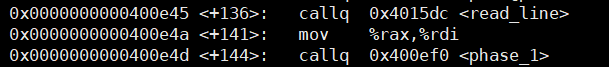
Lab3 BombLab Report

20220665 유영준

4주차 랩 시간에 GCC, GDB에 대한 간단한 설명과 어셈블리의 간단한 동작 방법에 대해 배웠다. 어셈블리의 syntax는 ‘opcode+operand’ 형식으로 구성되어 있고, opcode는 Unary operator, Binary operator가 존재한다. operand에는 실제 값, 레지스터 값, 메모리 세 종류가 존재한다. 어셈블리의 다양한 instruction에 대해 학습하였고, 예시를 통해 어셈블리의 작동 방식에 대해 알 수 있었다. 그리고 bomblab을 다운받기 위한 서버 접속 방법에 대한 설명을 듣고 해당 랩 시간을 마무리했다. 6주차 랩 시간에는 calling convention과 메모리에서 사용되는 stack의 작동 방식에 대해 학습했다. 해당 랩 시간과 관련하여 BombLab이 과제로 나왔고, phase1부터 phase6 그리고 secret phase까지의 풀이 과정은 다음과 같다. 과제는 xshell 환경에서 진행하였다.

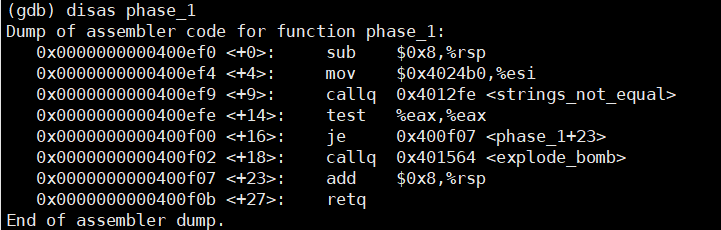
1. **Phase1**

* 먼저 main 함수를 disassemble 하여 전체 프로그램의 작동 방식을 살펴보았다.



각 phase에 들어가기 전 read\_line 함수를 통해 읽은 입력 값을 rdi 레지스터에 넣은 후 phase를 실행하는 것으로 보아 입력 값이 각 phase의 첫 번째 인자로 들어가 함수를 진행시킨다는 것을 알 수 있었다.

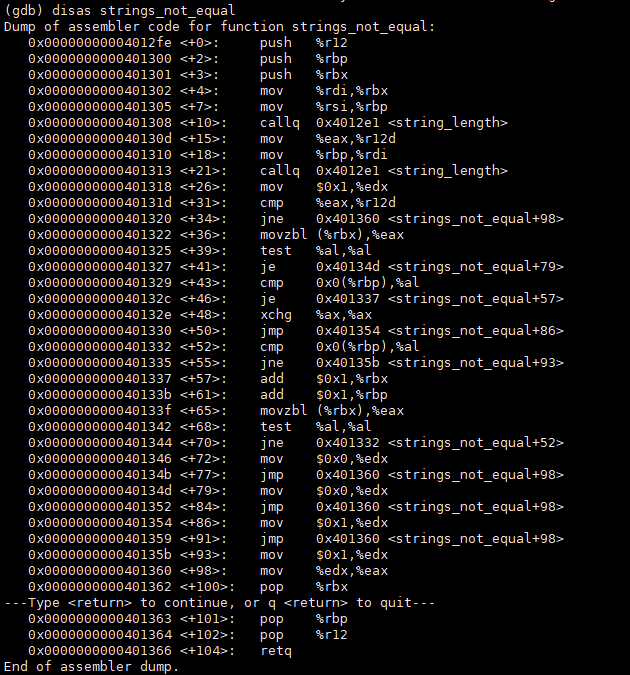
* phase\_1 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.



rsp값을 바꾸어 stack frame를 형성한 뒤 rsi 에 특정 값을 넣은 후 strings\_not\_equal 함수를 실행시킨다는 것을 알았다. 이떄 rsi에 넣은 주소에 어떤 값이 들어 있는지 x/s를 통해 확인해 보았다.



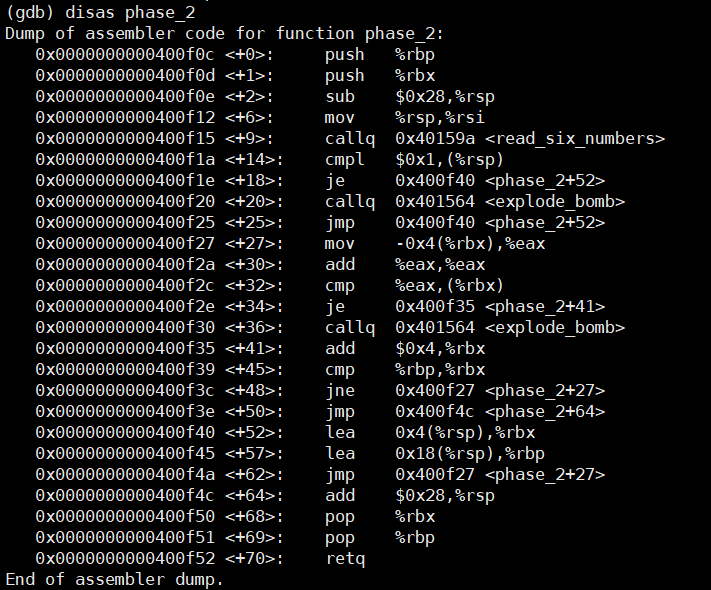
앞에서 살펴본 것처럼 rdi에는 input string이 들어가있고, rsi에 해당 string이 들어있으므로 strings\_not\_equal 함수에서 두 string이 동일한지 확인하여 같으면 0, 아니면 1을 return할 것이라고 추측하고, strings\_not\_equal 함수를 disassemble 해보았다.



