CSED211: Bomb Lab Report

20230499/KimJaeHwan

1. Overview

이번 랩에서는 6단계로 이루어진 폭탄을 해체하는 것을 목표로 한다. 폭탄은 실행 파일 형식으로 주어지며, 적절한 문자열 한 줄을 입력하여 각 단계를 해체할 수 있다. 폭탄은 해체자 마다 다른 폭탄을 받으며, 이 보고서에는 38번째 폭탄의 해체에 대하여 설명한다. 본 보고서에서 + 숫자 형식으로 위치를 표시하였는데, 이는 포스텍 프로그래밍 서버에서 어떠한 옵션 없이 gdb를 실행하였을 때, disas (함수 이름) 명령을 통해 얻은 어셈블리 코드를 기준으로 한다. 추가로, 폭탄을 어셈블리 코드로 변환하여 볼 수 없는 경우에는, 다음 링크를 통해 전체 어셈블리 코드를 볼 수 있다.

https://excalidraw.com/#json=MC3_0GSFgaM00eTLZ-AlF,JGBJxz_6_U3JTKtXtniylQ

2. Before start

폭탄을 본격적으로 해체하기 위해서, 몇몇 핵심적인 함수를 분석하여 전체 구조를 파악할 수 있다. 먼저, main 함수를 보면, +111 위치에서 폭탄을 초기화한 후, 문자열 입력, 단계 함수를 실행한 후, 폭발하지 않으면 폭탄을 해체하는 함수가 실행된다. 총 여섯 단계가 모두 해체된 후, 프로그램은 종료된다. 단계마다, read_line 함수를 통해 한 문자를 읽어온 후 입력 값을 rax 레지스터를 통해 반환한다. 이 값은 rdi 레지스터에 복사되어 각 phase_n 함수에 매개변수로서 전달된다. 또한, 이번 랩에서 폭탄이 터지면 감점 사항이 있다. 따라서, 다음 명령을 통해 폭탄이 터지지 않도록 할 수 있다.

```
(gdb) b explode_bomb
Breakpoint 1 at 0x401524
```

3. Question #1. phase_1

첫 단계를 뜻하는 함수인 phase_1 함수를 어셈블리로 표현하면 매우 간단하다. 가장 먼저, 폭탄을 회피하는 조건을 먼저 보면, +16 위치에서 ZF(zero flag)가 설정되어 있어야 한다. 이를 위해서 +14 위치에서 test 인스트럭션을 수행하는데, 두 오퍼랜드가 모두 eax 레지스터이므로 eax 레지스터에 0이 아닌 값이 있다면 폭발을 피할 수 있다. eax 레지스터는 함수의 반환 값을 저장하므로 strings_not_equal 함수의 반환 값을 저장하고 있을 것이다. 해당 함수의 이름이 올바르게 작성되어 있다면, 두 매개변수를 받아 두 매개변수 위치에 저장된 문자열이 같은지 확인하는 함수라고 추측할 수 있다. 어셈블리어의 콜링 컨벤션에 따르면, 처음 두 매개변수는 각각 rdi, rsi 레지스터를 통해 넘겨진다. rdi 레지스터의 값은 phase_1 함수 호출 전, 우리가 입력한 문자열이 저장되어 있다. esi 레지스터는 phase_1+4 위치에서 0x402520 메모리에 저장된 문자열을 복사한다. 따라서, 이번 단계의 정답은 0x402520 메모리에 저장되어 있는 문자열과 같은 문자열을 입력하는 것이다. 다음 명령을 통해 해당 메모리에 저장된 값을 출력할 수 있다.

```
(gdb) x/s 0x402520
0x402520:       “The Moon unit will be devided into two divisions.“
```

‘x’ 명령은 메모리에 저장된 값을 출력하라는 명령어이며, s 옵션은 문자열 형식으로 출력하라는 뜻이다. 따라서 첫 단계의 답은 다음과 같다.

```
Answer: The moon unit will be divided into two divisions.
```

