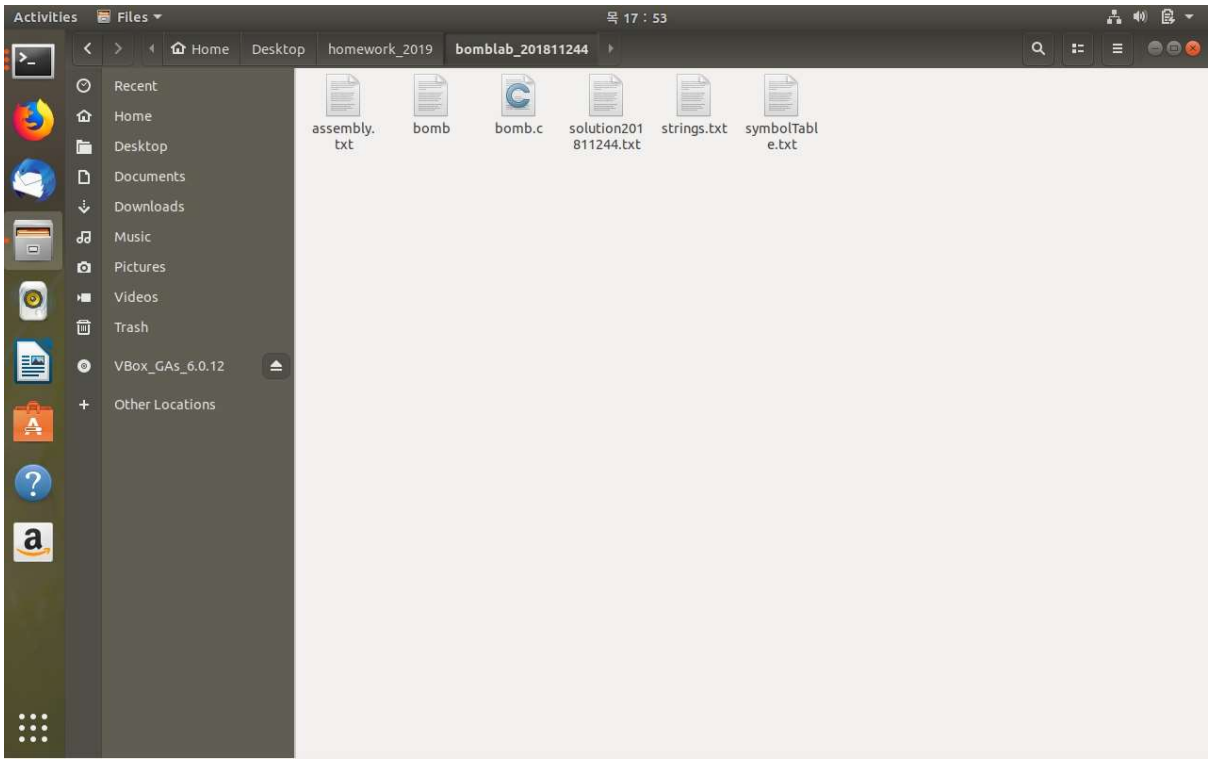
201811244 컴퓨터공학과 김창엽

— How to 폭탄해체 —

(Secret_Phase X)

Step 1.



필요한 준비물:

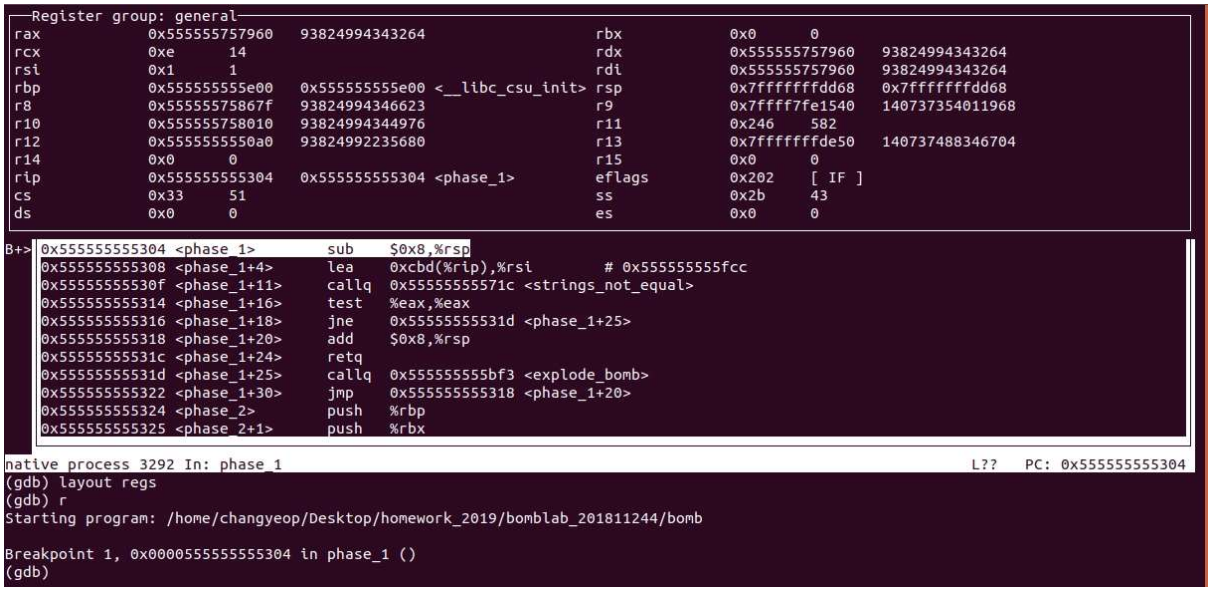
Bomb의 어셈블리 파일 (objdump -d bomb)

Bomb에 등장하는 출력가능한 문자열들 (strings)