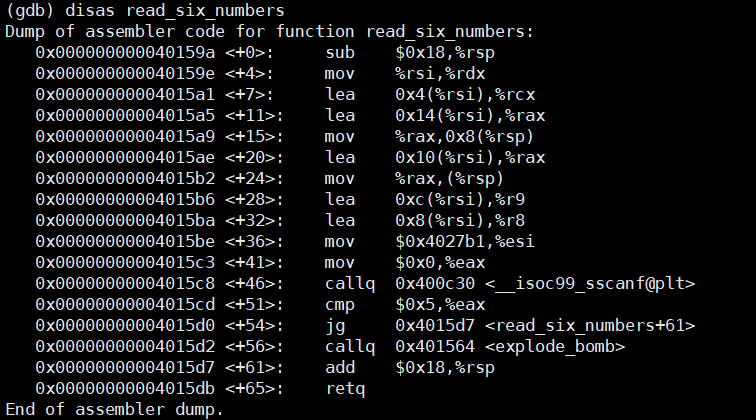
예상대로 strings\_not\_equal 함수에서 두 string이 같으면 0, 아니면 1을 rax에 넣어 return해준다는 것을 알았다. 해당 함수를 실행시킨 이후 test와 je를 통해 rax 자기 자신의 and 연산 후 그 값이 0이면 bomb을 터트리지 않고 함수를 종료한다는 것을 알았다. 이를 통해 phase\_1의 입력 string은 “He is evil and fits easily into most overhead storage bins.”임을 알았다.

1. **Phase2**

* phase\_2 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.



* rsp값을 바꾼 후 rsi에 rsp값을 넣은 뒤 read\_six\_numbers 함수를 호출하는 것을 알 수 있다. read\_six\_numbers를 disassemble 하여 함수의 작동 방식을 살펴보았다.



* <+36>에서 rsi 레지스터에 넣는 값을 x/s를 통해 알아보았다.

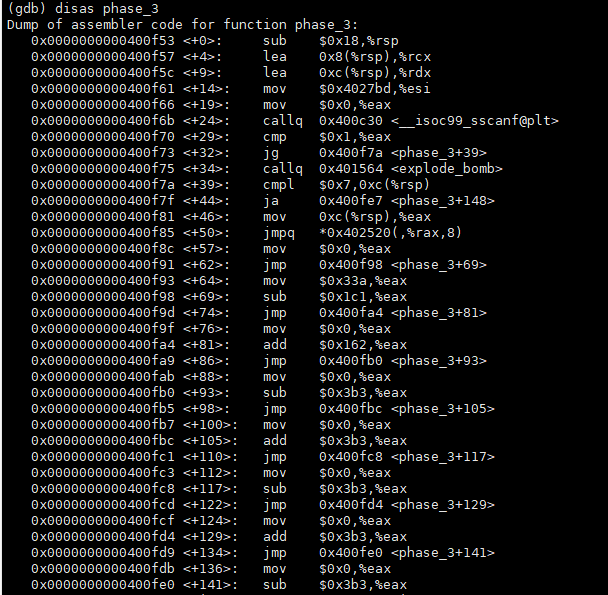


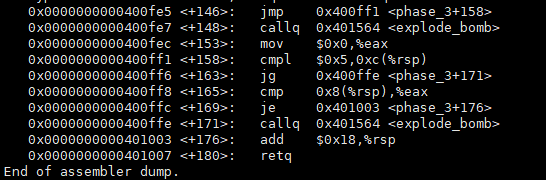
이를 통해 read\_six\_numbers 함수에서는 rsp부터 6개의 공간에 input string의 6개의 정수를 차례로 저장한다는 것을 알았다. 그런 뒤, <+51>부터 만약 scanf를 통해 입력 받은 정수의 개수가 5개 이하일 경우 bomb을 터트린다는 것을 알 수 있다.

* 다시 phase\_2 함수로 돌아가서 read\_six\_numbers 함수 호출 다음 rsp에 들어있는 주소의 값이 1인지 확인하고 만약 1이 아니라면 bomb을 터트린다. 이를 통해 input string의 첫 번째 정수는 1이라는 것을 알 수 있다.
* <+52>로 jump한 뒤, rbx, rbp 레지스터에 각각 rsp+4, rsp+24 값을 넣은 뒤, <+27>부터 <+50>까지 loop를 돌린다는 것을 알 수 있다. rbx-4를 한 값, 즉 input string의 전 정수 값에서 2배를 하여 rbx 값과 일치하는지 확인 후 일치하지 않으면 bomb을 터트리므로 input string의 6개의 정수는 1부터 2배를 한 “1 2 4 8 16 32”라는 것을 알 수 있다. 이때 rbp 레지스터에는 rsp+24 값을 넣어 loop에서 6개의 정수를 확인한 후 종료할 수 있도록 해준다.

1. **Phase3**

* phase\_3 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.



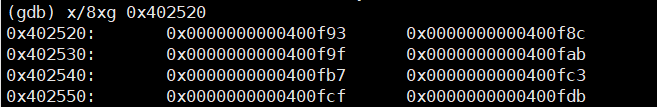


먼저 rsp에 24를 빼서 strack frame을 형성한 후 rcx, rdx를 적절한 위치에 저장한 뒤 scanf 함수를 호출하여 입력 값을 읽는 것을 알 수 있다. x/s를 통해 rsi 레지스터에 어떤 값의 주소가 저장되어 있는지 확인하였다.



