2020년 2학기

Assignment #2

과목 : 컴퓨터SW시스템개론

담당 교수 : 김종

학과 : 컴퓨터공학과

학번 : 20190439

이름 : 오승훈

povis id : sho0927

|  |
| --- |
| < 명예서약(Honor code) >  “나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.” |

**Problem : Bomb Lab**

1. 문제의 개요

본 프로그램은 다음의 내용을 포함한다.

- bomb을 disassemble하여 폭탄이 터지지 않는 input을 찾아내는 것이다.

- 총 6개의 phase가 존재한다.

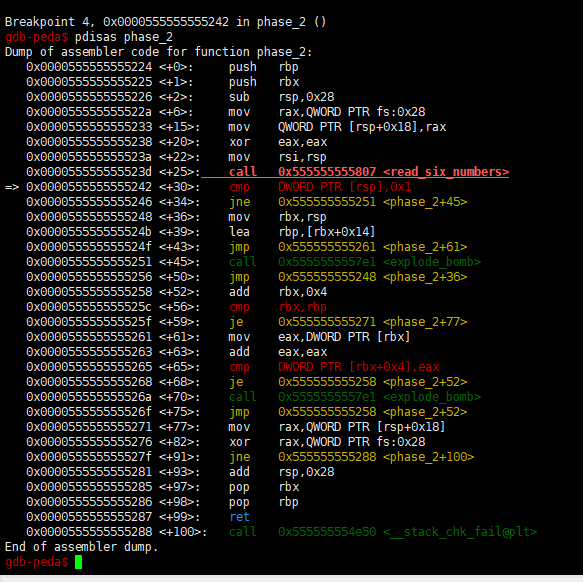
2. 프로그램 구조 및 설명

**1) Phase\_1**

<strings\_not\_equal> function을 호출하기 전 함수에 인자로 %RSI register로 넘겨주게 되는데 이 register에 들어간 값을 보면 “Houses will begat jobs, jobs will begat houses.” 라는 string이 들어가있는 것을 볼 수 있다. 그리고 strings\_not\_equal 함수는 말 그대로 string이 같으면 0을 반환하고 아니면 1을 반환한다는 것을 알 수 있다.

**2) Phase\_2**

<read\_six\_numbers> function에서 scanf가 호출되기 전 %esi에 대입되는 0x402593 주소의 memory 값이 “%d %d %d %d %d %d”인 것으로 보아 6개의 정수를 입력받음을 알 수 있다. 이후 RSP register의 첫 값이 1과 같은지를 비교하는 것을 알 수 있고, 그렇다면 첫 정수는 1이 들어가야함을 알 수 있다. 그 후 밑에 첨부된 사진의 phase\_2의 +61과 +63을 보면 eax에 현재 값을 넣고, 그 값을 그대로 더해주는 것을 알 수 있다. 이는 1 -> 2 -> 4 -> 8 -> 16 -> 32의 값을 유도하고, 이는 결과적으로 %d에 들어가야할 값이 1 2 4 8 16 32가 들어가야함을 알 수 있다.



**3) Phase\_3**

Scanf를 호출하기 전 해당 함수에 인자로 넘어가는 esi에 대입되는 값을 보면 “%d %d”이다. 따라서 정수 두 개를 입력받음을 알 수 있다. 이 후 <+45>에서 알 수 있듯이 첫 입력 값은 7보다 작거나 같아야한다는 것을 알 수 있다. 그리고 <+68> 부터 <+124>를 통해 이 문장은 switch 구문이라는 것을 알 수 있다. <+68> 구문을 통해 각 값이 어디로 유도가 되는지를 알 수 있는데

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| <+138> | <+77> | <+84> | <+91> | <+98> | <+105> | <+112> | <+119> |

현재 jmp된 곳에 존재하는 eax register 값을 2번째 %d에 넣어주면 답으로 인정된다는 것을 알 수 있다.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0x238 | 0x2c9 | 0x323 | 0x9b | 0xde | 0xd0 | 0x9e | 0x373 |

solution에는 0, 0x238 즉 0과 568을 넣었다.

**4) Phase\_4**

scanf 함수 호출 전 esi에 대입하는 주소의 값이 %d %d이므로 정수 두 개를 입력하는 것이다. 입력된 값의 개수인 eax의 값을 2와 비교하여 eax == 2가 아니면 폭탄이 터지며, 두 번째 입력 값을 2를 빼서 2보다 작거나 같으면 func4 함수를 시작을 하고 2보다 크게 되면 폭탄이 터지게 된다.