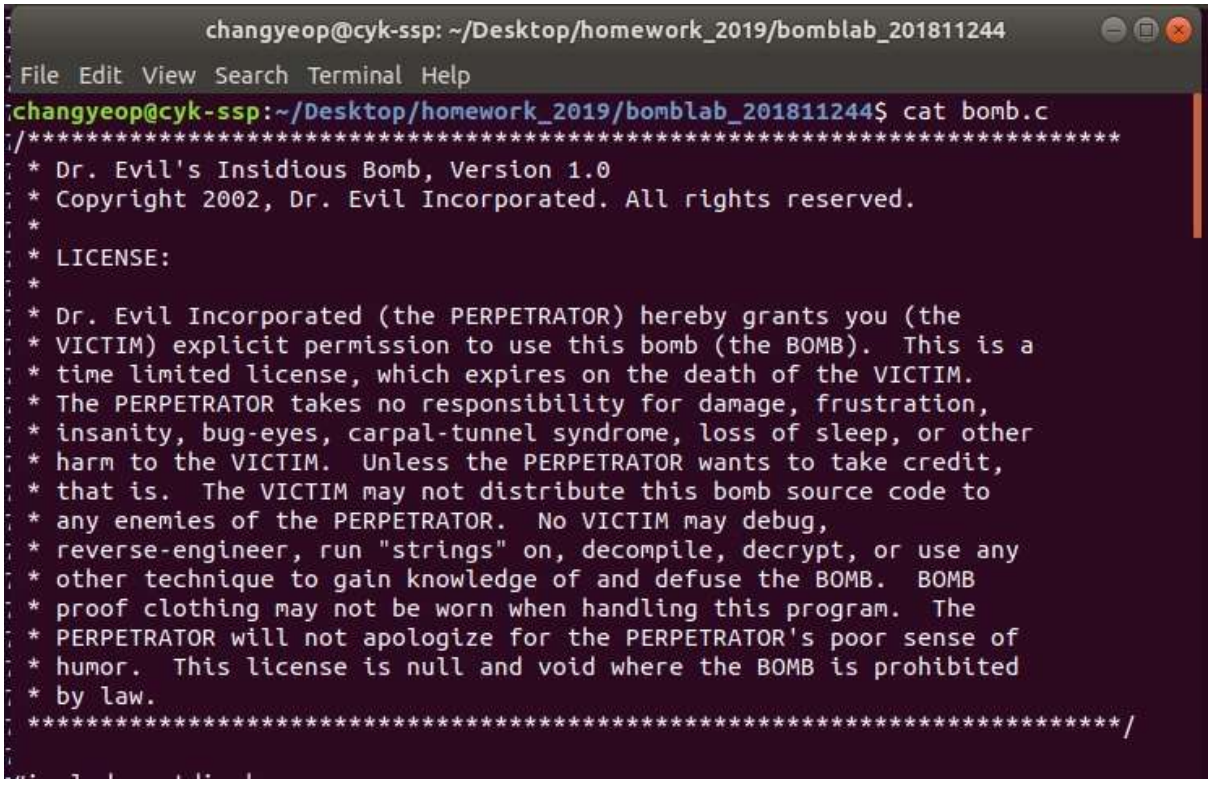
Bomb에 등장하는 함수들과 그 주소 (symbol table, objdump -t bomb)

GDB

사실상 위에 강조된 두 도구만으로도 폭탄을 해체하는 것이 가능하다.



참고로 **layout split, layout regs**란 명령어를 gdb상에서 입력하면 이렇게 편하게 레지스터들의 값들과 현재 내가 어느 라인을 읽고 있는지 알 수 있다.



우선 cat bomb.c로 내용을 확인해보았다. 대충 ***Dr. Evil***의 경고문 같은 것이었다. 밑에 내용에도 별게 없으니 패스.

1단계부터 차근차근 해체해보자. b phase_1으로 1단계에 breakpoint를 걸어야 한다.

===1단계===

```
0000000000001304 <phase_1>:
    1304:       48 83 ec 08             sub    $0x8,%rsp
    1308:       48 8d 35 bd 0c 00 00    lea    0xcbd(%rip),%rsi        # 1fcc <_IO_stdin_used+0x14c>
    130f:       e8 08 04 00 00          callq  171c <strings_not_equal>
    1314:       85 c0                   test   %eax,%eax
    1316:       75 05                   jne    131d <phase_1+0x19>
    1318:       48 83 c4 08             add    $0x8,%rsp
    131c:       c3                      retq
    131d:       e8 d1 08 00 00          callq  1bf3 <explode_bomb>
    1322:       eb f4                   jmp    1318 <phase_1+0x14>
```

수업 시간에 봤던게 1304쪽에 있다. 미리 8바이트를 스택에 비워두고, lea로 주소를 %rsi에게 넘겨준다. 그리고 strings_not_equal 함수를 호출한다.

우선 **lea 0xcbd(%rip), %rsi** 부터 보자. D(R) 형태, 저 주소값은 0xcbd + %rip 가 될 것이고 %rsi에는 저 주소값이 저장될 것이다. 그리곤 strings_not_equal 함수가 호출된다.

느낌상 %rsi 레지스터에 char* 형태로 선언된 문자열이 있을 것이다. 실제로 x/s $rsi로 확인해보면, %rsi 레지스터에 "qjxlftndjqtek."란 문자열이 저장되어 있는 것을 알 수 있다. 참고로 x/s 명령어는, %rsi 명령어에 있는 값(주소)에 있는 값을 문자열 형태(s)로 보여주는 것이다.

문자열을 비교하는 것이니, strings_not_equal에는 내가 입력한 문자열 (현재 %rdi – 함수의 첫 번째 인자를 받는 레지스터가 저장중)과 원래 정답 (%rsi에 있음) 을 비교하는 함수겠지?