이를 통해 2개의 정수를 입력해야 한다는 것을 알았고, scanf를 호출한 뒤 rax 값이 1보다 작거나 같은 경우 bomb을 터트린다는 것을 알았고, 첫 번째 정수는 rdx에 저장되고, 두 번째 정수는 rcx에 저장된다는 것을 알았다. rsp+12의 값, 즉 rdx의 값이 7보다 클 경우 <+148>에서 bomb을 터트리는 것으로 보아 입력 string의 첫 번째 정수는 7보다 작거나 같아야 함을 알 수 있다. <+46>부터 rax값에 rsp+12의 주소에 들어있는 값을 저장하고 이를 index로 하여 indirect jump를 하는 것을 알 수 있었다.

* x/8xg (memory) command를 통해 jump table이 어떻게 구성되어 있는지 확인하였다.



rax 값, 즉 index가 0일 경우 <+64>행으로, 1일 경우 <+57>행으로, 2일 경우 <+76>행으로, 3일 경우 <+88>행으로, 4일 경우 <+100>행으로, 5일 경우 <+112>행으로, 6일 경우 <+124>행으로, 7일 경우 <+136>행으로 jump를 한다는 것을 알 수 있다. 각각의 경우 jump를 한 뒤 rax 값을 필요한 경우 초기화 시킨 뒤 특정 값을 더하거나 빼는 것을 알 수 있다.

* 모든 instruction을 수행한 뒤 rsp+12 주소에 들어있는 값과 5를 비교하여 만약 5보다 크면 bomb을 터트리는 것으로 보아 첫 번째 정수는 5보다 작거나 같아야 한다는 것을 알 수 있었다. 각각의 index의 경우 16진수를 10진수로 바꾸어 연산한 결과가 rax에 저장되어 있고, 그 값을 rsp+8의 값, 즉 rcx와 비교하여 같지 않으면 bomb을 터트리는 것을 알 수 있다. 이를 통해 첫 번째 정수에는 5보다 작거나 같은 정수가 들어오고 두 번째 정수에는 첫 번째 정수를 index로 하여 연산한 결과가 들어가야 한다는 것을 알게되었다.
* 가능한 첫 번째 값을 x1이라 하고, 두 번째 값을 x2라 할 때 연산 과정은 다음과 같다.

x1 = 0인 경우: x2 = 826-449+354-947+947-947+947-947 = -216

x1 = 1인 경우: x2 = 0-449+354-947+947-947+947-947 = -1042

x1=2인 경우: x2 = 0+354-947+947-947+947-947 = -593

x1=3인 경우: x2= 0-947+947-947+947-947 = -947

x1=4인 경우: x2= 0+947-947+947-947 = 0

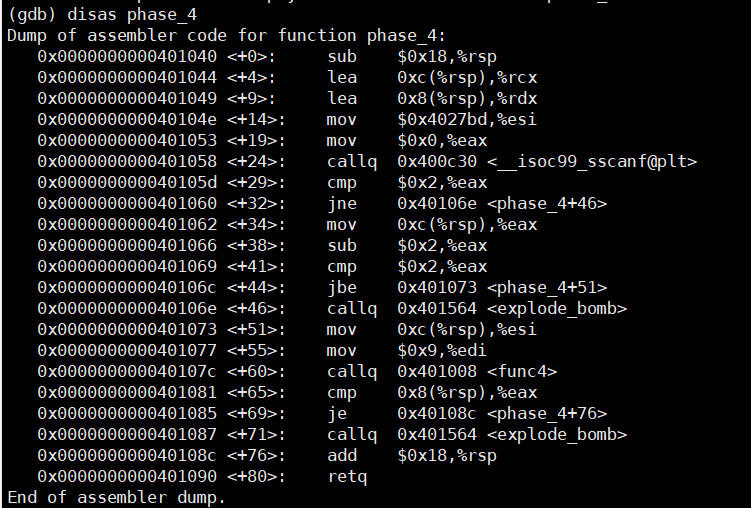
x1=5인 경우: x2= 0-947+947-947 = -947

따라서 가능한 입력 정수의 순서쌍은 다음과 같다.

(0 -216), (1 -1042), (2 -593), (3 -947), (4 0), (5 -947)

1. **Phase4**

* phase\_4 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

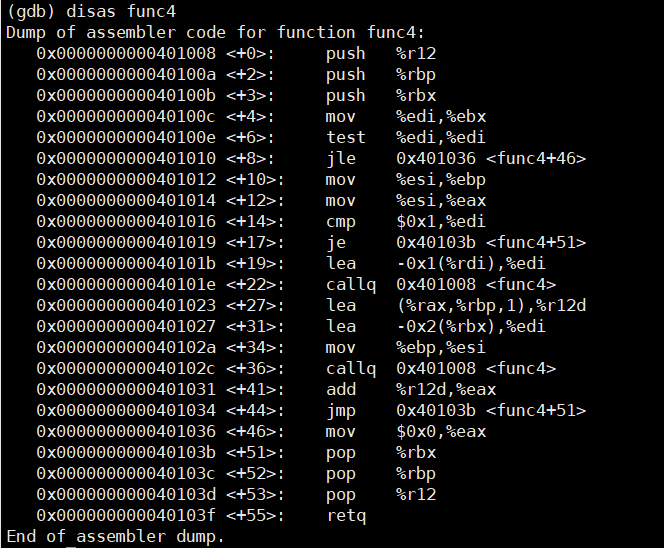


먼저 rsp에 24를 빼서 stack frame을 형성한 뒤 rsp+8, rsp+12에 각각 rdx, rcx 값을 저장한 뒤 rsi에 특정 메모리의 값을 저장하고 rax에 0을 넣은 후 scanf 함수를 호출하는 것을 알 수 있다. x/s를 통해 rsi에 저장된 값을 알아보면 다음과 같다.



