<201511646_나여영_Report>

문제 1

Bomb가 발생한 phase_1 함수에 브레이크 포인트를 걸어 놓고 info reg를 이용해 레지스터 정보들을 가져와서 x/d , x/s $reg 명령어를 이용해 레지스터 값들을 확인 했으나 답에 대한 정보는 찾을 수 없었다. (rsi 와 rdi값이 같은 것만 확인했다)

이후 si 명령어를 이용해 함수 내부로 진입했더니 si 3번째 단계에서 strings not equals이 호출됨을 확인하고, 다시 gdb run을 실행해서 phase_1 함수의 2번째 줄까지 진행한 후 info reg를 이용해 레지스터 정보를 확인했다.

그런데 이전과 다르게 rsi와 rdi값이 다름을 확인하여, x/d $rsi 명령어를 이용해 정답이 underflow 임을 찾았다.

답:underflow

문제 2

disas phase_2를 분석하였을 때, phase_2 함수에서 read_six_numbers 함수를 호출하여 6개의 값을 받아오는 것을 확인할 수 있었다. 또한, rsp 레지스터에 그 배열 값들을 차례로 저장함을 알 수 있다. read_six_number를 수행한 이후에는 첫 번째 배열값을 비교하기 시작한다.

```
0x0000000000400fba <+14>:   cmpl   $0x1,(%rsp)
0x0000000000400fbe <+18>:   je     0x400fe5 <phase_2+57>
0x0000000000400fc0 <+20>:   callq  0x40l7al <explode_bomb>
...
```

14 line에서 rsp의 시작주소에 저장된 값을 숫자 1과 비교하여 같을 때, 57번째 라인으로 진행하고, 같지 않으면 explode_bomb를 수행하는 것으로 보아 첫 번째 배열 값은 1임을 알 수 있다.

```
0x0000000000400fe5 <+57>:   lea    0x4(%rsp),%rbp
0x0000000000400fea <+62>:   mov    $0xl,%ebx
0x0000000000400fef <+67>:   jmp    0x400fc7 <phase_2+27>
```

rsp + 4의 주소에 위치한 값을 rbp로 저장한 후, ebx에 1을 저장하는데, 이는 for문에서 카운트를 하는 변수에 해당한다. 이렇게 값들을 초기화 한 뒤에 27 line으로 이동하여 본격적인 배열의 비교를 시작한다.

```
0x0000000000400fc7 <+27>:   add    $0xl,%ebx
0x0000000000400fca <+30>:   mov    %ebx,%eax
0x0000000000400fcc <+32>:   imul   -0x4(%rbp),%eax
0x0000000000400fd0 <+36>:   cmp    %eax,0x0(%rbp)
0x0000000000400fd3 <+39>:   je     0x400fda <phase_2+46>
0x0000000000400fd5 <+41>:   callq  0x40l7al <explode_bomb>
0x0000000000400fda <+46>:   add    $0x4,%rbp
0x0000000000400fde <+50>:   cmp    $0x6,%ebx
```

27line 이후에는 index값 역할을 하는 ebx 값을 배열의 이전 값과 곱해 eax에 저장하여 새로운 배열 위치의 값과 비교하는 반복문을 수행한다. 이를 통해 배열의 식은

(n번째 값) = (n-1번째 값)*n이 됨을 알 수 있다. 이를 통해 정답을 찾으면 1 2(1*2) 6(2*3) 24(6*4) 120(24*5) 720(120*6)이 된다.

정답 : 1 2 6 24 120 720

문제3

```
0x0000000000401006 <+14>:    mov    $0x40le7c,%esi
```

14 line을 수행한 후 esi 레지스터의 값을 확인해보면 "%d %d"임을 확인할 수 있다. 이를 통해 숫자 2개를 입력해야 함을 확인할 수 있다.

```
0x000000000040102a <+50>:    jmpq   *0x401b60(,%rax,8)
```