4. Question #2. phase_2

두 번째 단계는 첫 번째 단계에서는 보지 못했던 read_six_numbers 함수를 호출하는 부분이 있다. 해당 함수의 이름을 통해 6개의 정수를 입력받음을 추측할 수 있다. 해당 함수의 어셈블리 코드를 보면 입출력 라이브러리의 sscanf 함수를 호출하여, 해당 함수를 호출한 이전 함수의 스택 메모리에 정수를 저장한다. read_six_numbers+36 위치에서 esi 레지스터에 0x4032e8 메모리의 값을 복사하며 해당 메모리의 값을 확인해보면 다음과 같다.

```
(gdb) x/s 0x4032e8
0x4032e8:       “%d %d %d %d %d %d“
```

즉, 해당 주소는 입력받을 형식을 저장하는 메모리 주소이며, 따라서 본 함수가 정수 6개를 입력받는다는 사실을 다시 확인할 수 있다. 이 함수 내에서도 폭발할 수 있는 여지가 있는데, +51, +54 위치에서 입력한 정수가 5개보다 적거나 같은 경우 폭발한다. 즉, 이 함수를 요약하자면 phase_2 함수의 rsp 값이 들어있는 rsi 레지스터의 위치에, 입력받은 문자열이 들어있는 rdi 레지스터로부터 6개의 정수를 읽어 저장하는 함수이다.

다시 phase_2 함수로 돌아와 +14 위치부터 보자. +14부터 +20까지는 첫 번째 정수가 음수인지 확인하여, 음수라면 폭발시킨다. 첫 번째 정수가 0이나 양수라면 해당 분기를 통과하며, +56으로 이동한다. 다음으로, 점프 구문을 살펴보면 +27부터 +52까지 반복문으로 작성되어 있음을 알 수 있다. 가장 먼저 +56부터 +66 위치의 명령을 수행한 후 반복문을 진입하며, 반복문을 빠져나가기 위해서는 +49 위치에서 ebx 레지스터가 6이 되어야 한다. ebx 레지스터는 루프 카운터로서, 반복마다 1, 2, 3, 4, 5의 값을 가지고 반복문을 수행한 후, 6이 되면 종료된다.

이제 반복문 내에서 어떤 명령을 수행하는지 보자. rsp 레지스터의 위치에 입력받은 여섯 개의 정수가 보관되어 있으므로 rbp 레지스터는 처음에 +56 위치에서 입력 중 두 번째 정수를 보관하고 있다. 반복문이 한번 동작할 때마다, +45 위치에서 다음 정수의 값을 저장하게 된다.

반복문 내 +27, +29 위치에서 eax 레지스터에 현재 루프 카운터의 값과 rbp 레지스터가 가리키는 이전 값을 더해 rbp 레지스터가 가리키는 현재값과 같은지 확인한다. 만약 같다면 폭발을 피할 수 있다. 즉, 6개의 숫자를 각각 $a_1, a_2, \ldots, a_6$, ebx 레지스터의 값을 i 라고 하면, +27부터 +29 위치에서 eax 레지

스터의 값은 $a_i + i$ 로 계산된다는 사실을 알 수 있다. rbp 레지스터는 a_{i+1}의 주소를 뜻하며 +35 위치에서 $a_{i+1} = a_i + i \ (i = 1, 2, 3, 4, 5)$ 조건이 참인지 검사한다. 따라서 답은 다음과 같다.

Answer: 다음 두 규칙을 만족하는 모든 쌍 $(a_1, a_2, a_3, a_4, a_5, a_6)$. 예를 들어, "1 2 4 7 11 16"
1. $a_1 \geq 0$
2. $a_{i+1} = a_i + i \ (i = 1, 2, 3, 4, 5)$

5. Question #3. phase_3

세 번째 단계 함수의 어셈블리 코드를 본다면, 마치 이전 단계의 read_six_numbers 함수와 같이+_24 위치의 sscanf 함수를 통해 입력받은 문자열을 다르게 저장함을 알 수 있다. 이전과 마찬가지로 +14 위치에서 esi 레지스터에 저장되는 0x4032f4 메모리 주소에 들어 있는 값을 보면 입력 형식을 알 수 있다.

```
(gdb) x/s 0x4032f4
0x4032f4:        "%d %d"
```

따라서, 이번 단계의 정답은 두 정수 형태이다. 이 값은, sscanf 함수의 세 번째, 네 번째 매개변수로 전달되는 rdx, rcx 레지스터의 주소에 저장되며, 이 위치는 각각 rsp + 0xc, rsp + 0x8 주소이다.

