CSED211: Data Lab3 Bomb Lab Report

20200952 / Chaeyeon Jang

1. Overview

Bomb Lab 에서는 총 6 개 phase 의 assembly 코드를 gdb 로 분석하여 각 phase 의 flow 를 이해하고, correct 한 input 을 입력함으로써 폭탄이 터지지 않고 phase 를 무사히 통과해야 한다.

2. Phase #1

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x0000000000400ef0 <+0>:    sub    $0x8,%rsp
   0x0000000000400ef4 <+4>:    mov    $0x4025a0,%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 에서는 $0x4025a0 의 값을 esi 레지스터에 넣고 <strings_not_equal> 함수를 수행한다. 함수의 반환값인 eax 레지스터를 test 하여 eax & eax 값이 0 일 경우, 즉 eax 값이 0 일 경우 폭탄이 터지지 않고 무사히 phase_1 이 return 된다. <srings_not_equal> 함수를 disassemble 해보면 아래와 같다.

```
(gdb) disas strings_not_equal
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
   0x0000000000401348 <+10>:   callq  0x401321 <string_length>
   0x000000000040134d <+15>:   mov    %eax,%r12d
   0x0000000000401350 <+18>:   mov    %rbp,%rdi
   0x0000000000401353 <+21>:   callq  0x401321 <string_length>
   0x0000000000401358 <+26>:   mov    $0x1,%edx
   0x000000000040135d <+31>:   cmp    %eax,%r12d
   0x0000000000401360 <+34>:   jne    0x4013a0 <strings_not_equal+98>
   0x0000000000401362 <+36>:   movzbl (%rbx),%eax
   0x0000000000401365 <+39>:   test   %al,%al
   0x0000000000401367 <+41>:   je     0x40138d <strings_not_equal+79
   0x0000000000401369 <+43>:   cmp    0x0(%rbp),%al
   0x000000000040136c <+46>:   je     0x401377 <strings_not_equal+57>
   0x000000000040136e <+48>:   xchg   %ax,%ax
   0x0000000000401370 <+50>:   jmp    0x401394 <strings_not_equal+86>
   0x0000000000401372 <+52>:   cmp    0x0(%rbp),%al
   0x0000000000401375 <+55>:   jne    0x40139b <strings_not_equal+93>
   0x0000000000401377 <+57>:   add    $0x1,%rbx
   0x000000000040137b <+61>:   add    $0x1,%rbp
   0x000000000040137f <+65>:   movzbl (%rbx),%eax
   0x0000000000401382 <+68>:   test   %al,%al
   0x0000000000401384 <+70>:   jne    0x401372 <strings_not_equal+52>
   0x0000000000401386 <+72>:   mov    $0x0,%edx
   0x000000000040138b <+77>:   jmp    0x4013a0 <strings_not_equal+98>
```

```
   0x000000000040138d <+79>:    mov    $0x0,%edx
   0x0000000000401392 <+84>:    jmp    0x4013a0 <strings_not_equal+98>
   0x0000000000401394 <+86>:    mov    $0x1,%edx
   0x0000000000401399 <+91>:    jmp    0x4013a0 <strings_not_equal+98>
   0x000000000040139b <+93>:    mov    $0x1,%edx
   0x00000000004013a0 <+98>:    mov    %edx,%eax
   0x00000000004013a2 <+100>:   pop    %rbx
   0x00000000004013a3 <+101>:   pop    %rbp
   0x00000000004013a4 <+102>:   pop    %r12
   0x00000000004013a6 <+104>:   retq
End of assembler dump.
```

Strings_not_equal 함수에서는 함수의 argument 인 rsi, rdi 레지스터에 저장된 문자열을 처음부터 문자열의 끝까지 비교하여 두 문자열이 같다면 eax 에 0 을 저장하고 return 한다. 그렇다면 <phase_1 + 4>에서 esi 레지스터에 저장하는 $0x4025a0 의 문자열 값이 무엇인지 확인하고, 그 문자열과 같은 문자열을 input 으로 입력하면 explode_bomb 을 피해갈 것이다. $0x4025a0 의 값을 확인하였더니 아래와 같았다.

```
(gdb) x/s 0x4025a0
0x4025a0:      "There are rumors on the internets."
```

따라서 phase_1 의 Input 으로 There are rumors on the internets. 을 입력하였더니 phase_1 이 diffuse 되었다.

```
(gdb) r
Starting program: /home/std/jcy2749/bomb64/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
There are rumors on the internets.
Phase 1 defused. How about the next one?
```