에서 rax값에 따른 점프테이블을 이용해 switch문이 구성되어 있는데, 점프테이블은 아래와 같은 명령어로 확인할 수 있다.

```
(gdb) x/8a 0x401b60
0x401b60:    0x401038 <phase_3+64>    0x401031 <phase_3+57>
0x401b70:    0x401044 <phase_3+76>    0x40104e <phase_3+86>
0x401b80:    0x40105a <phase_3+98>    0x401066 <phase_3+110>
0x401b90:    0x401072 <phase_3+122>   0x40107e <phase_3+134>
```

첫 번째 인자(rax)값이 0일때와 1일때만 순서가 다르고, 나머지 경우에는 순서에 맞게 점프하도록 테이블이 구성되어 있다. 또한, 0일때만 초기값이 0x39로 다르고, 나머지의 경우에는 0x00으로 일치하여 점프 순서에 맞게 각 값에 따른 eax값을 새롭게 구할 수 있다.

그런데, 아래와 같은 조건에 의해 첫 번째 인자는 5보다 작거나 같아야 한다.

```
0x0000000000401094 <+156>:  cmpl   $0x5,0x8(%rsp)
0x0000000000401099 <+161>:  jg     0x40l0al <phase_3+169>
...
0x00000000004010al <+169>:  callq  0x40l7al <explode_bomb>
```

따라서 답은 각각 0 -188 / 1 -245 / 2 -18 / 3 -95 / 4 338 / 5 -253으로 5가지 경우가 가능하다.

답: 0 -188 (or 1 -245 , 2 -18 , 3 -95, 4 338, 5 -253)

문제4

```
0x00000000004010da <+9>:    mov    $0x401e7f,%esi
```

에서 esi의 레지스터 값을 확인해 본 결과 "%d"로, 하나의 정수 값을 받음을 확인할 수 있다.

```
0x00000000004010e9 <+24>:   cmp    $0x1,%eax
0x00000000004010ec <+27>:   jne    0x4010f5 <phase_4+36>
...
0x00000000004010f5 <+36>:   callq  0x4017a1 <explode_bomb>
```

에 의해 인자는 1보다 커야 함을 알 수 있다.

이후, 이 값을 edi 레지스터에 저장하여 func4 함수를 호출하는데 이 함수는 초기값으로 rax를 1로 잡고, 아래와 같은 작업을 edi에 저장된 횟수만큼 반복하는 것으로 아래와 같다.

```
0x00000000004010ab <+0>:    mov    $0x1,%eax
0x00000000004010b0 <+5>:    test   %edi,%edi
0x00000000004010b2 <+7>:    jle    0x4010cf <func4+36>
...
0x00000000004010bb <+16>:   callq  0x4010ab <func4>
0x00000000004010c0 <+21>:   lea    0x0(,%rax,8),%edx
0x00000000004010c7 <+28>:   sub    %eax,%edx
0x00000000004010c9 <+30>:   mov    %edx,%eax
```

위 어셈블리 코드를 분석하면 edx = rax*8이고, rax = edx-rax의 형태로 저장하여 최종적으로 rax = 8*rax – rax = 7*rax 형태가 됨을 알 수 있다. 이는 7^{edi} 를 구하는 재귀함수가 되고 구해진 값은 eax에 저장되어 리턴된다.

```
0x0000000000401103 <+50>:   cmp    $0xc90f7,%eax
0x0000000000401108 <+55>:   je     0x40110f <phase_4+62>
0x000000000040110a <+57>:   callq  0x4017a1 <explode_bomb>
```

에 의해 eax값은 0xc89f7(823,543)과 같아야 하는데, 823,543은 7^7 이므로, 답은 7이 된다.

답: 7

문제5

```
0x0000000000401122 <+14>:   mov    $0x40e7c,%esi
```

에서 esi에 넣은 값을 x/s 명령어를 이용해 확인하면 "%d %d"가 된다. 이를 통해 숫자 2개를 입력 값으로 받음을 확인할 수 있다.

```
0x000000000040113b <+39>:   mov    0x8(%rsp),%eax
0x000000000040113f <+43>:   and    $0xf,%eax
0x0000000000401142 <+46>:   mov    %eax,0x8(%rsp)
```