scanf 함수를 호출한 뒤 rax 값이 2가 아니면 bomb을 터트리는 것으로 보아 입력 string에는 2개의 정수가 포함되어야 함을 알 수 있고, 첫 번째 정수는 rdx에, 두 번째 정수는 rcx에 저장되는 것을 알 수 있다.

* scanf 함수를 호출한 뒤 <+34>부터 rax에 rsp+12에 저장된 메모리 안의 값, 즉 두 번째 입력 값을 저장하고 rax에 2를 빼준 뒤 2와 비교하여 2보다 크면 bomb을 터트리는 것을 알 수 있다. 이를 통해 두 번째 정수에는 4보다 작거나 같은 값이 들어와야 한다는 것을 알았다.
* <+51>부터는 rsi에는 rsp+12에 저장된 메모리 안의 수, 즉 두 번째 정수를 저장하고, rdi에 9을 저장한 뒤 func4 함수를 호출하는 것을 알 수 있다. 그리고 이 함수의 return 값을 rsp+8에 저장된 메모리의 수와 비교하여 같지 않으면 bomb을 터트리는 것으로 보아 첫 번째 정수에는 func4 함수에 9와 두 번째 정수를 인자로 한 것의 return 값이 들어가야 한다는 것을 알 수 있다.
* func4 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.



함수를 보았을 때 rdi, 즉 첫 번째 인자가 0보다 작거나 같은 경우 0을 return하고, 1인 경우 1을 return하고 그렇지 않은 경우 rdi에 1을 빼서 다시 func4를 호출하는 것을 알 수 있다. 그런 다음 r12에 함수의 return 값에 rbp, 즉 두 번째 인자를 더하여 저장하고, 첫 번째 인자에 2를 뺀 뒤 func4 함수를 한 번 더 호출하는 것을 알 수 있다. 그런 다음 return 값에 r12를 더하는 것을 알 수 있다. rdi가 2 이상일 경우를 점화식의 형태로 나타내면 다음과 같다.

func4(di, si) = func4(di-1, si) + func4(di-2, si) + si

* 우리가 구해야 하는 것은 func4(9, si)이므로 이를 식으로 표현하면

func4(9, si) = func4(8, si) + func4(7, si) + si

위에서 종료 조건을 통해 func4(1, si) = si, func4(0, si) = 0이라는 것을 알기 때문에 이를 이용하여 func4(9, si) 값을 계산하면 그 값은 88\*si가 된다.

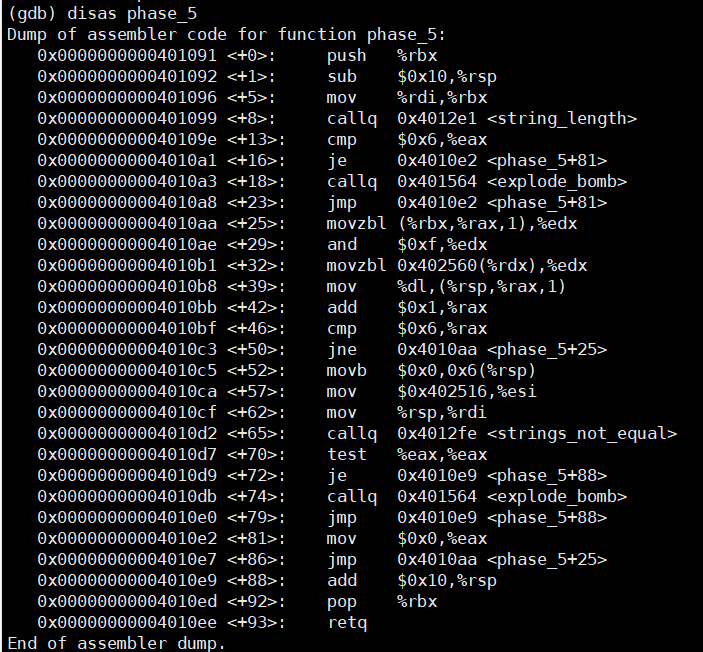
* func4 함수 호출이 종료된 후 return 값을 첫 번째 인자와 같은지 비교하기 때문에 두 번째 정수를 x라 할 때 첫 번째 정수는 88\*x의 형태여야 한다. 따라서 가능한 입력 정수의 순서쌍은 다음과 같다.

(176 2), (264 3), (352 4)

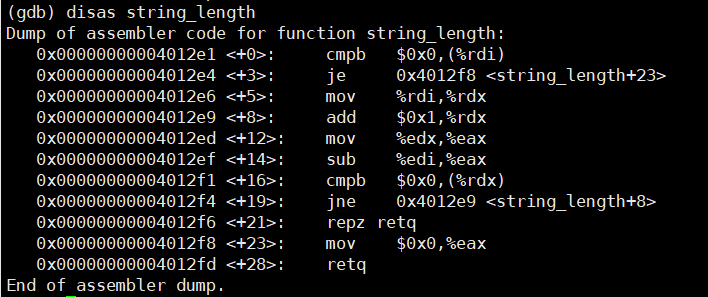
이때 두 번째 정수가 1인 경우는 불가능하다. <phase\_4+38>에서 두 번째 정수를 rax에 넣은 뒤 rax-2를 한 뒤 2와 비교하기 때문이다.

