어 phase_1 을 풀 수 있게 되는데, strings_not_eqaul 함수를 disassemble 해보면 함수 이름에서 유추할 수 있듯이 인자로 받은 주소에 저장된 두 개의 문자열이 같은 경우 0 을 다른 경우 1 을 반환하는 함수임을 확인할 수 있으므로 phase 1 의 정답이 “Wow! Brazil is big.”임을 알 수 있다.

```
(gdb) r
Starting program: /home/std/limyoojin/hw3/bomb29/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
```

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  0x4015da <read_six_numbers>
   0x0000000000400f1a <+14>:    cmpl   $0x0,(%rsp)
   0x0000000000400f1e <+18>:    jns    0x400f44 <phase_2+56>
   0x0000000000400f20 <+20>:    callq  0x4015a4 <explode_bomb>
   0x0000000000400f25 <+25>:    jmp    0x400f44 <phase_2+56>
   0x0000000000400f27 <+27>:    mov    %ebx,%eax
   0x0000000000400f29 <+29>:    add    -0x4(%rbp),%eax
   0x0000000000400f2c <+32>:    cmp    %eax,0x0(%rbp)
   0x0000000000400f2f <+35>:    je     0x400f36 <phase_2+42>
   0x0000000000400f31 <+37>:    callq  0x4015a4 <explode_bomb>
   0x0000000000400f36 <+42>:    add    $0x1,%ebx
   0x0000000000400f39 <+45>:    add    $0x4,%rbp
   0x0000000000400f3d <+49>:    cmp    $0x6,%ebx
   0x0000000000400f40 <+52>:    jne    0x400f27 <phase_2+27>
   0x0000000000400f42 <+54>:    jmp    0x400f50 <phase_2+68>
   0x0000000000400f44 <+56>:    lea    0x4(%rsp),%rbp
   0x0000000000400f49 <+61>:    mov    $0x1,%ebx
   0x0000000000400f4e <+66>:    jmp    0x400f27 <phase_2+27>
   0x0000000000400f50 <+68>:    add    $0x28,%rsp
   0x0000000000400f54 <+72>:    pop    %rbx
   0x0000000000400f55 <+73>:    pop    %rbp
   0x0000000000400f56 <+74>:    retq
End of assembler dump.
```

phase_2 함수를 disassemble 해본 결과는 위와 같다.

phase_2 함수에서는 스택 포인터의 값을 %rsi 로 복사한 후 read_six_numbers 함수의 호출이 일어난다. read_six_numbers 함수를 disassemble 해보면 sscanf 함수의 호출이 일어나고 있음을 살펴볼 수 있다. sscanf 의 두번째 인자가 데이터의 format 을 나타내는 문자열이므로, sscanf 의 호출 직전에서 함수의 두번째 인자를 저장하는 %rsi 레지스터가 가리키는 주소 0x402809 에 저장된 문자열을 출력해보면 “%d %d %d %d %d %d”를 얻을 수 있다. 이를 통해 read_six_numbers 라는 이름에서 유추할 수 있었듯이 phase 2 에서는 6 개의 정수를 입력해야 함을 알 수 있다. 또한 sscanf 의 반환값이 읽어들인 필드 수이며, 반환값을 저장하는 레지스터가 %rax 임을 고려할 때, read_six_numbers 함수에서는 읽어들인 정수의 개수가 6 개보다 적은 경우 explode_bomb 함수의 호출이 일어나게 됨을 확인할 수 있다.

phase_2 에서 read_six_numbers 함수를 통해 정수를 입력받은 후의 어셈블리어를 분석해보면, %rsp 가 가리키는 메모리 주소의 값과 0 을 비교하고 있으며, cmpl $0x0, (%rsp)에서 jns 인스트럭션을 통해 sign_flag 가 0 인 경우에 explode_bomb 을 뛰어넘게 되므로 %rsp 가 가리키는 메모리 주소에 저장된 값이 0 이상이어야 함을 알 수 있다. 해당 지점에서 %rsp 가 가리키는 메모리 주소 0x7fffffffe320 에 저장되는 값을 확인하기 위해 phase 2 에서 임의로 1 2 3 4 5 6 의 정수를 입력하고 해당 인스트럭션에 브레이크 포인트를 설정하여 실행시켜보았다.

```
(gdb) x/wx 0x7fffffffe320
0x7fffffffe320: 0x00000001
(gdb) x/wx 0x7fffffffe324
0x7fffffffe324: 0x00000002
(gdb) x/wx 0x7fffffffe328
0x7fffffffe328: 0x00000003
(gdb) x/wx 0x7fffffffe32c
0x7fffffffe32c: 0x00000004
(gdb) x/wx 0x7fffffffe330
0x7fffffffe330: 0x00000005
(gdb) x/wx 0x7fffffffe334
0x7fffffffe334: 0x00000006
```

0x7fffffffe320 부터 주소를 4 바이트씩 증가시키며 저장된 값을 확인한 결과 phase 2 에서 입력한 1 2 3 4 5 6 의 정수가 순서대로 저장되어 있음을 확인할 수 있었다. 따라서 cmpl $0x0, (%rsp)의 인스트럭션이 phase2 에서 입력하는 첫번째 정수가 0 이상이어야 함을 나타냄을 알 수 있었다.

이어서 phase_2 의 어셈블리어를 분석해보면 첫번째 정수가 조건을 만족하는 경우 〈phase_2 + 56〉의 인스트럭션으로 점프하게 되고, %rsp 값에 4 를 더한 값을 %rbp 에 저장하여 %rbp 가 입력한 두번째 정수가 저장된 메모리 주소를 가리키게 하고 %ebx 에는 1 을 저장한 후 〈phase_2 + 27〉으로 점프하는 것을 확인할 수 있다. 그 이후부터는 반복이 이루어짐을 확인할 수 있는데, %ebx 의 값을 %eax 에 복사하고, %rbp 값에 4 를 뺀 메모리 주소에 저장된 값을 %eax 에 더해준 후 %eax 의 값과 비교하여 같다면 explode_bomb 함수 호출을 건너뛰고, %ebx 의 값을 1, %rbp 의 값을 4 증가시킨 후 %ebx 가 6 과 같지 않으면 〈phase_2 + 27〉로 다시 점프한다. 이를 해석해보면 %ebx 가 1 부터 5 가 될 때까지 총 5 번의 반복이 이루어지며 이때 %eax 는 %ebx 와 같은 값을 가지므로, %ebx 의 값을 i 라고 할 때 (%rsp + 4 x i)의 메모리 주소에 저장된 값이 (%rsp + 4 x(i − 1))메모리 주소의 값 + i 의 연산 결과와 같아야 함을 확인할 수 있다.

```
Starting program: /home/std/limyoojin/hw3/bomb29/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
```

첫번째 숫자의 조건은 0 이상이었으므로 첫번째 숫자를 1 로 정하게 되면, 위의 조건을 만족하는 6 개의 정수는 1 2 4 7 11 16 이 됨을 알 수 있다.

Phase 3)

