**2022-2 CSED211 Lab03 Report**

학번 : 20210479

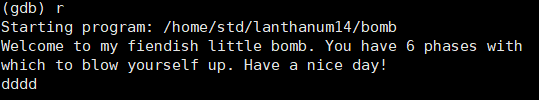
이름 : 이주현

**명예서약 (Honor Code)**

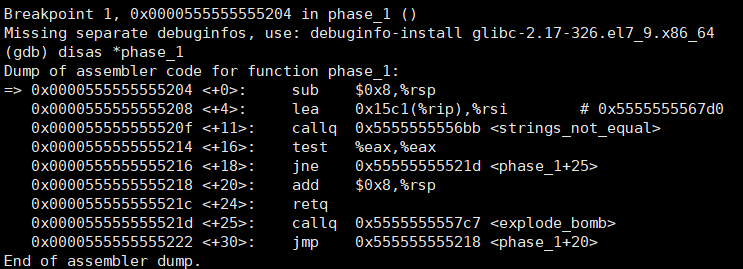
나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.

I completed this programming task without the improper help of others.

1. **문제별 코드 설명**
   * **Before Start Defusing Bomb**
     + 우선 해결에 앞서 서버에 bomb 파일을 넣고, ‘gdb bomb’ 명령어를 실행해 디버깅을 시작하였다. 시작한 당시에는 어떤 phase의 정답도 알고 있지 않으므로 breakpoint를 걸어 폭탄의 폭발을 막아주었다. 이를 위해 ‘info function’ 명령어를 실행해 어떤 함수들이 있는지 보고, 아래와 같이 ‘b\* phase\_1’, ‘b\* explode\_bomb’ 명령어를 실행해 해결에 진입했다.
     + 
       - phase\_1의 답을 모르는 상태에서 무언가를 입력하면 폭탄이 폭발하는 것으로 넘어갈 것 같아 phase\_1 실행 전에 멈추도록 했다.
     + 
       - 답이 틀렸을 경우에도 폭탄이 폭발하는 것을 막기 위해 breakpoint를 걸어주었다.
   * **Phase\_1**
     + breakpoint를 건 후 ‘r’ 명령어를 실행해 bomb을 실행했다.



* + - 답을 모르는 상황이므로 아무거나 입력해 다음 breakpoint로 이동하도록 만들었다. 그리고 디버깅을 위해 ‘disas \*phase\_1’을 입력해 phase\_1에 어떤 instruction들이 있는지 살펴보았다.

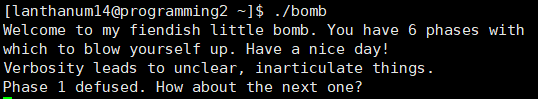


“strings\_not\_equal”이라는 함수를 부르는 것으로 보아, 어떤 string을 입력했을 시 그것과 답을 비교해 phase를 통과했는지 통과하지 못했는지 판단하는 것일 거라고 예상했다.

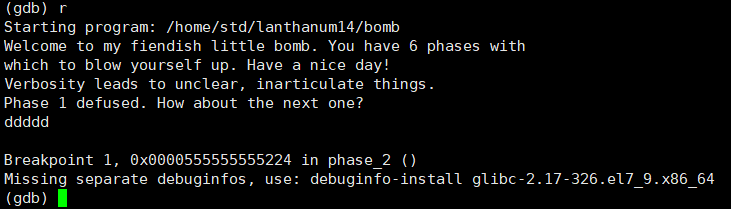
* + - 이때 strings\_not\_equal 함수를 바로 전에 0x15cl(%rip)를 %rsi에 저장하는 것을 보아, %rsi가 strings\_not\_equal 함수의 argument로 쓰일 것이라는 생각을 했다. 이에 ‘x/s 0x5555555567d0’ 명령어를 사용해 해당 주소에 어떤 string이 저장되어 있는지 확인했다.



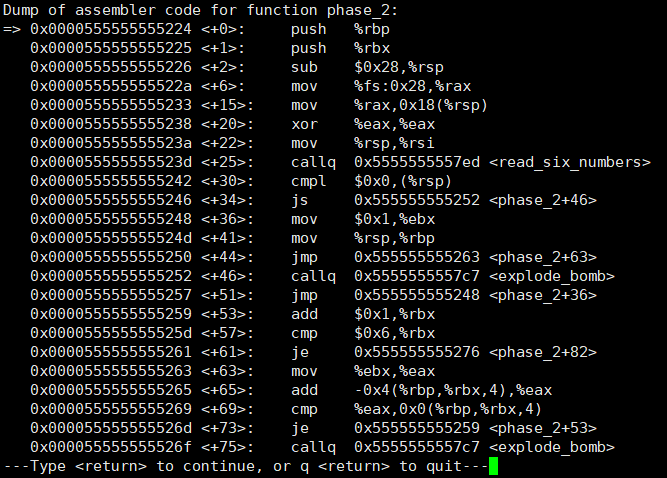
* + - 해당 string이 phase\_1의 답일 것이라고 생각해 bomb 실행 후 string을 써 넣었다. phase\_1을 해결했다.

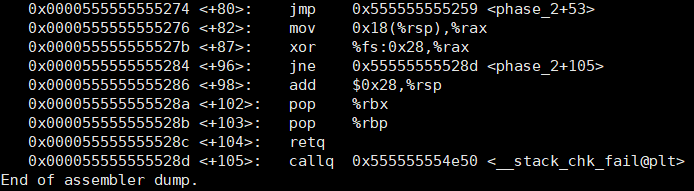


* + **Phase\_2**
    - 위에서 했던 것과 같이, ‘b\* phase\_2’, ‘b\* explode\_bomb’ 명령어를 이용해 breakpoint를 만든 후 ‘r’ 명령어로 프로그램을 실행했다.



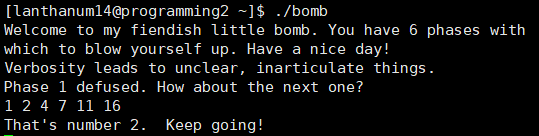
* + - phase\_2에 어떤 instruction들이 있는지 살펴보기 위해 ‘disas \*phase\_2’를 입력했다.



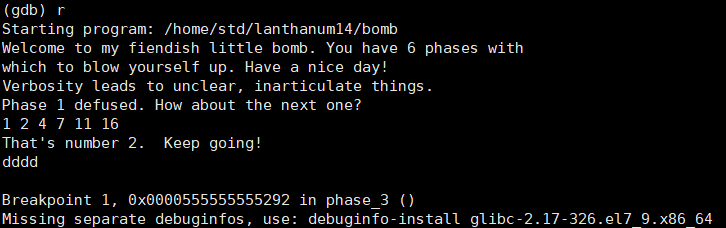


이번에는 “read\_six\_numbers”라는 함수로 보아 6자리 혹은 6개의 숫자를 읽어들이는 것이라고 생각했다. ‘disas\* read\_six\_numbers’ 명령어로 어셈블리 코드를 살펴 보았으나, 입력받은 숫자의 개수를 확인하는 함수로 보여 따로 분석하지 않고 phase\_2 어셈블리 코드 분석을 이어 나갔다.

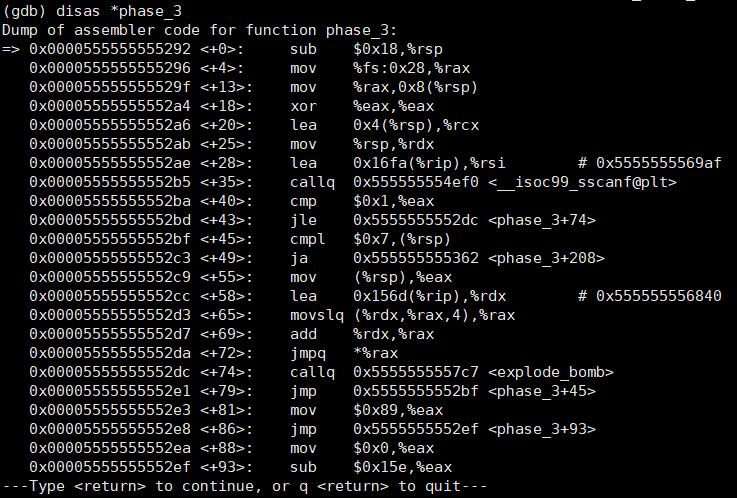
* + - <phase\_2+30>을 보면, (%rsp)-$0x0 값을 조사하는 것을 알 수 있다. 이때 이 값이 음수라면 ‘explode\_bomb’이 실행되는 것으로 보아, 첫 번째 입력하는 값이 양수여야 한다는 사실을 알 수 있다.
    - 만약 첫 번째 값으로 양수를 입력했다면, 뒤이어 %ebx에 0x1을 옮기고 %rsp를 %rbp로 옮긴 후, <phase\_2+63>으로 점프해 %ebx를 %eax로 옮기는 동작이 이어지는 것을 알 수 있었다. 이후 %eax에 %rbp+4\*%rbx-4를 더해 그 값을 %eax에 저장하고, %eax와 %rbp+4\*%rbx를 비교하는 동작이 이어졌다. 그 후 만약 둘이 같다면 <phase\_2+53>부터 <phase\_2+73>까지가 반복되는 것을 알 수 있었다. 반복을 끝내는 조건은 %rbx가 6과 같아지는 것이었다.
    - 즉 우리는 이 코드를 해석해 6회에 걸쳐 어떤 수를 입력해야 하는지를 알아낼 수 있다. 우선 첫 번째 숫자는 양수라는 것을 위에서 밝혀내었다. 나머지 숫자들은 일정한 규칙을 따르는 것으로 보이는데, %rbp+4\*%rbx-4와 %eax를 더한 값을 %rbp+4\*%rbx와 비교하는 것으로 보아, 직전에 입력한 값에 %eax(=%ebx)를 더한 값이 이번에 입력해야 하는 값이 되는 것으로 보였다. 즉, 처음에 입력한 수에서 1, 2, 3, 4, 5씩 점점 증가하는 수들을 입력하면 되는 것이다.
    - 해당 조건을 만족하는 ‘1 2 4 7 11 16’을 입력하자, phase\_2를 해결할 수 있었다.