다시 phase_3 함수로 돌아와서, 가장 먼저 +29 위치부터 +34 위치까지 eax 레지스터에 저장된 sscanf 함수의 반환 값이 1보다 작거나 같다면 폭발한다. 이는 입력으로 두 개가 주어졌는지 확인하는 부분이다. 사실 2개보다 더 많은 정수를 입력받을 수 있으며, 그 이후 값은 무시된다.

다음 부분을 보자. 가장 먼저, rsp + 0xc에 저장된 첫 번째 정수를 7과 비교하여, 7보다 크다면 폭발한다. 만약 7보다 작다면, (0x402580 + 첫 번째 정수 * 8) 위치로 Indirect Jump 한다. 이를 통해 해당 단계는 jump table을 갖는 switch – case 문을 분석해야 함을 알 수 있다. 첫 번째 입력의 검사형식에 따르면, 0부터 7 사이 정수가 가능하므로, 총 case 문은 8가지의 경우가 있을 것이다. 총 8개의 jump table을 다음 명령을 통해 확인할 수 있다.

```
(gdb) x/8xg 0x402580
0x402580:    0x0000000000400f90    0x0000000000400fcd
0x402590:    0x0000000000400f97    0x0000000000400f9e
0x4025a0:    0x0000000000400fa5    0x0000000000400fac
0x4025b0:    0x0000000000400fb3    0x0000000000400fba
```

gdb 디버거에서 x 명령은 메모리의 값을 출력하는 명령어이며, 옵션으로 준 8은 8개의 메모리의 값을 이어서 출력, x는 16진법으로 출력, g는 8바이트의 값을 출력하라는 의미이다. 해당 명령어를 통해 확인한 테이블의 주소는 각각 phase_3 함수의 +57, +118, +64, +71, +78, +85, +92, +99 위치와 같다. 모든 케이스에서, 해당 위치로 점프한 이후 eax 레지스터에 특정한 상수를 저장하고 +123 위치로 이동

한다. eax 레지스터에 저장되는 상수는 첫 번째 입력이 0, 1, 2, 3, 4, 5, 6, 7일 때, 각각 16진법으로는 0xf8, 0x56, 0x34, 0xf7, 0x287, 0x3d0, 0x2ca, 0x1ec이며, 10진법으로는 각각 248, 86, 52, 247, 647, 976, 714, 492이다. 해당 위치에서 폭발을 회피하기 위해서, eax 레지스터에 저장된 값과 rsp + 0x8 위치에 저장된 값이 같아야 한다. rsp + 0x8은 두 번째 입력을 뜻하므로, 정답은 다음과 같다.

Answer: "0 248", "1 86", "2 52", "3 247", "4 647", "5 976", "6 714", "7 792"
다음 중 하나를 입력하면 통과한다. (큰따옴표는 입력에 포함되지 않음)

