CSED211 Intro. to Computer SW Systems : Bomb Lab Report

October 15, 2025

Student 20240505 Hyunseong Kong

1. Overview

이번 랩은 연속된 6개의 문자열 정답을 입력함으로서 bomb을 무력화하는 것을 목적으로 한다. 바이너리 파일은 objdump를 이용해 디어셈블링하였으며, jump를 보다 쉽게 확인하게 해주는 플래그를 설정하였다. 이러한 플래그의 존재는 objdump의 linux manual을 확인하던 중 알게 되었다. 제공받은 bomb은 18번이다.

2. Disassembly Analysis

시작 전에, 모든 문자열은 <read_line>에서 읽어들인다. 입력한 문자열은 메모리 어딘가에 저장되며, 그 첫번째 주소는 rdi에 저장된다. 이에 대한 더 자세한 내용은 secret phase 파트에서 후술한다.

2-1. Phase 1

Answer:

```
Public speaking is very easy.
```

Explanation:

```
  400ef4:     be bc 24 40 00        mov    $0x4024bc,%esi
  400ef9:     e8 10 04 00 00        call   40130e <strings_not_equal>
  400efe:     85 c0                 test   %eax,%eax
  400f00:     74 05                 je     400f07 <phase_1+0x17>
```

함수 <strings_not_equal>는 인자 2개를 받아 두 인자가 가리키는 주소의 문자열이 정확히 동일한지 판별한다. 이 인자 2개는 함수 인자 convention에 의해 항상 edi와 esi이다. 만약 문자열이 정확히 동일하다면 0을 반환하고, 그렇지 않다면 1을 반환하게 된다. 이러한 반환값은 test를 통해 반환값이 0일 때만 ZF가 1로 설정되어 je로 다음 실행주소를 변경하게 된다. 만약 실행 주소가 변경되지 않는다면, 폭탄이 터지기 때문에 우리는 eax가 0이 되는, 즉 두 문자열이 정확히 같게 되는 입력 문자열을 찾으면 된다. 따라서 남은 작업은 0x4024bc 주소에 어떤 정보가 들어가있는지 확인하는 것이다.

```
 4024b0 79207468 6973206f 6e652e00 5075626c  y this one..Publ
 4024c0 69632073 7065616b 696e6720 69732076  ic speaking is v
 4024d0 65727920 65617379 2e006272 75696e73  ery easy..bruins
```

각 숫자는 16진수이기 때문에, 주어진 바이너리 데이터를 2개씩 묶으면 하나의 ascii 문자가 된다. 첫번째 묶음의 주소가 0x4024b0이기 때문에, 0x4024bc 주소의 값은 0x50여야 한다. 이는 ascii상에서 **P**에 해당한다. 문자열의 끝을 위해서는 0x00을 찾아야 한다. 이렇게 찾은 문자열이 올바른 입력이다.
따라서, 전체 흐름에서 폭탄을 터지지 않게 하는 입력값은 **Public speaking is very easy.**이다.

2-2. Phase 2

Answer:

```
1 2 4 8 16 32
```

Explanation:

```
  400f0e:           48 83 ec 28           sub    $0x28,%rsp
  400f12:           48 89 e6              mov    %rsp,%rsi
  400f15:           e8 90 06 00 00        call   4015aa <read_six_numbers>
```

우선 phase 2는 <read_six_numbers>를 실행하여 총 6개의 숫자 값을 읽어들인다. 해당 함수는 <__isoc99_sscanf@plt>를 실행하는데, cppreference에 의하면 sscanf는 처음 2개 인자로 buffer와 format을 받는다. 함수 인자에 대한 convention에 의하면, 함수의 처음 6개의 인자만 레지스터에서 읽어들인다. 해당 함수는 숫자 6개를 입력받는 것을 목적으로 하기에, 5 & 6번째 입력값을 위해 스택을 2개 쓴다(64비트 시스템에서 return 위치 + 스택 2개를 포함하여 총 8*3 바이트의 스택 공간 할당. rbp 값은 최적화로 인해 push하지 않는다). 함수를 실행하기 전에 rsp 값을 rsi로 함수에 넘겨주었고, 함수는 이 주소부터 6개의 연속된 4바이트를 각각 sscanf의 인자로 넘겨준다. 즉, 최종적으로 입력한 6개의 숫자는 rsp 위치부터 6개의 연속된 위치에 저장된다.

```
  400f1a:           83 3c 24 01           cmpl   $0x1,(%rsp)
  400f1e:    ,----- 74 20                 je     400f40 <phase_2+0x34>
  400f20:    |      e8 4f 06 00 00        call   401574 <explode_bomb>
```