1. **Phase5**

* phase\_5 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.



먼저 rsp에 16을 빼서 stack frame을 형성한 뒤, rbx에 rdi의 값을 넣은 후 string\_length 함수를 호출하는 것을 알 수 있다. string\_length 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.



loop문을 사용하여 입력 받은 string의 길이를 구하여 return하는 함수임을 알 수 있다.

* <phase\_5+13>부터 길이가 저장된 rax를 6과 비교하여 같지 않으면 bomb을 터트리는 것으로 보아 입력 해야하는 string의 길이가 6이라는 것을 알 수 있다.
* rax에 0을 넣은 후 <+25>부터 loop에 들어가는 것을 알 수 있다. rdx에 rbx값, 즉 입력 string의 주소에 rax를 더한 값을 넣고 이를 0xf와 and 연산을 한 값을 다시 rdx에 넣은 후 <+32>에서 특정 메모리 주소에 rdx를 더한 값을 다시 rdx에 넣는다. 그리고 이를 rsp+rax의 주소에 넣는 것을 알 수 있다. 그런 다음 rax를 6이 될 때까지 1씩 더해주면서 반복해주는 것을 알 수 있다. x/s를 통해 메모리 주소의 값을 확인해보면 다음과 같다.



이를 통해 함수를 해석해보면 길이가 6인 string을 입력 받아 string의 한 자리씩 0xf와 and 연산을 한 값을 n이라 하면, 위 string의 n+1번째 수가 rsp+rax에 저장되어 결국 rsp에 새로운 string이 생성되는 것을 알 수 있다. 이때 string의 주소는 string의 첫 자리의 주소와 같고, rsp+6의 값에 0을 저장하여 string을 형성하는 것을 알 수 있다.

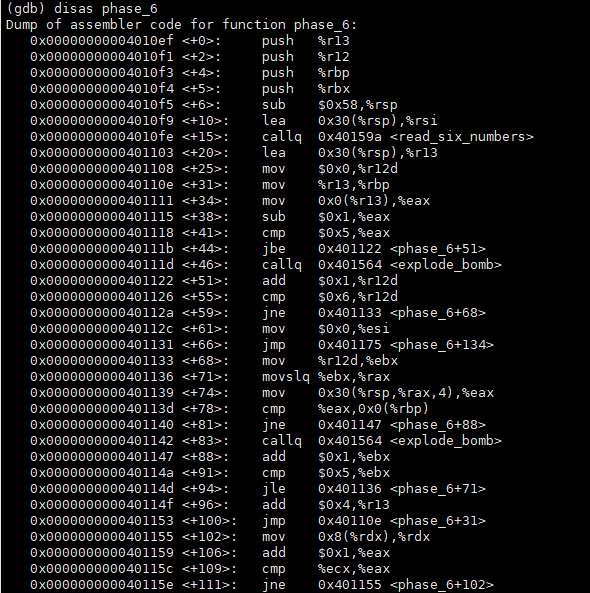
* 아스키코드를 통해 각 문자에 해당하는 수를 확인하여 0xf와 연산하면 아스키코드를 16으로 나눈 나머지가 나오는 것을 알 수 있다. 예를 들어 a는 아스키코드로 97이고 이를 0xf와 and 연산하면 결과는 1이 된다. 그리고 이는 위의 string의 2번째 수인 a를 rsp에 저장하게 만든다.
* rsp에 string을 형성한 뒤 rdi에 넣고, rsi에 특정 값을 넣은 후 strings\_not\_equal 함수를 실행시켜 두 string이 같은지 확인한다. 이 함수는 phase\_1에서 살펴본 바와 같이 두 string이 같으면 0, 다르면 1을 return하는 함수이다. 그리고 그 값이 0인지 test를 통해 확인하여 0이 아니면 bomb을 터트리는 것을 확인할 수 있다.
* x/s를 통해 rsi에 저장된 값을 확인해보면 다음과 같다.

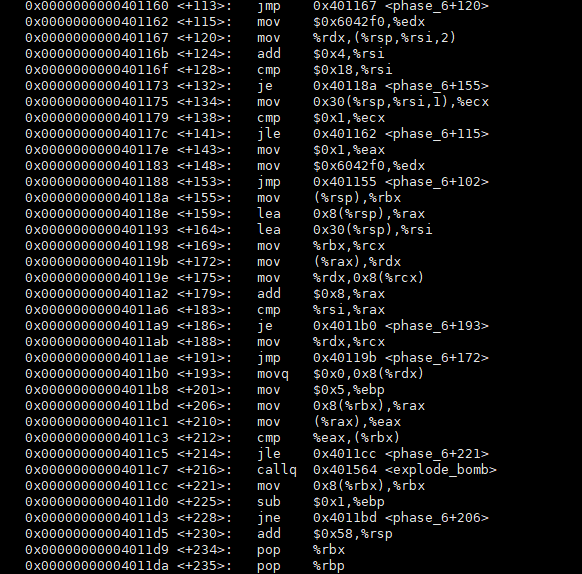


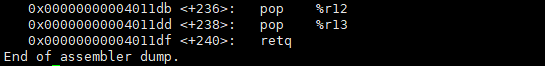
* rsp에 저장된 string이 flyers가 되도록 역추적하여 원래의 string이 무엇인지 찾아야 한다. 예를 들어 flyers의 첫 번째인 f는 위의 0x452560에 저장된 string에서 10번째에 해당하므로 0xf와 and 연산의 결과가 9가 되는 문자를 찾으면 아스키코드가 105인 i가 나온다. 이와 같은 방법으로 나머지 string을 완성시켜주면 입력 string에는 “ionefg”가 들어가야 한다는 것을 알 수 있다.