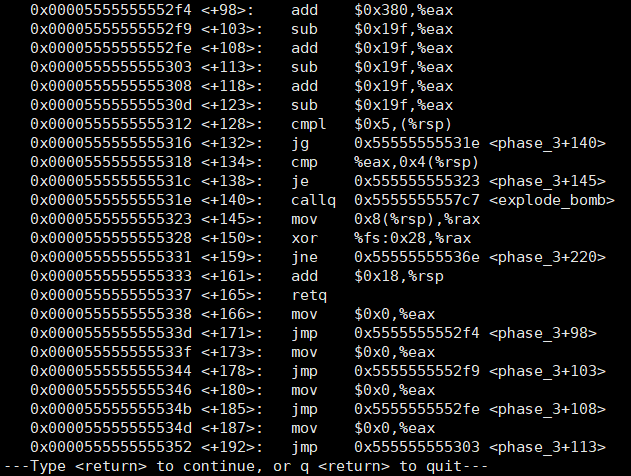


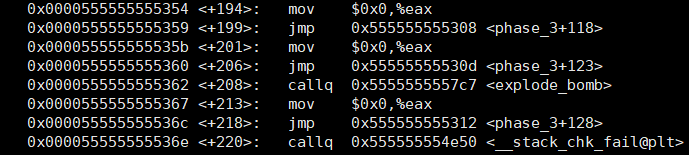
* + **Phase\_3**
    - 위에서 했던 것과 같이, ‘b\* phase\_3’, ‘b\* explode\_bomb’ 명령어를 이용해 breakpoint를 만든 후 ‘r’ 명령어로 프로그램을 실행했다.



* + - phase\_3에 어떤 instruction들이 있는지 살펴보기 위해 ‘disas \*phase\_3’를 입력했다.



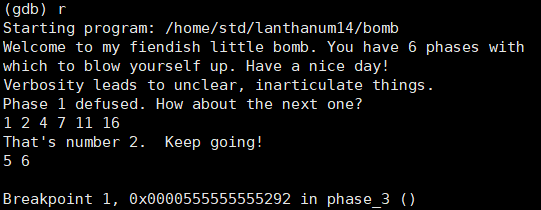




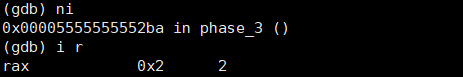
* + - <phase\_3+35>에서 scanf가 불리고 있으므로, 그 위에서 어떤 값을 받는지 알아보기 위해 ‘x/s 0x5555555569af’ 명령어로 scan 형식을 알아보았다. 그 결과, 10진수 두 개를 입력해야 함을 알 수 있었다.



* + - 따라서 입력 형식을 맞춰주기 위해 ‘k’ 명령어로 지금 실행 중인 프로그램을 끝내고, 다시 프로그램을 실행해 아래와 같은 값을 입력해 주었다.



* + - scanf 아래의 명령어들을 계속해서 분석하였다. 수를 입력받은 직후 <phase\_3+40>에서 $0x1과 %eax를 비교하는 것을 볼 수 있었다. 이때 %eax가 1보다 작거나 같으면 폭탄이 터지는 것을 확인할 수 있었다. 그래서 %eax 값은 무엇으로 결정되는지 보기 위해 ‘ni’ 명령어로 scanf를 지난 직후인 <phase\_3+40>까지 실행한 후, %eax 값을 살펴보았다. %eax 값이 2가 되어있는 것으로 보아 입력받은 숫자의 개수인 듯 했다. 이를 통해, 2개의 숫자를 입력하지 않으면 explode\_bomb이 실행된다는 것을 알 수 있었다.

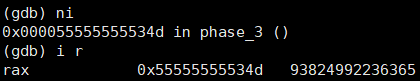


* + - 이렇게 입력받은 수가 두 개임을 확인하고 나면 <phase\_3+45> 동작을 실행하게 되는데, 여기에서는 (%rsp)와 $0x7을 비교하며, 만약 (%rsp)가 7보다 above라면 expode\_bomb이 실행되는 것을 확인할 수 있었다. 그래서 여기에서도 ‘ni’ 명령어로 instruction을 차례로 실행해 <phase\_3+45>를 실행한 뒤, %rsp 값을 확인해보았다.



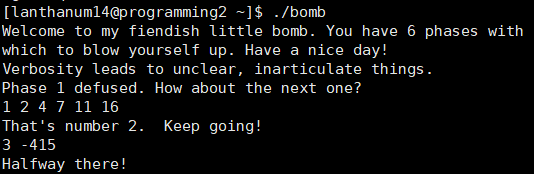
아까 입력했던 값으로 미루어 보아, %rsp가 첫 번째 입력한 값임을 알 수 있었다. 즉, 첫 번째로 입력하는 값은 7보다 작아야 하는 것이다. 현재는 그 조건을 만족하고 있으므로 다음 명령어들을 계속해서 분석했다.

* + - <phase\_3+55>부터 %rax에 관한 연산들을 하다가, <phase\_3+72>에서 \*%rax로 점프하는 것을 알 수 있었다. 그래서 ‘ni’ 명령어로 <phase\_3+72>까지 실행한 후, %rax 주소 값을 확인해 보았다.

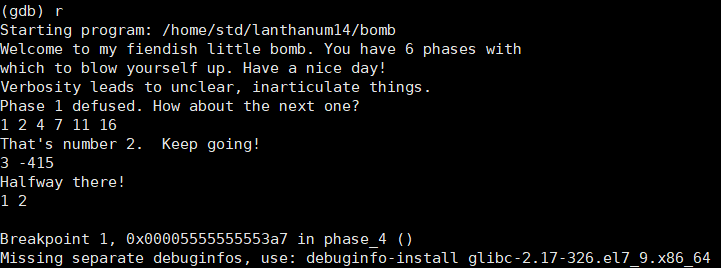


해당 주소는 <phase\_3+187> 자리였으며, $0x0을 %eax로 옮긴 후 <phase\_3+113>으로 이동하는 동작을 하는 곳이었다.