저 함수가 호출될 때 사용되는 %rdi와 %rsi (첫 번째 인자 담당 레지스터와 두 번째 인자 담당 레지스터)에도 분명 값이 저장됐을 것이고 그 값들을 strings_not_equal에서 확인해보자.

```
000000000000171c <strings_not_equal>:
    171c:       41 54                   push   %r12
    171e:       55                      push   %rbp
    171f:       53                      push   %rbx
    1720:       48 89 fb                mov    %rdi,%rbx
    1723:       48 89 f5                mov    %rsi,%rbp
    1726:       e8 d4 ff ff ff          callq  16ff <string_length>
    172b:       41 89 c4                mov    %eax,%r12d
    172e:       48 89 ef                mov    %rbp,%rdi
    1731:       e8 c9 ff ff ff          callq  16ff <string_length>
    1736:       ba 01 00 00 00          mov    $0x1,%edx
    173b:       41 39 c4                cmp    %eax,%r12d
    173e:       74 07                   je     1747 <strings_not_equal+0x2b>
    1740:       89 d0                   mov    %edx,%eax
    1742:       5b                      pop    %rbx
    1743:       5d                      pop    %rbp
    1744:       41 5c                   pop    %r12
    1746:       c3                      retq
    1747:       0f b6 03                movzbl (%rbx),%eax
    174a:       84 c0                   test   %al,%al
    174c:       74 27                   je     1775 <strings_not_equal+0x59>
    174e:       3a 45 00                cmp    0x0(%rbp),%al
    1751:       75 29                   jne    177c <strings_not_equal+0x60>
    1753:       48 83 c3 01             add    $0x1,%rbx
    1757:       48 83 c5 01             add    $0x1,%rbp
    175b:       0f b6 03                movzbl (%rbx),%eax
    175e:       84 c0                   test   %al,%al
    1760:       74 0c                   je     176e <strings_not_equal+0x52>
    1762:       38 45 00                cmp    %al,0x0(%rbp)
    1765:       74 ec                   je     1753 <strings_not_equal+0x37>
    1767:       ba 01 00 00 00          mov    $0x1,%edx
    176c:       eb d2                   jmp    1740 <strings_not_equal+0x24>
    176e:       ba 00 00 00 00          mov    $0x0,%edx
    1773:       eb cb                   jmp    1740 <strings_not_equal+0x24>
    1775:       ba 00 00 00 00          mov    $0x0,%edx
    177a:       eb c4                   jmp    1740 <strings_not_equal+0x24>
    177c:       ba 01 00 00 00          mov    $0x1,%edx
    1781:       eb bd                   jmp    1740 <strings_not_equal+0x24>
```

함수의 인자 두 개를 %rbx, %rbp에 할당받고, 두 문자열 각각의 길이를 비교한다. 길이가 같다면, 1747 로 jump 한다. 일단 phase_1은 원래 갖고 있는 문자열과, 내가 입력한 문자열 두 개를 비교하니 레지스터의 어느 곳에 정답을 갖고 있는 것은 분명하다. gdb로 저 값을 찾아내자.

strings_not_equal 함수에서 첫 번째 인자와 두 번째 인자를 각각 %rbx, %rbp에 넣었는데, 당연히 이 함수는 내가 입력한 문자열과 정답을 인자로 받을 것이다. 이 레지스터 값들을 x/s로 메모리의 값을 확인해서, 그 주소의 값을 문자열로 출력했다. .

```
Reading symbols from bomb...done.
(gdb) b phase_1
Breakpoint 1 at 0x1304
(gdb) r
Starting program: /home/changyeop/Desktop/homework_2019/bomblab_201811244/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
string

Breakpoint 1, 0x0000555555555304 in phase_1 ()
(gdb) si
0x0000555555555308 in phase_1 ()
(gdb) si
0x000055555555530f in phase_1 ()
(gdb) si
0x000055555555571c in strings_not_equal ()
(gdb) si
0x000055555555571e in strings_not_equal ()
(gdb) si
0x000055555555571f in strings_not_equal ()
(gdb) x/s $rsi
0x555555555fcc: "qjxlftndjqtek."
(gdb) x/s $rdi
0x555555757960 <input_strings>: "string"
```

난 string을 먼저 답으로 넣어봤고, strings_not_equal 함수의 두 인자의 값을 확인했다. 함수의 첫 번째 인자로는 내가 입력한 string이 있었고, 두 번째 인자로는 내가 맞춰야 하는 문자열인 "qjxlftndjqtek. "가 있었다. 정답은 "qjxlftndjqtek." (버틸수없다.) 가 확실하다.