에서 첫 번째 인자를 eax에 넣은 후 이를 16으로 나눈 나머지를 구하는 것을 볼 수 있다. 이를 통해 첫 번째 인자에 어떤 수를 넣던지에 상관없이 16으로 나눈 나머지로 변환되어 0~15 범위에 있는 숫자로 변환된다는 것을 알 수 있다.

```
0x0000000000401146 <+50>:   cmp    $0xf,%eax
0x0000000000401149 <+53>:   je     0x401177 <phase_5+99>
...
0x0000000000401177 <+99>:   callq  0x4017a1 <explode_bomb>
```

또한 위 어셈블 코드를 확인하면 첫 번째 인자를 16으로 나눈 나머지가 15가 되면 안된다는 점 또한 알 수 있다.

```
0x000000000040114b <+55>:   mov    $0x0,%ecx
0x0000000000401150 <+60>:   mov    $0x0,%edx
0x0000000000401155 <+65>:   add    $0x1,%edx
...
0x000000000040115a <+70>:   mov    0x401ba0(,%rax,4),%eax
0x0000000000401161 <+77>:   add    %eax,%ecx
0x0000000000401163 <+79>:   cmp    $0xf,%eax
0x0000000000401166 <+82>:   jne    0x401155 <phase_5+65>
```

전체적인 코드를 보면 반복문 형태로 구성 되어있다. (eax값이 15가 될 때까지 반복하는 구조이다)

초기 eax(rax) 값은 직접 암호로 입력한 첫 번째 인자인데, 이를 인덱스로 하는 int형 배열의 주소를 찾아 그 주소에 해당하는 정수 값을 가져오도록 되어있다(70 line). 그 가져온 값을 앞에서 0으로 초기화한 ecx에 더해서 넣고 그 값이 15가 되는지 확인하여 되지 않았을 경우 65line부터 다시 반복하는 구조이다. 반복문이 진행되는 동안 edx값은 0부터 시작해서 1씩 추가 된다.

```
0x0000000000401168 <+84>:   mov    %eax,0x8(%rsp)
0x000000000040116c <+88>:   cmp    $0xc,%edx
0x000000000040116f <+91>:   jne    0x401177 <phase_5+99>
0x0000000000401171 <+93>:   cmp    0xc(%rsp),%ecx
0x0000000000401175 <+97>:   je     0x40117c <phase_5+104>
0x0000000000401177 <+99>:   callq  0x4017a1 <explode_bomb>
```

이렇게 계속 반복하면서 eax값이 15가 되었을 때 두 가지를 확인하게 되는데, edx값(반복문 반복 횟수)이 12가 되어야 하고, 암호로 넣은 두 번째 인자 값이 ecx값(반복되면서 구해진 eax값을 모두 합친 값)과 같아야 한다. 이를 구하기 위해서는 0x401ba0 위치에 있는 배열을 확인해야 하는데, 확인해본 결과 arr[0] = 10, arr[1] = 2, … 14 7 8 12 15 11 0 4 1 13 3 9 6, … , arr[15] = 5 순서로 되어있음을 확인하였다.

그런데, 반복횟수가 12가 되기 위해서는 15가 나올 때까지 뒤로 12회 역추적 해야한다. 역추적 했을 때 15 – 6 – 14 – 2 – 1 – 10 – 0 – 8 – 4 – 9 – 13 – 11 – 7 – … 순서로 됨을 확인했다. 15로부터 12회 역추적하면 11이라는 값이 첫 eax값이 되어야 하는데, 이 eax값을 구하기 위해서는 11을 값으로 갖는 배열 인덱스 7이 첫 번째 암호로 들어가야 한다 (물론, 이를 16으로 나눈 나머지를 구하는 것이기 때문에 23, 39 … 등도 상관없이 사용이 가능하다.

역추적 된 순서대로 15부터 11까지 더한 값이 ecx값, 즉 두 번째 값이 되어야 하는데 이를 다 더하면 93이 된다.