바로 다음 줄에서는 첫번째 값이 위치한 rsp 주소의 값과 1을 비교한다. 만약 두 값이 다르다면 폭탄이 터지기 때문에, 첫번째 입력 값이 1로 고정됨을 확인해볼 수 있다.

```
  400f27: ,--|----> 8b 43 fc              mov    -0x4(%rbx),%eax
  400f2a: |  |      01 c0                 add    %eax,%eax
  400f2c: |  |      39 03                 cmp    %eax,(%rbx)
  400f2e: |  |  ,-- 74 05                 je     400f35 <phase_2+0x29>
  400f30: |  |  |   e8 3f 06 00 00        call   401574 <explode_bomb>
-------------------------------------------------------------------------
  400f40: |  '--|-> 48 8d 5c 24 04        lea    0x4(%rsp),%rbx
  400f45: |     |   48 8d 6c 24 18        lea    0x18(%rsp),%rbp
  400f4a: '-----|-- eb db                 jmp    400f27 <phase_2+0x1b>
```

그 후 0x400f40 위치로 이동한 후에, rbx에 2번째 입력값의 위치를 저장하고, rbp에 6번째 입력값 다음 위치를 저장한다. 그리고 0x400f27로 이동하게 되는데, 여기서 eax에 rbx의 이전 값(1번째 입력값)을 옮긴 후 2배하고, 그것을 ebx와 비교하므로 2번째 입력값은 첫번째 값의 2배인 2가 되어야 함을 알 수 있다.

```
  400f35: |  |  '-> 48 83 c3 04           add    $0x4,%rbx
  400f39: |  |      48 39 eb              cmp    %rbp,%rbx
  400f3c: +--|----- 75 e9                 jne    400f27 <phase_2+0x1b>
```

그 후에는, rbx가 가리키는 주소를 다음 입력값으로 옮기고 그것이 rbp(6번째 입력값의 다음 위치)와 다르다면 위의 과정을 반복한다. 즉, 결론적으로 3번째 입력값은 2번째 입력값의 2배인 4, 4번째는 3번째 입력값의 2배인 8, 5번째는 16, 6번째는 32가 되어야 함을 알 수 있다. 이러한 연속적인 5번의 비교 끝에, rbp와 가리키는 위치가 같아지면 phase 2는 종료된다.
따라서, 전체 흐름에서 폭탄을 터지지 않게 하는 입력값은 **1 2 4 8 16 32**이다.

2-3. Phase 3

Answer:

```
4 0
```

Explanation:

```
  400f57:        48 8d 4c 24 08        lea    0x8(%rsp),%rcx
  400f5c:        48 8d 54 24 0c        lea    0xc(%rsp),%rdx
  400f61:        be b5 27 40 00        mov    $0x4027b5,%esi
  400f6b:        e8 c0 fc ff ff        call   400c30 <__isoc99_sscanf@plt>
  400f70:        83 f8 01              cmp    $0x1,%eax
  400f73:        7f 05                 jg     400f7a <phase_3+0x27>
```

우선 phase 3는 <__isoc99_sscanf@plt>를 실행하여 값을 읽어들인다. esi에 저장된 주소값인 0x4027b5는 2564202564 (ascii로는 %d %d)를 가리키기 때문에 해당 phase가 숫자 2개를 받음을 유추해볼 수 있다. sscanf 실행 후에 eax와 1을 비교하여 2 미만이면 폭탄이 터지는데, cppreference에 의하면 sscanf는 성공적으로 읽어들인 변수의 개수를 반환하기 때문에 이러한 유추는 더욱 견고해진다.

```
  400f7a:    '-> 83 7c 24 0c 07        cmpl   $0x7,0xc(%rsp)
  400f7f: ,----- 77 66                 ja     400fe7 <phase_3+0x94>
  400f81: |      8b 44 24 0c           mov    0xc(%rsp),%eax
  400f85: |      ff 24 c5 f0 24 40 00  jmp    *0x4024f0(,%rax,8)
```

```
4024f0 930f4000 00000000 8c0f4000 00000000  ..@.......@.....
402500 9f0f4000 00000000 ab0f4000 00000000  ..@.......@.....
402510 b70f4000 00000000 c30f4000 00000000  ..@.......@.....
402520 cf0f4000 00000000 db0f4000 00000000  ..@.......@.....
```

