**CSED211 Lab3. Bomblab report**

20210643 김현준

**<Lab3>**

Lab3에서는 gdb 사용법과 Assembly에 대해 배웠다. Gdb는 gnu project debugger로 gdb exefile을 입력하여 디버거를 실행한다. 자주 쓰이는 명령으로는 disas, l, b, I, ni, d, r, display 등이 있다.

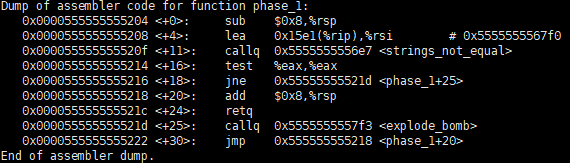
Assembly어는 Machine-friendly 언어로, 컴파일의 결과이다. Opcode와 operand 부분이 있고, opcode에는 unary operator와 binary operator가 들어온다. Operand에는 Immediate, Register, 또는 Memory 내용이 들어오게 된다. 어셈블리에서 General Reg들은 rax, rbx, rcx 등이 있고, 각각의 사용 용도가 다르다. Rsi와 rdi는 첫째, 둘째 arguments를 담당하고 rbp와 rsp는 stack frame 관리에 쓰인다.

기초적인 어셈블리 command로는 mov, add, sub등이 있고, cmp는 두 값을 비교하여 Flag를 바꾼다. Flag에는 CF, ZF, SF, OF가 있다. 또한, if나 for 같은 C 문법을 사용할 때 어셈블리는 jump를 이용하여, je, jne, jg, jl 등이 있다. 추가로, 자주 쓰이는 lea는 Load Effective Address로 mov와 달리 주소를 저장한다는 차이점이 있다.

**<phase\_1>**

우선, main 함수의 형태가 phase\_1~6 함수를 호출하고 각 함수마다 각 phase의 암호 코드를 갖고 있다는 것을 disas main에서 알 수 있었다.

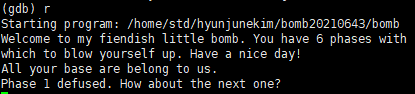
다음으로, phase\_1을 disas해보니, 아래와 같이 string\_not\_equal 함수를 호출하여 test로 0인지 확인 후, explode\_bomb이 호출되는 것을 확인할 수 있었다. 따라서 string\_not\_equal 함수의 이름처럼 무언가 string을 비교하여 맞으면 explode\_bomb 하지 않는 형태임을 짐작할 수 있었다.



이 때, +4 부분을 살펴보면, lea 명령이 rsi를 0x15el(rip)으로 주소를 넣고 있고 우측에 해당 주소가 # 0x5555555567f0으로 적혀 있는 것을 확인할 수 있었다. 따라서 x/s를 통해 해당 주소에 있는 값을 string으로 변환해본 결과 아래와 같이 “All your base are belong to us.”가 나오는 것을 확인할 수 있었다.



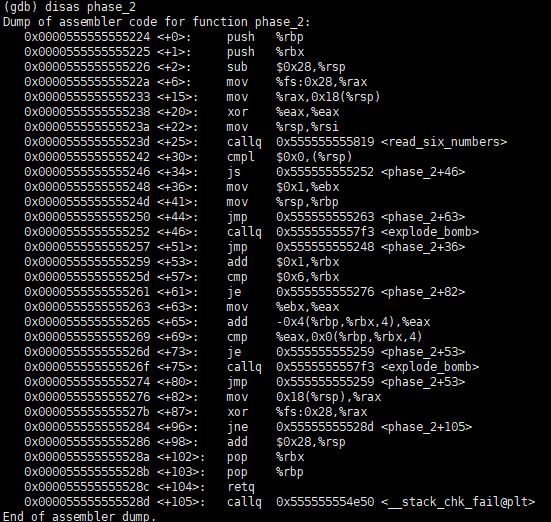
따라서 이 string과 입력 string을 strings\_not\_equal 함수로 비교하여 같으면 phase\_1이 defuse되는 것으로 생각하였고 해당 string을 입력한 결과 phase1이 defuse됨을 확인할 수 있었다.



답: All your base are belong to us.