1. **Phase6**

* phase\_6 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

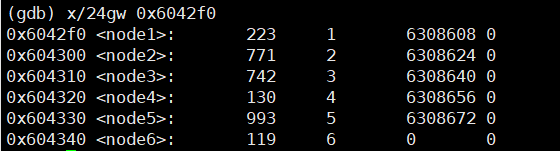






먼저 rsp에 88을 빼서 stack frame을 형성한 것을 알 수 있다. read\_six\_numbers는 phase\_2에서 확인해본 바와 같다. rsi에 rsp+48의 주소 값을 넣기 때문에 rsp+48부터 4byte씩 6개의 정수를 저장한다는 것을 알 수 있다.

* <+31>부터 <+100>까지 이중 loop를 돌리고 있다는 것을 알 수 있는데, 입력 받은 6개의 정수를 0번째부터 5번째라 할 때, 큰 loop에서는 rax에 n번째 정수 값을 넣어 <+38>부터 <+44>까지 그 값에 1을 뺀 값을 5보다 작거나 같은지 확인하고 있다. 즉, 입력 받은 6개의 정수가 모두 1이상 5이하의 정수라는 것을 알 수 있다. 그리고 0~5번째까지 모두 확인한 후에 r12에 들어있는 값이 6이 되었을 때 loop를 탈출하는 것을 알 수 있다.
* <+71>부터 <+94>까지는 큰 loop 안에 작은 loop를 형성하고 있는데, rbp에 n번째 수를 저장하고 있다면, rax에는 n+1번째부터 마지막까지의 수를 저장하도록 하여 두 수가 같을 경우 bomb을 터트린다. 즉, 두 loop에서 확인하고 있는 것은 입력 값이 서로 다른 6개의 1이상 6이하의 정수라는 것이다. 따라서 입력해야하는 6개의 수는 1, 2, 3, 4, 5, 6의 적절한 순서 조합이라는 것을 알았다.
* 앞의 loop에서 빠져나갈 때 <+134>행으로 이동하게 된다. <+102>부터 <+153>까지도 하나의 큰 loop를 형성하고 있는데, 작동 방식을 살펴보면 다음과 같다. <+134>에서 rcx에 48+(rsp+1\*rsi) 연산을 통해 rcx에 입력한 n번째 정수를 저장한다는 것을 알 수 있다. rsi는 <+124>에서 4씩 더하고 있으므로 index의 역할을 한다고 할 수 있다. rcx가 1일 경우 <+115>에서 rdx에 0x6042f0에 저장된 값을 넣고, 이를 다시 rsp+2\*rsi의 주소에 저장하는 것을 알 수 있다. rsi가 4씩 변하므로 rsp부터는 8byte 크기만큼의 pointer array를 형성하고 있다는 것을 알 수 있다.
* 0x6042f0에는 어떤 값이 저장되어 있는지 다음과 같이 확인해 보았다.

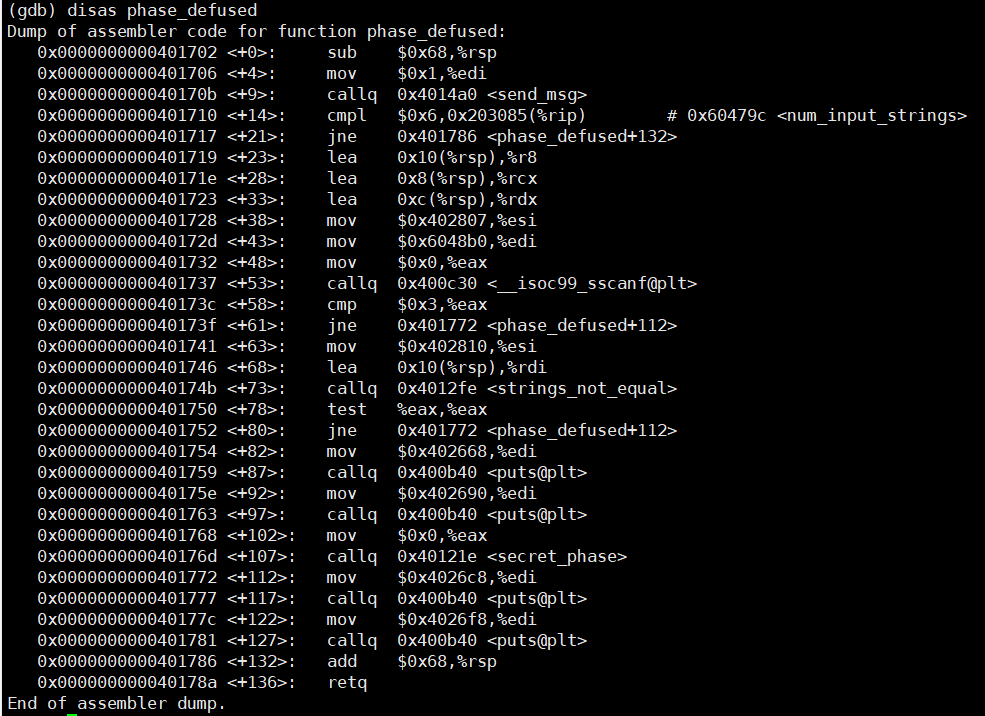


node1부터 6까지에는 세 자리의 정수와 1부터 6까지의 정수가 저장되어 있음을 확인할 수 있었다. rcx가 1일 경우, node1 pointer가 pointer array에 저장되고, 그 외의 경우에는 <+102>부터 <+111>까지 loop를 돌려 해당 숫자의 node pointer를 pointer array에 저장한다는 것을 알 수 있다. 즉, n번째 수가 X인 경우 nodeX를 pointer array의 n번째에 저장한다. <+128>과 <+132>에서 rsi 값을 24와 비교하여 같으면 loop를 탈출하는 것을 알 수 있다.