6. Question #4. phase_4

네 번째 단계의 어셈블리 코드를 보면, 처음 부분이 세 번째 단계와 같으며, 동일한 방식으로 입력이 두 개의 정수가 되어야 함을 알 수 있다. 마찬가지로 rsp+0xc, rsp+0x8 위치에 두 정수가 저장된다. 다음 부분을 보자. 함수의 전체적인 흐름을 파악해 보면, +34, +39 위치에서 첫 번째 입력이 0xe, 즉 14보다 큰 경우 폭발함을 알 수 있다. 이후 func4 함수를 호출하는데, 콜링 컨벤션에 따르면 매개변수는 각각 edi 레지스터가 갖는 첫 번째 입력, esi 레지스터에 저장되는 0, edx 레지스터에 저장되는 0xe = 14, 총 세 개의 매개변수를 갖음을 알 수 있다. 이후 +65 위치에서 func4 함수의 반환 값과 0xf = 15를 비교하며, 반환 값이 15가 아닌 경우 폭발한다. 이후, +70 위치에서 두 번째 입력값이 15인지 검사하며, 아니라면 폭발하게 된다. 즉, 본 함수를 요약하면 두 정수를 입력한 문자열에서 읽어온 후, func4 함수를 실행한 후, 함수의 반환 값과 두 번째 입력이 모두 15가 되어야 폭탄을 해체할 수 있다. 이제 남은 일은 func4를 분석하는 일이다.

func4는 실행 도중 자기 자신을 호출하는 재귀함수이다. +1부터 +12까지 계산을 수행한 후, 해당 값에 따라서 다음 부분에서 재귀를 중단할지, 새로운 재귀함수를 호출할지 결정한다. 편의상 세 매개변수를 각각 a, b, c라고 하자. 그러면 +1 위치부터 +12 위치까지, eax 레지스터에 ((c – b) + (c – b) >> 31) >> 1이 저장된다는 것을 알 수 있다. 일단은 첫 입력에서 b = 0 < c = 14이므로, (c – b) >> 31, 즉 부호비트는 0이다. 따라서 eax 레지스터에는 두 값의 평균이 저장된다.

이후에는 계산한 값과 첫 번째 매개변수와 비교하면서, 다음과 같은 연산을 수행한다.

- 계산한 값과 첫 번째 매개변수를 비교하여, 두 값이 같다면 +33, +37 위치를 거쳐 반환한다.
- 계산한 값이 더 크다면, edx 레지스터에 계산값 – 1을 대입한 이후, func4 함수를 호출한다. 즉, func4(a, b, c)에서 a < (b+c)/2인 경우, func4(a, b, (b+c)/2 – 1).
- 계산한 값이 더 작다면, esi 레지스터에 계산값 + 1을 대입한 이후, func4 함수를 호출한다. 즉, func4(a, b, c)에서 a > (b+c)/2인 경우, func4(a, (b+c)/2 + 1, c)

다음과 같은 재귀함수 중에서, 두 번째 매개변수를 저장하는 esi 레지스터에 저장될 값보다 세 번째 매개변수를 저장하는 edx 레지스터의 값이 항상 더 크므로, 계산 시 부호비트와 관련된 부분을 무시할 수 있다. 또한, 재귀함수의 반환 값은 모든 호출된 모든 재귀에서 계산된 두 번째, 세 번째 매개변수의

평균의 합이며, 이를 저장하기 위해 eax 레지스터와 ebx 레지스터를 이용한다. 모든 입력값에 대해 계산하는 과정을 그림으로 나타내면 다음과 같다.

		di														
Step	argument	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Step 1.	(di, 0, 14)	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
Step 1-1.	(di, 0, 6)	10	10	10	10	10	10	10		↓	↓	↓	↓	↓	↓	↓
Step 1-1-1.	(di, 0, 2)	11	11	11		↓	↓	↓		↓	↓	↓	↓	↓	↓	↓
Step 1-1-1-1.	(di, 0, 0)	11		↓		↓	↓	↓		↓	↓	↓	↓	↓	↓	↓
Step 1-1-1-2.	(di, 2, 2)			13		↓	↓	↓		↓	↓	↓	↓	↓	↓	↓
Step 1-1-2.	(di, 4, 6)					15	15	15		↓	↓	↓	↓	↓	↓	↓
Step 1-1-2-1.	(di, 4, 4)					19		↓		↓	↓	↓	↓	↓	↓	↓
Step 1-1-2-2.	(di, 6, 6)							21		↓	↓	↓	↓	↓	↓	↓
Step 1-2.	(di, 8, 14)									18	18	18	18	18	18	18
Step 1-2-1.	(di, 8, 10)									27	27	27		↓	↓	↓
Step 1-2-1-1.	(di, 8, 8)									35		↓		↓	↓	↓
Step 1-2-1-2.	(di, 10, 10)											37		↓	↓	↓
Step 1-2-2.	(di, 12, 14)													31	31	31
Step 1-2-2-1.	(di, 12, 12)													43		↓
Step 1-2-2-2.	(di, 14, 14)															45

di	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Finaly return value	11	11	13	10	19	15	21	7	35	27	37	18	43	31	45