**<phase\_2>**

먼저, phase\_2 함수를 disas해 보았다. 그 결과는 아래와 같이 나왔고, callq를 보니 explode\_bomb 외에는 read\_six\_numbers 함수를 호출한다는 것을 확인할 수 있었다. 함수 이름을 보고 우선 6개의 숫자를 비교하는 것이 phase\_2일 것이라고 추측하였다.



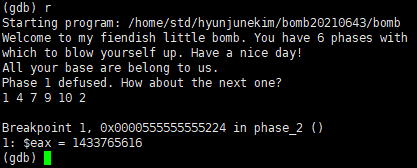
read\_six\_numbers 함수 호출 이후 explode\_bomb이 호출되는 곳을 보니, +46과 +75 부분이 있었다. 폭탄이 터지지 않으려면 이 함수가 호출되면 안 되므로, 먼저 +46이 발동하려면 어떻게 해야 하는지 살펴보니, 앞에서 +34에서 js로 점프해서 넘어오지 않으면 된다는 것을 알 수 있었다. 그 전 cmpl이 0과 rsp를 비교하므로, SF가 0일 때 js가 실행되므로 cmpl의 결과가 음수여야 하고, 그러려면 rsp가 음수여야 한다. 따라서 6개의 숫자는 음수가 아님을 알 수 있었다.

다음으로 +75부분을 보면, +73의 je가 실행되지 않았을 때 폭탄이 터진다. je 전을 보면 cmp가 있는데, 여기서 eax와 0x0(%rbp, %rbx, 4)를 비교하는 것을 볼 수 있었다. je가 실행되면 다시 +53으로 돌아가므로, 이런 루프를 돌아서 6개의 숫자를 모두 비교하는 것으로 생각되었다. 그리고 특정한 연산을 통해 얻은 결과를 %eax와 비교하고 있으므로 %eax를 추적해봐야겠다고 생각하였다. 따라서 phase\_2와 explode\_bomb에 breakpoint를 걸어놓고, display %eax를 해 놓고 run 해보았다.

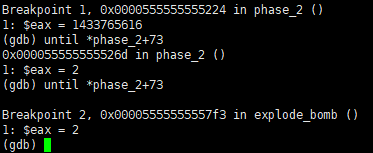




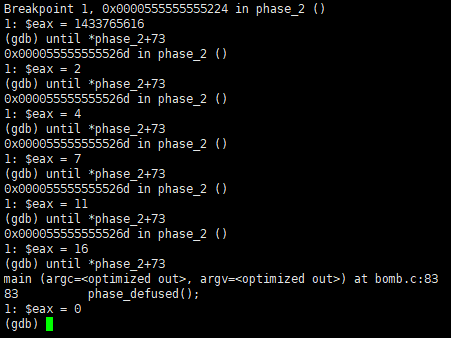
그리고 먼저 phase\_1을 defuse한 후, 아무 숫자나 6개 입력해보았다.



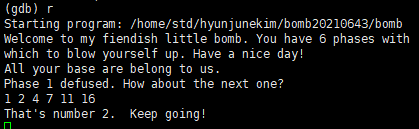
우선 모두 양수를 입력했으므로 +46의 explode\_bomb은 피할 것이므로, +73까지 실행되도록 until \*phase\_2+73을 입력해주었다.



그러자 위와 같이 나왔다. 해석해 보자면, 처음에 +73으로 갔을 때에는 je가 실행이 되었고, 두 번째 +73에서는 실행이 되지 않았고, 그때 eax는 2라는 것으로 생각하였다. 따라서 1은 맞는 답이고 그 다음 수는 2임을 확인할 수 있었고, 이와 같은 과정을 반복하면 6개의 수를 모두 알아낼 수 있겠다고 판단하였다. 따라서 이를 반복해보았다.



