시스템소프트웨어 HW2 REPORT(Bomb3)

정보컴퓨터공학부 2021555544 김태익

이번 HW2로 나온 bomblab은 실행 파일을 disassemble하고 디버깅 툴을 활용해 작동 원리를 파악하는 과제였다. 과제를 해나가는 과정에서 우리가 흔하게 사용하는 반복문, 분기문 등이 실제 기계어로는 어떻게 구현되는지 파악하게 되었고, gdb를 사용해 디버깅하는 과정에 대해 익히게 되었다.

먼저, 코드의 전체적인 구조가 어떻게 되어있는지 파악하기 위해 objdump 명령어를 사용해 disassemble된 코드를 텍스트 파일로 저장했다.

objdump -d 명령어를 통해 실행 가능한 부분을 disassemble하여 코드의 흐름을 파악하고,
objdump -s 명령어를 통해 각 섹션의 실제 바이트 값을 disassemble하여 저장된 텍스트 등의 content를 파악했다. 마지막으로,
objdump -t 명령어를 통해 심볼 테이블까지 준비를 끝냈다.

먼저 어셈블리어 코드를 분석 후 최대한 동작 원리를 파악하고자 했고, 구조가 복잡해서 눈으로만 파악이 어려운 부분은 gdb를 통해 디버깅하면서 각 레지스터에 들어있는 값들을 비교해 가며 동작 원리를 파악했다. (gdb) layout reg 명령어를 통해 어셈블리어와 레지스터 값을 한 번에 파악할 수 있는 레이아웃으로 변경하니 분석이 굉장히 편해졌다.

phase 1

인자로 받은 문자열이 특정 문자열과 같은지 검사하고, 같지 않다면 폭탄이 터진다.

문자열 비교는 <strings_not_equal> 함수를 호출해서 실행한다.

비교하는 문자열의 주소가 2ab0임을 확인 후,
bomb_content에 들어가 0x2ab0의 내용을 확인했다.

```
2ab0 54686572 65206172 65207275 6d6f7273   There are rumors
2ac0 206f6e20 74686520 696e7465 726e6574    on the internet
2ad0 732e0000 00000000 576f7721 20596f75   s.......Wow! You
2ae0 27766520 64656675 73656420 74686520   've defused the
2af0 73656372 65742073 74616765 21000000   secret stage!...
```

0x00으로 분리된 앞 문장 “There are rumors on the internets.” 이 답인 것을 확인할 수 있었다.
2e로 나타내진 문자 그대로의 ‘.’ 그리고 00인 null도 ‘.’으로 표시되어 헷갈렸다.

덤으로 secret stage가 있다는 것을 알게 되었다.

```
12c4:       85 c0                   test   %eax,%eax
12c6:       75 05                   jne    12cd <phase_1+0x19>
```

이후 이런 코드를 통해 결과를 판정하는데,
test %eax %eax면 무조건 같은 게 아닌가?라는 의문이 들었다.
이는 %eax = 0 인지 확인하는 코드와 똑같은 것이었다. <strings_not_equal>의 return 값이 0, 즉 비교

한 문자열이 똑같으면 점프 후 함수를 종료하고, 그렇지 않으면 폭탄이 터지는 구조였다.

phase 2

총 40바이트를 확보한 후 read_six_number을 호출한다.

read_six_number:
6개 정수를 입력으로 받고, 받은 정수를 스택에 차례대로 저장하는 함수이다.
반환값인 받은 입력 개수가 6보다 작으면 폭탄이 터진다.

그 후 <phase_2>로 return 한 후, 반복문을 실행한다.

```
12f2:    83 3c 24 00          cmpl   $0x0,(%rsp)
12f6:    78 0a                js     1302 <phase_2+0x2e>
...
1302:    e8 67 07 00 00       callq  1a6e <explode_bomb>
```

배열에 저장된 첫 숫자가 0보다 작으면 폭탄은 터진다. 즉, 음수가 아니어야 한다.

```
12f8:    bb 01 00 00 00       mov    $0x1,%ebx
12fd:    48 89 e5             mov    %rsp,%rbp
1300:    eb 11                jmp    1313 <phase_2+0x3f>
...
1313:    89 d8                mov    %ebx,%eax
1315:    03 44 9d fc          add    -0x4(%rbp,%rbx,4),%eax
1319:    39 44 9d 00          cmp    %eax,0x0(%rbp,%rbx,4
```

그 후, base를 1로 지정 한 후 (현재 숫자) + base == (다음 숫자) 인지 확인한다.
그렇지 않으면 폭탄은 터진다.

```
1309:    48 83 c3 01          add    $0x1,%rbx
130d:    48 83 fb 06          cmp    $0x6,%rbx
```

만약 조건을 만족한다면, base에 1을 더한 후 다음 숫자로 넘어가서 검사를 계속 반복한다.
base가 6이 되어 6개의 숫자에 대한 검사가 끝났다면 종료된다.