답 : 7 93

문제6

문제6은 결과적으로 fun6을 통해 구해진 값 (결과적으로 링크드 리스트형태)에서 5번째 인자를 암호로 넣어야 하는 형태이다. fun6을 살펴보면 매우 복잡한 구조로 되어있는데, fun6을 시작하기 전에 phase_6 함수에서 edi 레지스터에 하나의 주소를 할당하는 것을 볼 수 있다. 그 주소를 x/x 명령어로 확인해보면 Node라는 이름으로 값이 나오는 것을 보고 힌트를 얻어 x/36x로 그 값들을 확인해 보았다.

```
0x603320 <node1>:    0x0000020d    0x00000001    0x00603330    0x00000000
0x603330 <node2>:    0x0000008f    0x00000002    0x00603340    0x00000000
0x603340 <node3>:    0x000000ec    0x00000003    0x00603350    0x00000000
0x603350 <node4>:    0x00000103    0x00000004    0x00603360    0x00000000
0x603360 <node5>:    0x0000027c    0x00000005    0x00603370    0x00000000
0x603370 <node6>:    0x000000da    0x00000006    0x00603380    0x00000000
0x603380 <node7>:    0x00000334    0x00000007    0x00603390    0x00000000
0x603390 <node8>:    0x000002d4    0x00000008    0x006033a0    0x00000000
0x6033a0 <node9>:    0x0000032d    0x00000009    0x00000000    0x00000000
```

노드의 형태를 보면 16바이트 크기의 노드들로 구성되어 있는데, 노드들을 4바이트씩 나눠 보면 value / index / next_node_addr / flag(?) 정도로 확인할 수 있다. 이 때 node9는 next_node_addr가 0인데, 이를 통해 링크드 리스트의 끝임을 유추해볼 수 있다.

그런데 fun6을 한줄씩 진행하면서 이 노드들을 확인하면 중간중간 next_node_addr가 0이 되는 경우가 발생한다.

```
0x603320 <node1>:    0x0000020d    0x00000001    0x00603350    0x00000000
0x603330 <node2>:    0x0000008f    0x00000002    0x00000000    0x00000000
0x603340 <node3>:    0x000000ec    0x00000003    0x00603330    0x00000000
0x603350 <node4>:    0x00000103    0x00000004    0x00603340    0x00000000
0x603360 <node5>:    0x0000027c    0x00000005    0x00603370    0x00000000
0x603370 <node6>:    0x000000da    0x00000006    0x00603380    0x00000000
0x603380 <node7>:    0x00000334    0x00000007    0x00603390    0x00000000
0x603390 <node8>:    0x000002d4    0x00000008    0x006033a0    0x00000000
0x6033a0 <node9>:    0x0000032d    0x00000009    0x00000000    0x00000000
```

그런데, node9의 next_node_addr도 동시에 계속 0이고, 서로 다음 노드에 대한 주소가 바뀌는 것을 확인하고 연결해보니 두 개의 링크드 리스트가 생성되는 것을 알 수 있었다.

또한 disas을 이용해 어셈블 코드를 확인하면서 리스트의 값의 변화를 확인해 보니 value 크기가 큰 순서대로 새로운 링크드 리스트를 생성하는 것을 확인할 수 있었다. 이를 통해 fun_6은 내림차순 정렬 함수임을 확인 하였다.

그 후 정렬된 링크드 리스트의 5번째 숫자를 확인해보니 0x020d(525)임을 확인하여 정답을 찾았다.

답 : 525

추가문제

secret_phase를 찾기 위해 소스코드를 찾아 봤는데, 소스코드에서 사용하는 함수는 phase_1 ~ phase_6과 phase_defused, initialize_bomb 3가지였다. 그 중 phase_defused를 disas 명령어로 열어보니 secret_phase가 있음을 확인할 수 있었다.

[phase_defused]

```
0x00000000004018fe <+20>:    cmpl   $0x6,0x20216f(%rip)
0x0000000000401905 <+27>:    jne    0x401960 <phase_defused+118>
```

그런데, 일반적인 case에서는 secret_phase 근처로 아예 진입하지 못하고 계속 점프하였는데, 이유를 살펴 본 결과 0x20216f(%rip)가 문제마다 1씩 증가하는데, 이를 6과 비교했을 때 6과 다르면 계속 점프 하는 것을 확인할 수 있었다. 즉, secret_phase를 진행하기 위해선 1차적으로 6번 문제를 진행하는 중에 phase_defused가 진행되어야 함을 알 수 있었다.

```
0x0000000000401911 <+39>:    mov    $0x401ec6,%esi
0x0000000000401916 <+44>:    mov    $0x603b70,%edi
0x000000000040191b <+49>:    callq  0x400c90 <__isoc99_sscanf@plt>
0x0000000000401920 <+54>:    cmp    $0x2,%eax
0x0000000000401923 <+57>:    jne    0x401956 <phase_defused+108>
```