3. 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  0x40168e <read_six_numbers>
   0x0000000000400f1a <+14>:    cmpl   $0x1,(%rsp)
   0x0000000000400f1e <+18>:    je     0x400f40 <phase_2+52>
   0x0000000000400f20 <+20>:    callq  0x4015a4 <explode_bomb>
   0x0000000000400f25 <+25>:    jmp    0x400f40 <phase_2+52>
   0x0000000000400f27 <+27>:    mov    -0x4(%rbx),%eax
   0x0000000000400f2a <+30>:    add    %eax,%eax
   0x0000000000400f2c <+32>:    cmp    %eax,(%rbx)
   0x0000000000400f2e <+34>:    je     0x400f35 <phase_2+41>
   0x0000000000400f30 <+36>:    callq  0x4015a4 <explode_bomb>
   0x0000000000400f35 <+41>:    add    $0x4,%rbx
```

```
   0x0000000000400f39 <+45>:    cmp    %rbp,%rbx
   0x0000000000400f3c <+48>:    jne    0x400f27 <phase_2+27>
   0x0000000000400f3e <+50>:    jmp    0x400f4c <phase_2+64>
   0x0000000000400f40 <+52>:    lea    0x4(%rsp),%rbx
   0x0000000000400f45 <+57>:    lea    0x18(%rsp),%rbp
   0x0000000000400f4a <+62>:    jmp    0x400f27 <phase_2+27>
   0x0000000000400f4c <+64>:    add    $0x28,%rsp
   0x0000000000400f50 <+68>:    pop    %rbx
   0x0000000000400f51 <+69>:    pop    %rbp
   0x0000000000400f52 <+70>:    retq
End of assembler dump.
```

Phase_2 에서는 <read_six_numbers> 함수를 실행하는데, 아래는 <read_six_numbers>의 assembly code 이다.

```
(gdb) disas read_six_numbers
Dump of assembler code for function read_six_numbers:
   0x000000000040168e <+0>:    sub    $0x18,%rsp
   0x0000000000401692 <+4>:    mov    %rsi,%rdx
   0x0000000000401695 <+7>:    lea    0x4(%rsi),%rcx
   0x0000000000401699 <+11>:   lea    0x14(%rsi),%rax
   0x000000000040169d <+15>:   mov    %rax,0x8(%rsp)
   0x00000000004016a2 <+20>:   lea    0x10(%rsi),%rax
   0x00000000004016a6 <+24>:   mov    %rax,(%rsp)
   0x00000000004016aa <+28>:   lea    0xc(%rsi),%r9
   0x00000000004016ae <+32>:   lea    0x8(%rsi),%r8
   0x00000000004016b2 <+36>:   mov    $0x403368,%esi
   0x00000000004016b7 <+41>:   mov    $0x0,%eax
   0x00000000004016bc <+46>:   callq  0x400c30 <__isoc99_sscanf@plt>
   0x00000000004016c1 <+51>:   cmp    $0x5,%eax
   0x00000000004016c4 <+54>:   jg     0x4016cb <read_six_numbers+61>
   0x00000000004016c6 <+56>:   callq  0x4015a4 <explode_bomb>
   0x00000000004016cb <+61>:   add    $0x18,%rsp
   0x00000000004016cf <+65>:   nop
   0x00000000004016d0 <+66>:   retq
End of assembler dump.
```

이때 esi 레지스터에 $0x403368 값을 옮기고 입력 값을 scan 받으므로, 0x403368 값의 형태를 확인하면 phase_2 의 input 형태를 확인할 수 있다. 아래와 같이 6 개의 정수를 입력 받는 형태이므로, phase_2 는 6 개의 정수를 input 으로 입력해야 한다.

```
(gdb) x/s 0x403368
0x403368:       "%d %d %d %d %d %d"
```

이때 <phase_2 +14>에서, 1 과 (rsp+ 0)의 값을 비교해서 같아야 explode_bomb 을 피해가므로 가장 첫번째 정수는 1 이 되어야 한다. 첫번째 정수가 1 이라면, <phase_2 +52>로 jump 하여 rsp + 4 의 주소를 rbx 에 넣고(즉 다음 정수의 주소를 rbx 에 넣고) rsp + 18 의 주소를 rbp 에 넣은 후 <phase_2 + 27>로 jump 한다. 이후 마지막 정수를 읽을 때까지 반복문을 반복하며 <phase_2 + 30>에서 이전 정수를 2 배 한 값이 현재 정수의 값과 같으면 <phase_2 + 41>로 Jump 하여 현재 정수가 마지막 정수 값과 같은지 비교한다. 이때, <phase_2 + 34>, <phase_2 + 36>에서 이전

정수를 2 배한 값이 현재 정수의 값과 다르면 explode_bomb 이 실행되기 때문에, phase_2 에 대한 input 은 1 2 4 8 16 32 가 되어야 한다.
아래와 같이 phase_2 의 input 으로 1 2 4 8 16 32 를 입력했더니 phase_2 가 diffuse 되었다.