결국 숫자 사이의 차이가 1에서부터 시작해 1씩 늘어가는 수열이 정답이다.
ex) 1 2 4 7 11 16

phase 3

```
135e:    48 8d 35 48 1a 00 00 lea 0x1a48(%rip),%rsi      # 2dad <array.3417+0x28d>
1365:    e8 26 fc ff ff       callq  f90 <__isoc99_sscanf@plt>
```

sscanf로 받은 값을 0x2dad에 있는 배열에 저장한다.
0x2dad는 정수가 2개 들어갈 공간이 있어, 입력이 정수 2개라는 것을 유추할 수 있었다.
또한 이후 함수의 return 값, 즉 받은 입력의 수가 1개 이하면 폭탄이 터지는 것으로 확인할 수도 있다.

```
136f:       83 3c 24 07             cmpl   $0x7,(%rsp)
1373:       0f 87 99 00 00 00       ja     1412 <phase_3+0xd0>
...
1412:       e8 57 06 00 00          callq  1a6e <explode_bomb>
```

두 입력 중 첫번째는 0에서 7사이의 값이어야 한다.

```
1379:       8b 04 24                mov    (%rsp),%eax
137c:       48 8d 15 7d 17 00 00    lea    0x177d(%rip),%rdx        # 2b00 <t+0x1a0>
1383:       48 63 04 82             movslq (%rdx,%rax,4),%rax
1387:       48 01 d0                add    %rdx,%rax
138a:       ff e0                   jmpq   *%rax
```

이후 0x2b00에 있는 array에서 array[첫 정수] 이런 형식으로 접근하고 그 값이 나타내는 address로 이동한다.

```
2b00 93e8ffff 9ae8ffff e8e8ffff efe8ffff  ................
2b10 f6e8ffff fde8ffff 04e9ffff 0be9ffff  ................
```

근데 0x2b00에 있는 array에는 이런 값이 저장되어 있었다. disassemble한 텍스트 파일에 나타나있는 주소의 형식과 달랐다.

알고 보니 objdump에서 나오는 주소는 실행 전 주소이기 때문에 상대주소(오프셋),
0x2b00에 저장되어 있는 값은 실행 이후에 알 수 있는 절대주소이었다.
따라서 objdump 파일만으로 파악 어려워 0을 직접 넣고 %rax의 값을 직접 관찰하며 알아보기로 했다.

관찰 결과, 첫 숫자에 따라 점프하는 반복문의 위치가 달라, 각 입력마다 다른 연산을 진행하는 것을 알게 되었다. 그 결과 0을 넣었을때 연산 후의 값이 -664가 나오는 것을 찾았고, 이것이 입력 두번째 정수와 같으면 폭탄은 해제된다. 따라서 답은 0 -664이다.

입력 첫 정수 0 ~ 7에 따른 다양한 조합의 답이 존재할 것이다.

phase 4

정수 2개를 입력 받은 후 func4를 실행한다.
이때 첫번째 정수의 값은 14 이하여야 폭탄이 터지지 않는다.

func4를 실행한 후 return 값과 %rsp + 4에 저장된 값이 31이 아니면 폭탄이 터진다.
%rsp + 4에 저장된 값은 입력 받은 두 번째 숫자이므로 입력의 두 번째 정수는 31인 것을 파악했다.

인자인 %rdi = z, %rsi = x, %rdx = y, 함수 내부에서 사용하는 %rbx = temp라 생각하고
func4를 분석해보면,

```
temp = y - x;
if (temp % 2 == 1) temp++;
temp = temp/2 + x
```

```
if (temp > z)
  y = temp - 1
  return func4() + temp

if (temp < z)
  x = temp + 1
  return func4() + temp

if (temp == z)
  return temp
```

이러한 형식으로 이루어진 것을 알 수 있다.
생소하고 어떤 동작을 하는 코드인지 감이 안 잡혔으나, 인터넷으로 검색도 해보고 gdb로 디버깅 해본 결과 0~14 사이에서 z를 이진 탐색하는 과정에서 사용하는 중간값의 합계를 출력하는 함수임을 발견했다.
따라서 입력의 첫 정수의 유효한 범위인 0 ~ 14 중에서 이진 탐색 과정 중 중간 값들의 합계가 31이 되는 것이 13이라는 것을 알아냈고, 결국 답은 13 31 인 것을 알아냈다.

phase 5
먼저 문자열을 입력받는다.
문자열의 길이가 6이 아니면 폭탄이 터진다.