이렇게 입력받은 두 정수는 각각 rdx 위치인 `rsp + 0xc`와 rcx 위치인 `rsp + 0x8`로 들어가게 된다. 그 첫번째 값은 cmpl을 통해 7과 비교되는데 부호없는 비교인 ja를 사용했기 때문에, 0부터 7 사이가 아닌 정수(음수도 포함)은 전부 폭탄으로 직결된다. 첫번째 값이 주어진 범위 사이로 올바르게 입력되었다면, 그 값을 eax로 옮기고 `0x4024f0 + rax*8` 위치로 jump한다. `0x4024f0` 주소에는 주소 값이 연속적으로 나열되어 있기에, 이는 jump table로 추측된다. 따라서, 전체적인 구조를 살펴보았을 때, 해당 코드는 switch문을 활용한 것으로 생각된다. jump table은 little endian으로 저장되었기 때문에 실제 주소는 포인터의 8바이트를 각 2글자씩 쪼개어 뒤집어주어야 확인할 수 있음을 참고하자.

```
  400f91: |  ,-- eb 05               jmp    400f98 <phase_3+0x45>
  400f93: |  |   b8 d3 03 00 00      mov    $0x3d3,%eax
  400f98: |  '-> 2d 78 02 00 00      sub    $0x278,%eax
  400f9d: |  ,-- eb 05               jmp    400fa4 <phase_3+0x51>
------------------------------------------------------------------------
  400fe0: |  '-> 2d 17 02 00 00      sub    $0x217,%eax
  400fe5: |  ,-- eb 0a               jmp    400ff1 <phase_3+0x9e>
  400fe7: '--|-> e8 88 05 00 00      call   401574 <explode_bomb>
  400fec:    |   b8 00 00 00 00      mov    $0x0,%eax
  400ff1:    '-> 83 7c 24 0c 05      cmpl   $0x5,0xc(%rsp)
```

또한, switch 문에 해당하는 코드는 위와 같은 구조가 반복된다. 즉, 만약 한번 switch 문의 case를 타고 들어가면 break 없이 모든 후속된 코드를 실행함을 유추해볼 수 있다.

```
  400ff1:    '-> 83 7c 24 0c 05      cmpl   $0x5,0xc(%rsp)
  400ff6: ,----- 7f 06               jg     400ffe <phase_3+0xab>
```

마지막으로, 첫번째 입력값인 `rsp + 0xc`를 5와 다시한번 비교하여 만약 크면 폭탄이 터진다. 즉, 우리가 실제로 선택할 수 있는 첫번째 입력값은 0부터 5 사이이다. 또한, 위의 연속적인 switch 문을 살펴보면 같은 eax에 0x217라는 값을 계속 더하고 빼고 있음을 확인해볼 수 있다.

```
  400ff8:        3b 44 24 08         cmp    0x8(%rsp),%eax
```

위의 명령어에서 두번째 입력값이 eax와 비교되고 있기 때문에, 0x217는 두번째 입력값을 계산하는데 중요하게 사용된다. 그러나 여기서는 위의 더하고 빼는 과정이 정확히 상쇄되어 eax가 0이 되게 만드는 것이 가장 간단한 입력 케이스라 판단했고, 첫번째 입력값에 4를 넣으면 그런 경우가 발생함을 알 수 있다. 이렇게 잘 설정된 첫번째 입력값을 통해 두번째 입력값은 바로 0을 넣으면 된다.
따라서, 전체 흐름에서 폭탄을 터지지 않게 하는 입력값은 **4 0**이다.

2-4. Phase 4

Answer:

```
13 3
```

Explanation: phase 4는 <__isoc99_sscanf@plt>를 실행하여 값을 읽어들이며, phase 3과 동일한 이유로 정수 2개를 동일한 상대적인 스택 주소 위치에 받는다. 여기서는 그 설명을 생략한다.

```
  401068: |      83 7c 24 0c 0e      cmpl   $0xe,0xc(%rsp)
  40106d: |  ,-- 76 05               jbe    401074 <phase_4+0x2e>
```

우선, 입력된 첫번째 입력값은 15와 비교된다. 여기서 jbe는 unsigned 비교이기 때문에 첫번째 입력값이 0부터 14 사이여야 함을 확인할 수 있다.

```
  401074:    '-> ba 0e 00 00 00      mov    $0xe,%edx
  401079:        be 00 00 00 00      mov    $0x0,%esi
  40107e:        8b 7c 24 0c         mov    0xc(%rsp),%edi
  401082:        e8 81 ff ff ff      call   401008 <func4>
  401087:        83 f8 03            cmp    $0x3,%eax
```