```
(gdb) disas phase_3
Dump of assembler code for function phase_3:
   0x0000000000400f57 <+0>:     sub    $0x18,%rsp
   0x0000000000400f5b <+4>:     lea    0x8(%rsp),%rcx
   0x0000000000400f60 <+9>:     lea    0xc(%rsp),%rdx
   0x0000000000400f65 <+14>:    mov    $0x402815,%esi
   0x0000000000400f6a <+19>:    mov    $0x0,%eax
   0x0000000000400f6f <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x0000000000400f74 <+29>:    cmp    $0x1,%eax
   0x0000000000400f77 <+32>:    jg     0x400f7e <phase_3+39>
   0x0000000000400f79 <+34>:    callq  0x4015a4 <explode_bomb>
   0x0000000000400f7e <+39>:    cmpl   $0x7,0xc(%rsp)
   0x0000000000400f83 <+44>:    ja     0x400fe9 <phase_3+146>
   0x0000000000400f85 <+46>:    mov    0xc(%rsp),%eax
   0x0000000000400f89 <+50>:    jmpq   *0x402520(,%rax,8)
   0x0000000000400f90 <+57>:    mov    $0x0,%eax
   0x0000000000400f95 <+62>:    jmp    0x400f9c <phase_3+69>
   0x0000000000400f97 <+64>:    mov    $0x2b5,%eax
   0x0000000000400f9c <+69>:    sub    $0x13a,%eax
   0x0000000000400fa1 <+74>:    jmp    0x400fa8 <phase_3+81>
   0x0000000000400fa3 <+76>:    mov    $0x0,%eax
   0x0000000000400fa8 <+81>:    add    $0x4c,%eax
   0x0000000000400fab <+84>:    jmp    0x400fb2 <phase_3+91>
   0x0000000000400fad <+86>:    mov    $0x0,%eax
   0x0000000000400fb2 <+91>:    sub    $0x212,%eax
   0x0000000000400fb7 <+96>:    jmp    0x400fbe <phase_3+103>
   0x0000000000400fb9 <+98>:    mov    $0x0,%eax
   0x0000000000400fbe <+103>:   add    $0x212,%eax
   0x0000000000400fc3 <+108>:   jmp    0x400fca <phase_3+115>
   0x0000000000400fc5 <+110>:   mov    $0x0,%eax
   0x0000000000400fca <+115>:   sub    $0x212,%eax
   0x0000000000400fcf <+120>:   jmp    0x400fd6 <phase_3+127>
   0x0000000000400fd1 <+122>:   mov    $0x0,%eax
   0x0000000000400fd6 <+127>:   add    $0x212,%eax
   0x0000000000400fdb <+132>:   jmp    0x400fe2 <phase_3+139>
   0x0000000000400fdd <+134>:   mov    $0x0,%eax
   0x0000000000400fe2 <+139>:   sub    $0x212,%eax
   0x0000000000400fe7 <+144>:   jmp    0x400ff3 <phase_3+156>
   0x0000000000400fe9 <+146>:   callq  0x4015a4 <explode_bomb>
   0x0000000000400fee <+151>:   mov    $0x0,%eax
   0x0000000000400ff3 <+156>:   cmpl   $0x5,0xc(%rsp)
   0x0000000000400ff8 <+161>:   jg     0x401000 <phase_3+169>
   0x0000000000400ffa <+163>:   cmp    0x8(%rsp),%eax
   0x0000000000400ffe <+167>:   je     0x401005 <phase_3+174>
   0x0000000000401000 <+169>:   callq  0x4015a4 <explode_bomb>
   0x0000000000401005 <+174>:   add    $0x18,%rsp
   0x0000000000401009 <+178>:   retq
End of assembler dump.
```

```
(gdb) x/s 0x402815
0x402815:       "%d %d"
```

phase_3 함수를 disassemble 해본 결과는 위와 같으며, sscanf 함수가 호출되고 있으므로 phase 2 에서와 마찬가지로 sscanf 의 두번째 인자인 format string 을 알아내기 위해 %rsi 에 저장되는 0x402825 의 주소에 저장되는 문자열을 출력해보면 “%d %d”가 나오고, 따라서 phase 3 에서는 두 개의 정수를 입력해야 함을 알 수 있다.

sscanf 함수가 호출되기 전 phase_3 함수에서 lea 인스트럭션을 통해 (%rsp 의 값 + 0x8)을 %rcx 에, (%rsp 의 값 + 0xc)을 %rdx 에 복사하고 있음을 확인할 수 있다. %rdx 와 %rcx 는 각각 함수의 3 번째, 4 번째 인자를 저장하는 레지스터이고 sscanf 의 3 번째, 4 번째 인자의 값은 각각 입력하는 첫번째

정수의 주소, 두번째 정수의 주소가 된다. 레지스터의 값을 확인해보면 %rsp 는 0x7fffffffe340 의 값을 가지므로 %rdx 가 저장하는 0x7fffffffe34c 주소의 메모리에 입력하는 첫번째 정수가, %rcx 가 저장하는 0x7fffffffe348 주소의 메모리에 입력하는 두번째 정수가 저장될 것임을 알 수 있다. sscanf 의 호출 이후 반환값, 즉 sscanf 가 입력 받은 필드의 개수가 1 이하인 경우에는 jg 인스트럭션이 실행되지 않아 explode_bomb 이 호출되는데, 이는 앞서 format string 을 통해 확인했던 것과 마찬가지로 두 개의 정수를 입력해야 함을 의미한다. 이후 cmpl $0x7,0xc(%rsp)이 실행되고 ja 인스트럭션이 실행되면 explode_bomb 이 호출되게 되는데, 이는 메모리의 (%rsp 의 값 + 0xc) 주소에 저장된 값인 입력한 첫번째 정수의 값이 7 보다 같거나 작아야 함을 의미한다. 이후 입력한 첫번째 정수의 값을 %eax 에 복사하며, indirect jump 를 수행하게 된다. jmpq *0x402520(, %rax, 8)의 indirect jump 인스트럭션은 메모리의 (0x402520 + %rax * 8) 주소에 저장된 값을 점프 목적지로 읽어들여 점프한다.

```
(gdb) x/wx 0x402528
0x402528:       0x00400f90
(gdb) x/wx 0x402530
0x402530:       0x00400fa3
(gdb) x/wx 0x402538
0x402538:       0x00400fad
(gdb) x/wx 0x402540
0x402540:       0x00400fb9
(gdb) x/wx 0x402548
0x402548:       0x00400fc5
(gdb) x/wx 0x402550
0x402550:       0x00400fd1
(gdb) x/wx 0x402558
0x402558:       0x00400fdd
```

직전에 첫번째 입력한 정수의 값을 %eax 레지스터에 복사했으므로 첫번째 입력한 정수의 값에 따라 점프 목적지가 달라지게 됨을 알 수 있다. 첫번째 입력한 정수를 7 이하의 조건에 따라 0~7 사이의 수로 가정하고, 각각의 경우에 점프 목적지를 x/wx command 를 통해 읽어보았다. 왼쪽과 같은 결과를 얻을 수 있었으며, 첫번째 정수를 4 로 선택하여 해당 경우에 대해 더 분석해보기로 하였다.

첫번째 수로 4 를 입력한 경우, jmpq 인스트럭션을 통해 0x00400fb9 로 점프하게 된다. 이후의 인스트럭션을 분석해보면 우선 %eax 에 0x0 을 복사한 후 add, jmp, sub 의 인스트럭션을 통해 %eax 레지스터의 값을 변경시켜주고 있으며, 이와 같은 연산을 마친 후 조건 제어 인스트럭션이 실행되며 이때 %eax 레지스터 값은 최종적으로 0x0 + 0x212 – 0x212 + 0x212 – 0x212 = 0x0 이 된다.

cmpl $0x5,0xc(%rsp) 인스트럭션 이후 jg 인스트럭션의 점프 조건이 만족되는 경우 explode_bomb 의 호출 인스트럭션으로 이동하게 되므로, 첫번째 입력한 정수의 값이 5 이하여야 했음을 알 수 있다. 해당 경우는 첫번째 정수를 4 로 선택했으므로 점프가 이루어지지 않으며, 다음 조건 제어 인스트럭션을 보면 cmp 0x8(%rsp), %eax 인스트럭션 이후 je 인스트럭션의 점프 조건이 만족되는 경우 explode_bomb 호출을 뛰어넘어 phase 3 을 해결할 수 있다. 따라서 두번째 입력한 정수의 값과 %eax 레지스터의 값이 같아야 함을 알 수 있고, 앞서 계산했듯 이는 0 이다.