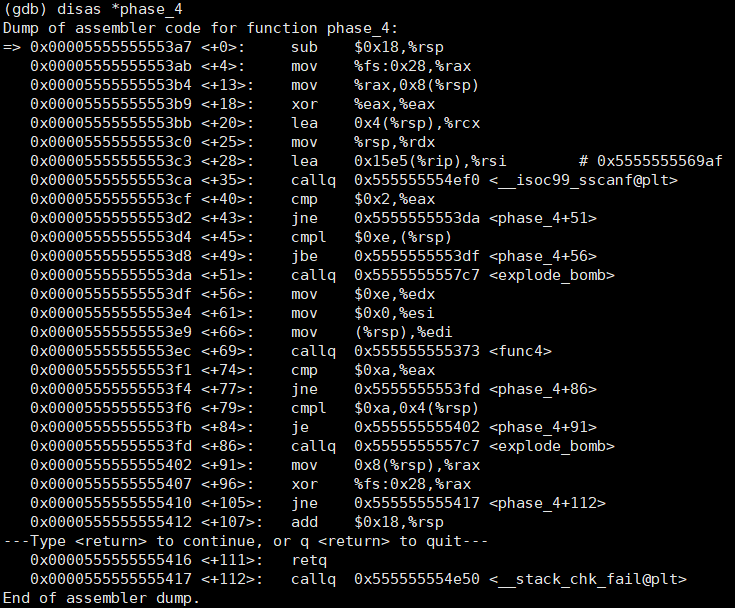
* + - 그래서 <phase\_3+113>으로 이동해 해당 명령어를 분석했다. <phase\_3+113>에서는 %eax에 대해 -$0x19f+$0x19f-$0x19f 동작을 하는 것을 알 수 있었다. 상쇄되는 것을 제하면 결국 %eax를 0으로 초기화한 뒤 $0x19f를 빼는 것과 같았다. 0x19f를 십진수로 나타내면 415이므로, 결국 %eax의 값은 -415가 되는 것이다.
    - 그 이후 <phase\_3+128>에서는 (%rsp)를 5와 비교하는 동작이 일어났다. 이때 (%rsp)가 5보다 크면 explode\_bomb이 실행되는 것을 알 수 있었고, 입력한 값이 5였으므로 다음 단계로 진행했음을 알 수 있었다.
    - <phase\_3+134>에서는 (%rsp)+4와 %eax를 비교하고 있었다. 이는 즉 %rsp 이후 입력된 integer, 즉 두 번째 입력된 값과 %eax를 비교하는 것과 같다. 그리고 <phase\_3+138>에서 두 번째 입력된 값이 %eax와 같지 않다면 explode\_bomb을 실행하는 것을 알 수 있었다. 즉, 두 번째 입력할 값은 %eax에 저장한 값, 다시 말해 -415여야 한다는 것이다. 만약 -415를 입력했다면 몇 단계를 더 거쳐 phase\_3이 종료되는 것을 확인할 수 있었다.
    - 정리하자면, phase\_3에는 5보다 작은 한 숫자와 -415를 입력해야 하는 것으로 보였다. 이를 확인하기 위해 3 -415를 입력했다. phase\_3을 해결한 것을 확인할 수 있었다.



* + **Phase\_4**
    - 위에서 했던 것과 같이, ‘b\* phase\_4’, ‘b\* explode\_bomb’ 명령어를 이용해 breakpoint를 만든 후 ‘r’ 명령어로 프로그램을 실행했다.



* + - phase\_4에 어떤 instruction들이 있는지 살펴보기 위해 ‘disas \*phase\_4’를 입력했다.

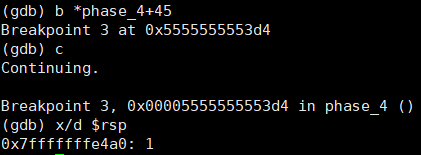


* + - phase\_3과 유사하게 <phase\_4+35>에서도 scanf가 불리는 것을 확인할 수 있었다. 그래서 어떤 형식의 값을 받는지 알아보기 위해 ‘x/s 0x5555555569af’ 명령어롤 통해 scan 형식을 알아보았다.



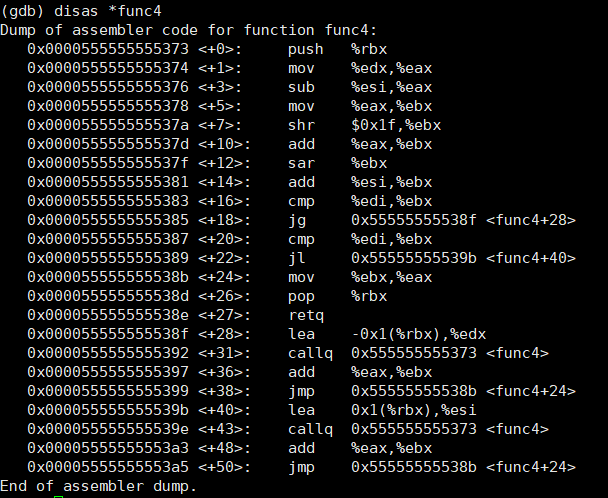
그 결과, 10진수 두 개를 입력해야 함을 알 수 있었다. 이번에는 10진수 두 개를 입력한 채 시작했으므로, 계속해서 분석을 진행했다.

* + - 이번에도 <phase\_4+40>에서 %eax와 $0x2를 비교하는 동작이 있었다. phase\_3과 유사하게 %eax가 1 이하일 경우 explode\_bomb을 실행하는 것으로 보아, 입력받은 숫자의 개수를 확인하는 것임을 알 수 있었다. 따라서 분석을 계속 진행했다.
    - 이번에는 <phase\_4+45>에서 (%rsp)와 $0xe를 비교했다. 그런 후 (%rsp)가 $0xe 보다 below이거나 같을 경우 다음으로 넘어가는 과정을 거쳤다. 이것도 phase\_3과 유사하게 입력받은 값을 비교하는 과정일 것 같아 breakpoint를 만들고 ‘c’ 명령어로 <phase\_4 + 45>까지 프로그램을 실행한 뒤, %rsp의 값을 읽어 보았다.



앞서 입력했던 값으로 미루어 보아, 이번에도 %rsp는 첫 번째 입력한 값을 읽고 있음을 알 수 있다. 즉 phase\_4에서는 첫 번째 입력하는 값이 14 이하이어야 함을 알 수 있다.

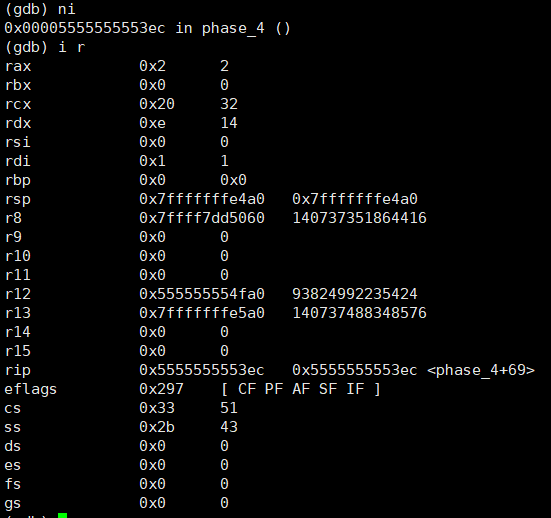
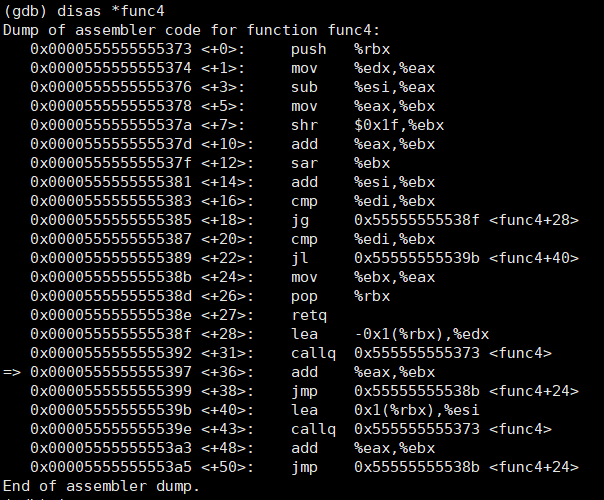
* + - 그 다음 과정인 <phase\_4+56>에서는 %edx에 $0xe를 옮기고, %esi에는 $0x0을 옮기고, %edi에는 (%rsp)를 옮기는 과정이 진행됨을 볼 수 있었다. 그런 후 “func4”라는 함수를 부르는 것을 볼 수 있었다. 따라서 ‘disas \*func4’ 명령어를 이용해 func4 함수의 어셈블리 코드를 살펴보았다.



<func4+31> 등을 통해 func4 함수는 자기 자신을 호출하는 재귀함수임을 알 수 있었다. 우선 func4가 종료되고 반환된 %rax 값을 확인해보았다.



* + - <phase\_4+74>에서 %eax와 $0xa를 비교하고, 같지 않으면 explode\_bomb을 실행하는 것을 볼 수 있었다. 즉 func4가 종료된 시점에서 %rax 값이 10진수로 10이어야 한다는 것이다. func4를 통과해 10이 되는 %eax의 값을 찾기 위해서는 이전에 정해진 %edx, %esi, %edi 값을 이용해 재귀함수를 역으로 계산해야 한다.
    - 다시 func4의 어셈블리 코드로 돌아와 분석을 시작하였다. 우선 func4로 진입할 당시 %eax = 2, %edx = 14, %esi = 0, %edi = 1 (첫 번째로 입력해 준 값)임을 기억하고, func4의 instruction들을 읽었다.