우리가 원하는 값인 15를 반환하기 위해서는 첫 번째 매개변수, 즉 첫 번째 입력이 5가 되어야 한다. 따라서 답은 다음과 같다.

Answer: 5 15

7. Question #5. phase_5

phase_5 함수를 어셈블리어로 나타내면 꽤 단순해 보인다. 분기 문을 본다면, 다음과 같은 정보를 얻을 수 있다.

- +4 위치에서 string_length 함수를 호출하며, 반환 값은 6이어야 폭발을 피할 수 있다.
- +29 위치부터 +51 위치까지 반복문이 존재한다.
- +53 위치의 분기에서 폭발을 피하기 위해서 edx 레지스터에 저장된 값이 0x3c, 즉 60이어야 한다.
- edx 레지스터는 +24 위치에서 0으로 초기화되며, 반복문 도중 업데이트된다.

즉, 이제 남은 것은 반복문을 분석하는 것이다.

반복문에 진입하기 전에는 +19, +24 위치의 명령을 수행한 후, +29 위치부터 +36 위치까지 연산한 값을 edx 레지스터에 더해주고, rax 레지스터에 1을 더하고, rax 레지스터가 6이라면 반복문을 탈출한다. 이때 eax 레지스터를 0으로 초기화하였지만, 레지스터의 구조에 따르면 rax 레지스터를 통해 동일한

값에 접근할 수 있으므로, 해당 부분에서 eax 레지스터와 rax 레지스터는 동일하게 취급할 수 있다. 이는 rcx 레지스터와 ecx 레지스터 또한 마찬가지이다.

반복문 내에서, 각 레지스터는 다음과 같은 값을 갖는다.

- rax 도는 eax : 루프 카운터로, 반복마다 0, 1, 2, 3, 4, 5를 가지며, 6번의 반복 후 6이 되며 반복을 종료한다.
- rbx : +1 위치에서 rdi, 즉 입력한 문자열의 값을 복사해 저장하고 있는 레지스터
- ecx : +29 위치에서 값을 받으며, rbx 레지스터의 주소로부터 rax 레지스터의 값 * 1 바이트만큼 떨어져 있는 주소의 값. 즉, 입력의 rax 번째 문자. (char 자료형의 크기가 1바이트이므로)

그다음, 반복문을 분석해보면, +29 위치에서 각 문자를 가져와서, +33 위치에서 15와 and 연산한 후, +36 위치에서 0x4025c0 위치로부터 4 * rcx 바이트만큼 떨어져 있는 값을 edx 레지스터에 더한다. 0x4025c0 주소를 출력해보면 다음과 같다.

```
(gdb) x/16wu 0x4025c0
0x4025c0 <array.3161>:   2        10       6        1
0x4025d0 <array.3161+16>:        12       16       9        3
0x4025e0 <array.3161+32>:        4        7        14       5
0x4025f0 <array.3161+48>:        11       8        15       13
```

해당 위치에는 array.3161이라는 변수명이 저장되어 있으며, 다음 주소부터는 변수명이 붙어있지 않다. gdb의 x 명령 중, u 옵션은 10진법으로 출력하라는 의미이며, w는 word, 즉 4바이트로 끊어서 출력하라는 의미이다.