===2단계===

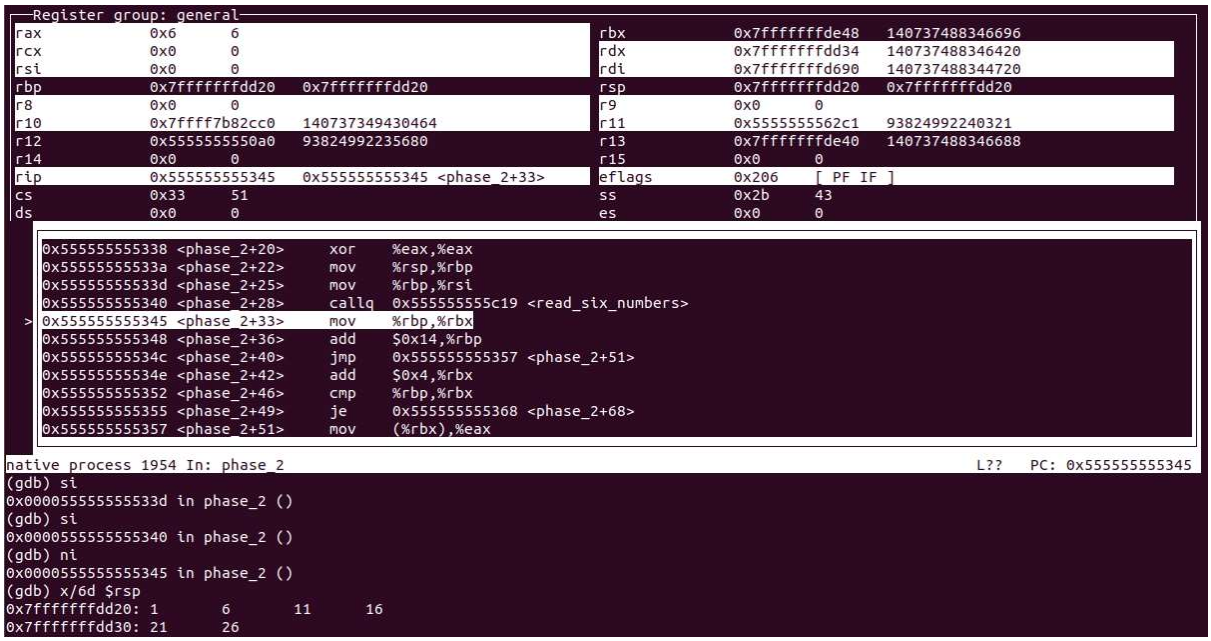
```
0000000000001324 <phase_2>:
    1324:       55                      push   %rbp
    1325:       53                      push   %rbx
    1326:       48 83 ec 28             sub    $0x28,%rsp
    132a:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    1331:       00 00
    1333:       48 89 44 24 18          mov    %rax,0x18(%rsp)
    1338:       31 c0                   xor    %eax,%eax
    133a:       48 89 e5                mov    %rsp,%rbp
    133d:       48 89 ee                mov    %rbp,%rsi
    1340:       e8 d4 08 00 00          callq  1c19 <read_six_numbers>
    1345:       48 89 eb                mov    %rbp,%rbx
    1348:       48 83 c5 14             add    $0x14,%rbp
    134c:       eb 09                   jmp    1357 <phase_2+0x33>
    134e:       48 83 c3 04             add    $0x4,%rbx
    1352:       48 39 eb                cmp    %rbp,%rbx
    1355:       74 11                   je     1368 <phase_2+0x44>
    1357:       8b 03                   mov    (%rbx),%eax
    1359:       83 c0 05                add    $0x5,%eax
    135c:       39 43 04                cmp    %eax,0x4(%rbx)
    135f:       74 ed                   je     134e <phase_2+0x2a>
    1361:       e8 8d 08 00 00          callq  1bf3 <explode_bomb>
    1366:       eb e6                   jmp    134e <phase_2+0x2a>
    1368:       48 8b 44 24 18          mov    0x18(%rsp),%rax
    136d:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
    1374:       00 00
    1376:       75 07                   jne    137f <phase_2+0x5b>
    1378:       48 83 c4 28             add    $0x28,%rsp
    137c:       5b                      pop    %rbx
    137d:       5d                      pop    %rbp
    137e:       c3                      retq
    137f:       e8 6c fb ff ff          callq  ef0 <__stack_chk_fail@plt>
```

처음부터 천천히 뜯어보자. 스택에 %rbp 레지스터와 %rbx 레지스터를 push 해주고, 그리고 스택에 40비트만큼 할당해주었다. (%rsp -= 40)

문제와 관련 없는 mov %fs:0x28 부분은 제쳐두자. (stack-guard value)

대충 이 phase_2 함수는 6개의 숫자를 읽고 (read_six_numbers) 무언가를 한다!

read_six_numbers 호출 후를 잘 보면, %rsp (%rbp) 위로 내가 입력했던 6개의 숫자들이 차례대로 쌓인다.



그러곤 %rbp의 값(%rsp의 주소)를 %rbx에게 넘겨줬으니, %rbx도 역시 내가 입력한 6개의 숫자들의 시작 주소를 가지고 있다.

이후에 %eax 레지스터를 사용해서, 숫자에 5를 더해 그 다음 숫자와 같은지를 계속 비교하고 있다. 바로 내가 입력한 6개의 숫자들이 공차가 5인 등차수열인지를 반복문을 돌면서 체크하고 있다. 따라서 내가 입력해야될 6개의 숫자들은 그저 공차가 5인 등차수열이기만 하면 된다.

1, 6, 11, 16, 21, 26 / 2, 7, 12, 17, 22, 27 등등 정답은 다양하다!

2단계 clear!