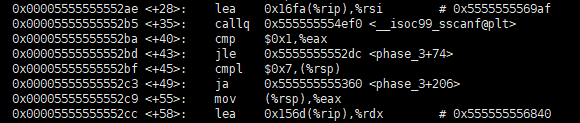
아무 숫자나 입력하여 하나씩 알아내다 보니, 6개의 숫자가 +1, +2, +3, …씩 증가하는 수열임을 알 수 있었고, 따라서 1, 2, 4, 7, 11, 16이 답이 됨을 알 수 있었다. 또한, 숫자 간의 차이가 +1, +2, +3, …임을 만족하는 모든 수열은 phase\_2를 defuse시킨다는 것을 알 수 있었다.



답: 1 2 4 7 11 16

**<phase\_3>**

우선 phase\_3 함수를 disas 해보았다. 그 결과에서 먼저 phase\_1을 푼 것처럼 lea 명령어에서의 주소를 먼저 체크해보았다.



+28과 +58의 주소를 x/s로 살펴보니 아래와 같았고, %d %d로 +35에 있는 scanf관련 함수의 입력으로 받는 형태이지 않을까 하는 추론을 할 수 있었다. +58의 주소에서는 얻을 만한 정보가 보이지 않았다.

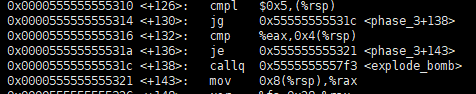


다음으로, 정수 두 개를 아무거나 1 1로 입력해 놓고 breakpoint를 걸어 보았다. 그런데, phase\_3의 disas를 보면 +40부터 cmp, jle, cmpl, ja가 있고 점프하는 곳은 모두 explode\_bomb이다.



따라서 %eax는 1보다 커야 하고, %rsp는 7보다 작거나 같아야 했고, 각각 display하고 x/d하여 확인해 보니 1, 1을 입력했을 때 %eax 는 2, %rsp는 입력한 1이 있는 것을 확인할 수 있었다. 따라서 다음으로 넘어갈 수 있었고, 다음 코드를 보니 jmpq \*%rax하는 부분이 있었다. %rax를 확인해 보니 +88 부분의 주소가 들어가 있었다.

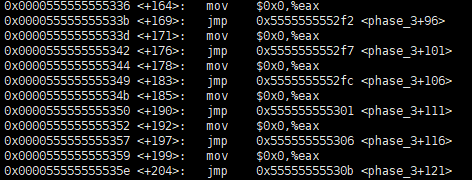
다음 코드 부분은 eax를 0으로 설정하고 다양한 연산을 수행하는 부분이었다. 각 연산은 -123, +246, -352, +352, -352, +352, -352이고, 모두 계산하니 %eax가 -229가 되는 것을 확인할 수 있었고, 다음 코드를 보니 아래와 같은 부분이 있었다.



%rsp가 5보다 작거나 같아야 하고 %eax의 값이 0x4(%rsp), 즉 두 번째 입력값으로 예상되는 값과 같아야 explode하지 않음을 추론할 수 있었다. 따라서 다시 bomb를 실행하여 1, -229를 입력해보니 phase\_3이 defuse되는 것을 확인할 수 있었다.



또한, 1 이외에 2~7의 값을 입력했을 때에는 jmpq \*%rax 부분에서 +164 이후로 넘어가 각각 연산 단계에 중간으로 점프하여 연산이 이뤄지는 것을 알 수 있었고, 따라서 1 -229 이외에도 여러 답이 존재할 수 있음을 알 수 있었다.



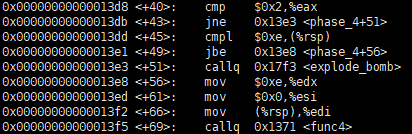
답: 1 -229

**<phase\_4>**

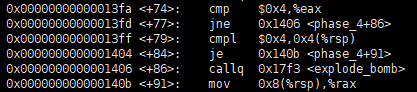
우선 phase\_4를 disas하고, phase\_3와 비슷한 형태의 앞부분을 발견하였다. 따라서 lea 부분의 오른쪽 주소를 체크한 결과 %d %d로 두 개의 정수를 입력하는 것을 확인하였다.