```
Starting program: /home/std/limyoojin/hw3/bomb29/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
4 0
Halfway there!
```

phase 3 는 첫번째 입력한 정수의 값에 따라 점프 목적지가 다르기 때문에 %eax 레지스터가 최종적으로 가지게 되는 값 또한 다르게 되어, 여러가지 답이 존재할 수 있고 그 중 하나의 답이 4 0 이 된다.

Phase 4)

```
(gdb) disas phase_4
Dump of assembler code for function phase_4:
   0x0000000000401048 <+0>:     sub    $0x18,%rsp
   0x000000000040104c <+4>:     lea    0x8(%rsp),%rcx
   0x0000000000401051 <+9>:     lea    0xc(%rsp),%rdx
   0x0000000000401056 <+14>:    mov    $0x402815,%esi
   0x000000000040105b <+19>:    mov    $0x0,%eax
   0x0000000000401060 <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x0000000000401065 <+29>:    cmp    $0x2,%eax
   0x0000000000401068 <+32>:    jne    0x401071 <phase_4+41>
   0x000000000040106a <+34>:    cmpl   $0xe,0xc(%rsp)
   0x000000000040106f <+39>:    jbe    0x401076 <phase_4+46>
   0x0000000000401071 <+41>:    callq  0x4015a4 <explode_bomb>
   0x0000000000401076 <+46>:    mov    $0xe,%edx
   0x000000000040107b <+51>:    mov    $0x0,%esi
   0x0000000000401080 <+56>:    mov    0xc(%rsp),%edi
   0x0000000000401084 <+60>:    callq  0x40100a <func4>
   0x0000000000401089 <+65>:    cmp    $0x1,%eax
   0x000000000040108c <+68>:    jne    0x401095 <phase_4+77>
   0x000000000040108e <+70>:    cmpl   $0x1,0x8(%rsp)
   0x0000000000401093 <+75>:    je     0x40109a <phase_4+82>
   0x0000000000401095 <+77>:    callq  0x4015a4 <explode_bomb>
   0x000000000040109a <+82>:    add    $0x18,%rsp
   0x000000000040109e <+86>:    retq
End of assembler dump.
```

```
(gdb) x/s 0x402815
0x402815:       "%d %d"
```

phase_4 함수를 disassemble 해보면 sscanf 가 호출되는 것을 확인할 수 있으며, sscanf 의 두번째 인자인 format string 을 알아내기 위해 %rsi 에 저장되는 0x402815 의 주소에 저장되는 문자열을 출력해보면 “%d %d”가 나오고, 따라서 phase 4 에서 두 개의 정수를 입력해야 함을 알 수 있다. sscanf 에 3 번째, 4 번째 인자를 저장하는 %rdx, %rcx 에는 각각 (%rsp + 0xc), (%rsp + 0x8)이 저장되며, 따라서 메모리의 (%rsp + 0xc), (%rsp + 0x8) 주소에 각각 입력한 첫번째 정수와 두번째 정수가 저장됨을 알 수 있다. sscanf 의 호출 이후 반환값, 즉 읽어들인 필드의 개수가 2 개가 아닌 경우 explode_bomb 함수가 호출되며 이는 앞서 확인했듯이 2 개의 값을 입력해야 함을 의미한다. 이후 메모리의 (%rsp + 0xc) 주소에 저장된 값인 입력한 첫번째 정수가 0xe 보다 같거나 작은 경우에 explode_bomb 의 호출을 뛰어넘는다. 따라서 입력하는 첫번째 정수의 값은 0xe 이하여야 한다. 다음으로 %edx 에 0xe, %esi 에 0x0, %edi 에 입력한 첫번째 정수값을 복사하고 func4 함수를 호출하게 된다. 함수 호출 후의 어셈블리어를 살펴보면 %eax 와 0x1 을 비교하여 같지 않으면 explode_bomb 의 호출 인스트럭션으로 점프하며, 따라서 func4 함수의 반환값이 1 이 되어야 함을 확인할 수 있다. 다음으로 메모리의 (%rsp + 0x8) 주소에 저장된 값, 즉 두번째로 입력한 정수 값과 0x1 을 비교하여 같은 경우 explode_bomb 호출 인스트럭션을 건너뛰고 함수 종료부로 넘어감을 확인할 수 있다. 따라서 두번째 입력 값은 1 이 되어야 함을 알 수 있다.

```
(gdb) disas func4
Dump of assembler code for function func4:
   0x000000000040100a <+0>:     sub    $0x8,%rsp
   0x000000000040100e <+4>:     mov    %edx,%eax
   0x0000000000401010 <+6>:     sub    %esi,%eax
   0x0000000000401012 <+8>:     mov    %eax,%ecx
   0x0000000000401014 <+10>:    shr    $0x1f,%ecx
   0x0000000000401017 <+13>:    add    %ecx,%eax
   0x0000000000401019 <+15>:    sar    %eax
   0x000000000040101b <+17>:    lea    (%rax,%rsi,1),%ecx
   0x000000000040101e <+20>:    cmp    %edi,%ecx
   0x0000000000401020 <+22>:    jle    0x40102e <func4+36>
   0x0000000000401022 <+24>:    lea    -0x1(%rcx),%edx
   0x0000000000401025 <+27>:    callq  0x40100a <func4>
   0x000000000040102a <+32>:    add    %eax,%eax
   0x000000000040102c <+34>:    jmp    0x401043 <func4+57>
   0x000000000040102e <+36>:    mov    $0x0,%eax
   0x0000000000401033 <+41>:    cmp    %edi,%ecx
   0x0000000000401035 <+43>:    jge    0x401043 <func4+57>
   0x0000000000401037 <+45>:    lea    0x1(%rcx),%esi
   0x000000000040103a <+48>:    callq  0x40100a <func4>
   0x000000000040103f <+53>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401043 <+57>:    add    $0x8,%rsp
   0x0000000000401047 <+61>:    retq
End of assembler dump.
```

첫번째 입력값의 조건을 알아내기 위해 func4 함수를 disassemble 해보면 우선 func4 함수는 함수 내부에서 다시 func4 함수의 호출이 이루어지는 재귀적인 함수이다. 또한 %rdx, %rsi, %rdi 레지스터가 인자로 받은 값을 저장하고 있음을 유추할 수 있고, phase_4 에서의 분석을 통해 func4 가 처음 호출될 때 첫번째 인자가 입력한 첫번째 정수, 두번째 인자가 0x0, 세번째 인자가 0xe 임을 알 수 있다. func4 의 어셈블리어를 분석해보면 func4 에서는 %ecx 의 값이 %edi 의 값보다 큰 경우 세번째 인자의 값을 변경하여 func4 를 다시 호출하며 그 반환 값의 두 배를 반환하며, %ecx 의 값이 %edi 의 값보다 작은 경우 두번째 인자의 값을 변경하여 func4 를 다시 호출하며 그 반환값의 2 배에 0x1 을 더한 값을 반환한다. 마지막으로 %ecx 의 값이 %edi 와 같은 경우에는 0 을 반환함을 알 수 있다. func4 는 가장 마지막으로 호출된 func4 가 0 을 반환할 때까지 재귀적으로 호출될 것이며, phase_4 에서 func4 함수의 반환 값이 1 이 되어야 함을 고려할 때 가장 간단한 해답은 첫번째로 호출된 func4 가 func4 를 다시 호출하며 그 반환값의 2 배에 0x1 을 더한 값을 반환하며, 그 안에서 두번째로 호출되는 func4 가 0 을 반환하는 것이다. func4 가 처음 호출될 때의 값을 통해 연산해보면 조건 제어를 통해 분기가 일어나기 직전 %rdi, %rsi, %rdx, %rax, %rcx 는 각각 입력한 첫번째 정수, 0x0, 0xe, 7, 7 의 값을 가질 것이다. 이때 위에서 설명한 것과 같은 조건 분기가 일어나기 위해서 %rdi 는 7 보다 큰 값을 가져야 한다. 두번째로 호출되는 func4 는 첫번째로 입력한 정수, 0x8, 0xe 를 인자로 호출된다. 두번째로 호출되는 func4 에서 조건 제어로 분기가 일어나기 직전 %rdi, %rsi, %rdx, %rax, %rcx 는 각각 첫번째 입력한 정수, 0x8. 0xe, 0x3, 0xb 이 되며, 두번째로 호출되는 func4 의 반환 값이 0 이 되기 위해서는 %rdi, 즉 첫번째로 입력한 정수의 값이 0xb, 즉 11 이 되어야 한다.