즉, 0x4025c0 메모리 주소에는 배열이 들어있다. 따라서 입력한 문자를 c라 할 때, array.3161[c & 15] 값을 edx 레지스터에 더한다. 예를 들어보자. 만약 입력 문자열이 abcdef인 경우, a 아스키코드는 97이고, 15와 and 연산하면 1이 된다. 따라서 edx에 array.3161[1] = 10을 더한다. 마찬가지로 b와 15를 and 연산하면 2이므로 edx에 array.3161[2] = 6을 더하고, edx 레지스터의 값은 16이 된다. 이렇게 6개의 문자를 돌아가면서 배열에서 값을 골라 더하면, edx는 54가 된다. 이 방식으로, 영어 소문자 a부터 p는 다음과 같은 값을 갖게 된다.

index of array	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
value of array	2	10	6	1	12	16	9	3	4	7	14	5	11	8	15	13
alphabet	p	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o

sum of green	60	concat of green:	abdfmn

이때 편의상 영어 소문자만으로 예를 들었지만, 숫자나 영어 대문자와 같이 입력할 수 있는 아스키코드는 모두 가능하다.

우리의 목적은 배열의 값 중 6개의 값을 더해 60이 되도록 하는 값이다. 이때 그림과 같이 10 + 6 + 12 + 9 + 8 + 15 = 60임을 이용해, 배열의 인덱스가 1, 2, 4, 6, 13, 14가 되는 알파벳인 abdfmn 또한 답이지만, phase_5에서는 문자열에서 중복을 검사하지 않으므로 간단하게 aaaaaa 또한 정답이다. 이 문제의 답은 다음과 같다.

Answer: C++ 코드로 작성하자면 다음과 같다.

```
string str; //input
int arr[16] = {2, 10, 6, 1, 12, 16, 9, 3, 4, 7, 14, 5, 11, 8, 15, 13};
int sumv = 0;
for(auto c: str) sumv += arr[c & 15];
if(sumv == 60 and str.length() == 6) pass();
else explode();
```

예를 들면, aaaaaa, abdfmn 등이 가능하다.

8. Question #6. phase_6

마지막 단계는 이전 단계와는 비교할 수 없을 정도로 길고 복잡하므로, 점프 구문을 관찰하여 작은 단위로 쪼개어 생각할 것이다. 점프 구문을 분석하면 다음과 같은 사실을 파악할 수 있다.

- +31 위치부터 +100 위치까지 반복문이 있으며, 반복문 내에 +71 위치부터 +94 위치까지 반복문이 이중으로 존재한다.
- +102 위치부터 +153 위치까지 반복문이 존재한며, 반복문 내에 이중 반복문이 두 개 존재한다.
- +172 위치부터 +191 위치까지 반복문이 존재한다.
- +206 위치부터 +228 위치까지 반복문이 존재한다.

크게 반복문을 기준으로 부분을 나누어 생각하자. 각 반복문을 첫 번째, 두 번째, 세 번째, 네 번째 반복문이라 하자. 첫 번째 반복문 내에 반복문이 존재한다. 이제 각 단계를 보자.

1) 첫 번째 반복문 이전

- +10, +15 위치에서, rsp + 0x30 위치에 6개의 정수를 저장한다. r13 레지스터 또한 첫 번째 정수의 주소를 갖는다.
- r12d 레지스터에 0을 저장한다.
- eax 레지스터에 첫 번째 정수의 값을 저장한다.

2) 첫 번째 반복문

첫 번째 반복문은 이중 반복문을 갖는다. 첫 번째 반복문에서 사용되는 레지스터의 용도는 다음과 같다.

- r12d: 루프 카운터. 0부터 5까지 반복문을 수행하며, 6이 되면 종료된다. 즉, 0, 1, 2, 3, 4, 5.
- r13: 입력한 6개의 정수 중, rax번째 정수. 0-base index로, 0번째부터 센다.
- rbp: r13과 동일하나, 이중 반복문 내에서 사용되며, 이중 반복문에 들어가기 전 r12d에 1이 더해지므로, 입력한 6개의 정수 중 (r12d – 1)번째 정수.
- ebx: 이중 반복문 속의 두 번째 루프 카운터.
- eax: 바깥 반복문에서 사용되는 경우 입력한 6개의 정수 중 r12d번째 정수,
  이중 반복문 속에서 사용되는 경우 입력한 6개의 정수 중 ebx번째 정수.