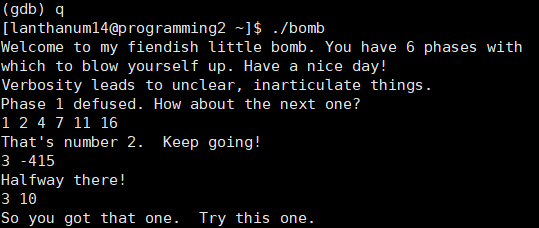
func4는 우선 %eax를 14로, %ebx를 7로 초기화한 뒤, 입력해준 값인 %edi와 %ebx를 비교해 %ebx가 더 클 경우 %edx에 (%rbx)-1을 대입해 다시 func4를 호출하는 형식의 함수이다. 만약 %ebx가 작을 경우에는 %esi에 (%rbx)+1을 대입해 다시 func4를 호출하는 형식이었다. 그리고 종료되기 전 %eax와 %ebx가 더해져 %eax에 대입되는 것을 알 수 있었다.

* + - 이러한 func4의 특성을 참고해 c코드로 나타내어 보면 아래와 같았다.

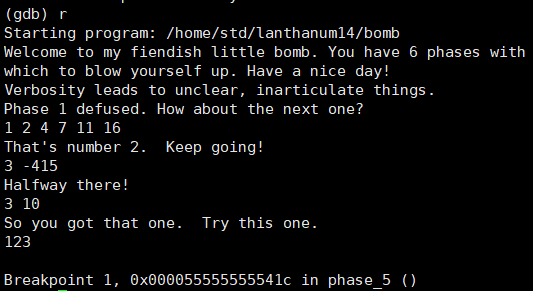
|  |
| --- |
| int func4(int dx, int di, int si)  {  int bx = (dx + si) / 2;  if (di < bx)  {  return func4(bx - 1, di, si) + bx;  }    else if (di > bx)  {  return func4(dx, di, bx + 1) + bx;  }    else  {  return bx;  }  } |

따라서 여기에 들어가서 최종값을 10으로 만드는 %edi를 찾아보면 3이다. 즉, 첫 번째 값으로 3을 입력하면 되는 것이다.

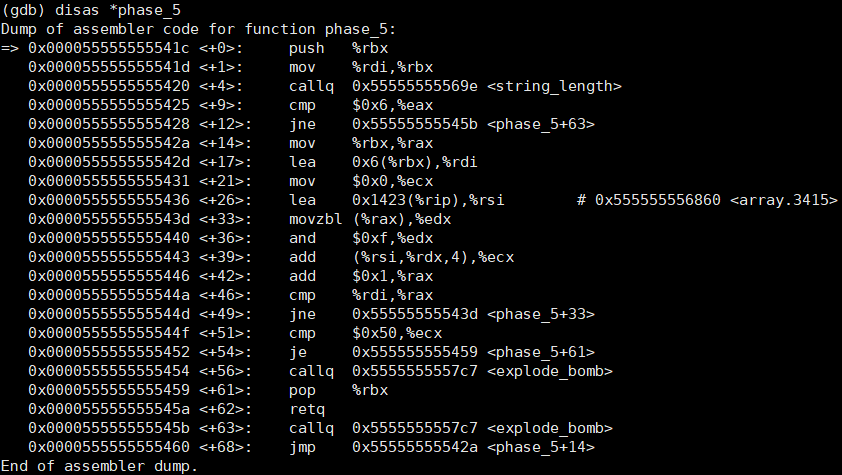
* + - 이제 남은 phase\_4를 보면, <phase\_4+79>에서 (%rsp)+4를 $0xa와 비교하는 부분이 있다. 그리고 거기에서 둘이 같지 않다고 판단되면 explode\_bomb을 실행한다. 이때 (%rsp)+4는 첫 번째 입력한 값 바로 위에 저장된 integer, 즉 두 번째로 입력한 값일 것이다. 따라서 두 번째로 입력하는 값은 10이어야 함을 알 수 있다.
    - 정리하면 phase\_4 defuse를 위해서는 ‘3 10’을 입력하면 된다고 할 수 있다. 이를 아래와 같이 직접 입력해 확인하였다. 이로써 phase\_4를 해결했다.



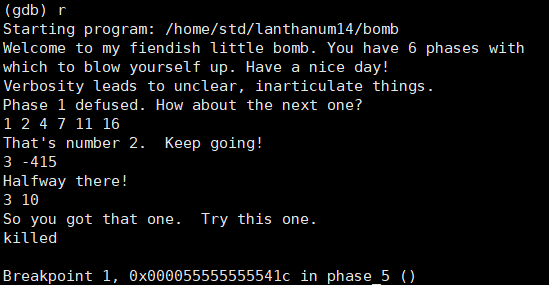
* + **Phase\_5**
    - 위에서 했던 것과 같이, ‘b\* phase\_5’, ‘b\* explode\_bomb’ 명령어를 이용해 breakpoint를 만든 후 ‘r’ 명령어로 프로그램을 실행했다.



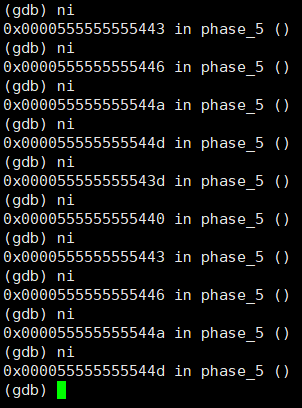
* + - phase\_5에 어떤 instruction들이 있는지 살펴보기 위해 ‘disas \*phase\_5’를 입력했다.



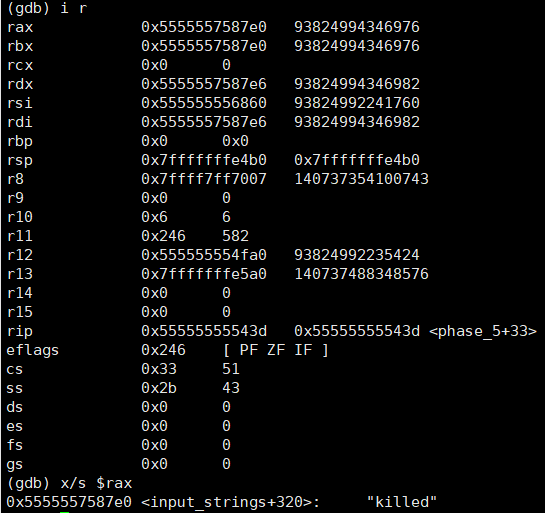
* + - 우선 <phase\_5+9>에서 %eax와 $0x6을 비교하고, 둘이 같지 않으면 explode\_bomb을 실행시키는 것으로 보아 %eax가 6이어야 함을 알 수 있다. 그리고 <phase\_5+4>에 “string\_length” 함수를 호출하는 부분이 있는 것으로 보아, 6자로 된 string을 입력해야 하는 것으로 보인다. (즉 %eax는 입력한 string의 길이를 나타내는 것이다.)
    - 따라서 기존에 입력했던 것을 삭제하고, 6글자로 된 string을 넣어 다시 phase\_5 분석을 시작했다.



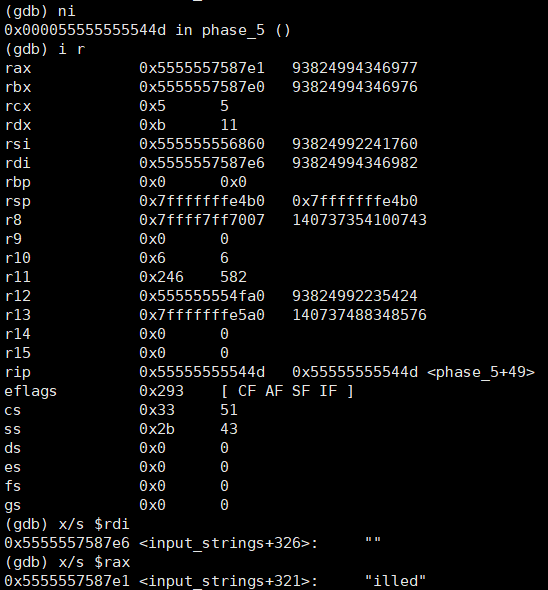
* + - ‘killed’ 라는 6글자 string을 넣고 ‘ni’ 명령어로 계속 한 단계씩 실행한 결과, 아래와 같이 <phase\_5+33>부터 <phase\_5+49> 사이를 계속해서 반복하고 있는 것이 보였다.



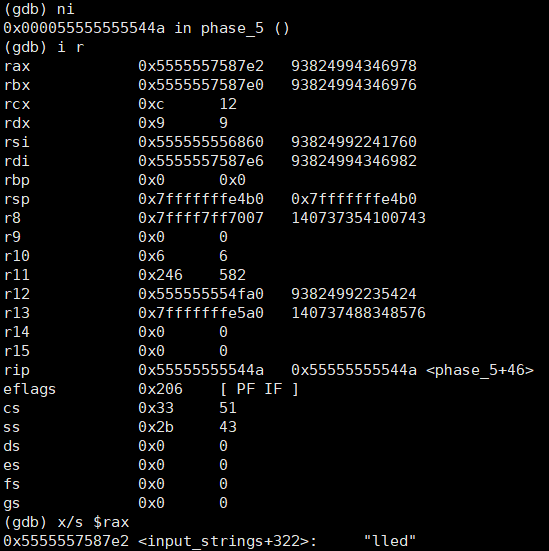
즉 이 말은 <phase\_5+46>에 있는 %rdi와 %rax 비교에서 not equal이 나온 것임을 알 수 있다. 이 부분의 코드를 분석하기 위해 <phase\_5+33>에 breakpoint를 걸고, 한 단계씩 진행하며 레지스터의 값들을 살폈다.