입력한 문자열에서 한 문자씩 불러오고, 0xf와 and 연산을 해서 하위 4비트만 가져온다
이후 연산 결과를 인덱스 처럼 사용해 array 2b20에서 index번째 값을 추출해오고 %rcx에 더한다,
helloo를 예로 들자면,

%rdx = 0x68(h) -> 0x08
%rcx = %rsi + 4*%rdx -> 시작에서 32바이트 떨어져있는 0x4를 더함 -> 0x04

%rdx = 0x65(e) -> 0x05
%rcx = %rsi + 4*%rdx -> 시작에서 20바이트 떨어져있는 0x10를 더함 -> 0x14
....
이 과정을 모든 문자에 반복한 후, %rcx의 값이 0x37이 되는 문자열을 찾는다.

```
2b20 02000000 0a000000 06000000 01000000  ................
2b30 0c000000 10000000 09000000 03000000  ................
2b40 04000000 07000000 0e000000 05000000  ................
2b50 0b000000 08000000 0f000000 0d000000  ................
```

array 2b20에 저장된 값 중 6개를 골라 0x37이 나오는 값을 찾으면 된다.
0x05 + 0x06 + 0x07 + 0x08 + 0x0e + 0x0f = 0x37임을 발견,
이제 0x0f와 AND 연산을 해서 저 값들의 인덱스가 나오는 문자를 찾으면 되는데,
각각 11, 2, 9, 13, 10, 14 가 인덱스임을 확인했다.
따라서 하위 4비트만 가져왔을때 0x0b, 0x02, 0x09, 0x0d, 0x0a, 0x0e가 나오는 문자를 사용하면 된다.
알파벳 소문자 'a' ~ 'z'의 아스키 코드가 0x61 ~ 0x7a이므로 알파벳 소문자를 이용하기로 했다.
0x6b(k), 0x62(b), 0x69(i), 0x6d(m), 0x6a(j), 0x6e(n)으로 이루어진 문자열을 입력하면 폭탄은 해제된다.

phase 6

우선 phase 2와 마찬가지로 6개의 정수를 입력 받고 시작한다.
이중 반복문을 통해 6개의 정수 중에 중복되는 값이 있는지 확인하고, 있으면 폭탄은 터진다.
또한 각 반복마다 검사하는 값 - 1이 5보다 크면 폭탄이 터지므로, 6보다 큰 숫자가 존재해선 안된다.
따라서 입력은 1, 2, 3, 4, 5, 6 으로 이루어진 숫자 조합이다.

이중 반복문을 탈출하고 나서 입력한 숫자대로 node를 스택에 넣는 작동을 볼 수 있었다.
예를 들어, 입력이 2, 3, 1, 4, 6, 5이면, node 2, node 3, node 1, node 4, node 6, node 5
이런 순서로 저장한다.

이후 node 뒷부분에 있는 주소를 다음 node에 연결하며, linked list를 생성한다.

생성된 linked list를 바탕으로, 첫 노드에 저장된 값이 다음 노드에 저장된 값보다 크면 폭탄이 터진다.
즉, 최종 생성된 리스트는 값이 오름차순으로 정렬되어 있어야한다.

```
15a4:       48 8d 15 85 2c 20 00    lea     0x202c85(%rip),%rdx        # 204230 <node1>
```

여기서 node1이 저장되어있는 위치가 204230임을 확인할 수 있었다.

```
204110 aa010000 06000000 00000000 00000000  ..............  // node 6 1aa
...
204230 50030000 01000000 40422000 00000000  P.......@B .....  // node 1 350
204240 c2020000 02000000 50422000 00000000  ........PB .....  // node 2 2c2
204250 4e000000 03000000 60422000 00000000  N.......`B .....  // node 3  04e
204260 38010000 04000000 70422000 00000000  8.......pB .....  // node 4 138
204270 79010000 05000000 10412000 00000000  y........A .....  // node 5 179
```

이렇게 각 node에 저장된 값을 확인할 수 있었고, 값이 오름차순 정렬이 되려면
node 3 -> node 4 -> node 5 -> node 6 -> node 2 -> node 1 순서로 되어있어야 한다.
따라서 답은 3 4 5 6 2 1 이다.

Secret phase

phase_defused를 살펴보면, 정수 2개를 입력받는 부분에 %d %d %s으로 되어있어 정수 2개 뒤에 추가적으로 입력하는 문자열을 검사하는 부분이 있다. 무슨 문자열인지 확인해보니 DrEvil이었고, 이에 phase 4의 답 뒤에 DrEvil을 작성하고 phase 6까지 통과하니 secret phase로 진입했다.

secret phase에선 문자를 입력받고, 그걸 long으로 바꾸는 strtol을 사용한다.
그 후 %rdi에 n1의 주소, %rsi에는 입력한 숫자를 넣고 fun7을 실행해, 결과값이 1이면 폭탄은 해체된다.

n1을 살펴보니 트리 구조로 값들이 저장되어있음을 확인했고, fun7은 이진 탐색을 하는 함수임을 확인했다. 왼쪽으로 내려가면 fun7*2, 오른쪽으로 내려가면 (fun7*2)+1을 return하므로 최종 return 값이 1이 되려면 시작에서 한 번 오른쪽으로 내려간 0x32, 즉 50을 입력하면 폭탄이 해제된다.