```
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
```

4. Phase #3

```
(gdb) disas phase_3
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    $0x403374,%esi
   0x0000000000400f66 <+19>:   mov    $0x0,%eax
   0x0000000000400f6b <+24>:   callq  0x400c30 <__isoc99_sscanf@plt>
   0x0000000000400f70 <+29>:   cmp    $0x1,%eax
   0x0000000000400f73 <+32>:   jg     0x400f7a <phase_3+39>
   0x0000000000400f75 <+34>:   callq  0x4015a4 <explode_bomb>
   0x0000000000400f7a <+39>:   cmpl   $0x7,0xc(%rsp)
   0x0000000000400f7f <+44>:   ja     0x400fe7 <phase_3+148>
   0x0000000000400f81 <+46>:   mov    0xc(%rsp),%eax
   0x0000000000400f85 <+50>:   jmpq   *0x402600(,%rax,8)
   0x0000000000400f8c <+57>:   mov    $0x0,%eax
   0x0000000000400f91 <+62>:   jmp    0x400f98 <phase_3+69>
   0x0000000000400f93 <+64>:   mov    $0x23e,%eax
   0x0000000000400f98 <+69>:   sub    $0x342,%eax
   0x0000000000400f9d <+74>:   jmp    0x400fa4 <phase_3+81>
   0x0000000000400f9f <+76>:   mov    $0x0,%eax
   0x0000000000400fa4 <+81>:   add    $0xc6,%eax
   0x0000000000400fa9 <+86>:   jmp    0x400fb0 <phase_3+93>
   0x0000000000400fab <+88>:   mov    $0x0,%eax
   0x0000000000400fb0 <+93>:   sub    $0x39b,%eax
   0x0000000000400fb5 <+98>:   jmp    0x400fbc <phase_3+105>
   0x0000000000400fb7 <+100>:  mov    $0x0,%eax
   0x0000000000400fbc <+105>:  add    $0x39b,%eax
   0x0000000000400fc1 <+110>:  jmp    0x400fc8 <phase_3+117>
   0x0000000000400fc3 <+112>:  mov    $0x0,%eax
   0x0000000000400fc8 <+117>:  sub    $0x39b,%eax
   0x0000000000400fcd <+122>:  jmp    0x400fd4 <phase_3+129>
   0x0000000000400fcf <+124>:  mov    $0x0,%eax
   0x0000000000400fd4 <+129>:  add    $0x39b,%eax
   0x0000000000400fd9 <+134>:  jmp    0x400fe0 <phase_3+141>
   0x0000000000400fdb <+136>:  mov    $0x0,%eax
   0x0000000000400fe0 <+141>:  sub    $0x39b,%eax
   0x0000000000400fe5 <+146>:  jmp    0x400ff1 <phase_3+158>
   0x0000000000400fe7 <+148>:  callq  0x4015a4 <explode_bomb>
   0x0000000000400fec <+153>:  mov    $0x0,%eax
   0x0000000000400ff1 <+158>:  cmpl   $0x5,0xc(%rsp)
   0x0000000000400ff6 <+163>:  jg     0x400ffe <phase_3+171>
   0x0000000000400ff8 <+165>:  cmp    0x8(%rsp),%eax
   0x0000000000400ffc <+169>:  je     0x401003 <phase_3+176>
   0x0000000000400ffe <+171>:  callq  0x4015a4 <explode_bomb>
   0x0000000000401003 <+176>:  add    $0x18,%rsp
   0x0000000000401007 <+180>:  retq
End of assembler dump.
```

우선 <phase_3 + 14>에서, $0x403374 의 값을 esi 레지스터에 넣고 입력값을 scan 받으므로 $0x403374 에 저장된 값을 확인하여 phase_3 의 input 형태를 확인한다.

```
(gdb) x/s 0x403374
0x403374:       "%d %d"
```

저장된 형태로 보아 phase_3 은 2 개의 정수를 입력 받는 것을 알 수 있다. 또한 첫 번째 정수는 rsp + 0xc 에, 두 번째 정수는 rsp + 0x8 에 저장된다.
이때, <phase_3 + 39>에서 7 과 (rsp + 0xc)에 저장된 값을 비교하여 (rsp + 0xc)의 값이 7 보다 큰 경우 explode_bomb 을 실행하므로 첫 번째 정수는 7 보다 작아야 한다. 그리고 <phase_3 + 50>에서 jump table 을 통해 rax 에 저장된 값에 따라 정해진 주소로 이동하는데, jump table 을 복원하면 아래와 같다.