* + - 

루프 진행 전에는 %rax, %rbx에 모두 입력한 string인 “killed”가 저장되어 있는 것을 볼 수 있다.



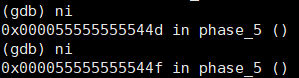
계속해서 진행하던 중 루프 한 번을 돌자 %rax에 변화가 생겼다. 맨 앞 “k”가 사라진 것을 볼 수 있다.



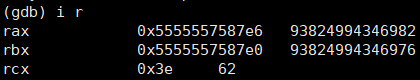
두 번째 루프를 돌자 %rax의 맨 앞이었던 “I”도 사라진 것을 알 수 있다.



위와 같이 루프를 한 번 돌 때마다 %rax에 저장된 string이 앞에서부터 사라지는 것을 볼 수 있다.

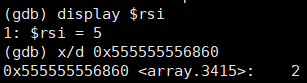


그리고 이로써 %rax와 %rdi가 같아졌으므로 루프를 빠져 나왔다. 이제 explode\_bomb을 피하기 위해서 남은 것은 위에 나온 <phase\_5+51>, 즉 $0x50과 %ecx의 비교뿐이다.

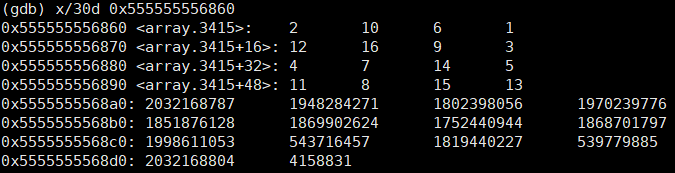
* + - 

비교 전에 %rcx 값을 보았다. $0x3e라는 값을 가지고 있음을 알 수 있다. 이 값이 어떻게 정해지는지 알기 위해서 <phase\_5+51> 위쪽의 구문들을 다시 한번 살폈다.

* + - %ecx에 관한 부분은 <phase\_5+21>에서 처음 0으로 리셋하는 부분과, 루프에서 (%rsi + 4\*%rdx)를 더해 업데이트하는 부분에서 찾을 수 있었다. 이때 위의 루프를 다시 한번 참고하니, %rdx는 계속해서 값이 바뀌고, %rsi는 그 값이 ‘0x555555556860’으로 계속 일정했음을 알 수 있다. 이 %rsi의 기원은 <phase\_5+26>에서 찾을 수 있었다.
    - <phase\_5+26>에서, %rsi 옆에 <array.3415>이라고 표시가 되어있다.

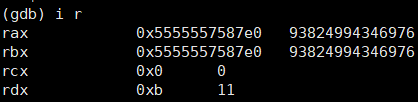
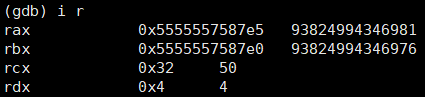
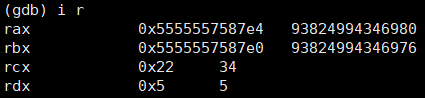
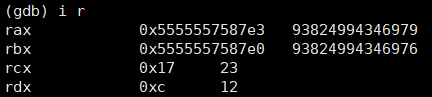
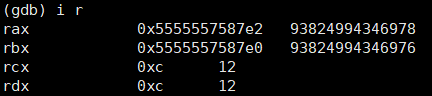
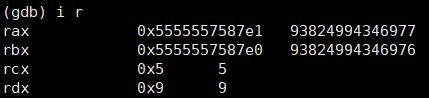


%rsi가 array의 형태로 저장되어 있는데 제대로 출력이 되지 않는 것이라 판단해, ‘x/30d 0x555555556860’ 명령어를 사용해 배열을 출력하도록 해보았다.



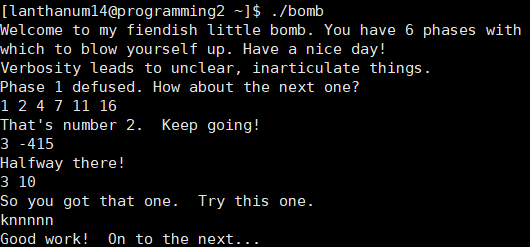
확인 결과 %rsi는 nested array 형태로 저장되어 있었다. 그리고 이 값은 변하지 않고 계속 유지되었으므로, 결국 %ecx 값이 변하는 것은 %rdx에 의한 것임을 짐작할 수 있다.

* + - ‘killed’라는 string을 넣었을 때 %rdx의 변화를 지속적으로 관찰하였다. <phase\_5+33>에서 (%rax)와 %rdx에 대해 movzbl 동작이 수행되는 것으로 보아, %rax에 저장된 string과 %rdx의 변화가 관련이 있을 것이라고 생각했다. 그래서 해당 movzbl 동작 뒤에 있으며, 루프 안에 있는 구문인 <phase\_5+39>에 breakpoint를 걸고, 계속해서 루프를 돌게 만들어 %rdx의 변화를 관찰했다 아까와 동일하게 루프를 돌 때마다 string의 앞 글자가 사라지며, %rdx는 그에 따라 11, 9, 12, 12, 5, 4로 계속해서 변했다.

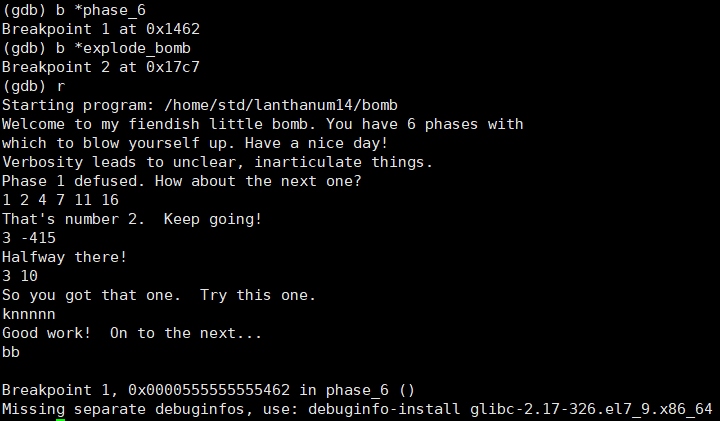
 

movzbl 동작은 (%rax)와 %rdx에 대해 수행된 것이다. 즉, %rax에 저장된 string 중 가장 앞에 있는 element와 상호작용한 것임을 알 수 있으며, 이것에 관련지어 생각해보면 %rdx는 %rax에 저장된 string의 가장 앞에 있는 알파벳의 순서를 10진수로 나타낸 것을 저장하고 있는 것이라고 해석할 수 있다.

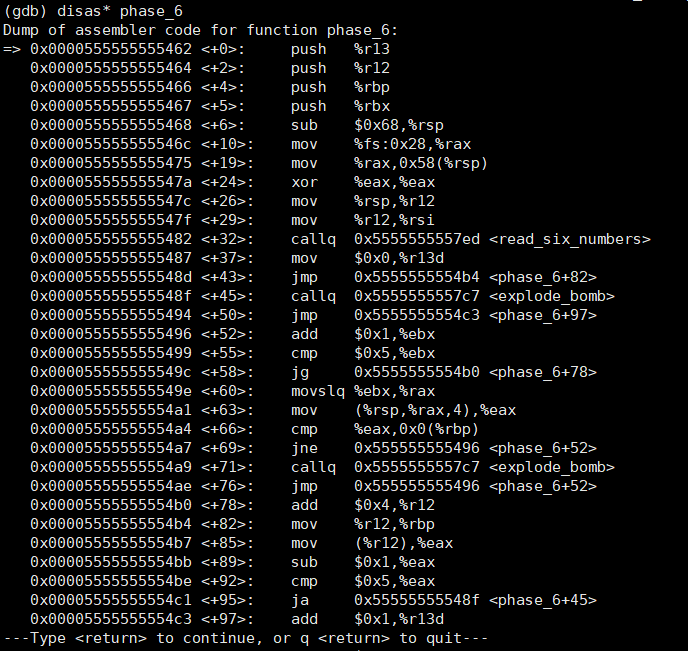
* + - 이제 %rdx와 %rsi로 %ecx를 계산하는 과정을 모두 분석했다. %ecx는 (%rsi + 4\*%rdx)의 누적합이므로, %rsi 배열 속의 적당한 값 6개를(중복 가능) 더해 80이 되도록 만들고, 그 수들을 가리키도록 만들 수 있는 적당한 %rdx를 찾는 순서로 암호를 풀어야 한다.
    - 이 방법을 이용해 생각해보면, 배열 속 숫자들 중 15와 5를 이용해 15\*5+5 계산을 하면 80을 만들 수 있다. 이때 15는 <array.3415+56>이고, 5는 <array.3415+44>이므로, 14번째 알파벳 5개와 11번째 알파벳 1개를 조합한 string이 해답이 될 수 있다.
    - 정리하자면, n 5개와 k하나로 이루어진 string이 해답이 될 수 있다는 것이다. 그래서 해당 조건을 만족하는 ‘knnnnn’을 입력해 보았고, phase\_5가 defuse된 것을 확인했다. 이로서 phase\_5를 해결했다.