=== 3 단계 ===

```
0000000000001384 <phase_3>:
    1384:       48 83 ec 18             sub    $0x18,%rsp
    1388:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    138f:       00 00
    1391:       48 89 44 24 08          mov    %rax,0x8(%rsp)
    1396:       31 c0                   xor    %eax,%eax
    1398:       48 8d 4c 24 04          lea    0x4(%rsp),%rcx
    139d:       48 89 e2                mov    %rsp,%rdx
    13a0:       48 8d 35 15 0f 00 00    lea    0xf15(%rip),%rsi        # 22bc <array.3409+0x2ac>
    13a7:       e8 14 fc ff ff          callq  fc0 <__isoc99_sscanf@plt>
    13ac:       83 f8 01                cmp    $0x1,%eax
    13af:       7e 19                   jle    13ca <phase_3+0x46>
    13b1:       83 3c 24 07             cmpl   $0x7,(%rsp)
    13b5:       77 4b                   ja     1402 <phase_3+0x7e>
    13b7:       8b 04 24                mov    (%rsp),%eax
    13ba:       48 8d 15 2f 0c 00 00    lea    0xc2f(%rip),%rdx        # 1ff0 <_IO_stdin_used+0x170>
    13c1:       48 63 04 82             movslq (%rdx,%rax,4),%rax
    13c5:       48 01 d0                add    %rdx,%rax
    13c8:       ff e0                   jmpq   *%rax
    13ca:       e8 24 08 00 00          callq  1bf3 <explode_bomb>
    13cf:       eb e0                   jmp    13b1 <phase_3+0x2d>
    13d1:       b8 76 01 00 00          mov    $0x176,%eax
    13d6:       eb 3b                   jmp    1413 <phase_3+0x8f>
    13d8:       b8 30 02 00 00          mov    $0x230,%eax
    13dd:       eb 34                   jmp    1413 <phase_3+0x8f>
    13df:       b8 e9 00 00 00          mov    $0xe9,%eax
    13e4:       eb 2d                   jmp    1413 <phase_3+0x8f>
    13e6:       b8 a6 01 00 00          mov    $0x1a6,%eax
    13eb:       eb 26                   jmp    1413 <phase_3+0x8f>
    13ed:       b8 18 01 00 00          mov    $0x118,%eax
    13f2:       eb 1f                   jmp    1413 <phase_3+0x8f>
    13f4:       b8 d5 00 00 00          mov    $0xd5,%eax
    13f9:       eb 18                   jmp    1413 <phase_3+0x8f>
    13fb:       b8 02 02 00 00          mov    $0x202,%eax
    1400:       eb 11                   jmp    1413 <phase_3+0x8f>
    1402:       e8 ec 07 00 00          callq  1bf3 <explode_bomb>
    1407:       b8 00 00 00 00          mov    $0x0,%eax
    140c:       eb 05                   jmp    1413 <phase_3+0x8f>
```

딱봐도 Switch문이라는게 눈에 보이지 않는가? Jmpq *%rax가 switch(case), 그리고 밑에 있는 수많은 jmp와 mov들은 cases이라는 것을 딱 직감적으로 알아차릴 수 있다. 일단 이 함수의 입력은 무엇일까? 우선 저기 밑에 있는 scanf의 입력에 따라 그 후에 %eax의 값이 변하는데, 잘못된 데이터 형식이면 그 밑에 있는 **cmp $0x1, %eax** 를 통과할 수 없다. scanf후에 바뀌는 %eax의 값의 기준은, 입력 개수와 데이터형에 의해 좌우되는 것 같다. 물론 레지스터를 x/s로 뜯어서 "%d %d", 즉 정수 2개를 입력받는 것으로 확인할 수도 있다. 그래도 scanf 함수 호출 후 %eax의 값이 바뀐다는 것을 알면 왜 **cmp $0x1, %eax**이 있는지 대충 짐작할 수 있다.

정수 2개를 입력 후, 첫번째 입력한 숫자가 7이하인지 검사한다. 그리고 그 후, 이 줄에 집중해라.

movslq (%rdx, %rax, 4), %rax => %rax = *(%rdx + 4 * %rax)

참고로 %rax에는 내가 처음에 입력한 7 이하 숫자가 들어있다 (%rsp). 저 수식의 의미는, 내가 입력한 숫자 * 4 + %rdx에 있는 값을 다시 %rax에게 넣으라는 의미이다. 만일 내가 첫번째 숫자로 5를 입력했다면 *(%rdx + 20)의 값이 %rax에 들어갔을 것이다. slq니까 당연히 4바이트에서 8바이트로, Sign-extended하게 넣으라는 뜻이다.

저게 의미하는게 뭘까? 바로 Jump-table이다.

입력한 숫자에 따라서 특정 주소(cases) 로 점프를 하고, 숫자들과 비교를 한다!

0	1	2	3	4	5	6	7
79	374	560	233	422	280	213	514

0을 입력하면, %rax에는 case 0으로 가는 주소가 담기고, 79란 값 내가 두번째로 입력한 값과 비교를 한다!

즉, 정답은 저 테이블에 있는 (0, 79), (1, 374), ... (7, 514) 까지 다 된다!

3단계 clear

=== 4단계 ===

```
0000000000001454 <phase_4>:
    1454:       48 83 ec 18             sub    $0x18,%rsp
    1458:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    145f:       00 00
    1461:       48 89 44 24 08          mov    %rax,0x8(%rsp)
    1466:       31 c0                   xor    %eax,%eax
    1468:       48 8d 54 24 04          lea    0x4(%rsp),%rdx
    146d:       48 8d 35 4b 0e 00 00    lea    0xe4b(%rip),%rsi        # 22bf <array.3409+0x2af>
    1474:       e8 47 fb ff ff          callq  fc0 <__isoc99_sscanf@plt>
    1479:       83 f8 01                cmp    $0x1,%eax
    147c:       75 07                   jne    1485 <phase_4+0x31>
    147e:       83 7c 24 04 00          cmpl   $0x0,0x4(%rsp)
    1483:       7f 05                   jg     148a <phase_4+0x36>
    1485:       e8 69 07 00 00          callq  1bf3 <explode_bomb>
    148a:       8b 7c 24 04             mov    0x4(%rsp),%edi
    148e:       e8 a5 ff ff ff          callq  1438 <func4>
    1493:       3d 80 89 05 00          cmp    $0x58980,%eax
    1498:       74 05                   je     149f <phase_4+0x4b>
    149a:       e8 54 07 00 00          callq  1bf3 <explode_bomb>
    149f:       48 8b 44 24 08          mov    0x8(%rsp),%rax
    14a4:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
    14ab:       00 00
    14ad:       75 05                   jne    14b4 <phase_4+0x60>
    14af:       48 83 c4 18             add    $0x18,%rsp
    14b3:       c3                      retq
    14b4:       e8 37 fa ff ff          callq  ef0 <__stack_chk_fail@plt>
```

```
0000000000001438 <func4>:
    1438:       b8 01 00 00 00          mov    $0x1,%eax
    143d:       83 ff 01                cmp    $0x1,%edi
    1440:       7f 02                   jg     1444 <func4+0xc>
    1442:       f3 c3                   repz retq
    1444:       53                      push   %rbx
    1445:       89 fb                   mov    %edi,%ebx
    1447:       8d 7f ff                lea    -0x1(%rdi),%edi
    144a:       e8 e9 ff ff ff          callq  1438 <func4>
    144f:       0f af c3                imul   %ebx,%eax
    1452:       5b                      pop    %rbx
    1453:       c3                      retq
```