```
Starting program: /home/std/limyoojin/hw3/bomb29/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
4 0
Halfway there!
11 1
So you got that one.  Try this one.
```

이를 통해 phase 4 를 풀 수 있는 정답이 11 1 이 됨을 알 수 있다.

Phase 5)

```
(gdb) disas phase_5
Dump of assembler code for function phase_5:
   0x000000000040109f <+0>:     sub    $0x18,%rsp
   0x00000000004010a3 <+4>:     lea    0x8(%rsp),%rcx
   0x00000000004010a8 <+9>:     lea    0xc(%rsp),%rdx
   0x00000000004010ad <+14>:    mov    $0x402815,%esi
   0x00000000004010b2 <+19>:    mov    $0x0,%eax
   0x00000000004010b7 <+24>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x00000000004010bc <+29>:    cmp    $0x1,%eax
   0x00000000004010bf <+32>:    jg     0x4010c6 <phase_5+39>
   0x00000000004010c1 <+34>:    callq  0x4015a4 <explode_bomb>
   0x00000000004010c6 <+39>:    mov    0xc(%rsp),%eax
   0x00000000004010ca <+43>:    and    $0xf,%eax
   0x00000000004010cd <+46>:    mov    %eax,0xc(%rsp)
   0x00000000004010d1 <+50>:    cmp    $0xf,%eax
   0x00000000004010d4 <+53>:    je     0x401102 <phase_5+99>
   0x00000000004010d6 <+55>:    mov    $0x0,%ecx
   0x00000000004010db <+60>:    mov    $0x0,%edx
   0x00000000004010e0 <+65>:    add    $0x1,%edx
   0x00000000004010e3 <+68>:    cltq
   0x00000000004010e5 <+70>:    mov    0x402560(,%rax,4),%eax
   0x00000000004010ec <+77>:    add    %eax,%ecx
   0x00000000004010ee <+79>:    cmp    $0xf,%eax
   0x00000000004010f1 <+82>:    jne    0x4010e0 <phase_5+65>
   0x00000000004010f3 <+84>:    mov    %eax,0xc(%rsp)
   0x00000000004010f7 <+88>:    cmp    $0xf,%edx
   0x00000000004010fa <+91>:    jne    0x401102 <phase_5+99>
   0x00000000004010fc <+93>:    cmp    0x8(%rsp),%ecx
   0x0000000000401100 <+97>:    je     0x401107 <phase_5+104>
   0x0000000000401102 <+99>:    callq  0x4015a4 <explode_bomb>
   0x0000000000401107 <+104>:   add    $0x18,%rsp
   0x000000000040110b <+108>:   retq
End of assembler dump.
```

```
(gdb) x/s 0x402815
0x402815:       "%d %d"
```

phase_5 함수를 disassemble 해보면 sscanf 가 호출되는 것을 확인할 수 있으며, sscanf 의 두번째 인자인 format string 을 알아내기 위해 %rsi 에 저장되는 0x402815 의 주소에 저장되는 문자열을 출력해보면 “%d %d”가 나오고, 따라서 phase 5 에서 두 개의 정수를 입력해야 함을 알 수 있다. 또한 Phase 3, Phase 4 와 마찬가지로 sscanf 의 3 번째, 4 번째 인자를 저장하는 %rdx, %rcx 에 각각 (%rsp + 0xc), (%rsp + 0x8)이 저장되어 메모리의 (%rsp + 0xc), (%rsp + 0x8) 주소에 각각 입력한 첫번째 정수와 두번째 정수가 저장된다. 또한 sscanf 의 반환값, 즉 읽어들인 필드의 개수가 1 이하일 경우 explode_bomb 이 호출된다. 다음으로 첫번째로 입력한 정수의 값을 %eax 로 옮기고, 0xf 와의 and 연산을 수행하며 결과값을 기존에 첫번째 입력한 정수가 저장되어 있던 주소에 덮어씌운다. 이로 인해 메모리의 (%rsp + 0xc) 주소에 저장된 값이 0x0~0xf 사이의 값을 가지게 됨을 알 수 있다. 그리고 %eax 의 값을 0xf 와 비교하여 같은 경우 explode_bomb 의 호출 인스트럭션으로 점프하게 된다. 이를 통해 입력하는 첫번째 정수의 하위 4 비트가 모두 1 이여서는 안됨을 알 수 있다. 이후에는 우선 %ecx, %edx 에 0x0 을 저장한 후, %edx 의 값을 1 증가시키고, cltq 인스트럭션을 통해 %eax 에 저장된 값을 sign extension 한다. 이후 메모리의 (0x402560 + 4 * %rax) 주소의 값을 %eax 에 저장하고 %ecx 에 %eax 값을 더해준 후 %eax 와 0xf 가 같지 않은 경우 〈phase_5 + 65〉로 점프하여 %edx 의 값을 1 증가시키는 것부터의 인스트럭션을 반복한다. %eax 가 0xf 와 같아지게 되면

다음으로 %edx 와 0xf 를 비교하여 같지 않은 경우 explode_bomb 호출 인스트럭션으로 점프하게 된다. 이는 〈phase_5 + 65〉 ~ 〈phase_5 + 82〉의 인스트럭션이 15 번 반복 실행되어야 함을 의미하며, 따라서 %eax 는 15 번의 반복 후 0xf 의 값을 가지게 되어야 한다.

```
(gdb) x/16wd 0x402560
0x402560 <array.3161>:  10      2       14      7
0x402570 <array.3161+16>:       8       12      15      11
0x402580 <array.3161+32>:       0       4       1       13
0x402590 <array.3161+48>:       3       9       6       5
```

이 반복문에서 %eax 의 값은 메모리의 (0x402560 + 4 * %rax) 주소에 저장된 값이 되므로, 우선 x/16wd command 를 통해 해당 주소 부근에 저장되어 있는 값들을 확인해보았다. (x/32wd 0x402560 의 결과 0x4025a0 부터는 관련되지 않은 값이 저장되어 있음을 확인할 수 있어 위 사진의 주소까지만 확인하였다.) 〈phase_5 + 65〉 ~ 〈phase_5 + 82〉의 인스트럭션이 마지막으로 반복 실행될 때 %eax 가 가져야 하는 값이 0xf, 즉 15 이므로 15 가 저장되어 있는 0x402578 부터 역추적을 해나갔다.

```
%rax = 15
0x402560 + 4 * %rax = 0x402578
-> %rax = 6
0x402560 + 4 * %rax = 0x402598
-> %rax = 14
0x402560 + 4 * %rax = 0x402568
-> %rax = 2
0x402560 + 4 * %rax = 0x402604
-> %rax = 1
0x402560 + 4 * %rax = 0x402588
-> %rax = 10
0x402560 + 4 * %rax = 0x402560
-> %rax = 0
0x402560 + 4 * %rax = 0x402580
-> %rax = 8
0x402560 + 4 * %rax = 0x402570
-> %rax = 4
0x402560 + 4 * %rax = 0x402584
-> %rax = 9
0x402560 + 4 * %rax = 0x402594
-> %rax = 13
0x402560 + 4 * %rax = 0x40258c
-> %rax = 11
0x402560 + 4 * %rax = 0x40257c
-> %rax = 7
0x402560 + 4 * %rax = 0x40256c
-> %rax = 3
0x402560 + 4 * %rax = 0x402590
-> %rax = 12
0x402560 + 4 * %rax = 0x402574
-> %rax = 5
```

왼쪽과 같은 계산을 통해 15 번의 반복 결과 %eax 의 값이 0xf 가 되게 하는 초기 %rax 의 값이 5 임을 발견할 수 있었다. 뿐만 아니라 각 반복에서 %eax 에 메모리의 값을 저장한 후 %ecx 에 더하는 연산이 수행되므로 모든 반복이 끝난 후 %ecx 레지스터에는 12+3+7+11+13+9+4+8+0+10+1+2+14+6+15 =115 의 값이 저장되어 있을 것임을 예상할 수 있다.

```
Starting program: /home/std/limyoojin/hw3/bomb29/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
4 0
Halfway there!
11 1
So you got that one.  Try this one.
5 115
Good work!  On to the next...
```

따라서 두번째로 입력하는 정수가 115 여야 함을 확인할 수 있고, 첫번째 입력하는 정수의 경우 하위 4 비트가 0101 이어야 함을 알 수 있다. 따라서 5 115 가 phase_5 의 정답이 된다.

Phase 6)