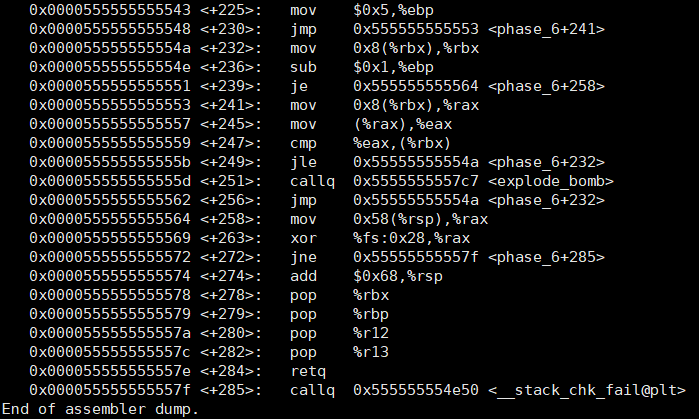
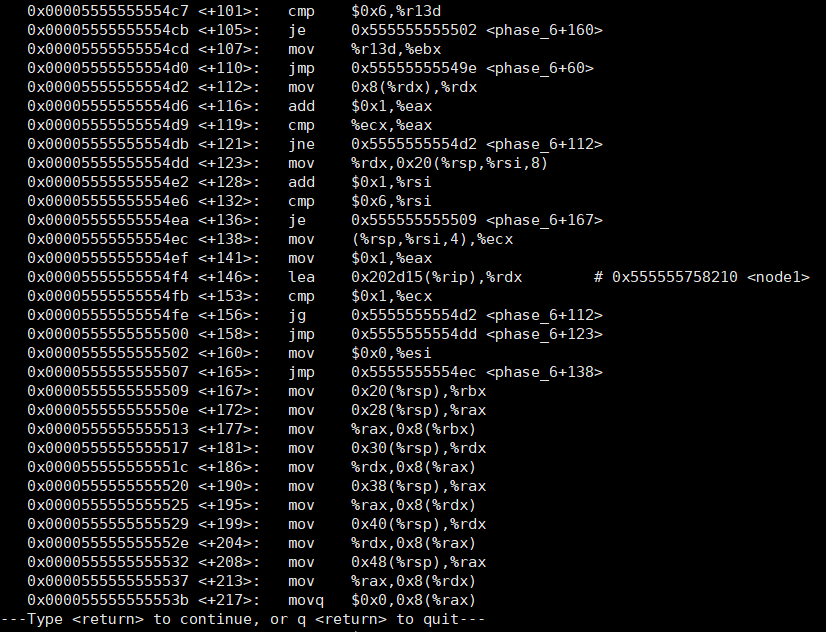


* + **Phase\_6**
    - 위에서 했던 것과 같이, ‘b\* phase\_6’, ‘b\* explode\_bomb’ 명령어를 이용해 breakpoint를 만든 후 ‘r’ 명령어로 프로그램을 실행했다.

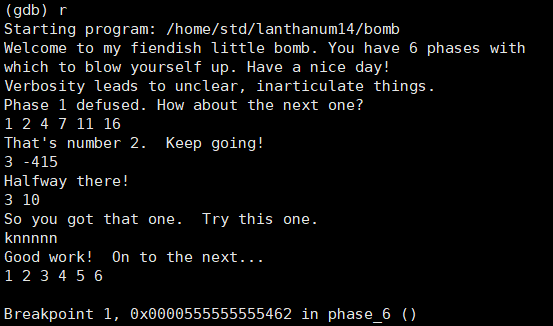


* + - phase\_6에 어떤 instruction들이 있는지 살펴보기 위해 ‘disas \*phase\_6’를 입력했다.

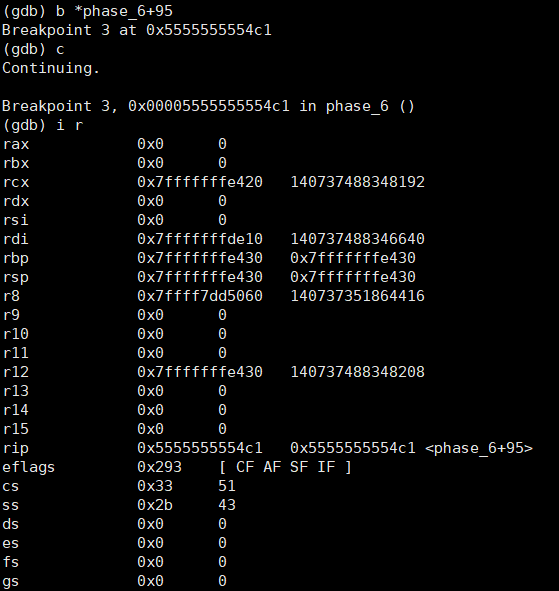




* + - <phase\_6+32>에서 “read\_six\_numbers”를 부르는 것으로 보아, 이번에도 6개의 숫자를 입력해야 한다는 것을 알 수 있었다. 따라서 1 2 3 4 5 6을 재입력해 프로그램을 재실행했다.



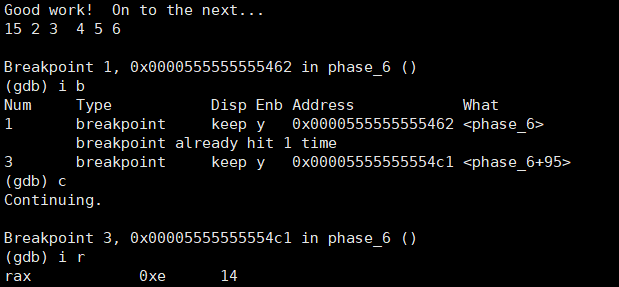
* + - ‘read\_six\_numbers’ 함수가 호출된 뒤 instruction들을 분석하면, %r13d로 0을 옮긴 후 <phase\_6+82>로 unconditional jump를 한다는 것을 알 수 있었다. 점프를 한 후에는 %rbp에 %r12를 옮기고, %eax에 (%r12)를 옮기고, %eax에서 1을 빼는 과정이 이어진다는 것을 알 수 있었다. 그런 후 %eax와 $0x5 사이에 비교가 일어남을 볼 수 있었는데, 이때 %eax가 5보다 above이면 explode\_bomb이 실행되는 것을 볼 수 있었다. 즉 여기에서 %eax는 5보다 작거나 같아야 하며, 이 조건에 대해 알아보기 위해 <phase\_6+95>를 breakpoint로 지정하고 ‘c’ 명령어로 breakpoint까지 프로그램을 실행해 %eax의 값을 알아보았다.



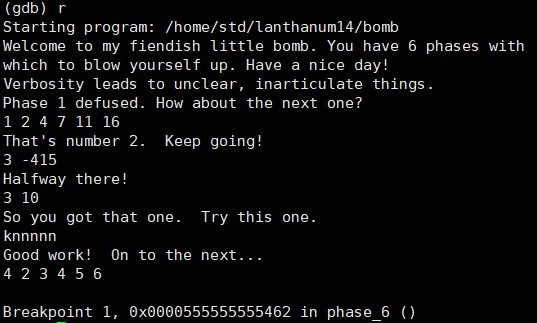
이 breakpoint에서 %eax는 0인 것을 확인할 수 있다. 이때 <phase\_6+95>에 이르기까지의 과정에서 %eax의 값에 명시적인 변화가 생긴 것은 (%r12)가 옮겨지고 1이 빼진 것뿐이다. 즉, 1이 빼지기 전에 %eax는 1이었다는 소리이며, %r12도 1 값을 갖는다는 소리가 된다.



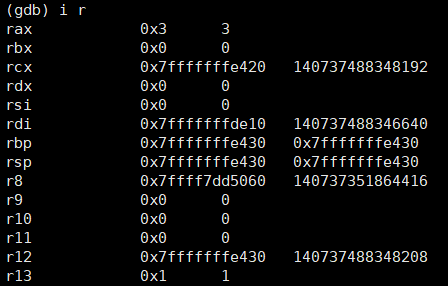
지금까지 입력해 준 수들을 보면 의심되는 것은 첫 번째 입력해준 수이다. 첫 번째 입력을 15로 바꾸어 정말 해당 값이 %eax에 저장되는 것인지 알아보았다.