그 후, func4로 들어가게 되면 RSI는 두 번째 대입된 값을 RDI는 7 값을 가지고 있다는 것을 알 수 있다.

Func4 내부에서는 먼저 rdi가 0인지를 체크를 하는데 0이라면 <+16>으로 jmp를 하게 된다. 그리고 rdi가 1보다 작거나 같으면 +18로 jmp를 하는 구조를 띄는 것을 알 수 있다. 그 후

r12d = rsi , rbx = rdi, rdi = rdi – 1 으로 값을 변경 한 후 func4 콜을 다시 하게된다.

그 후 rbp = rax + r12 , rdi = RBX – 2, RSI = R12d 가 된 후 func4 콜을 다시 하게 된다. 그 후 ebp register 값을 모두 rax 값으로 더하고 값을 반환하게 되는데 RBX 와 RDI를 중심으로 보면 된다. RBX와 RDI는 7 / 7로 시작하는데 (7,6) (6,5) (5,4) (4,3) (3,2) …. 로 다시 가지를 생기게 되고, 이 값들은 총 33번의 반복을 유도를 하게 되므로 두 번째 넣은 값을 계속 더해서 return을 하는 형식의 recursion 구문이므로 이 값은 두 번째 입력 받은 인자 \* 33 = 첫 번째 입력 받은 인자로 가는 값을 입력하면 되므로

132 4

99 3

66 2

이 셋 중 아무 답이나 쓰면 된다 132 4 를 답으로 제출 하였다. 또한 secret\_phase로 접근을 하기 위한 DrEvil을 뒤에 적어둔다. 이는 secret\_phase를 설명하는 곳에서 더 자세하게 설명하겠다.

**5) Phase\_5**

여기서는 <+35>에 breakpoint를 사용하게 되면 두 개의 인자를 입력 받는다는 것을 알 수 있다. 또한 RSI register에 rip+0x15c3이라는 주소를 넣어둔다는 것을 알 수 있다. 여기서 RSI register가 갖고 있는 것은 array의 start 주소를 가지고 있는데 이는 이러한 형태의 array를 가진다.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| A | 2 | E | 7 | 8 | C | F | B |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 0 | 4 | 1 | D | 3 | 9 | 6 |  |

이 때 <+81>에 의해 이는 arr의 시작 주소에 4 \* rax 값을 더해서 주소값을 접근한다는 것을 알 수 있는데 그렇다면 해당 index에 저장되어 있는 값으로 다음번에는 접근을 한다는 소리가 된다. 그리고 만약 RAX에 0xf가 들어오게 되면 이 반복이 15번이 반복이 되었는지를 판단을 하게 되는데 이 index 값들을 따라 가보게 되면 5 C 3 7 B D 9 4 8 0 A 1 2 E 6 F 순으로 가야 15번의 반복 이후 0xF 값이 들어올 수 있게 됨을 알 수 있다. 그러므로 첫 값은 5를 대입 해야하고, 그 index 값에 저장되어 있는 값들을 모두 더해 RCX에 넣게 되므로 위의 C 3 7 B D 9 4 8 0 A 1 2 E 6 F 를 모두 더하면 된다. 그러면 115가 나오는 것을 알 수 있고, RSP + 0x4 와 RCX register를 비교하게 되므로 115를 두 번째 인자로 받으면 된다는 것을 알 수 있다.

Solution : 5 115

**6) Phase\_6**

<read\_six\_numbers>에서 숫자 6개를 읽어온 후, 숫자의 개수가 6개가 맞는지 확인한다. 이후 <+43>부터 <+95>까지의 이중 반복문을 돌며 겹치는 input이 있는지, 1~6의 범위를 벗어나는 input이 있는지 확인한다.

Node 6개로 이루어진 Linked List의 순서를 각 Node가 가지고 있는 정보를 내림차순으로 정렬을 하는 것을 목표로 하는 것을 <+232> ~ <+247>을 통해서 알 수 있다. 0x10b 0x37a 0x3aa 0x161 0x14f 0x206의 값을 1번 node 부터 6번 노드까지 가지고 있으므로 이 값들을 정렬하는 순서를 적어주라는 뜻이므로 3 2 6 4 5 1의 NODE 순서를 가져야 이 NODE가 내림차순으로 정렬이 되었다는 것을 확인할 수 있기 때문에 3 2 6 4 5 1을 solution으로 넣어야한다.