* loop를 탈출한 다음 rbx, rax, rsi를 각각 initialize 해준 뒤, <+172>부터 <+191>까지 loop에 들어가는 것을 알 수 있다. rcx, rdx에는 각각 array의 n번째, n+1번째 노드의 값이 들어가고, rax에 n+1번째 노드의 주소가 들어가도록 하여 <+175>행에서 해당 주소에 각 노드의 세 자리 정수 값이 저장되도록 하였다. rsi에는 array의 마지막 주소의 다음 주소로 하여 loop가 끝날 수 있도록 했다.
* 그런 다음 <+201>에서 rbp를 5로 초기화 해준 뒤, <+206>부터 <+228>까지 loop에 들어간다. loop를 실행할 때마다 rbp 값을 1씩 감소시킨 뒤 0이 되면 loop를 탈출한다. 해당 loop에서 rbx는 array의 n번째의 주소를 의미하고 rax는 n+1번째에 들어있는 값을 의미한다. rbx에 저장된 메모리에 들어있는 값과 rax와 비교하여 rbx에 저장된 메모리에 들어있는 값이 rax값보다 작으면 bomb을 터트린다. 이를 통해 node 안에 저장된 수들이 점점 커지는 순서로 node를 배열해야 한다는 것을 알았다. node1~6까지 저장된 수는 각각 223, 771, 742, 130, 993, 119이므로 해당 수들을 커지는 순으로 배열하기 위한 입력 값은 “6 4 1 3 2 5”임을 알았다.

1. **Secret Phase**

* main 함수에서 호출하는 phase\_defused 함수를 disassemble 하였을 때 <+107> 행에 secret\_phase가 존재한다는 것을 알 수 있었다.



* <+38>과 <+43>에서 rsi에 들어있는 주소에 어떤 값이 들어있는지 확인하면 다음과 같다.

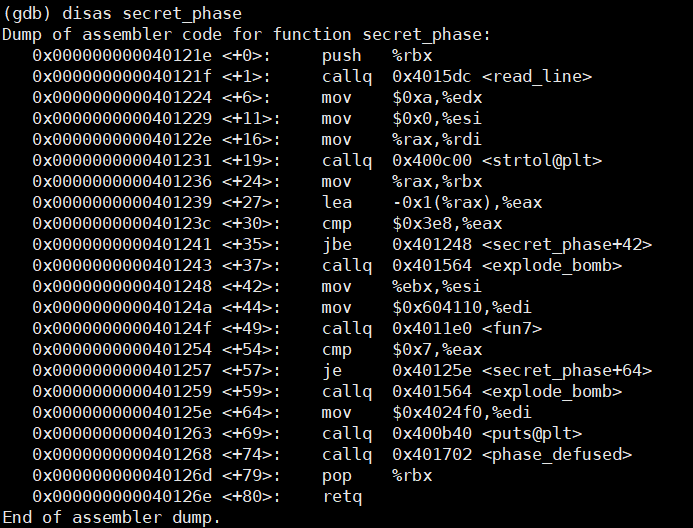


그리고 rdi에 들어있는 주소의 값은 phase\_4에서 입력한 string이 나왔다. scanf를 호출한 뒤에 <+58>에서 rax 값이 3이면 rsi에 특정 메모리 주소를 저장하고 strings\_not\_equal 함수를 호출하는 것을 알았다. 해당 메모리에 어떤 값이 저장 되어있는지 확인하면 다음과 같다.

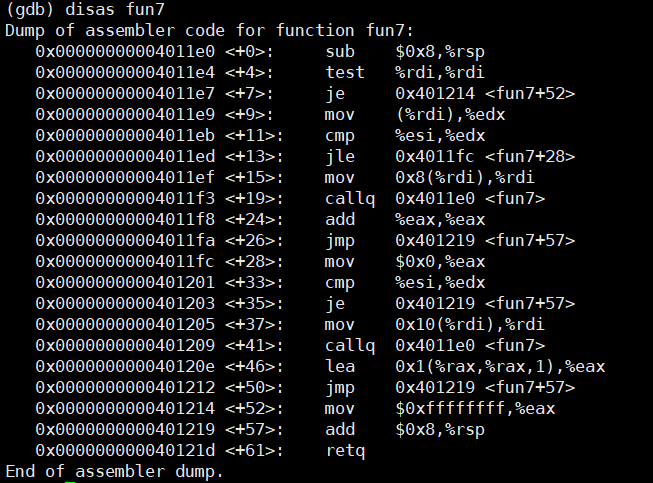


그리고 두 string이 같으면 secret\_phase 함수를 호출한다. 이를 통해 phase\_4의 입력을 받을 때 “DrEvil”을 추가로 입력하면 secret\_phase가 호출된다는 것을 알았다.