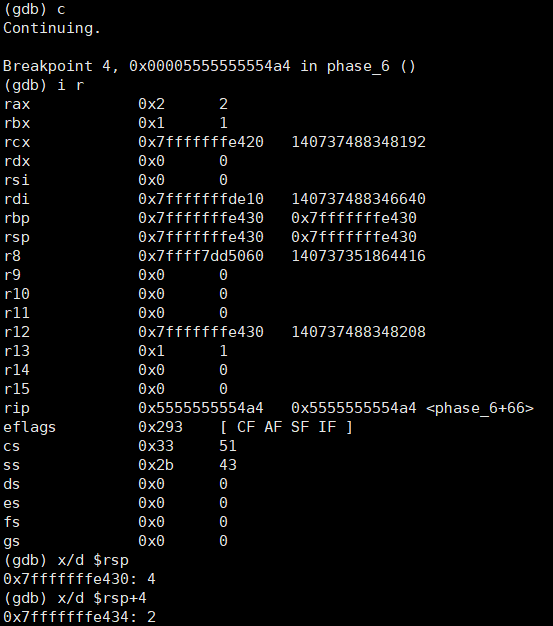


<phase\_6+89>에 의해 1이 빼진 14가 %rax에 저장되어 있는 것으로 미루어 보아, %rax에는 첫 번째 입력한 값이 저장된다는 것을 알 수 있다. 그리고 이를 통해서 첫 번째 입력하는 값은 7 이하여야 함을 알 수 있다. (1이 빼지는 연산 동작이 중간에 있으므로 7을 입력하더라도 6이 되어 explode\_bomb이 실행되지 않는다.) 따라서 다음 동작들이 실행될 수 있도록 프로그램을 재실행한 후 첫 번째 값을 4로 바꾸어 넣었다.



* + - 이제 첫 번째 값의 조건은 넘겼으므로 다음 동작들을 살펴봐야 한다. ja 동작이 일어나지 않는다면 그 다음 동작은 <phase\_6+97>부터이며, 여기서는 %r13d에 1을 더하고 그것을 $0x6과 비교하는 작업이 이루어진다. 그런데 지금까지의 동작에서 %r13d는 <phase\_6+37>에서 0으로 바뀐 후 어떤 변동도 없었다. 그리고 <phase\_6+97>에서 1을 더하게 되므로 %r13d의 값은 무조건 1일 것이며, 따라서 <phase\_6+105>의 je 구문은 절대 실행되지 않는다. 실제 ‘I r’ 명령어를 통해 확인한 결과도 이와 같았다.



* + - 따라서 <phase\_6+107>부터의 동작이 재개될 것이며, 여기서는 %ebx에 %r13d(=1)이 옮겨지는 동작이 일어났다. 그런 후 <phase\_6+60>으로 unconditional jump가 일어났다. 여기에서는 다시 %ebx를 %rax로 옮겼으며, %eax에 (%rsp + 4\*%rax)가 옮겨졌다. 이때 %ebx는 1이므로 movslq 동작이 일어날 때 %rax에는 1이 들어가며, 그 다음 동작은 즉 %rsp + 4를 %eax에 넣는 것과 동일하다는 것을 알 수 있다. 이때 ‘I r’ 명령어로 값을 확인해보면 %eax에는 2가 들어가있다. 즉 %rsp+4는 두 번째 입력된 값이라는 소리이다. 직접 값을 확인해보면 %rsp에는 4가, %rsp+4에는 2가 들어가 있음을 확인할 수 있다.
    - 
    - 그런 다음 <phase\_6+66>에서는 바로 %eax와 %rbp의 비교가 일어난다. 그리고 jne 밑에 바로 explode\_bomb 호출 구문이 있는 것으로 보아, %eax와 %rbp가 같지 않아야 한다는 것을 알 수 있다. 이때 %rbp 값을 읽어보면 다음과 같다.
    - 
    - 이는 첫 번째 입력한 값이거나 네 번째 입력한 값일 수 있다. 이를 확인하기 위해 각 숫자를 겹치지 않게 ‘3 5 7 9 11 13’으로 바꾸어 입력한 결과, %rbp 값이 아래와 같이 3으로 나타났다. 즉, %rbp는 첫 번째 입력한 값임을 알 수 있으며, 이 instruction들을 통해 두 번째 입력한 값은 첫 번째 입력한 값과 같아서는 안된다는 것을 알 수 있었다. 새로 입력한 ‘3 5 7 9 11 13’은 지금까지 나온 모든 조건을 만족하므로, 그 다음 조건 분석을 진행했다.



* + - 위에서 %eax와 %rbp가 같지 않음을 확인했으므로, 다음 구문은 <phase\_6+52>가 된다. 여기에서는 %ebx에 1을 더하고 그것을 $0x5와 비교하는 작업이 일어났다. 이때 아까 %ebx가 1이 된 이후로 %ebx에 수정이 일어나지 않았으므로, %ebx에 1을 더하는 동작이 일어나면 %ebx는 무조건 2가 된다. ‘I r’ 명령어로 값을 확인해본 결과 정말 2 값을 갖는다는 것을 알 수 있었다.

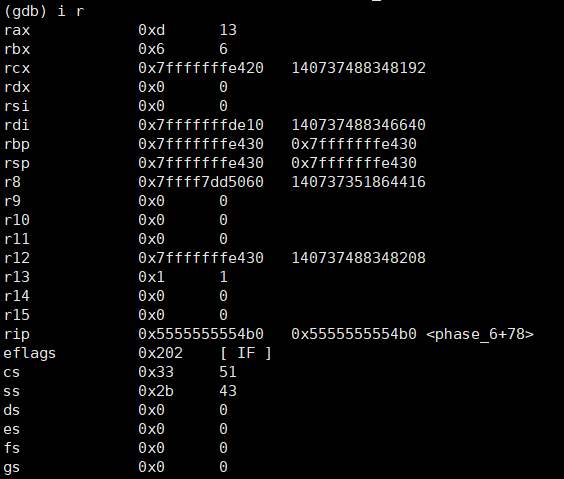


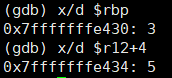
그러면 다음 동작은 <phase\_6+60>, 즉 movslq 동작이 될 것이다. 여기서는 %ebx 값이 %rax로 옮겨지며, %eax에 (%rsp + 4\*%rax) 값이 대입됨을 확인할 수 있었다. 이것은 이전 단계와 동일한 form으로, 대신 %ebx의 값이 2였으므로 %rax에는 최종적으로 %rsp+8이 저장되게 된다. 즉, 세 번째로 입력받은 값이 저장될 것임을 알 수 있다.



이 동작들 이후에는 위에서와 같이 이어서 %eax와 %rbp의 비교 동작이 일어나고, 둘의 값이 같지 않다면 <phase\_6+52>로 돌아가 %ebx의 값을 1 올려 동일한 동작을 반복하는 것을 볼 수 있었다. 즉 이것은 %ebx의 값을 1씩 증가시켜 (%ebx+1)번째로 입력한 값과 첫 번째 입력한 값을 비교하는 반복 구문임을 알 수 있다. 그리고 이들 중 하나라도 첫 번째 입력한 값과 같다면 explode\_bomb이 실행된다. 즉, 2~6번째에 입력하는 값들은 모두 첫 번째 입력한 값과 같으면 안 된다는 사실을 알 수 있다.

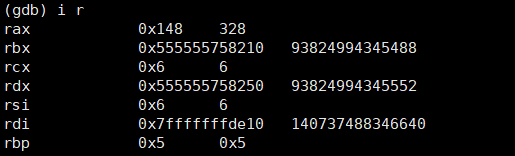
* + - 이 루프를 빠져나오는 방법은 %ebx가 5 초과가 되는 것이다. <phase\_6+58>에 의해 %ebx가 5보다 크면 <phase\_6+78>으로 jump하게 되어 있다. 즉 6번째 입력한 수까지 첫 번째 입력한 수와 비교를 끝냈으면 루프를 빠져나올 수 있는 것이다. 바로 다음 동작을 이어서 살펴보면, %r12에 4를 더하고 그것을 %rbp로 옮기고 있음을 볼 수 있다.