순서대로 분석해보자. 처음에, 입력된 정수의 값을 eax 레지스터에 저장한 이후, 1을 뺀 값이 5보다 큰 경우 폭발한다. 이때 조건을 검사하는 인스트럭션으로 jbe가 사용되는데, 해당 인스트력션은 부호 없는(unsigned) 분기이기 때문에, 음수를 매우 큰 양수로 취급한다. 따라서 입력이 6보다 큰 경우와 입력이 0인 경우 폭발하게 된다. 따라서 입력된 6개의 정수는 1, 2, 3, 4, 5, 6 중 하나여야 한다.

다음으로, 반복문 내에 존재하는 이중 반복문을 분석해보자. 안쪽 반복문은 r12d가 1, 2, 3, 4, 5인 경우 동작하며, ebx는 r12d부터 5까지 반복한다. 해당 반복문 내에서는, rbp 레지스터, 즉 r12d – 1번째 정수와 ebx번째 정수를 같은지 검사하며, 둘이 같다면 폭발한다. 한글로 설명하면 복잡하므로, 다음과 같이 C++ 코드로 나타낼 수 있다.

```
for(int r12d = 0; r12d != 6; r12d = r12d + 1)
       for(int ebx = r12d; ebx <= 5; ebx = ebx + 1)
              if(r13[r12d – 1] == r13[ebx]) explode_bomb();
```

우리는 첫 번째 반복문이 입력 양식을 검사한다는 사실을 알아냈다. 입력되는 6개의 정수는 1 이상 6 이하여야 하며, 중복이 있으면 폭발한다. 즉 1부터 6까지의 수들의 순열(permutation)이다.

3) 두 번째 반복문

두 번째 반복문은 +102부터 +153까지이며, 중간인 +134 부터 시작한다. 두 번째 반복문에서 등장하는 레지스터는 다음을 뜻한다.

- rsi: 루프 카운터로, 0부터 4씩 증가하여 24가 되면 종료된다. 즉, 0, 4, 8, 12, 16, 20.
- ecx: rsi/4번째 입력값. 즉, rsi가 0, 4, 8, 12, 16, 20일 때 각각 0, 1, 2, 3, 4, 5번째 입력값을 가리킴.

n번째 반복에서는 다음을 수행한다. ecx에 입력 중 n번째 수를 저장한다. 이후,

- ecx 레지스터에 1이 저장된 경우: +115 위치에서 edx에 0x6042f0 주소를 저장한다.
- ecx 레지스터의 값이 1보다 큰 경우: +143, +148, +102 ~ +113에서, 각각 eax 레지스터에 1을 저장하고 edx에 0x6042f0 값을 저장한 후, ecx 레지스터와 eax 레지스터가 같아질 때까지 eax 레지스

터에는 1을, edx 레지스터에는 edx + 8 위치에 존재하는 값을 저장한다.
이후 rdx 레지스터에 저장된 값을 rsp + rsi * 2 위치에 저장하고, 루프 카운터를 증가시킨다.
단순히 위 반복문을 본다면 rdx 레지스터에 무엇이 들어가는지 이해할 수 없다. 이를 이해하기 위해서, 두 번째 반복문에서 등장하는 0x6042f0 주소에 들어있는 값을 자세히 살펴보아야 한다. 이를 16진수로 쭉 출력해보면 96바이트에 걸쳐 node라는 변수가 들어있는 것을 볼 수 있다.

```
(gdb) x/24xw 0x6042f0
0x6042f0 <node1>:    0x000001a6    0x00000001    0x00604300    0x00000000
0x604300 <node2>:    0x000000ed    0x00000002    0x00604310    0x00000000
0x604310 <node3>:    0x00000332    0x00000003    0x00604320    0x00000000
0x604320 <node4>:    0x000002d7    0x00000004    0x00604330    0x00000000
0x604330 <node5>:    0x00000061    0x00000005    0x00604340    0x00000000
0x604340 <node6>:    0x0000021a    0x00000006    0x00000000    0x00000000
```

해당 노드를 관찰한다면 다음과 같은 사실을 알 수 있다.