그 후, <func4>가 호출되는데 그 인자로 (첫번째 입력값, 0, 14)을 순서대로 넣어준다. 또한, 함수 호출 후의 eax가 3과 비교되기 때문에, <func4>의 최종 반환값은 3이어야 함을 알 수 있다.
우선, <func4>는 그 구조를 살펴보면 0x401023, 0x401038 위치의 call로 인해 재귀 함수임을 쉽게 알 수 있다.

```
40100c: |         89 d0                mov    %edx,%eax
40100e: |         29 f0                sub    %esi,%eax
401010: |         89 c1                mov    %eax,%ecx
401012: |         c1 e9 1f             shr    $0x1f,%ecx
401015: |         01 c8                add    %ecx,%eax
401017: |         d1 f8                sar    $1,%eax
401019: |         8d 0c 30             lea    (%rax,%rsi,1),%ecx
```

위는 <func4> 내부에서 입력받은 값에 연산을 초기 연산을 수행하는 명령어를 나타낸다. eax에 edx-esi를 저장하고, 그 값을 0x1f만큼 logical right shift한 값을, 다시 eax에 더한 후에, 1개 비트 칸만큼 signed right shift한다. 해당 연산은 edx-esi를 나누기 2 연산한 것이라 생각해볼 수 있다. 여기서 shr 연산을 수행하면 부호만 남게 되는데, 음수 여부에 따라 그 값을 보정하여 나누기 연산이 올바르게 수행되도록 하였다. 해당 결과는 edi와 비교되며, 비교 결과에 따라 조건문으로 다음에 실행할 코드가 분기된다.

```
40102c: |  |  '-> b8 00 00 00 00        mov    $0x0,%eax
401031: |  |      39 f9                 cmp    %edi,%ecx
401033: |  +----- 7d 0c                 jge    401041 <func4+0x39>
401035: |  |      8d 71 01              lea    0x1(%rcx),%esi
401038: '--|----- e8 cb ff ff ff        call   401008 <func4>
40103d:    |      8d 44 00 01           lea    0x1(%rax,%rax,1),%eax
```

만약 ecx <= edi이면, eax에 0이 들어간 후, 다시 ecx >= edi인지 비교한다. 만약, 그렇다면 그 eax를 그대로 반환하고 그렇지 않다면 재귀적으로 <func4>를 호출한 후, 그 결과에 2배 후 1을 더한 값을 반환한다.

```
401020: |     |   8d 51 ff              lea    -0x1(%rcx),%edx
401023: +-----|-- e8 e0 ff ff ff        call   401008 <func4>
401028: |     |   01 c0                 add    %eax,%eax
```

만약 ecx > edi이면, 재귀적으로 <func4>를 호출한 후, 그 결과에 2배 후 반환한다. 우리가 원하는 반환값 3=2*(2*(0)+1)+1이기 때문에, 첫번째 분기에서 2번 재귀 호출된 후에 0을 반환하면 된다. 재귀 호출에는 각 파라미터 값들이 조금씩 변화하기 때문에 이 점을 유의하며 입력값을 찾으면 된다.

- 첫번째 재귀 호출: esi == 0, edx == 0xE라 생각하고 조건 분기 전까지 명령을 그대로 따라가면 ecx가 0x7이 된다. 즉, edi는 7보다는 커야 한다.
- 두번째 재귀 호출: esi == 0x8, edx == 0xE라 생각하고 조건 분기 전까지 명령을 그대로 따라가면 ecx가 0xB가 된다. 즉, edi는 11보다는 커야 한다.
- 마지막 재귀 호출: esi == 0xC, edx == 0xE라 생각하고 조건 분기 전까지 명령을 그대로 따라가면 ecx가 0xD가 된다. 즉, edi는 13이어야 한다.

따라서, 초기 edi가 13여야 2번의 재귀 호출 후 0을 반환하고, 그러면 순서대로 1, 3을 반환하여 최종 결과가 3이 됨을 알게 되었다.

```
40108c: |      83 7c 24 08 03        cmpl   $0x3,0x8(%rsp)
```

입력한 2번째 인자는 마지막에 한 번, 3과 비교되기 때문에 그 값이 정확히 3이어야 한다.
따라서, 전체 흐름에서 폭탄을 터지지 않게 하는 입력값은 **13 3**이다.