```
(gdb) disas phase_6
Dump of assembler code for function phase_6:
   0x000000000040110c <+0>:     push   %r14
   0x000000000040110e <+2>:     push   %r13
   0x0000000000401110 <+4>:     push   %r12
   0x0000000000401112 <+6>:     push   %rbp
   0x0000000000401113 <+7>:     push   %rbx
   0x0000000000401114 <+8>:     sub    $0x50,%rsp
   0x0000000000401118 <+12>:    lea    0x30(%rsp),%r13
   0x000000000040111d <+17>:    mov    %r13,%rsi
   0x0000000000401120 <+20>:    callq  0x4015da <read_six_numbers>
   0x0000000000401125 <+25>:    mov    %r13,%r14
   0x0000000000401128 <+28>:    mov    $0x0,%r12d
   0x000000000040112e <+34>:    mov    %r13,%rbp
   0x0000000000401131 <+37>:    mov    0x0(%r13),%eax
   0x0000000000401135 <+41>:    sub    $0x1,%eax
   0x0000000000401138 <+44>:    cmp    $0x5,%eax
   0x000000000040113b <+47>:    jbe    0x401142 <phase_6+54>
   0x000000000040113d <+49>:    callq  0x4015a4 <explode_bomb>
   0x0000000000401142 <+54>:    add    $0x1,%r12d
   0x0000000000401146 <+58>:    cmp    $0x6,%r12d
   0x000000000040114a <+62>:    je     0x40116e <phase_6+98>
   0x000000000040114c <+64>:    mov    %r12d,%ebx
   0x000000000040114f <+67>:    movslq %ebx,%rax
   0x0000000000401152 <+70>:    mov    0x30(%rsp,%rax,4),%eax
   0x0000000000401156 <+74>:    cmp    %eax,0x0(%rbp)
   0x0000000000401159 <+77>:    jne    0x401160 <phase_6+84>
   0x000000000040115b <+79>:    callq  0x4015a4 <explode_bomb>
   0x0000000000401160 <+84>:    add    $0x1,%ebx
   0x0000000000401163 <+87>:    cmp    $0x5,%ebx
   0x0000000000401166 <+90>:    jle    0x40114f <phase_6+67>
   0x0000000000401168 <+92>:    add    $0x4,%r13
   0x000000000040116c <+96>:    jmp    0x40112e <phase_6+34>
   0x000000000040116e <+98>:    lea    0x48(%rsp),%rsi
   0x0000000000401173 <+103>:   mov    %r14,%rax
   0x0000000000401176 <+106>:   mov    $0x7,%ecx
   0x000000000040117b <+111>:   mov    %ecx,%edx
   0x000000000040117d <+113>:   sub    (%rax),%edx
   0x000000000040117f <+115>:   mov    %edx,(%rax)
   0x0000000000401181 <+117>:   add    $0x4,%rax
   0x0000000000401185 <+121>:   cmp    %rsi,%rax
   0x0000000000401188 <+124>:   jne    0x40117b <phase_6+111>
   0x000000000040118a <+126>:   mov    $0x0,%esi
   0x000000000040118f <+131>:   jmp    0x4011b1 <phase_6+165>
   0x0000000000401191 <+133>:   mov    0x8(%rdx),%rdx
   0x0000000000401195 <+137>:   add    $0x1,%eax
   0x0000000000401198 <+140>:   cmp    %ecx,%eax
   0x000000000040119a <+142>:   jne    0x401191 <phase_6+133>
   0x000000000040119c <+144>:   jmp    0x4011a3 <phase_6+151>
   0x000000000040119e <+146>:   mov    $0x6042f0,%edx
   0x00000000004011a3 <+151>:   mov    %rdx,(%rsp,%rsi,2)
   0x00000000004011a7 <+155>:   add    $0x4,%rsi
   0x00000000004011ab <+159>:   cmp    $0x18,%rsi
   0x00000000004011af <+163>:   je     0x4011c6 <phase_6+186>
   0x00000000004011b1 <+165>:   mov    0x30(%rsp,%rsi,1),%ecx
   0x00000000004011b5 <+169>:   cmp    $0x1,%ecx
   0x00000000004011b8 <+172>:   jle    0x40119e <phase_6+146>
   0x00000000004011ba <+174>:   mov    $0x1,%eax
   0x00000000004011bf <+179>:   mov    $0x6042f0,%edx
   0x00000000004011c4 <+184>:   jmp    0x401191 <phase_6+133>
   0x00000000004011c6 <+186>:   mov    (%rsp),%rbx
   0x00000000004011ca <+190>:   lea    0x8(%rsp),%rax
   0x00000000004011cf <+195>:   lea    0x30(%rsp),%rsi
   0x00000000004011d4 <+200>:   mov    %rbx,%rcx
   0x00000000004011d7 <+203>:   mov    (%rax),%rdx
   0x00000000004011da <+206>:   mov    %rdx,0x8(%rcx)
   0x00000000004011de <+210>:   add    $0x8,%rax
   0x00000000004011e2 <+214>:   cmp    %rsi,%rax
   0x00000000004011e5 <+217>:   je     0x4011ec <phase_6+224>
   0x00000000004011e7 <+219>:   mov    %rdx,%rcx
   0x00000000004011ea <+222>:   jmp    0x4011d7 <phase_6+203>
   0x00000000004011ec <+224>:   movq   $0x0,0x8(%rdx)
   0x00000000004011f4 <+232>:   mov    $0x5,%ebp
   0x00000000004011f9 <+237>:   mov    0x8(%rbx),%rax
   0x00000000004011fd <+241>:   mov    (%rax),%eax
   0x00000000004011ff <+243>:   cmp    %eax,(%rbx)
   0x0000000000401201 <+245>:   jge    0x401208 <phase_6+252>
   0x0000000000401203 <+247>:   callq  0x4015a4 <explode_bomb>
   0x0000000000401208 <+252>:   mov    0x8(%rbx),%rbx
   0x000000000040120c <+256>:   sub    $0x1,%ebp
   0x000000000040120f <+259>:   jne    0x4011f9 <phase_6+237>
   0x0000000000401211 <+261>:   add    $0x50,%rsp
   0x0000000000401215 <+265>:   pop    %rbx
   0x0000000000401216 <+266>:   pop    %rbp
   0x0000000000401217 <+267>:   pop    %r12
   0x0000000000401219 <+269>:   pop    %r13
   0x000000000040121b <+271>:   pop    %r14
   0x000000000040121d <+273>:   retq
```

phase_6 함수를 disassemble 해보면 우선 %r14, %r13, %r12, %rbp, %rbx 레지스터의 데이터를 스택에 추가하고, 스택 포인터의 값을 0x50 감소시킨다. %r13 레지스터에 %rsp + 0x30 의 값을 저장한 후, 이를 %rsi 레지스터에 복사해주고 read_six_numbers 함수의 호출이 일어난다.

이 과정에서 stack pointer 의 변화를 살펴보면 우선 phase_6 함수에서 read_six_numbers 의 호출 직전 %rsp 의 값은 0x7fffffffe2e0 이며, read_six_numbers 를 호출하면서 return address 가 stack 에 push 되어 %rsp 가 8 감소한다. 따라서 read_six_numbers 를 처음 호출하였을 때 %rsp 의 값은 0x7fffffffe2d8 이 되며 이후 0x18 만큼 감소하여 0x7fffffffe2c0 이 된다. 이후 lea 와 mov 인스트럭션을 통해 레지스터들의 값을 변경시켜주는데, 그 결과 함수의 3, 4, 5, 6 번째 인자의 값을 저장하는 %rdx, %rcx, %r8, %r9 의 값이 차례대로 0x7fffffffe310, 0x7fffffffe314, 0x7fffffffe318, 0x7fffffffe31c 이 되며, %rsp, %rsp+8 이 가리키는 메모리에 각각 0x7fffffffe320, %0x7fffffffe324 의 값이 저장된다. 이후 sscanf 호출이 이루어지는데, 이때 format string 을 확인하기 위해 함수의 두번째 인자로 들어가는 레지스터 %rsi 가 저장하고 있는 0x402809 에 저장된 string 을 확인해보면 "%d %d %d %d %d %d"임을 알 수 있고, read_six_numbers 에서 유추할 수 있듯 phase_6 에서는 6 개의 정수를 입력해야 한다. 따라서 sscanf 의 인자가 6 개를 넘어가게 되므로 %rdx, %rcx, %r8, %r9 가 가리키는 주소에 처음 4 개의 정수가 저장되고, 메모리의 %rsp, %rsp + 0x8 주소에 위치한 값이 주소가 되는 메모리에 남은 2 개의 정수가 저장될 것이다.