```
(gdb) x/8gx 0x402600
0x402600:       0x0000000000400f93      0x0000000000400f8c
0x402610:       0x0000000000400f9f      0x0000000000400fab
0x402620:       0x0000000000400fb7      0x0000000000400fc3
0x402630:       0x0000000000400fcf      0x0000000000400fdb
```

이때, <phase_3 + 165>에서 eax 레지스터의 값과 (rsp + 0x8)에 저장된 값이 같아야 explode_bomb 을 피할 수 있으므로, phase_3 의 input 에서 두 번째 정수는 첫 번째 정수를 통해 jump table 로 이동한 주소에서부터 eax 의 값을 계산한, 최종적인 eax 값과 같은 정수이어야 한다. 즉 phase_3 input 의 첫 번째 정수는 7 보다 작은 정수 중 하나, 두 번째 정수는 첫 번째 정수에 맞게 이동한 주소에부터 계산된 eax 값이면 된다.
첫 번째 정수로 3 을 선택했을 때, jump table 에 의해 0x0000000000400fab (<phase_3 + 88>)주소로 이동한다. 그럼 순차적으로 eax 레지스터에 0 을 대입하고, eax 레지스터에서 0x39b (=10 진수로 923)을 빼고, 다시 923 을 더하고, 다시 923 을 빼고, 다시 923 을 더하고, 마지막으로 923 을 빼는 과정이 진행된다. 최종적으로 eax 에 저장된 값은 -923 이므로, 첫 번째 정수가 3 일 경우 두 번째 정수는 -923 이 된다.
따라서, Phase_3 의 input 으로 3 -923 을 입력했더니 phase_3 의 bomb 가 diffuse 되었다.

```
That's number 2.  Keep going!
3 -923
Halfway there!
```

5. Phase #4

```
(gdb) disas phase_4
Dump of assembler code for function phase_4:
   0x0000000000401046 <+0>:    sub    $0x18,%rsp
   0x000000000040104a <+4>:    lea    0x8(%rsp),%rcx
   0x000000000040104f <+9>:    lea    0xc(%rsp),%rdx
   0x0000000000401054 <+14>:   mov    $0x403374,%esi
   0x0000000000401059 <+19>:   mov    $0x0,%eax
   0x000000000040105e <+24>:   callq  0x400c30 <__isoc99_sscanf@plt>
   0x0000000000401063 <+29>:   cmp    $0x2,%eax
   0x0000000000401066 <+32>:   jne    0x40106f <phase_4+41>
   0x0000000000401068 <+34>:   cmpl   $0xe,0xc(%rsp)
   0x000000000040106d <+39>:   jbe    0x401074 <phase_4+46>
   0x000000000040106f <+41>:   callq  0x4015a4 <explode_bomb>
   0x0000000000401074 <+46>:   mov    $0xe,%edx
   0x0000000000401079 <+51>:   mov    $0x0,%esi
   0x000000000040107e <+56>:   mov    0xc(%rsp),%edi
   0x0000000000401082 <+60>:   callq  0x401008 <func4>
   0x0000000000401087 <+65>:   cmp    $0x3,%eax
   0x000000000040108a <+68>:   jne    0x401093 <phase_4+77>
   0x000000000040108c <+70>:   cmpl   $0x3,0x8(%rsp)
   0x0000000000401091 <+75>:   je     0x401098 <phase_4+82>
   0x0000000000401093 <+77>:   callq  0x4015a4 <explode_bomb>
   0x0000000000401098 <+82>:   add    $0x18,%rsp
   0x000000000040109c <+86>:   retq
End of assembler dump.
```

우선 $0x403374 에 저장된 값을 확인하여 phase_4 의 input 형태를 확인한다.

```
(gdb) x/s 0x403374
0x403374:       "%d %d"
```

위를 통해 phase_4 에서는 두 개의 정수를 입력해야 함을 알 수 있다. 또한 첫 번째 정수는 rsp + 0xc 에, 두 번째 정수는 rsp + 0x8 에 저장된다.

이때 <phase_4 + 34>에서 (rsp + 0xc)의 값이 14 보다 작거나 같아야 explode_bomb 을 건너뛰므로 첫 번째 정수는 14 보다 작거나 같아야 한다.

그리고, edx 레지스터에 14, esi 레지스터에 0, edi 레지스터에 (rsp + 0xc)의 값을 저장하고 func4 함수를 호출한다. Func4 를 실행한 후 반환값인 eax 레지스터의 값이 3 과 같고, 동시에 (rsp + 0x8)의 값이 3 과 같아야 폭탄이 터지지 않고 phase_4 가 return 된다. 이를 통해 두 번째 정수는 3 이어야 함을 알 수 있고, func4 를 실행하고 return 값이 3 이어야 함을 알 수 있다. 이제 func4 의 assembly code 를 분석해보자.