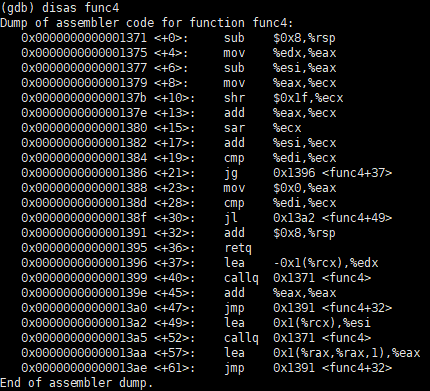
다음으로, 코드를 해석해보면, 2개가 입력되면 +43에서 점프가 안 일어나 explode를 피하고, 이후 0xe와 %rsp를 cmpl한다. 따라서 첫 입력값이 14보다 작거나 같아야 explode가 일어나지 않는 것을 알 수 있었고, 다음으로 $edi에 첫 입력값을 전달하여 func4로 넘어가는 것을 확인할 수 있다. func4의 매개변수르 %edx에는 14, %esi에는 0이 들어간다.



func4를 호출 이후를 살펴보면, func4의 리턴값이 4여야 하고, 두 번째 입력값이 4여야 한다는 것을 알 수 있다. 따라서, 두 번째 입력값은 4이고, 첫 번째 입력값은 14보다 작으면서 func4의 리턴값이 4가 되도록 하는 값을 찾으면 된다.



이제 func4를 disas해보면, +19에서 %ecx<%edi이면 점프하고, func4를 재귀함수로 callq한다. +30에서 %edi<%ecx이면 func4를 call한다. 또한, 어디에도 점프하지 못하면 +36에서 리턴하게 된다. 즉, 재귀함수를 부르는 if, else if, else문와 같은 형태인 것으로 생각할 수 있었다.



또한, 해석해 보면 ecx는 (edx-esi)/2로 구하고, ecx와 입력값을 비교하여 입력값이 더 작을 경우 edx=ecx-1로 재귀함수를 호출하고, 클 경우 esi=ecx+1로 재귀함수를 호출하고 같을 경우 0을 리턴함을 확인할 수 있었다. 이러한 형태를 가지므로, 재귀함수를 반복하다 보면 edx와 esi의 간격이 좁아져 결국 모두 ecx와 같아져서 리턴하기 시작할 것이라고 생각하였다. 또한, 최종 리턴값이 4가 되어야 하고, ecx보다 입력값이 작을 때, 재귀함수를 다녀온 이후 해당 리턴값을 다시 두 배 하여 리턴하므로, 0, 1, 2, 4로 리턴하여 최종으로 4를 리턴하면 될 것이로 생각하였다.

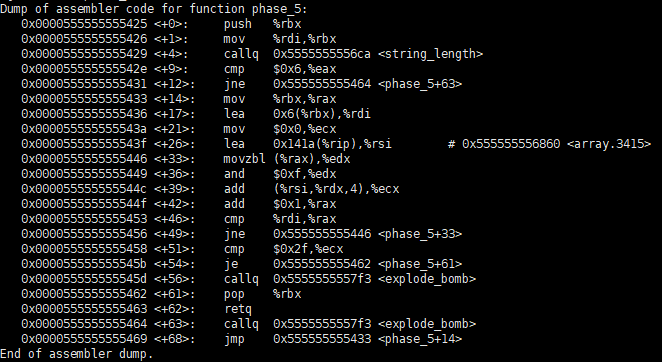
이러한 조건을 만족하는 수를 찾으니 2였다. 2를 입력하면 (edx, esi, edi, ecx) 쌍이 처음에는 (14, 0, 2, 7) 다음 재귀에서는 (6, 0, 2, 3), 다음에는 (2, 0, 2, 1), (2, 2, 2, 0)이 되어 리턴을 시작하여 0, 1, 2, 4로 최종적으로 4가 리턴되는 것을 확인할 수 있었다.



답: 2 4

**<phase\_5>**

제일 먼저 disas 해보았다.



string\_length라는 이름의 함수가 호출되니, 입력된 문자열의 길이를 구할 것이라 추측하였고, 아래에서 6과 비교하므로 입력된 문자열의 길이가 6일 것으로 추측하였다. 또한, phase\_5가 explode하지 않고 리턴하려면 ecx가 0x2f와 같아야 한다는 것을 여기서 알 수 있었다.