```
(gdb) x/wx 0x7fffffffe310
0x7fffffffe310: 0x00000001
(gdb)  x/wx 0x7fffffffe314
0x7fffffffe314: 0x00000002
(gdb)  x/wx 0x7fffffffe318
0x7fffffffe318: 0x00000003
(gdb)  x/wx 0x7fffffffe31c
0x7fffffffe31c: 0x00000004
(gdb)  x/wx 0x7fffffffe320
0x7fffffffe320: 0x00000005
(gdb)  x/wx 0x7fffffffe324
0x7fffffffe324: 0x00000006
```

앞서 확인한 각 레지스터에 저장된 값을 통해 이는 메모리의 0x7fffffffe310 주소부터 입력한 6 개의 정수가 저장됨을 의미한다는 것을 알 수 있으며, 임의로 1 2 3 4 5 6 의 6 개의 정수를 입력하고 read_six_numbers 의 호출이 끝난 후 해당 주소의 값을 x/wx command 로 확인해보면 왼쪽과 같이 됨을 통해서 이를 확인해볼 수 있다.

〈phase_6 + 12〉를 통해 %r13 이 입력된 첫번째 정수의 주소를 저장하고 있음을 고려해 〈phase_6 + 25〉부터의 인스트럭션을 분석해보자. %r14, %rbp 에 %r13 의 값을, %r12d 에 0x0 을 복사한 후 %r13 이 가리키는 메모리의 값을 %rax 레지스터에 저장하고 1 을 뺀 후 5 보다 작거나 같은 경우 explode_bomb 을 뛰어넘는다. 이후 %r12d 의 값을 1 증가시키고 6 과 같은 경우 〈phase_6+98〉로 점프하는 것을 확인할 수 있고 반복문이 존재함을 유추할 수 있으며, 이는 〈phase_6+96〉에서 〈phase_6+34〉로의 점프를 통해 확인할 수 있다. 또한 점프 직전에 %r13 의 값을 1 증가시켜주고 있는데 이는 입력한 첫번째 정수부터 6 번째 정수까지의 주소를 차례차례 가리키게 되도록 해 줌을 알 수 있고, 따라서 입력된 6 개의 정수가 모두 5 보다 작거나 같아야 함을 알 수 있다. 반복되는 인스트럭션 부분에서 〈phase_6+64〉부터의 인스트럭션도 분석해보면, 우선 먼저 확인했던 반복문에서 반복 횟수를 나타내는 %r12d 의 값을 %ebx 에 복사한 후 이를 다시 %rax 에 복사한다. 메모리에서

(0x30 + %rsp + 4 * %rax), 즉 %rax 의 값을 i 라 할 때 (입력한 첫번째 정수가 저장된 주소 + 4 * i) 주소에 저장된 값과 %rbp 에 저장된 값을 비교해서 같지 않은 경우 explode_bomb 의 호출을 건너뛰게 되고, %ebx 의 값을 1 증가시킨 후 5 이하이면 〈phase_6 + 67〉로 점프한다. 따라서 먼저 확인했던 반복문 속에 또 다른 반복문이 존재함을 알 수 있으며, 바깥쪽의 반복문에서 반복 횟수를 나타내는 %r12d 의 값을 %ebx 에 복사한 후 안쪽의 반복문이 실행되는 것을 생각하면 이중으로 이루어진 반복문이 메모리의 (0x30 + %rsp + 4 * i) 주소의 값이 (0x30 + %rsp + 4 * (i+1)), ..., (0x30 + %rsp + 4 * 5)의 각각의 주소에 저장된 값과 모두 달라야 함을 확인하고 있음을 알 수 있다. 따라서 입력한 6 개의 정수는 모두 달라야 한다.

이후 %rsi, %rax, %ecx 에 각각 %rsp + 0x48, %r14 의 값인 %rsp + 0x30, 0x7 를 복사한다. 이후에도 반복문의 형태를 확인할 수 있는데, %rax 의 값을 0x4 씩 증가시키며 %rsi 와 같아질 때까지 반복이 이루어진다. 각 반복에서는 %edx 에 %ecx 의 값인 0x7 을 복사하고 %edx 의 값에서 %rax 가 가리키는 주소에 저장된 값을 빼서 %rax 가 가리키는 주소에 복사한다. %rsp + 0x30 이 첫번째 정수가 저장된 주소였으므로, 이는 입력한 여섯 개의 정수에 대해 7 에서 해당 값을 뺀 값으로 변환시키고 있음을 의미한다.

다음으로 %rsi 에 0 을 저장한 후, 〈phase_6+131〉 ~ 〈phase_6+184〉에서 또 다시 점프 인스트럭션을 통해 반복이 일어나고 있는 것을 확인할 수 있다.

```
(gdb) x/24wd 0x6042f0
0x6042f0 <node1>:       559     1       6308608 0
0x604300 <node2>:       388     2       6308624 0
0x604310 <node3>:       234     3       6308640 0
0x604320 <node4>:       559     4       6308656 0
0x604330 <node5>:       115     5       6308672 0
0x604340 <node6>:       765     6       0       0
```

이때 0x6042f0 이라는 상수가 사용되고 있어 메모리의 0x6042f0 주소에 저장된 데이터를 확인해본 결과 node1~node6 의 6 개의 node 가 존재하는 것을 확인할 수 있었다. 또한 0x604300, 0x604310, 0x604320, 0x604330, 0x604340 이 각각 십진수로 6308608, 6308624, 6308640, 6308656, 6308672 임을 고려하면, 각각의 node 가 다른 node 를 가리키고 있는 연결 리스트의 구조를 가지고 있음을 알 수 있다. 반복되는 인스트럭션을 분석해보면, %rsi 가 0x0 부터 시작해 0x4 씩 증가하여 0x18 이 될 때까지 (0x30 + %rsp + %rsi) 주소에 저장된 값을 %ecx 에 저장하는데 이는 앞서 입력했던 정수 각각에 대해 7 에서 해당 값을 뺀 값을 차례대로 %ecx 에 저장하는 것을 의미한다. 이후 해당 값이 1 이하인 경우에는 %edx 에 node1 의 주소를, 1 보다 큰 경우에는 %eax 의 값을 1 로, %edx 의 값을 node1 의 주소로 초기화한 후 %eax 의 값이 %ecx 와 같아질 때까지 1 씩 증가시키면서 (0x8 + %rdx) 주소의 값, 즉 다음 node 의 주소를 %rdx 에 저장한다. 이를 통해 이 반복문이 의미하는 것은 (%rsp + 0x30 + 4 * i) 주소에 저장된 정수 값 k 에 대해, node(k)의 주소가 (%rsp + 8 * i) 주소에 저장되는 것임을 알 수 있다.

다음으로, %rbx, %rcx 에 %rsp 가 가리키는 주소의 값, %rax 에 0x8 + %rsp, %rsi 에 0x30 + %rsp 의 값을 저장한다. 〈phase_6+203〉 ~ 〈phase_6+222〉에서는 반복이 이루어지는데, %rsp+0x30 의 주소부터 저장되어 있는 여섯 개의 정수를 n1, n2, n3, n4, n5, n6 이라고 한다면 %rbx, %rcx 이 node_n1 의 주소, node_n2 의 주소, ..., node_n5 의 주소의 값을 차례대로 가지고, %rax 가 node_n2 의 주소가 저장된 메모리의 주소, ..., node_n6 의 주소가 저장된 메모리의 주소의 값을 차례대로 가지는 반복이 이루어지는 것으로 볼 수 있다. 반복문의 인스트럭션을 분석해보면 결국 (%rsp + 0x30 + 4 * i)의 주소에 저장된 정수 번째 node 가 (%rsp + 0x30 + 4 * (i + 1))의 주소에 저장된 정수 번째 node 를 가리키게끔 하고 있음을 알 수 있다. 이처럼 node 의 순서를 바꾼 이후에는 %ebp 의 값을 5 부터 0 까지 줄여나가며 또 다시 반복이 일어난다. 이때 (node 의 주소 + 0x8)의 주소에 해당 node 가 가리키는 node 의 주소가 저장되어 있고, node 의 주소에는 node 마다 각기 다른 세자리의 숫자가 저장되어 있음을 고려하여 반복문의 코드를 분석해보면, node 의 순서가 바뀐 이후에 i 번째 node 주소에 저장된 수가 i+1 번째 node 주소에 저장된 수보다 크거나 같은 경우에 explode_bomb 의 호출을 건너뛸 수 있음을 알 수 있다. 따라서 각 node 의 주소에 저장되어 있는 수가 내림차순을 이루도록 node 의 정렬이 이루어져야 함을 알 수 있고, 이때 765, 559, 559, 388, 234, 115 의 순으로 이루어지면 되므로 가능한 node 순서쌍이 두 가지지만 node6-〉node4-〉node1-〉node2-〉node3-〉node5 의 순서로 정렬되게 하는 6 개의 정수를 답으로 사용하기로 하였다. node 가 위와 같은 순서로 정렬되기 위해서는 node 를 정렬하는 반복문이 수행될 때 메모리의 (0x30 + %rsp) 주소부터 저장된 정수가 순서대로 6 4 1 2 3 5 여야 하지만 이것은 해당 반복문이 수행되기 전 7 에서 원래 값을 뺀 값이었다는 것을 고려하면 처음 입력해야 하는 정수는 1 3 6 5 4 2 임을 찾을 수 있다.