첫 번째 과정을 통과한 후에 어느 특정한 값을 가져오게 되는데, 0x401ec6 주소로 접근하여 값을 살펴보니 "%d %s"였다. 이는 뒤에 나올 scanf(49line)에서 정수, 문자열 두 개의 자료를 받아온다는 뜻이다. 그런데, 0x603b70에 접근하여 값을 출력해보니 phase_4의 정답으로 넣은 solution 파일의 4번째 줄에 작성한 내용이 출력이 되었다. 즉, 기존에 답을 7로 작성하였기 때문에 7이 그대로 출력이 됨을 확인할 수 있었다.

한편, 54line에서 eax값이 2인지 비교하는 부분이 있는데, 이 때 eax값은 scanf 결과 얻은 변수의 개수를 의미한다. 그런데, 위 0x603b70에서 edi 레지스터로 가져온 값이 7밖에 없으므로 scanf에서는 변수를 한 개 밖에 가져오지 못해 eax값은 1이 되어 secret_phase 근처로 접근하지 못하고

57 line에 의해 점프하게 된다. 결과적으로 이 부분에서 4번 문제의 정답 줄에 숫자와 더불어 하나의 문자열을 입력해야 함을 알 수 있다.

```
0x0000000000401925 <+59>:   mov    $0x401ecc,%esi
0x000000000040192a <+64>:   lea    0x10(%rsp),%rdi
0x000000000040192f <+69>:   callq  0x401358 <strings_not_equal>
0x0000000000401934 <+74>:   test   %eax,%eax
```

이후 0x401ecc값을 가져와 입력 값으로 받은 문자열과 비교하는데, 0x401ecc에 접근하여 값을 확인해보니 "austinpowers"라는 값이 나왔다. 이를 통해 4번 문제 정답 줄에 "austinpowers"를 추가해야 secret_phase 근처로 접근할 수 있음을 확인했다.

[secret_phase]

```
0x0000000000401269 <+1>:    callq  0x401805 <read_line>
...
0x0000000000401283 <+27>:   lea    -0x1(%rax),%eax
0x0000000000401286 <+30>:   cmp    $0x3e8,%eax
0x000000000040128b <+35>:   jbe    0x401292 <secret_phase+42>
0x000000000040128d <+37>:   callq  0x4017a1 <explode_bomb>
```

secret_phase에서는 새로운 라인의 값을 가져와 그 값에서 1을 뺀 값을 1000과 비교하여 큰 경우 폭탄이 터지고, 작거나 같은 경우 계속해서 진행할 수 있도록 되어있다.

```
0x0000000000401292 <+42>:   mov    %ebx,%esi
0x0000000000401294 <+44>:   mov    $0x603140,%edi
0x0000000000401299 <+49>:   callq  0x40122a <fun7>
0x000000000040129e <+54>:   cmp    $0x1,%eax
0x00000000004012a1 <+57>:   je     0x4012a8 <secret_phase+64>
0x00000000004012a3 <+59>:   callq  0x4017a1 <explode_bomb>
```

이후 받아온 값을 esi레지스터에 입력하고, 0x603140을 edi 레지스터에 입력한 후 fun7을 호출하는데, 호출 후 eax값이 1이 되어야 secret_phase가 defuse 되는 구조이다. 그런데 0x603140에 접근하여 값을 확인해보니

```
(gdb) x/x 0x603140
0x603140 <n1>:0x24
```