```
(gdb) disas func4
Dump of assembler code for function func4:
   0x0000000000401008 <+0>:    sub    $0x8,%rsp
   0x000000000040100c <+4>:    mov    %edx,%eax
   0x000000000040100e <+6>:    sub    %esi,%eax
   0x0000000000401010 <+8>:    mov    %eax,%ecx
   0x0000000000401012 <+10>:   shr    $0x1f,%ecx
   0x0000000000401015 <+13>:   add    %ecx,%eax
   0x0000000000401017 <+15>:   sar    %eax
   0x0000000000401019 <+17>:   lea    (%rax,%rsi,1),%ecx
   0x000000000040101c <+20>:   cmp    %edi,%ecx
   0x000000000040101e <+22>:   jle    0x40102c <func4+36>
   0x0000000000401020 <+24>:   lea    -0x1(%rcx),%edx
```

```
0x0000000000401023 <+27>:   callq  0x401008 <func4>
0x0000000000401028 <+32>:   add    %eax,%eax
0x000000000040102a <+34>:   jmp    0x401041 <func4+57>
0x000000000040102c <+36>:   mov    $0x0,%eax
0x0000000000401031 <+41>:   cmp    %edi,%ecx
0x0000000000401033 <+43>:   jge    0x401041 <func4+57>
0x0000000000401035 <+45>:   lea    0x1(%rcx),%esi
0x0000000000401038 <+48>:   callq  0x401008 <func4>
0x000000000040103d <+53>:   lea    0x1(%rax,%rax,1),%eax
0x0000000000401041 <+57>:   add    $0x8,%rsp
0x0000000000401045 <+61>:   retq
End of assembler dump.
```

func4 의 어셈블리 코드를 통해 func4 의 flow 를 복원해보면 아래와 같으며, 조건에 따라 재귀적으로 func4 를 호출하는 형태이다.

```
func4(edi, esi, edx):
    eax = edx
    eax -= esi
    ecx = eax
    ecx >> 31 //logical right shift
    eax += ecx
    eax >> 1 //arithmatic right shift
    ecx = rax + rsi
    if(edi >= ecx):
        eax = 0
        if(edi <= ecx): return
        esi = rcx + 1
        func4(edi, esi, edx)
        eax = 2 * rax + 1
        return
    if(edi < ecx):
        edx = rcx - 1
        func4(edi, esi, edx)
        eax = 2 * eax
        return
```

이때, return 값이 3 이 되어야 하므로, {edi < ecx}가 아닌 {edi >= ecx} 일 때의 if 문을 수행해야 함을 알 수 있다. 처음 edx 레지스터에 14, esi 레지스터에 0, edi 레지스터에 (rsp + 0xc) 값이 저장되어 있으므로, func4 함수를 따라 각 레지스터 값을 업데이트하면 eax=0, esi=8, ecx 는 7 이 되어 edi 는 7 보다 크거나 같아야 한다. 다시 func4 를 호출하고, 레지스터 값을 업데이트하면 eax=0, esi=12, ecx 는 11 이 되어 edi 는 11 보다 크거나 같아야 한다. 한 번 더 func4 를 호출하고 레지스터 값을 업데이트하면 ecx 는 13 이 되고, edi 는 13 보다 크거나 같아야 한다. 이때, edi <= ecx 이어야 재귀함수가 return 되므로 동시에 edi 는 13 보다 작거나 같아야 하고, edi 값, 즉 (rsp + 0xc)의 값은 13 이 된다. 재귀함수 func4 가 처음 return 되면 eax = 1, 한번 더 return 되면 eax = 2 * 1 + 1 = 3 이 되고, 최종적으로 3 이 반환된다. 그러므로 phase_4 에서 폭탄이 터지지 않고 무사히 phase_4 가 return 된다.

따라서, phase_4 의 input 은 첫 번째 정수 13, 두 번째 정수 3 이고, 아래와 같이 phase_4 가 diffuse 된다.

```
Halfway there!