```
Starting program: /home/std/limyoojin/hw3/bomb29/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
4 0
Halfway there!
11 1
So you got that one.  Try this one.
5 115
Good work!  On to the next...
1 3 6 5 4 2
Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
```

Secret Phase)

```
(gdb) disas phase_defused
Dump of assembler code for function phase_defused:
   0x0000000000401742 <+0>:     sub    $0x68,%rsp
   0x0000000000401746 <+4>:     mov    $0x1,%edi
   0x000000000040174b <+9>:     callq  0x4014e0 <send_msg>
   0x0000000000401750 <+14>:    cmpl   $0x6,0x203045(%rip)        # 0x60479c <num_input_strings>
   0x0000000000401757 <+21>:    jne    0x4017c6 <phase_defused+132>
   0x0000000000401759 <+23>:    lea    0x10(%rsp),%r8
   0x000000000040175e <+28>:    lea    0x8(%rsp),%rcx
   0x0000000000401763 <+33>:    lea    0xc(%rsp),%rdx
   0x0000000000401768 <+38>:    mov    $0x40285f,%esi
   0x000000000040176d <+43>:    mov    $0x6048b0,%edi
   0x0000000000401772 <+48>:    mov    $0x0,%eax
   0x0000000000401777 <+53>:    callq  0x400c30 <__isoc99_sscanf@plt>
   0x000000000040177c <+58>:    cmp    $0x3,%eax
   0x000000000040177f <+61>:    jne    0x4017b2 <phase_defused+112>
   0x0000000000401781 <+63>:    mov    $0x402868,%esi
   0x0000000000401786 <+68>:    lea    0x10(%rsp),%rdi
   0x000000000040178b <+73>:    callq  0x40133e <strings_not_equal>
   0x0000000000401790 <+78>:    test   %eax,%eax
   0x0000000000401792 <+80>:    jne    0x4017b2 <phase_defused+112>
   0x0000000000401794 <+82>:    mov    $0x4026c0,%edi
   0x0000000000401799 <+87>:    callq  0x400b40 <puts@plt>
   0x000000000040179e <+92>:    mov    $0x4026e8,%edi
   0x00000000004017a3 <+97>:    callq  0x400b40 <puts@plt>
   0x00000000004017a8 <+102>:   mov    $0x0,%eax
   0x00000000004017ad <+107>:   callq  0x40125c <secret_phase>
   0x00000000004017b2 <+112>:   mov    $0x402720,%edi
   0x00000000004017b7 <+117>:   callq  0x400b40 <puts@plt>
   0x00000000004017bc <+122>:   mov    $0x402750,%edi
   0x00000000004017c1 <+127>:   callq  0x400b40 <puts@plt>
   0x00000000004017c6 <+132>:   add    $0x68,%rsp
   0x00000000004017ca <+136>:   retq
End of assembler dump.
```

각 Phase 에 올바른 정답을 입력한 경우 호출되는 phase_defused 함수를 disassemble 해본 결과 secret_phase 가 존재함을 확인할 수 있었다. secret_phase 에 진입하기 위해 phase_defused 함수를 살펴보면 num_input_strings 의 값을 6 과 비교하여 같지 않은 경우 스택 포인터를 증가시키며 함수가 종료되도록 점프가 이루어지고 있음을 볼 수 있다. num_input_strings 의 값이 6 과 같은 경우를 분석해보기 위해 점프 인스트럭션 다음의 인스트럭션인 〈phase_defused+23〉에 브레이크 포인트를 만들고 실행한 결과 phase_6 에서 정답을 맞힌 경우 해당 브레이크 포인트에 도달하게 되는 것을 확인할 수 있었고, 이를 통해 num_input_strings 라는 이름에서 유추할 수 있듯이 입력한 string 의 개수가 6 개인지 확인하고 있음을 알 수 있다. 이후의 인스트럭션을 분석해보면 1 번째 인자를 0x6048b0, 2 번째 인자를 0x40285f, 3 번째 인자를 %rsp + 0x8, 4 번째 인자를 %rsp+0xc, 5 번째 인자를 %rsp+0x10 으로 하여 sscanf 의 호출이 일어나고 있고, 이때 반환값이 3 이 아닌 경우 "Congratulations! You've defused the bomb!"를 출력과 스택 포인터의 증가 후 함수가 종료되는 것을 볼 수 있다.

```
(gdb) x/s 0x6048b0
0x6048b0 <input_strings+240>:   "11 1"
(gdb) x/s 0x402868
0x402868:       "DrEvil"
```

```
(gdb) x/s 0x40285f
0x40285f:       "%d %d %s"
```

sscanf 가 데이터를 얻는 문자열인 첫번째 인자가 가리키는 메모리에 저장된 값을 확인해보면 “11 1”이 저장되어 있으며, 이는 phase_4 에서 입력했던 답이다. 따라서 해당 부분이 phase_4 에서 입력한 데이터의 개수가 3 개인지 확인하고 있음을 알 수 있으며, sscanf 의 format string 이 되는 두번째 인자가 가리키는 메모리에 “%d %d %s”가 저장되어 있는 것을 통해 정수 2 개와 문자열 1 개를 입력 받고 있음을 알 수 있다.

다음 인스트럭션을 보면 %rdi, %rsi 즉 첫번째 인자와 두번째 인자를 각각 %rsp+0x10, 0x402868 로 하여 strings_not_equal 이 호출되고 있는데, 이 함수는 phase_1 에서 확인했던 바에 의해 인자로 받은 두 주소에 저장된 문자열을 비교해 같은 경우 0 을, 다른 경우 1 을 반환하는 함수이다. 함수 호출 이후 test 인스트럭션을 통해 반환값을 저장하고 있는 %eax 의 자기자신과의 and 연산 후, jne 인스트럭션을 통해 zero flag 가 0 인 경우에 대해 secret_phase 에 도달할 수 있게 됨을 확인할 수 있다. strings_not_equal 의 인자를 확인해보면, 첫번째 인자는 %r8 과 같은 값을 가지고 있음을 볼 수 있고 따라서 sscanf 에서 3 번째로 입력된 데이터, 즉 입력한 문자열이 저장된 메모리를 가리키고 있음을 알 수 있다. 두번째 인자의 경우 x/s command 를 이용하면 해당 주소에 “DrEvil”이 저장되어 있음을 볼 수 있다. 이후 0x4026c0, 0x4026e8 주소에 저장된 문자열을 출력한 후 secret_phase 함수를 호출하게 되므로, 결국 secret_phase 에 진입하는 방법은 phase_4 에서 11 1 DrEvil 을 입력하는 것이다.