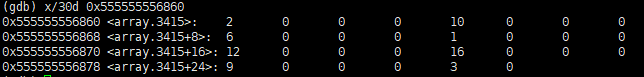
그리고, +42~+49 부분을 살펴보면 rax를 1씩 증가하고 점프를 하는데, 이 곳이 루프를 도는 부분일 수도 있다고 추측하였다. 따라서 rax를 확인해 보니, 입력한 문자열이 들어가 있었다.



따라서, 이 부분이 루프를 돌면서 문자 하나 하나를 처리하여 6번 도는 곳일 거라고 추측할 수 있었다.

처리부분을 분석해 보면, aaaaaa를 입력했을 때 edx가 97로 a의 아스키코드 번호가 나왔고, 다음으로 0xf와 and연산되어 rdx가 아래 4비트만 남아 1이 되었고, +42 부분에서 ecx가 0에서 10이 되었다. 따라서 이 과정을 6번 반복하면 ecx가 60이 되는데, ecx가 0x2f, 즉 47이 되어야 phase\_5가 풀리므로, 47가 되도록 하는 문자열 조합을 찾으면 될 것으로 생각하였다.

그런데, 루프에서 ecx를 업데이트할 때, add (%rsi, %rdx, 4), %ecx로 하는 것을 볼 수 있는데, 이것은 rsi의 배열에서 값을 꺼내오는 것으로 생각하였다. 그 이유는 +26 부분에서 0x555555556860을 x/30d로 살펴보니 정수의 배열이 나타났기 때문이다. 따라서, rdx만큼의 배열 index에서 값을 꺼내와 더하여 47이 되도록 하면 되는 것이었다.



47=10+10+10+10+6+1으로 쪼갤 수 있고, 10이 되려면 +4이므로 rdx=1, 6이 되려면 rdx=2, 1이 되려면 rdx=3임을 알 수 있다. a가 rdx=1이므로, 결과적으로 aaaabc는 답이 된다.



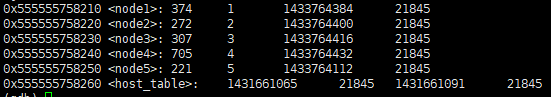
답: aaaabc

**<phase\_6>**

우선 phase\_6함수를 disas하니 phase\_2에서 나왔던 read\_six\_numbers가 있었고, 6개의 입력이 필요할 것이라고 생각하였다.



그래서, 1 2 3 4 5 6을 입력하고 breakpoint를 설정하여 ni로 한 줄씩 실행해보았다. 계속해서 실행해보다 +187에서 0x555555758210을 확인해보니, 아래와 같은 노드들을 찾았다.



또한, 계속해서 실행하다 +281 부분에서 eax와 rbx를 cmp하여 jge하는 코드가 있었고, 여기서 jump하지 못하면 explode하므로 jump해야 했다. 여기서 rbx를 확인해 보니 6번째 노드를 찾을 수 있었다.



그리고 eax를 확인해 보니 221이 들어가 있었다. 이 숫자는 node5의 값과 똑같으므로 노드의 값과 eax가 연관된 것으로 추정하였다. 그런데 eax의 주소의 값과 rbx의 주소의 값을 비교하여 rbp 쪽이 더 작아야 폭탄이 작동되지 않으므로, 노드 값에 따라 번호를 내림차순으로 정렬한 것이 답이 될 것이라고 생각하였다. 하지만 그렇게 정렬한 4 1 3 2 5 6이 답이 아니었기 때문에 앞부분을 다시 살펴보았다.



앞부분의 내용은 루프를 이용해 입력값이 6이하이면서 모두 다르게 판별하는 부분으로 해석하였고 그 중 +122~+129부분을 보면 edx에 7을 넣고 eax에 edx를 넣은 후 r12를 빼는 부분이 나온다. 즉, 이 부분을 통해 7-node number로 치환되는 것이 아닐까 생각했다. 따라서 7-a로 치환한 3 6 4 5 2 1을 입력해보았고, phase\_6이 defuse되는 것을 확인할 수 있었다.