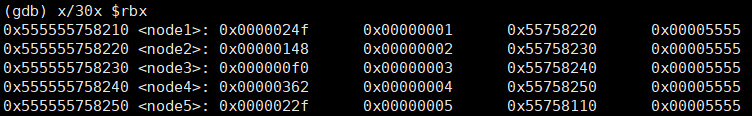
이때 각 레지스터의 값을 확인하면, <phase\_6+78> ~ <phase\_6+82> 구문을 실행한 결과는 두 번째 입력한 값을 %rbp에 넣는 것임을 알 수 있다. 그리고 이어서 (%r12)를 %eax로 옮기고 1을 뺀 뒤, %eax를 5와 비교하는 동작이 이어진다. 이때 %eax는 두 번째 입력하는 값이 들어가게 되며, 이것이 5보다 클 경우 explode\_bomb이 실행된다. 따라서 두 번째 입력하는 값은 6 이하여야 함을 알 수 있다. (1이 빼지므로 6을 입력해도 5가 된다.)

* + - 입력한 값으로는 두 번째 입력값의 조건을 지킬 수 있으므로, 바로 다음 동작을 분석했다. <phase\_6+97>부터는 %r13d에 1을 더하고 그것을 6과 비교하는 작업이 이루어졌다. 둘의 값이 같지 않으면 <phase\_6+60>으로 jump가 일어나며, 이곳에서 또 루프를 돌게 됨을 알 수 있다. 이번 루프는 <phase\_6+97~101> -> <phase\_6+52~69> -> <phase\_6+78~92>를 거치는 큰 루프이다. 이 루프도 2~6번째 정수를 검사하는 것이 목적이며, %rbp를 계속해서 다음에 입력한 정수값으로 바꾸어가며 하나씩 비교한다. 즉 입력한 6개의 정수 중 그 어떤 것도 같아서는 안된다는 조건이 있는 것이다. 그리고 <phase\_6+79~92>가 루프에 포함되므로, 앞서 언급한 조건에 더해 3~6번째 정수들도 6 이하여야 한다는 조건이 달리게 된다.
    - 다시 말해, 이번에 입력해야 할 6개의 정수는 모두 6 이하이며 서로 같아서는 안 되는 것이다. 이에 프로그램을 재실행해 ‘1 2 3 4 5 6’을 넣었다.
    - 남은 동작들을 살펴보면, 6번째 입력한 값까지 비교를 끝냈을 경우, <phase\_6+105>의 je 구문을 통해 <phase\_6+160>으로 jump하게 된다. 이곳에서는 %esi에 0을 넣고 <phase\_6+138>으로 unconditional jump를 하게 된다. 해당 자리에서는 %ecx에 (%rsp + 4\*%rsi)를 넣고, %eax에 1을 넣게 된다. 이때 %rsi는 0이므로 우리는 %ecx에 %rsp가 저장될 것임을 알 수 있다. 그리고 %ecx가 1보다 크면 <phase\_6+112>로 jump하게 된다. 이때 조건을 고려해 재입력해준 값을 보면 1과 같으므로 <phase\_6+123>으로 jump하게 된다. 이곳에서 횟수 카운팅 계산 동작 등을 진행한 후 두 번째 입력값을 1과 비교하게 되고, 두 번째 입력값이 1보다 크므로 <phase\_6+112>로 jump하게 된다.
    - <phase\_6+112>에서도 %rdx에 담긴 값을 0x20(%rsp, %rsi, 8)에 담는 등의 동작이 이어지며, 횟수 카운팅 계산 동작이 일어난다. 이것 역시 입력받은 값과 1을 비교하며 총 6번 반복하게 되는 루프이다. 이 역시 횟수를 세던 %rsi가 6이 되면 루프를 빠져 나오도록 설계되어 있으며, 루프를 빠져나온 이후에는 <phase\_6+167>로 jump하며, 이곳에서 여러 값 이동을 거쳐 <phase\_6+247>에서 (%rbx)와 %eax를 비교하게 된다. 그리고 여기에서 (%rbx)가 %eax보다 작거나 같아야 폭탄이 터지지 않음을 알 수 있다.
    - 위에서 (%rbx)가 %eax보다 작거나 같아야 함을 만족했으므로 <phase\_6+232~247>까지 작은 루프가 형성됨을 알 수 있다. 이 루프는 최초 0x5로 설정되었던 %ebp가 0이 되면 끝난다는 것을 알 수 있다. 즉 이 루프 또한 6번을 돌아 종료되는 루프인 것이다.
    - 여기까지 프로그램 정상 종료를 위한 조건을 분석했다. 그리고 이 조건을 만족하기 위해서는 (%rbx) <= %eax를 만족해야 함을 알 수 있었는데, 이들의 값을 알기 위해서는 앞서 <phase\_6+167~217>에 걸쳐 나타난 값의 이동을 이해할 필요가 있다. 우선 <phase\_6+247> 시점에서 각 레지스터의 값을 알아보면 다음과 같다.





이때 (%rbx) > %eax이므로 지금 입력한 값은 phase\_6의 해답이 될 수 없다는 것을 알았다. 그렇다면 그 이유를 분석해야 한다. 우선 위에서 %rbx를 알아보기 위해 입력한 명령어에서 <node1>이라는 구문을 볼 수 있었다. 이것을 좀 더 자세히 살펴볼 필요가 있을 것 같아 위에서 다른 phase를 해결할 때 array를 examine했던 것처럼 %rbx를 펼쳐보았다.

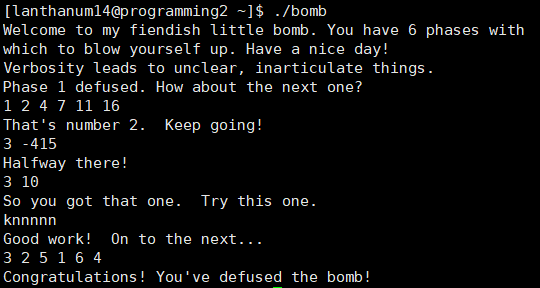


펼쳐본 결과, ‘node’라는 이름에서 짐작할 수 있었듯이 이는 linked list임을 알 수 있다.

* + - 위에서 밝혀낸 linked list를 좀 더 면밀히 살펴볼 필요가 있다. 우선 뒤에서 두 번째까지는 다음 node를 가리키는 주소가 저장되어 있는 것임을 알 수 있다. node\* next와 같은 역할을 하는 것이다. 따라서 우리에게 중요한 것은 앞의 두 개이다. 두 번째 값은 node의 숫자를 따라 증가하고 있음을 알 수 있다. 그리고 맨 첫 번째 값은 <phase\_6+167~217>에 걸쳐 계속해서 저장되고 이동했던 값들임을 짐작할 수 있다. 또한 node5의 세 번째, 네 번째 값을 이용해 node6을 찾아낼 수 있었다. <phase\_6+167~217>에 걸쳐 %rbx에 이 값들이 저장되었음을 알 수 있다.



* + - %rbx의 구조와 각 node들의 값을 염두에 두고 다시 <phase\_6+232>를 살펴보면, %rbx가 루프를 돌 때마다 계속 다음 노드를 가리키며, %rax는 %rbx의 다음 노드를 가리키게 됨을 알 수 있다. 그리고 %eax에는 이 %rax로부터 얻어낸 각 노드의 값이 담기는 것이다. 즉, <phase\_6+247>은 어떤 노드와 그 다음 노드의 값을 비교하는 과정임을 알 수 있다.
    - 이렇게 instruction을 해석해 낸 것과, 앞서 밝혀냈던 조건을 합치면 다음과 같을 것이다. 앞서 <phase\_6+232~247> 루프를 도는 동안 (%rbx)는 %eax보다 작거나 같아야 한다는 조건을 알아냈었다. 따라서 1 ~ 6까지의 노드의 순서를 적절히 배치해 항상 앞 노드의 값이 뒤의 노드보다 작도록 만들어야 한다. 위에서 알아낸 노드들의 값을 이용해 값을 오름차순으로 정렬하면 3(240) – 2(328) – 5(559) – 1(591) – 6(667) – 4(866)이다. 괄호 안의 숫자는 편의를 위해 각 node의 값을 10진수로 바꾸어 나타낸 것이다.
    - 정리하자면, phase\_6의 해답은 3 2 5 1 6 4임을 알 수 있다. 그리고 아래와 같이 직접 값을 넣어 phase\_6이 defuse됨을 확인하였다.



1. **Lab03에서 배운 점**

* Lab03을 해결하면서, gdb를 이용해 debugging하는 방법을 익힐 수 있었다. 다양한 명령어를 사용해 중지점을 만들거나 한 단계씩 실행하여 프로그램이 동작하는 과정을 세세하게 분석하는 방법을 익힐 수 있었다.
* 또한 high level language에서 지원되는 array, recursion, loop, linked list와 같은 data type들이 어셈블리어에서는 어떻게 나타나는지 잘 이해할 수 있었다. 이미 학습한 각 data type들의 특성을 이용해 그것이 어셈블리어로 표현된 과정을 이해하고 응용해 bomb을 defuse할 수 있었다.