이번 단계는 재귀함수다. func4를 보면, 어떤 숫자를 받았을 때, 1이면 1을 리턴하고 1보다 크면 그 인자에서 1을 빼고 함수를 다시 부른다. 함수를 부르고 나서 **imul %ebx, %eax**로 리턴값들 하나씩 곱해준다. 즉, 팩토리얼 함수이다. 이 단계는 어떤 정수 (num이라 하자)를 입력받고, func4(num) == 362880이란 값과 비교한다. num! = 362880, 계산기를 사용해서 알아보면 9!의 값이 저 값인 줄 알 수 있다. 정답은 9. 매우 간단하다!

4단계 clear.

=== 5단계 ===

```
0000000000014b9 <phase_5>:
    14b9:       53                      push   %rbx
    14ba:       48 83 ec 10             sub    $0x10,%rsp
    14be:       48 89 fb                mov    %rdi,%rbx
    14c1:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    14c8:       00 00
    14ca:       48 89 44 24 08          mov    %rax,0x8(%rsp)
    14cf:       31 c0                   xor    %eax,%eax
    14d1:       e8 29 02 00 00          callq  16ff <string_length>
    14d6:       83 f8 06                cmp    $0x6,%eax
    14d9:       75 55                   jne    1530 <phase_5+0x77>
    14db:       b8 00 00 00 00          mov    $0x0,%eax
    14e0:       48 8d 0d 29 0b 00 00    lea    0xb29(%rip),%rcx        # 2010 <array.3409>
    14e7:       0f b6 14 03             movzbl (%rbx,%rax,1),%edx
    14eb:       83 e2 0f                and    $0xf,%edx
    14ee:       0f b6 14 11             movzbl (%rcx,%rdx,1),%edx
    14f2:       88 54 04 01             mov    %dl,0x1(%rsp,%rax,1)
    14f6:       48 83 c0 01             add    $0x1,%rax
    14fa:       48 83 f8 06             cmp    $0x6,%rax
    14fe:       75 e7                   jne    14e7 <phase_5+0x2e>
    1500:       c6 44 24 07 00          movb   $0x0,0x7(%rsp)
    1505:       48 8d 7c 24 01          lea    0x1(%rsp),%rdi
    150a:       48 8d 35 ca 0a 00 00    lea    0xaca(%rip),%rsi        # 1fdb <_IO_stdin_used+0x15b>
    1511:       e8 06 02 00 00          callq  171c <strings_not_equal>
    1516:       85 c0                   test   %eax,%eax
    1518:       75 1d                   jne    1537 <phase_5+0x7e>
    151a:       48 8b 44 24 08          mov    0x8(%rsp),%rax
    151f:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
    1526:       00 00
    1528:       75 14                   jne    153e <phase_5+0x85>
    152a:       48 83 c4 10             add    $0x10,%rsp
    152e:       5b                      pop    %rbx
    152f:       c3                      retq
    1530:       e8 be 06 00 00          callq  1bf3 <explode_bomb>
    1535:       eb a4                   jmp    14db <phase_5+0x22>
    1537:       e8 b7 06 00 00          callq  1bf3 <explode_bomb>
    153c:       eb dc                   jmp    151a <phase_5+0x61>
    153e:       e8 ad f9 ff ff          callq  ef0 <__stack_chk_fail@plt>
```

솔직히 이 단계를 풀면서 크게 감탄했었다.

일단 어떤 문자열을 입력받는데, 그 길이가 6인 문자열을 입력받는다. (**cmp $0x6, %eax**)

(참고: 내가 입력한 문자열들은 %rbx가 가리키는 주소에 있다)

그러곤 문자열의 각각 문자들을 $0xf와 and 시키고 (**and $0xf, %edx)**

%edx에 (%rcx + %rdx * 1)의 값을 넣는다. **(movzbl (%rcx, %rdx, 1), %edx)**

%rcx에 문자열이 하나 저장되어 있는데,

```
0x555555556010 <array.3409>:    "isrveawhobpnutfgWow! You've defused the secret stage!"
```

저 문자열 대로라면 %rcx에 저장되어 있는 %rdx번째 문자를 %edx에 저장하는 것이다.

"isrveawhobpnutfg"는 길이가 16인 문자열이고, 이 문자열의 문자가 %edx에 저장될 것이다. (뒤는 무시해도 좋다. 어느 값을 15와 and시키면 그 결과 값은 무조건 15이하일테니까.)

mov %dl, 0x1(%rsp, %rax, 1)로 *(%rsp + %rax * 1) 에 차곡차곡 쌓아놓는다.

이렇게 해서 모인 문자들(%rsp + 1에 차곡차곡 쌓여있다)을 %rsi에 저장되어 있는 주소에 있는 값 "giants"와 비교한다. (1단계 처럼 말이다).

즉, 내가 입력한 문자열의 문자들이 $0xf과 and시켰을 때 나오는 숫자들이 "isrveawhobpnutfg"에서 순서대로 g, i, a, n, t, s를 가리켜야 한다. 즉, 내가 입력한 문자열의 문자들이 $0xf과 and 시켰을 때 각각 15, 0, 5, 11, 13, 1이 나와야 하므로 어떤 문자를 0xf와 and 시켜서 저 숫자들이 나올 수 있을지 아스키코드 표를 참고하면서 알아내야한다.