- node는 4바이트 형의 두 정수와, 8바이트의 주소를 갖는다. 즉 edx 레지스터에 8을 더한 주소는 다음 노드의 주소를 가리킨다.

즉, ecx가 1보다 큰 경우, edx + 8 주소에 저장된 값을 다시 edx 레지스터에 저장하는 과정은 다음 노드로 넘어가기 위한 연산임을 알 수 있다. 따라서 ecx번째 노드의 첫 번째 정수가 저장될 것이다.

요약하자면, 두 번째 반복문은 입력된 순서에 따라 메모리에 저장되어 있던 노드를 가져와 순서를 입력된 정수에 따라 섞은 후 노드에 있는 값을 스택에 저장한다. 예를 들어, 1, 4, 5, 2, 3, 6을 입력으로 받았다면, rsp에서 낮은 주소 방향으로 차례로 node1, node4, node5, node2, node3, node6에 저장된 값을 저장하게 될 것이다. 이를 통해 phase_6 함수 맨 처음에서 6개의 정수를 저장할 때, rsp 레지스터로부터 0x30바이트, 즉 48바이트 떨어진 위치에 저장한 이유가 여기서 노드를 재배열한 값을 저장하기 위함이라는 것을 알 수 있다.

4) 세 번째 반복문

세 번째 반복문에서는, 스택에 임시로 저장했던 노드의 순서를 0x6042f0 메모리에 저장되어 있었던 원래 노드에 업데이트한다. 이때 필요한 주소는 rdx 레지스터에 변하지 않고 저장되어 있다. 예를 들어 입력이 1, 4, 5, 2, 3, 6인 경우, 노드 1의 주소는 노드 4를 가리키고, 노드 4는 노드 5의 주소를 가리키게 된다. gdb 디버거를 이용해, 실행 중일 때 동일한 메모리 주소를 다시 출력해보면 다음과 같이 바뀌어 있음을 확인할 수 있다.

```
(gdb) x/24xw 0x6042f0
0x6042f0 <node1>:    0x000001a6    0x00000001    0x00604320    0x00000000
0x604300 <node2>:    0x000000ed    0x00000002    0x00604310    0x00000000
0x604310 <node3>:    0x00000332    0x00000003    0x00604340    0x00000000
0x604320 <node4>:    0x000002d7    0x00000004    0x00604330    0x00000000
0x604330 <node5>:    0x00000061    0x00000005    0x00604300    0x00000000
0x604340 <node6>:    0x0000021a    0x00000006    0x00000000    0x00000000
```

5) 네 번째 반복문

네 번째 반복문에서는, 재배열된 연결된 리스트를 순회하면서, +212 위치에서 현재값을 갖는 eax 레지스터와 이전값을 저장하는 rbx 레지스터를 비교해 현재 값이 이전 값보다 크거나 같아야 한다. 이때, 루프 카운터인 ebp가 5부터 시작해서 역순으로 내려가므로, 내림차순으로 정렬되어 있어야 폭탄이 터지지 않는다. 노드에 처음 저장되어 있던 값을 찾아보면 차례대로 422, 237, 818, 727, 97, 568이다. 따라서 이번 단계의 정답은 초기 노드의 값을 역순으로 정렬하는 3 4 6 1 2 5가 되어야 한다.

Answer: 3 4 6 1 2 5

9. Result

중단점을 잘 활용하여 explode_bomb 함수가 작동하지 않게 함으로써 폭탄을 터뜨리지 않고 해체하였다. 따라서 최대 점수인 70점을 획득하였다.

10	bomb38	Wed Oct 2 14:53	6	0	70	valid

10. Reference

Computer Systems: A Programmer's Perspective, 3/E (CS:APP3e), Randal E. Bryant and David R. O'Hallaron, Carnegie Mellon University.