**<secret\_phase>**

우선 phase\_defused 함수를 disas해 보았다. 그 결과에서 +72의 주소를 x/s로 찾아가니 %d %d %s가 나오는 것을 확인하였다. 따라서 이를 통해 phase\_4에서 추가로 string을 입력하면 된다는 것을 알게 되었고, 우선 아무 문자열 abc를 입력해보았다.

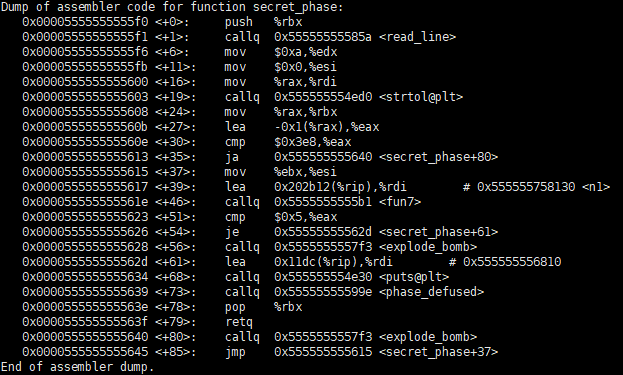


그러고 나서 strings\_not\_equal 함수 호출 직전 코드의 주소에서 DrEvil을 찾을 수 있었다. 따라서 phase\_4에서 DrEvil을 추가로 넣으면 secret\_phase로 이동하는 것으로 생각했고, 실제로 secret\_phase로 넘어가는 것을 확인할 수 있었다.

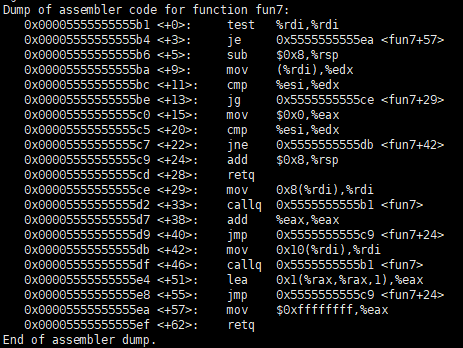




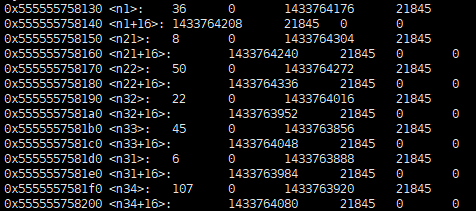
다음으로, secret\_phase함수를 disas하였다.



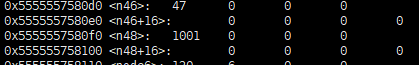
먼저, eax가 0x3e8보다 작아야 하는 것을 알 수 있다. 그리고, +54에서 fun7함수의 리턴값이 5여야 defuse된다는 것을 알 수 있었다. 따라서 다음으로 fun7함수를 disas해보았다.



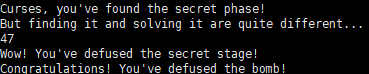
%rdi의 주소를 계속 업데이트하는 것이 코드의 내용이라고 생각되어, %rdi를 x/d로 값을 확인하였다. 그러자 36이 나왔고, ni 명령어로 코드를 쭉 실행하면서 %rdi를 확인해보았다.



그러자 위와 같이 36, 8, 50, 22, 45, 6, 107이 나왔고, 이어서 더 조사하니 47, 1001이 나왔다.



이러한 숫자들 중에 하나가 답이 될 것이라고 생각하였다. 그런데, fun7에서 +11에서 cmp하여 이동시키는 것을 확인하고, 이것이 tree 자료구조와 비슷하다고 생각했다. 또한 binary tree에서 자식 노드는 left의 경우 2배, right의 경우 2배+1이 노드 index이므로, 이 node index가 5가 되는 것을 찾아보았다. 나온 값들로 binary tree를 그려 보았을 때, 47이 5가 되는 것을 알 수 있었고, 실제로 47을 입력하니 secret\_phase가 defuse되었다.



답: 47

**최종 답:**

All your base are belong to us.

1 2 4 7 11 16

1 -229

2 4 DrEvil

aaaabc

3 6 4 5 2 1

47