13 3

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

6. Phase #5

```
(gdb) disas phase_5
Dump of assembler code for function phase_5:
   0x000000000040109d <+0>:    sub    $0x18,%rsp
   0x00000000004010a1 <+4>:    lea    0x8(%rsp),%rcx
   0x00000000004010a6 <+9>:    lea    0xc(%rsp),%rdx
   0x00000000004010ab <+14>:   mov    $0x403374,%esi
   0x00000000004010b0 <+19>:   mov    $0x0,%eax
   0x00000000004010b5 <+24>:   callq  0x400c30 <__isoc99_sscanf@plt>
   0x00000000004010ba <+29>:   cmp    $0x1,%eax
   0x00000000004010bd <+32>:   jg     0x4010c4 <phase_5+39>
   0x00000000004010bf <+34>:   callq  0x4015a4 <explode_bomb>
   0x00000000004010c4 <+39>:   mov    0xc(%rsp),%eax
   0x00000000004010c8 <+43>:   and    $0xf,%eax
   0x00000000004010cb <+46>:   mov    %eax,0xc(%rsp)
   0x00000000004010cf <+50>:   cmp    $0xf,%eax
   0x00000000004010d2 <+53>:   je     0x401100 <phase_5+99>
   0x00000000004010d4 <+55>:   mov    $0x0,%ecx
   0x00000000004010d9 <+60>:   mov    $0x0,%edx
   0x00000000004010de <+65>:   add    $0x1,%edx
   0x00000000004010e1 <+68>:   cltq
   0x00000000004010e3 <+70>:   mov    0x402640(,%rax,4),%eax
   0x00000000004010ea <+77>:   add    %eax,%ecx
   0x00000000004010ec <+79>:   cmp    $0xf,%eax
   0x00000000004010ef <+82>:   jne    0x4010de <phase_5+65>
   0x00000000004010f1 <+84>:   mov    %eax,0xc(%rsp)
   0x00000000004010f5 <+88>:   cmp    $0xf,%edx
   0x00000000004010f8 <+91>:   jne    0x401100 <phase_5+99>
   0x00000000004010fa <+93>:   cmp    0x8(%rsp),%ecx
   0x00000000004010fe <+97>:   je     0x401105 <phase_5+104>
   0x0000000000401100 <+99>:   callq  0x4015a4 <explode_bomb>
   0x0000000000401105 <+104>:  add    $0x18,%rsp
   0x0000000000401109 <+108>:  retq
End of assembler dump.
```

우선, 0x403374 의 값을 확인해보면 phase_5 는 2 개의 정수를 입력 받음을 알 수 있고, 첫 번째 정수는 rsp + 0xc 에, 두 번째 정수는 rsp + 0x8 에 저장된다. <phase_5 + 50>에 의해, (rsp+0xc) 값의 하위 4bit 는 1111 이 되면 안되고, <phase_5 + 55>에서 ecx 에 0, edx 에 0 을 초깃값으로 집어넣는다. 그 후 eax 값이 15 와 같아질 때까지 <phase_5 + 65> ~ <phase_5 + 82>의 반복문을 반복한다. 반복문에서, rax 값에 따라 0x402640 array 에서 해당되는 주소에 있는 값을 eax 에 저장하고, ecx 에 eax 값을 누적해서 더한다. 이때, 0x402640 array 를 복원하면 아래와 같다.

```
(gdb) x/8gx 0x402640
0x402640 <array.3161>:  0x000000020000000a      0x000000070000000e
0x402650 <array.3161+16>:       0x0000000c00000008      0x0000000b0000000f
0x402660 <array.3161+32>:       0x0000000400000000      0x0000000d00000001
0x402670 <array.3161+48>:       0x0000000900000003      0x0000000500000006
```

이때, <phase_5 + 88>에 의해 eax 가 15 와 같아지는 iteration 에서 edx 값이 15 가 되어야, 즉 정확히 반복문을 15 번을 반복한 후 eax 가 15 와 같아져야 explode_bomb 을 피할 수 있다. 즉, phase_5 에서는 input 으로 입력한 첫 번째 정수부터 시작하여, array 에서 그 정수번째 인덱스에 저장되어 있는 값을 eax 에 넣고, 다시 그 값에 해당하는 인덱스에 저장되어 있는 값을 eax 에 넣는 과정을 15 번 반복했을 때 eax 가 최초로 15 가 되어야 한다. 이때, 최초 rax 값이 5 일 경우

5 → 12 → 3 → 7 → 11 → 13 → 9 → 4 → 8→ 0 → 10 → 1 → 2 → 14 → 6 → 15 의 업데이트 과정을 거쳐 15 번째만에 eax 가 15 가 된다. 따라서 input 의 첫번째 정수는 5 가 된다. 두번째 정수는 <phase_5 + 93>에 의해 ecx 에 저장된 값과 같아야 하고, ecx 에 저장된 누적합은 12 + 3 + 7 + 11 + 13 + 9 + 4 + 8 + 0 + 10 + 1 + 2 + 14 + 6 + 15 = 115 이므로, 두번째 정수는 115 가 된다. 따라서 아래와 같이 5 115 를 입력하면 phase_5 가 diffuse 된다.

```
So you got that one.  Try this one.
5 115
Good work!  On to the next...
```