**7) Secret\_phase**

Secret\_phase를 알기 위해서는 main 함수에 존재하는 phase\_defused 함수를 뜯어 봐야하는데 이 함수의 경우 <+115>에서 strings\_not\_equal에 breakpoint를 찍으면 phase\_4에서 DrEvil이라는 단어를 더 적으면 secret\_phase로 자연스럽게 들어와진다는 것을 알 수 있다.

Secret\_phase의 경우 read\_line을 통해서 받아온 값을 string to long으로 전환을 시켜주는 함수를 이용하고, 0x3e9보다 작은 값을 넣어줘야한다는 것을 <+30>을 통해서 알 수 있다. <+39> 코드를 보면 rdi에 rip + 0x202b25라는 주소로 접근을 한다는 것을 알 수 있는데 여기에는 특정한 배열 값들이 저장이 되어있다. 그 후 fun7을 return 한 후 return 값이 0이 나와야한다는 것을 <+51> 코드를 통해서 알 수 있는데 그렇다면 fun7 함수 내부에서 return 되는 값들을 잘 봐야한다.

Fun7 함수에서는 return 값이 바뀌는 부분은 <+15> <+38> <+57> 부분 밖에 없다는 것을 알 수 있는데 만약 <+57>을 거쳐서 나오게 되면 RAX = RAX \* 2 or RAX = RAX \* 2 + 1을 통해서는 절대로 return 값이 0이 될 수 없다는 것을 알 수 있다. 그러므로 우리의 함수는 <+57>을 return값으로 사용하지 않고, <+15>와 <+33>을 이용하여 나오는 함수면 된다는 뜻이다. 그렇다면 <+33> call만을 이용하여 return 값을 이용해야 하는데 rdi 값에는 0x24가 처음 들어가 있게 되고, 그 후 <+33> call을 이용하기 위해 36보다 작은 수를 넣게 되면 0x8이 들어가 있는 것을 확인할 수 있다. 그렇다면 0x8가 동일하면 rax return 값을 0으로 반환하기 때문에 rax = 2\*rax를 해도 0이 ret 한다는 것을 알 수 있고, 그렇기 때문에 8을 넣으면 phase가 풀리게 된다.

이 함수는 결론적으로 BST를 assembly 코드로 변경을 시켜둔 것이다. 그렇기 때문에 우리가 반환해야하는 값은 test rax, rax 이기에 0이 되면 되는데 이는 결론적으로 bst 트리의 가장 왼쪽 tree를 구하라는 것과 동일한 결과를 유도하게 된다. Assembly 코드의 func7을 조금 해석을 해보자면 <+33>의 값은 왼쪽 node로 탐색을 들어가는 것이고, <+46>은 오른쪽 node로 탐색을 들어가는 것이다. 그렇기 때문에 +38과 +51 코드에 왼쪽을 다녀오면 2배를 해주는 것이고, 오른쪽을 다녀오면 2배 +1을 해주는 것이다. 그렇기 때문에 결론적으로 반환하는 값은 현재 NODE에 저장된 값이 오름차순 배열에 넣었다고 생각했을 때 몇 번째 index에 존재하는지를 반환하라는 것과 동일하다. 이는 BST 중에서도 왼쪽을 우선적으로 탐색하고, 그 후 오른쪽을 탐색하는 preorder으로 tree traversal을 돌게되는 binary search tree를 구현을 해둔 것이다.

Solution 0x8 -> 8

3. 토론

처음 이 bomb\_lab을 접했을 때는 대체 왜 이런 assembly 코드들이 존재하는지도 이해를 하지 못했고, 어떻게 우리의 코드들이 동작하는지 자세히 알 수는 없었다고 생각한다. 이 부분에 대한 수업을 듣기 전까지 나는 왜 이 함수는 이런 값들을 register에 넣게 되고, 어떤 register에 어떤 parameter들이 들어가는지 이해를 하지도 못했었는데 이제는 어떤 register에 어떤 값들이 들어가고, callee / caller save register가 어떤 것들이 있는지 알고 있는 내가 된 것 같다. 하지만 아직까지도 recursion을 이용하는 함수의 경우 왜 이렇게 동작을 하는지를 이해하는데는 상당한 시간이 걸리는 것 같고, phase\_4의 경우 왜 그렇게 동작을 하냐고 더 자세히 증명을 해보라고 한다면 명확히 설명을 하지 못한다는 것은 아쉬움으로 남는 것 같다. 하지만 퍼즐 조각을 푸는 듯한 느낌의 ASSN이라 결론적으로 즐기면서 마무리를 할 수 있는 과제였던 것 같다.