```
Starting program: /home/std/limyoojin/hw3/bomb29/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
4 0
Halfway there!
11 1 DrEvil
So you got that one.  Try this one.
5 115
Good work!  On to the next...
1 3 6 5 4 2
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

```
(gdb) disas secret_phase
Dump of assembler code for function secret_phase:
   0x000000000040125c <+0>:     push   %rbx
   0x000000000040125d <+1>:     callq  0x40161c <read_line>
   0x0000000000401262 <+6>:     mov    $0xa,%edx
   0x0000000000401267 <+11>:    mov    $0x0,%esi
   0x000000000040126c <+16>:    mov    %rax,%rdi
   0x000000000040126f <+19>:    callq  0x400c00 <strtol@plt>
   0x0000000000401274 <+24>:    mov    %rax,%rbx
   0x0000000000401277 <+27>:    lea    -0x1(%rax),%eax
   0x000000000040127a <+30>:    cmp    $0x3e8,%eax
   0x000000000040127f <+35>:    jbe    0x401286 <secret_phase+42>
   0x0000000000401281 <+37>:    callq  0x4015a4 <explode_bomb>
   0x0000000000401286 <+42>:    mov    %ebx,%esi
   0x0000000000401288 <+44>:    mov    $0x604110,%edi
   0x000000000040128d <+49>:    callq  0x40121e <fun7>
   0x0000000000401292 <+54>:    cmp    $0x7,%eax
   0x0000000000401295 <+57>:    je     0x40129c <secret_phase+64>
   0x0000000000401297 <+59>:    callq  0x4015a4 <explode_bomb>
   0x000000000040129c <+64>:    mov    $0x4025a0,%edi
   0x00000000004012a1 <+69>:    callq  0x400b40 <puts@plt>
   0x00000000004012a6 <+74>:    callq  0x401742 <phase_defused>
   0x00000000004012ab <+79>:    pop    %rbx
   0x00000000004012ac <+80>:    retq
End of assembler dump.
```

secret_phase 진입에 성공했으므로, secret_phase 의 정답을 찾기 위해 인스트럭션을 분석해보자. 진법으로 표기된 문자열을 정수로 변환해 반환하는 strtol 함수의 호출이 일어나고 있으며, 이때 %rdx 가 0xa 의 값을 가지므로 입력한 문자열을 10 진수로 변환하고 있으며 변환된 값은 %rax 레지스터에 저장된다. 이후 %rax-0x1 이 0x3e8 보다 큰 경우 explode_bomb 의 호출이 이루어지므로, 입력한 수는 0x3e9 이하여야 한다. 이후 첫번째 인자를 0x604110, 두번째 인자를 %esi 에 입력한 수로 하여 fun7 의 호출이 이루어진다. 〈secret_phase+57〉을 보면 fun7 의 반환값이 7 이어야 secret_phase 를 풀게 됨을 알 수 있다.

```
(gdb) x/120wx 0x604110
0x604110 <n1>:  0x00000024      0x00000000      0x00604130      0x00000000
0x604120 <n1+16>:       0x00604150      0x00000000      0x00000000      0x00000000
0x604130 <n21>: 0x00000008      0x00000000      0x006041b0      0x00000000
0x604140 <n21+16>:      0x00604170      0x00000000      0x00000000      0x00000000
0x604150 <n22>: 0x00000032      0x00000000      0x00604190      0x00000000
0x604160 <n22+16>:      0x006041d0      0x00000000      0x00000000      0x00000000
0x604170 <n32>: 0x00000016      0x00000000      0x00604290      0x00000000
0x604180 <n32+16>:      0x00604250      0x00000000      0x00000000      0x00000000
0x604190 <n33>: 0x0000002d      0x00000000      0x006041f0      0x00000000
0x6041a0 <n33+16>:      0x006042b0      0x00000000      0x00000000      0x00000000
0x6041b0 <n31>: 0x00000006      0x00000000      0x00604210      0x00000000
0x6041c0 <n31+16>:      0x00604270      0x00000000      0x00000000      0x00000000
0x6041d0 <n34>: 0x0000006b      0x00000000      0x00604230      0x00000000
0x6041e0 <n34+16>:      0x006042d0      0x00000000      0x00000000      0x00000000
0x6041f0 <n45>: 0x00000028      0x00000000      0x00000000      0x0000---Type <retu
0000
0x604200 <n45+16>:      0x00000000      0x00000000      0x00000000      0x00000000
0x604210 <n41>: 0x00000001      0x00000000      0x00000000      0x00000000
0x604220 <n41+16>:      0x00000000      0x00000000      0x00000000      0x00000000
0x604230 <n47>: 0x00000063      0x00000000      0x00000000      0x00000000
0x604240 <n47+16>:      0x00000000      0x00000000      0x00000000      0x00000000
0x604250 <n44>: 0x00000023      0x00000000      0x00000000      0x00000000
0x604260 <n44+16>:      0x00000000      0x00000000      0x00000000      0x00000000
0x604270 <n42>: 0x00000007      0x00000000      0x00000000      0x00000000
0x604280 <n42+16>:      0x00000000      0x00000000      0x00000000      0x00000000
0x604290 <n43>: 0x00000014      0x00000000      0x00000000      0x00000000
0x6042a0 <n43+16>:      0x00000000      0x00000000      0x00000000      0x00000000
0x6042b0 <n46>: 0x0000002f      0x00000000      0x00000000      0x00000000
0x6042c0 <n46+16>:      0x00000000      0x00000000      0x00000000      0x00000000
0x6042d0 <n48>: 0x000003e9      0x00000000      0x00000000      0x00000000
---Type <return> to continue, or q <return> to quit---c
0x6042e0 <n48+16>:      0x00000000      0x00000000      0x00000000      0x00000000
```

```
(gdb) disas fun7
Dump of assembler code for function fun7:
   0x000000000040121e <+0>:     sub    $0x8,%rsp
   0x0000000000401222 <+4>:     test   %rdi,%rdi
   0x0000000000401225 <+7>:     je     0x401252 <fun7+52>
   0x0000000000401227 <+9>:     mov    (%rdi),%edx
   0x0000000000401229 <+11>:    cmp    %esi,%edx
   0x000000000040122b <+13>:    jle    0x40123a <fun7+28>
   0x000000000040122d <+15>:    mov    0x8(%rdi),%rdi
   0x0000000000401231 <+19>:    callq  0x40121e <fun7>
   0x0000000000401236 <+24>:    add    %eax,%eax
   0x0000000000401238 <+26>:    jmp    0x401257 <fun7+57>
   0x000000000040123a <+28>:    mov    $0x0,%eax
   0x000000000040123f <+33>:    cmp    %esi,%edx
   0x0000000000401241 <+35>:    je     0x401257 <fun7+57>
   0x0000000000401243 <+37>:    mov    0x10(%rdi),%rdi
   0x0000000000401247 <+41>:    callq  0x40121e <fun7>
   0x000000000040124c <+46>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401250 <+50>:    jmp    0x401257 <fun7+57>
   0x0000000000401252 <+52>:    mov    $0xffffffff,%eax
   0x0000000000401257 <+57>:    add    $0x8,%rsp
   0x000000000040125b <+61>:    retq
End of assembler dump.
```

fun7 을 disassemble 해보면 재귀함수의 형태임을 우선 확인할 수 있고, 첫번째 인자에 저장된 값인 0x604110 의 주소에 저장된 값을 확인해보면 위와 같이 데이터가 저장되어 있는 것을 볼 수 있다.

또한 fun7 을 분석해보면 첫번째 인자로 받은 주소에 저장된 값이 입력한 수와 같을 경우 0, 입력한 수보다 작은 경우 첫번째 인자가 되는 %rdi 의 값을 메모리의 (%rdi + 0x10) 주소에 저장된 값으로 변경하여 fun7 을 다시 호출하고, 입력한 수보다 클 경우 %rdi 의 값을 메모리의 (%rdi + 0x8) 주소에 저장된 값으로 변경하여 fun7 을 다시 호출한다는 것을 알 수 있다.

```
   fun7(0x604110, x) - 7 반환
-> fun7(0x604150, x) - 3 반환
-> fun7(0x6041d0, x) - 1 반환
-> fun7(0x6042d0, x) - 0 반환
```

이와 같은 재귀함수의 형태와 위에서 확인했던 데이터를 고려해보면 phase_defused 에서 받게 되는 fun7 의 반환값이 7 이기 위해서는 왼쪽과 같이 함수의 호출이 일어나면 될 것이다. 이를 위해서는 마지막으로 호출되는 fun7 에서의 반환값이 0 이 되어야 하므로 x 의 값이 0x6042d0 주소에 저장된 0x3e9, 즉 1001 이 정답의 가능성이 있다. 그런데 이때 0x6041d0, 0x604150, 0x604110 에 저장된 값은 각각 0x6b, 0x32, 0x24 로 모두 0x3e9 보다 작아 각각의 함수 호출에서 2 * fun7 + 1 의 값을 반환하게 되며, 0x3e9 는 secret_phase 함수에서 확인했던 입력한 수의 조건인 0x3e9 이하여야함을 만족하므로 secret_phase 의 정답임을 알 수 있다.

```
Starting program: /home/std/limyoojin/hw3/bomb29/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2.  Keep going!
4 0
Halfway there!
11 1 DrEvil
So you got that one.  Try this one.
5 115
Good work!  On to the next...
1 3 6 5 4 2
Curses, you've found the secret phase!
But finding it and solving it are quite different...
1001
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
```