7. Phase #6

```
(gdb) disas phase_6
Dump of assembler code for function phase_6:
   0x000000000040110a <+0>:    push   %r14
   0x000000000040110c <+2>:    push   %r13
   0x000000000040110e <+4>:    push   %r12
   0x0000000000401110 <+6>:    push   %rbp
   0x0000000000401111 <+7>:    push   %rbx
   0x0000000000401112 <+8>:    sub    $0x50,%rsp
   0x0000000000401116 <+12>:   lea    0x30(%rsp),%r13
   0x000000000040111b <+17>:   mov    %r13,%rsi
   0x000000000040111e <+20>:   callq  0x40168e <read_six_numbers>
   0x0000000000401123 <+25>:   mov    %r13,%r14
   0x0000000000401126 <+28>:   mov    $0x0,%r12d
   0x000000000040112c <+34>:   mov    %r13,%rbp
   0x000000000040112f <+37>:   mov    0x0(%r13),%eax
   0x0000000000401133 <+41>:   sub    $0x1,%eax
   0x0000000000401136 <+44>:   cmp    $0x5,%eax
   0x0000000000401139 <+47>:   jbe    0x401140 <phase_6+54>
   0x000000000040113b <+49>:   callq  0x4015a4 <explode_bomb>
   0x0000000000401140 <+54>:   add    $0x1,%r12d
   0x0000000000401144 <+58>:   cmp    $0x6,%r12d
   0x0000000000401148 <+62>:   je     0x40116c <phase_6+98>
   0x000000000040114a <+64>:   mov    %r12d,%ebx
   0x000000000040114d <+67>:   movslq %ebx,%rax
   0x0000000000401150 <+70>:   mov    0x30(%rsp,%rax,4),%eax
   0x0000000000401154 <+74>:   cmp    %eax,0x0(%rbp)
   0x0000000000401157 <+77>:   jne    0x40115e <phase_6+84>
   0x0000000000401159 <+79>:   callq  0x4015a4 <explode_bomb>
   0x000000000040115e <+84>:   add    $0x1,%ebx
   0x0000000000401161 <+87>:   cmp    $0x5,%ebx
   0x0000000000401164 <+90>:   jle    0x40114d <phase_6+67>
   0x0000000000401166 <+92>:   add    $0x4,%r13
   0x000000000040116a <+96>:   jmp    0x40112c <phase_6+34>
   0x000000000040116c <+98>:   lea    0x48(%rsp),%rsi
   0x0000000000401171 <+103>:  mov    %r14,%rax
   0x0000000000401174 <+106>:  mov    $0x7,%ecx
   0x0000000000401179 <+111>:  mov    %ecx,%edx
   0x000000000040117b <+113>:  sub    (%rax),%edx
   0x000000000040117d <+115>:  mov    %edx,(%rax)
   0x000000000040117f <+117>:  add    $0x4,%rax
   0x0000000000401183 <+121>:  cmp    %rsi,%rax
   0x0000000000401186 <+124>:  jne    0x401179 <phase_6+111>
   0x0000000000401188 <+126>:  mov    $0x0,%esi
   0x000000000040118d <+131>:  jmp    0x4011af <phase_6+165>
   0x000000000040118f <+133>:  mov    0x8(%rdx),%rdx
   0x0000000000401193 <+137>:  add    $0x1,%eax
   0x0000000000401196 <+140>:  cmp    %ecx,%eax
   0x0000000000401198 <+142>:  jne    0x40118f <phase_6+133>
   0x000000000040119a <+144>:  jmp    0x4011a1 <phase_6+151>
   0x000000000040119c <+146>:  mov    $0x6042f0,%edx
   0x00000000004011a1 <+151>:  mov    %rdx,(%rsp,%rsi,2)
   0x00000000004011a5 <+155>:  add    $0x4,%rsi
   0x00000000004011a9 <+159>:  cmp    $0x18,%rsi
   0x00000000004011ad <+163>:  je     0x4011c4 <phase_6+186>
   0x00000000004011af <+165>:  mov    0x30(%rsp,%rsi,1),%ecx
```

```
0x00000000004011b3 <+169>:   cmp    $0x1,%ecx
0x00000000004011b6 <+172>:   jle    0x40119c <phase_6+146>
0x00000000004011b8 <+174>:   mov    $0x1,%eax
0x00000000004011bd <+179>:   mov    $0x6042f0,%edx
0x00000000004011c2 <+184>:   jmp    0x40118f <phase_6+133>
0x00000000004011c4 <+186>:   mov    (%rsp),%rbx
0x00000000004011c8 <+190>:   lea    0x8(%rsp),%rax
0x00000000004011cd <+195>:   lea    0x30(%rsp),%rsi
0x00000000004011d2 <+200>:   mov    %rbx,%rcx
0x00000000004011d5 <+203>:   mov    (%rax),%rdx
0x00000000004011d8 <+206>:   mov    %rdx,0x8(%rcx)
0x00000000004011dc <+210>:   add    $0x8,%rax
0x00000000004011e0 <+214>:   cmp    %rsi,%rax
0x00000000004011e3 <+217>:   je     0x4011ea <phase_6+224>
0x00000000004011e5 <+219>:   mov    %rdx,%rcx
0x00000000004011e8 <+222>:   jmp    0x4011d5 <phase_6+203>
0x00000000004011ea <+224>:   movq   $0x0,0x8(%rdx)
0x00000000004011f2 <+232>:   mov    $0x5,%ebp
0x00000000004011f7 <+237>:   mov    0x8(%rbx),%rax
0x00000000004011fb <+241>:   mov    (%rax),%eax
0x00000000004011fd <+243>:   cmp    %eax,(%rbx)
0x00000000004011ff <+245>:   jge    0x401206 <phase_6+252>
0x0000000000401201 <+247>:   callq  0x4015a4 <explode_bomb>
0x0000000000401206 <+252>:   mov    0x8(%rbx),%rbx
0x000000000040120a <+256>:   sub    $0x1,%ebp
0x000000000040120d <+259>:   jne    0x4011f7 <phase_6+237>
0x000000000040120f <+261>:   add    $0x50,%rsp
0x0000000000401213 <+265>:   pop    %rbx
0x0000000000401214 <+266>:   pop    %rbp
0x0000000000401215 <+267>:   pop    %r12
0x0000000000401217 <+269>:   pop    %r13
0x0000000000401219 <+271>:   pop    %r14
0x000000000040121b <+273>:   retq
End of assembler dump.
```