15	0	5	11	13	1
g	i	a	n	t	s
o	p	e/u	k	m	a / q

opekma 나 opukmq 등의 문자열을 입력하면 정답이다. 다양한 정답이 나올 수 있다.

5단계 clear.

===6단계===

대망의 6단계. 좀 많이 어려웠었지만 일일이 어셈블리어를 계속 따라가서 레지스터 값들의 변화를 쭉 찾아본 끝에 알 수 있었다.

```
0000000000015a0 <phase_6>:
    15a0:       53                      push   %rbx
    15a1:       ba 0a 00 00 00          mov    $0xa,%edx
    15a6:       be 00 00 00 00          mov    $0x0,%esi
    15ab:       e8 f0 f9 ff ff          callq  fa0 <strtol@plt>
    15b0:       48 89 c3                mov    %rax,%rbx
    15b3:       48 8d 3d 66 20 20 00    lea    0x202066(%rip),%rdi        # 203620 <node1>
    15ba:       e8 84 ff ff ff          callq  1543 <fun6>
    15bf:       48 8b 40 08             mov    0x8(%rax),%rax
    15c3:       48 8b 40 08             mov    0x8(%rax),%rax
    15c7:       48 8b 40 08             mov    0x8(%rax),%rax
    15cb:       48 8b 40 08             mov    0x8(%rax),%rax
    15cf:       48 8b 40 08             mov    0x8(%rax),%rax
    15d3:       48 8b 40 08             mov    0x8(%rax),%rax
    15d7:       48 8b 40 08             mov    0x8(%rax),%rax
    15db:       39 18                   cmp    %ebx,(%rax)
    15dd:       74 05                   je     15e4 <phase_6+0x44>
    15df:       e8 0f 06 00 00          callq  1bf3 <explode_bomb>
    15e4:       5b                      pop    %rbx
    15e5:       c3                      retq
```

저기에 node가 보이는가? 맞다. 얘네들은 node다. struct로 만들어진 node이다. %rdi를 뜯어본 결과, node의 구성을 대략적으로 알 수 있었다.

```
(gdb) x/4wx $rdi
0x555555757620 <node1>: 0x000000ce      0x00000001      0x55757630      0x00005555
```

int data;	int num;	struct node* next;
4 bytes	4 bytes	8 bytes

```
struct node {
   int data;
   int num;
   struct node* next;
};
```

이런식으로 linked list가 9개의 node로 구성되어있었다.

fun6도 뜯어본 결과, %rcx, %r8, %rax, %rdx 주로 이 4개의 레지스터를 사용해서 bubble-sort를 사용해 내림차순으로 노드들을 정렬하는 것이었다. 만약 왼쪽 노드가 오른쪽 보다 작으면 swap을 따로 함수를 이용하지 않고 node의 next들을 바꿔가면서 했다.

node1	node2	node3	node4	node5	node6	node7	node8	node9
206	433	1000	913	287	990	72	506	608

(정렬 후)

node3	node6	node4	node9	node8	node2	node5	node1	node7
1000	960	913	608	506	433	287	206	72

```
0x5555555555ba <phase_6+26>     callq  0x555555555543 <fun6>
0x5555555555bf <phase_6+31>     mov    0x8(%rax),%rax
0x5555555555c3 <phase_6+35>     mov    0x8(%rax),%rax
0x5555555555c7 <phase_6+39>     mov    0x8(%rax),%rax
0x5555555555cb <phase_6+43>     mov    0x8(%rax),%rax
0x5555555555cf <phase_6+47>     mov    0x8(%rax),%rax
0x5555555555d3 <phase_6+51>     mov    0x8(%rax),%rax
0x5555555555d7 <phase_6+55>     mov    0x8(%rax),%rax
0x5555555555db <phase_6+59>     cmp    %ebx,(%rax)
```

정렬 후 linked list의 8번째 node (7번 넘어갔으니)의 값에 대해서 묻고 있으니 206을 입력해주면 무사히 폭탄은 해체된다.

+++ Tips +++

참고로 주의할 점은 node1~8이 놀랍게도 바로 메모리 상에서 딱 붙어 있으므로, x/x $rdi + 16하면 node2가 나온다는거..!

이게 바로 수업시간 때 나왔던 그 어처구니 없는 예다. 배열 여러 개 선언하다 보면 메모리상으로 딱 붙어서 나올 때가 은근히 있다는 것!

char a[4], char b[4]가 있다고 치면..

a[0]	a[1]	a[2]	a[3]	b[0]	b[1]	b[2]	b[3]
0x100	0x101	0x102	0x103	0x104	0x105	0x106	0x107

가끔 이렇게 메모리 상에 배치되니, *(a + 4) 하면 b[0]의 값이 나온다!

일일이 next의 주소로 봐야 헷갈리지 않는다! 그리고 또 웃긴 건 node9만 메모리상으로 따로 동떨어져있다는거! 그래서 쉽사리 놓치기 딱 좋다. node가 8까지만 있는 줄 알았다 (나도 이거 놓쳐서 이 문제에 하루를 더 날렸다!)

이렇게 6단계 clear...?

== 짧은 후기 ==

2단계에 2~3일, 3~4까지 일사천리로 풀다가 5단계에 7시간, 그리고 6에 2~3일. 정말 모든 걸 쏟아 부었다. 정말 어려운 과제였지만 그래도 되게 재미있었다. 근데 너무 어려워서 secret phase까지 풀지 못했으나 나중에 시간 나면 풀어야겠다.

+ 게시판에 많은 질문들이 올라와 있었는데 그 많은 질문들에 일일이 정성스레 답변해드린 멘토님들과 조교님들 수고하셨습니다!!