* secret\_phase 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.

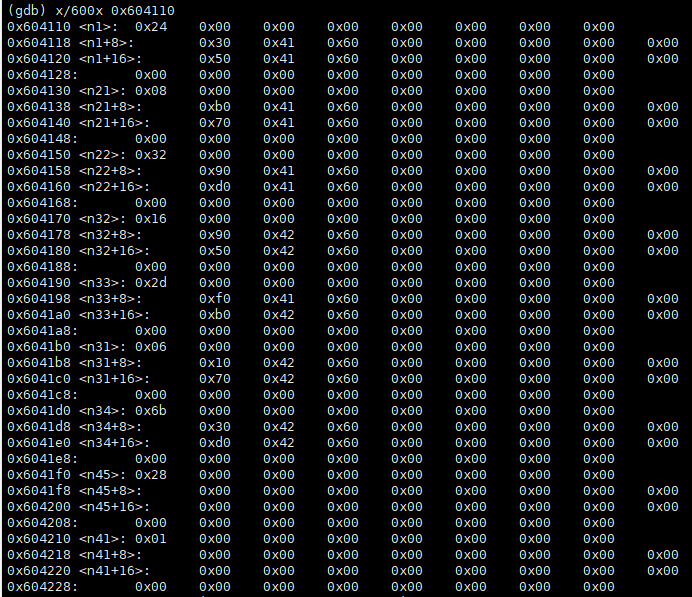


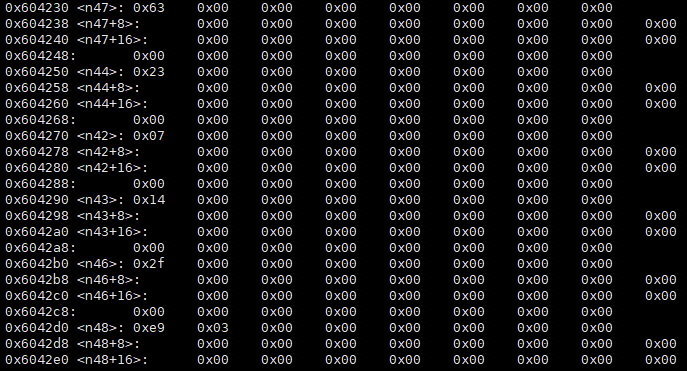
* 먼저 read\_line 함수를 호출하여 입력 받은 string을 rdi에 저장하고, rsi, rdx에 각각 0, 10을 저장한 뒤 strtol 함수를 호출한다. strtol 함수는 string을 long 타입의 정수로 변환해주는 함수이다. 변환해준 정수를 rbx에 저장하는 것을 알 수 있다.
* rax에 1을 빼준 뒤 1000과 비교한 뒤 1000보다 크면 bomb을 터트리는 것으로 보아 입력하는 정수 값은 1이상 1001이하라는 것을 알았다.
* 그 다음 rdi에 특정 메모리 주소를 넣고 rsi에 변환된 정수를 넣어 fun7 함수를 호출하는 것을 알았다. 그 다음 rax를 7과 비교하여 같지 않으면 bomb을 터트리는 것으로 보아 fun7 함수의 return 값이 7이 되도록 하는 적절한 정수를 입력해야 한다는 것을 알았다.
* fun7 함수를 disassemble 하여 함수의 작동 방식을 살펴보았다.



첫 번째 인자가 0일 경우 rax에 0xffffffff을 넣어 return하는 것을 알 수 있다. rdx에 첫 번째 인자로 받은 주소에 들어있는 값을 넣고, rdx가 두 번째 인자보다 클 경우 rdi에 rdi의 주소에 8을 더한 주소에 들어있는 값을 넣은 뒤 다시 fun7 함수를 호출하고, return 값에 2를 곱한다. 작거나 같을 경우에는 rax에 0을 넣고 rdx와 rsi가 같으면 함수를 return, 아니면 rdi에 16을 더한 주소에 들어있는 값을 넣고 fun7 함수를 호출한 뒤 return 값에 2를 곱하고 1을 더하는 연산을 수행한다.

* fun7 함수의 첫 번째 인자로 들어오는 0x604110에 어떤 값이 들어있는지 확인하였다.





메모리에는 먼저 <nX>에 정수가 들어있고, <nX+8>과 <nX+16>에 주소가 little endian의 방식으로 적혀 있는 것을 알 수 있다. <nX>를 하나의 노드라고 하면 이 메모리에는 tree형식으로 값이 저장되어 있다.

* 메모리에 들어있는 값과 fun7 함수를 종합적으로 볼 때, fun7 함수는 두 번째 인자와 노드의 값을 비교하여 다음 노드의 값을 두 번째 인자로 하여 다시 fun7 함수를 호출하는 것을 반복하면서 위에서 언급한 규칙대로 return 값을 변화시킨다.
* fun7 함수의 최종 return 값이 7이 되어야 하는데, 이 값은 0, 2\*0+1 = 1, 2\*1+1 = 3, 2\*3+1 = 7의 과정을 거쳐 만들 수 있다. 이를 위해서는 tree에서 오른쪽으로 이동하는 동작을 3번 수행해야 하므로 두 번째 인자로는 <n48> 노드의 값이 들어있어야 한다.
* <n48> 노드의 수 또한 little endian의 방식으로 적혀 있으므로 해당하는 수는 ‘0x3e9’이고, 이는 10진수로 1001이다. 따라서 secret\_phase의 입력 값은 1001임을 알았다.