Phase_6 은 6 개의 정수를 입력받고, <phase+6 + 37 ~ 47>에 의해 첫 번째 정수는 6 보다 작거나 같아야 한다. 우선, <phase_6 + 67 ~ 90>에서 반복문을 분석해보면 입력한 여섯 개의 숫자는 서로 다르면서 6 보다 작거나 같고 0 은 아닌 1, 2, 3, 4, 5, 6 중 각각 하나여야 한다. 그 후 <phase_6 + 111 ~ 124> 반복문을 돌면서 여섯 개의 정수 입력에 대해 각각 7 – 입력 값으로 바꿔준다. 이후 <phase_6 + 133 ~ 184>에서도 반복문이 수행되는데, 이때 $0x6042f0 에 저장되어 있는 값을 확인해보면, 아래와 같이 linked list 형태로 6 개의 node 가 연결되어 있음을 알 수 있고, 각 node 에는 924, 568, 158, 704, 365, 322 값이 저장되어 있다.

```
(gdb) x/48wd 0x6042f0
0x6042f0 <node1>:     924    1    6308608 0
0x604300 <node2>:     568    2    6308624 0
0x604310 <node3>:     158    3    6308640 0
0x604320 <node4>:     704    4    6308656 0
0x604330 <node5>:     365    5    6308672 0
0x604340 <node6>:     322    6    0       0
```

<phase_6 + 133 ~ 184>에서는 반복문을 돌며 rsp + 48, rsp + 52, rsp+ 56, … , rsp + 68 주소에 input 으로 입력한 정수값을 저장하고, rsp + 0, rsp + 8, …, rsp + 40 주소에 각 정수값에 해당하는 node 번호에 저장되어 있는 값(앞으로 노드값이라 표현)을 저장한다. 그 후 어셈블리 코드를 보면, <phase_6 + 245>를 보면 알 수 있듯 이전 노드값이 다음 노드값보다 커야, 즉 노드값이 내림차순이 되어야 explode_bomb 을 피할 수 있다. 이때, 노드 값이 큰 순서대로 정렬하면 1, 4, 2, 5, 6, 3 인데, 앞에서 7 – 입력값으로 입력값을 대체하는 과정이 있었으므로 input 값은 7 – 1, 7 – 4, 7 – 2, 7 – 5, 7 – 6, 7 – 3 인 6, 3, 5, 2, 1, 4 가 되어야 한다.

```
Good work!  On to the next...
6 3 5 2 1 4
Congratulations! You've defused the bomb!
```