+ 전 이미 질문하려고 하는 것들이 게시판에 몇 개 있어서 답변을 참고했습니다!

+++ Secret Phase 들어가기 +++

어셈블리 코드를 뜯어보면, 분명 fun7과 secret_phase를 적어도 한 번쯤은 봤을 것이다. 그럼 이 함수를 call하는 부분도 '어딘가에는' 있을 것! 한 번 찾아보자.

```
0000000000001d5b <phase_defused>:
    1d5b:       48 83 ec 78             sub    $0x78,%rsp
    1d5f:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    1d66:       00 00
    1d68:       48 89 44 24 68          mov    %rax,0x68(%rsp)
    1d6d:       31 c0                   xor    %eax,%eax
    1d6f:       83 3d ca 1b 20 00 06    cmpl   $0x6,0x201bca(%rip)        # 203940 <num_input_strings>
    1d76:       74 15                   je     1d8d <phase_defused+0x32>
    1d78:       48 8b 44 24 68          mov    0x68(%rsp),%rax
    1d7d:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
    1d84:       00 00
    1d86:       75 6e                   jne    1df6 <phase_defused+0x9b>
    1d88:       48 83 c4 78             add    $0x78,%rsp
    1d8c:       c3                      retq
    1d8d:       48 8d 4c 24 10          lea    0x10(%rsp),%rcx
    1d92:       48 8d 54 24 0c          lea    0xc(%rsp),%rdx
    1d97:       48 8d 35 68 05 00 00    lea    0x568(%rip),%rsi        # 2306 <array.3409+0x2f6>
    1d9e:       48 8d 3d ab 1c 20 00    lea    0x201cab(%rip),%rdi        # 203a50 <input_strings+0xf0>
    1da5:       e8 16 f2 ff ff          callq  fc0 <__isoc99_sscanf@plt>
    1daa:       83 f8 02                cmp    $0x2,%eax
    1dad:       74 0e                   je     1dbd <phase_defused+0x62>
    1daf:       48 8d 3d 2a 03 00 00    lea    0x32a(%rip),%rdi        # 20e0 <array.3409+0xd0>
    1db6:       e8 15 f1 ff ff          callq  ed0 <puts@plt>
    1dbb:       eb bb                   jmp    1d78 <phase_defused+0x1d>
    1dbd:       48 8d 7c 24 10          lea    0x10(%rsp),%rdi
    1dc2:       48 8d 35 43 05 00 00    lea    0x543(%rip),%rsi        # 230c <array.3409+0x2fc>
    1dc9:       e8 4e f9 ff ff          callq  171c <strings_not_equal>
    1dce:       85 c0                   test   %eax,%eax
    1dd0:       75 dd                   jne    1daf <phase_defused+0x54>
    1dd2:       48 8d 3d a7 02 00 00    lea    0x2a7(%rip),%rdi        # 2080 <array.3409+0x70>
    1dd9:       e8 f2 f0 ff ff          callq  ed0 <puts@plt>
    1dde:       48 8d 3d c3 02 00 00    lea    0x2c3(%rip),%rdi        # 20a8 <array.3409+0x98>
    1de5:       e8 e6 f0 ff ff          callq  ed0 <puts@plt>
    1dea:       b8 00 00 00 00          mov    $0x0,%eax
    1def:       e8 31 f8 ff ff          callq  1625 <secret_phase>
    1df4:       eb b9                   jmp    1daf <phase_defused+0x54>
    1df6:       e8 f5 f0 ff ff          callq  ef0 <__stack_chk_fail@plt>
    1dfb:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

보이는가? phase_defused에 보면 secret_phase를 call하는 부분이 있다!

```
(gdb) x/s $rsi
0x555555556306: "%d %s"
```

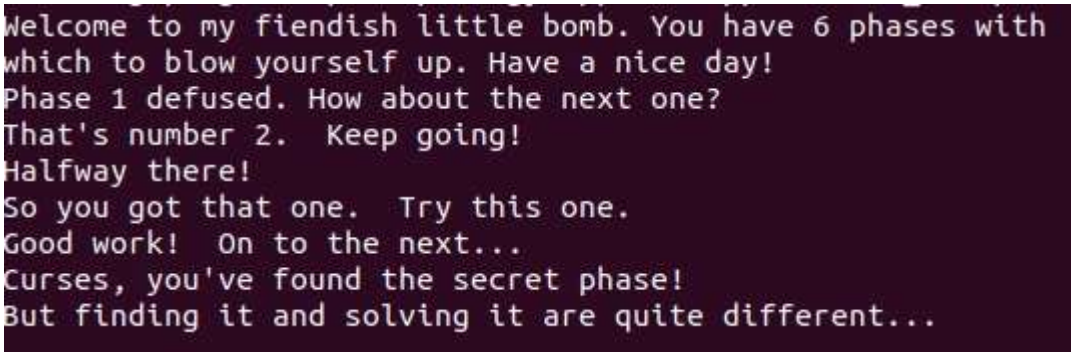
6단계가 끝나고 phase_defused에 보면 다시 한 번 4단계의 입력을 확인한다. 이 때

cmp $0x2, %eax로 인자를 확인하는데, 저기서 내가 뒤에 문자열을 입력하지 않았으면 %eax = 1 이고 거기서 축하메세지를 띄우며 폭탄은 해체되고 secret_phase로 진입하지 못할 것이다.

아무 문자열이나 4단계의 입력에 넣고 다시 확인해보면...

```
(gdb) x/s $rsi
0x55555555630c: "austinpowers"
```

내가 입력해야 할 값은 "austinpowers"이다.



이렇게 secret phase로 들어갈 수 있다!

많이 부족한 풀이 봐주셔서 감사합니다.

201811244 컴퓨터공학과 김창엽