라고 출력이 되는 것을 확인하였다. n1이 의미하는 바가 node 1이라는 추측을 하여 x/64x 명령어로 출력을 해보았다. 내용은 아래와 같다.

```
(gdb) x/64x 0x603140
0x603140 <n1>:0x24    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x603148 <n1+8>:      0x60    0x31    0x60    0x00    0x00    0x00    0x00    0x00
0x603150 <n1+16>:     0x80    0x31    0x60    0x00    0x00    0x00    0x00    0x00
0x603158:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x603160 <n21>:       0x08    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x603168 <n21+8>:     0xe0    0x31    0x60    0x00    0x00    0x00    0x00    0x00
0x603170 <n21+16>:    0xa0    0x31    0x60    0x00    0x00    0x00    0x00    0x00
0x603178:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

n1+8의 위치에 n21의 주소가 저장이 되어있었고, x/96x 명령어로 접근해보니 n1+16의 위치에 있는 값이 n22의 주소라는 것을 확인 하였다. 같은 방식으로 n21의 주소(0x603160)와 n22의 주소(0x603180)를 x/96x등의 형태로 접근해보니 결과적으로 n1, n21, n22, n31... 등은 트리의 노드 역할을 하여 전체적으로 트리모양의 구조를 가짐을 확인하였다. 트리 구조는 다음과 같다.

n1 : 0x24

n21 : 0x08 / n22 : 0x32

n31 : 0x06 / n32 : 0x16 / n33 : 0x2d / n34 : 0x6b

n41 : 0x01 / n42 : 0x07 / n43 : 0x14 / n44 : 0x23 / n45 : 0x28 / n46 : 0x2f / n47 : 0x63 / n48 : 0x3e9

```
0x0000000000401233 <+9>:    mov    (%rdi),%edx
0x0000000000401235 <+11>:   cmp    %esi,%edx
0x0000000000401237 <+13>:   jle    0x401246 <fun7+28>
```

한편, fun7에서는 처음에 rdi 레지스터에 저장된 주소(root node)에 접근하여 값을 edx에 저장하고 이 값과 사용자 입력값(esi, 암호값)을 비교하여 작거나 같을 때와 클 때로 나누어 연산을 진행한다.

1) 작거나 같을 때

```
0x0000000000401239 <+15>:   mov    0x8(%rdi),%rdi
0x000000000040123d <+19>:   callq  0x401220 <fun7>
0x0000000000401242 <+24>:   add    %eax,%eax
0x0000000000401244 <+26>:   jmp    0x401263 <fun7+57>
```

암호값이 rdi 레지스터에 있는 주소로 접근했을 때 얻는 값보다 작을 때 rdi+8 위치의 값(left child node의 주소)을 새로운 rdi값을 바꾸고 fun7을 재귀호출 한다. 이후 eax값에 eax+eax를 저장한다. 즉, eax = eax*2의 값이 되는 것이다.

2) 클 때

```
0x0000000000401246 <+28>:   mov    $0x0,%eax
0x000000000040124b <+33>:   cmp    %esi,%edx
0x000000000040124d <+35>:   je     0x401263 <fun7+57>
0x000000000040124f <+37>:   mov    0x10(%rdi),%rdi
0x0000000000401253 <+41>:   callq  0x401220 <fun7>
0x0000000000401258 <+46>:   lea    0x1(%rax,%rax,1),%eax
0x000000000040125c <+50>:   jmp    0x401263 <fun7+57>
```

암호값이 rdi 레지스터에 있는 주소로 접근했을 때 얻는 값보다 크거나 같을 때는 eax값을 0으로 초기화 한 뒤, 만약 같으면 secret_phase를 종료하고, 크면 rdi+10(right child node의 주소)의 값을 rdi값으로 바꾸고 fun7을 재귀호출 한다. 이후 eax값에 rax+rax+1을 저장한다. 즉, eax = 2*rax + 1 이 되는 것이다.

정리하자면, fun7은 암호값에 따라 root node로부터 left 혹은 right로 트리를 재귀적으로 탐색하는 방식인데, left로 가느냐 right로 가느냐에 따라 eax 값이 어떻게 변화하는지를 정리하면 답을 구할 수 있다. 결과적으로 1이 나오려면 left탐색 횟수에 상관없이 right탐색만 마지막으로 하면 된다 (eax right 연산시 안의 연산들이 다 0으로 초기화 되기 때문에 left탐색 횟수에 상관없이 결과적으로 1이 된다.) 따라서 처음에 right 연산을 하고 left 연산을 하게되는 0x2d(45), 0x28(40), 0x32(50)가 답이 된다.

답 : 40, 45, 50