2-5. Phase 5

Answer:

```
MFCDHG
```

Explanation:

```
4010a2:           48 89 fb              mov    %rdi,%rbx
4010a5:           e8 47 02 00 00        call   4012f1 <string_length>
4010aa:           83 f8 06              cmp    $0x6,%eax
4010ad: ,-------- 74 43                 je     4010f2 <phase_5+0x55>
4010af: |         e8 c0 04 00 00        call   401574 <explode_bomb>
```

phase 5는 <string_length> 함수를 실행하여 입력 문자열의 길이가 6이 아니면 폭탄이 폭발한다. 입력한 문자열의 위치는 rbx에도 복사된다는 점에 유의하자.

```
  4010f2: '--|--|-> b8 00 00 00 00        mov    $0x0,%eax
  4010f7:     '--|-- eb bd                jmp    4010b6 <phase_5+0x19>
```

만약 문자열의 길이가 정확히 6이면, 위를 실행하여 eax를 0으로 설정하고 아래의 명령을 실행한다.

```
  4010b6: |  ,----> 0f b6 14 03           movzbl (%rbx,%rax,1),%edx
  4010ba: |  |      83 e2 0f              and    $0xf,%edx
  4010bd: |  |      0f b6 92 30 25 40 00  movzbl 0x402530(%rdx),%edx
  4010c4: |  |      88 14 04              mov    %dl,(%rsp,%rax,1)
  4010c7: |  |      48 83 c0 01           add    $0x1,%rax
  4010cb: |  |      48 83 f8 06           cmp    $0x6,%rax
  4010cf: |  +----- 75 e5                jne    4010b6 <phase_5+0x19>
```

우선 rbx에 0xF를 AND 연산해서 16으로 나눈 나머지를 계산한다. 그 값을 이용해 0x402530 + rdx 위치의 값을 edx로 다시 집어넣는다. 이것을 eax를 늘려가며 총 6번을 수행한다. 각 값들은 rsp + rax 위치에 저장된다.

```
  4010d1: |  |      c6 44 24 06 00        movb   $0x0,0x6(%rsp)
  4010d6: |  |      be da 24 40 00        mov    $0x4024da,%esi
  4010db: |  |      48 89 e7              mov    %rsp,%rdi
  4010de: |  |      e8 2b 02 00 00        call   40130e <strings_not_equal>
  4010e3: |  |      85 c0                 test   %eax,%eax
```

그 다음에 rsp의 6번째 위치에 0x0을 삽입함으로써 rsp + 0x0부터 rsp + 0x5까지 6개 문자를 문자열의 형태로 만든다. 이제 해당 string은 0x40130e 위치의 문자열인 bruins과 비교된다. 결론적으로, 우리가 해야하는 것은 문자 테이블에서 bruins의 각 글자로 매핑되는 숫자 6개를 찾고, 그것을 ascii의 형태로 적절히 입력하는 것이다.

```
 402530 6d616475 69657273 6e666f74 7662796c  maduiersnfotvbyl
```

그러기 위해, 우선 0x402530 위치를 보자. 해당 위치에는 위와 같은 문자열의 리스트가 있다. 각각이 b,r,u,i,n,s로 매핑되려면 아스키 문자로 0xD,0x6,0x3,0x4,0x8,0x7을 입력하면 된다. 그러나, 키보드로는 해당 번호의 ascii 문자를 입력하기는 어렵다. 대신, 코드의 중간에 입력받은 1바이트 문자가 AND 연산을 이용해 16으로 나누는 부분이 있었기에, 찾은 숫자에 적당한 16의 배수를 더해가며 키보드로 입력할 수 있는 문자를 찾으면 된다. 각 숫자에 64를 더해보면 모든 숫자를 대문자 영문 형태로 입력할 수 있는 것을 확인할 수 있다.
따라서, 전체 흐름에서 폭탄을 터지지 않게 하는 입력값은 **MFCDHG**이다.

2-6. Phase 6

Answer:

```
3 5 1 6 2 4
```

Explanation:
우선 코드가 상당히 길기 때문에, 이를 두 파트로 분리해서 생각한다. 이러한 아이디어는 반복문이 윗쪽에서 한번, 아랫쪽에서 한번 발생하며, 아래쪽 반복문에서 다시 위쪽으로 jump할 수 없다는 명령어의 구조를 관찰하여 얻었다. 결과적으로 첫번째 파트는 입력값의 형태를 제한하고, 두번째 파트는 그러한 입력값으로 특정한 연산을 수행한다.

2-6-1. Part 1

```
  401109:              48 8d 74 24 30        lea    0x30(%rsp),%rsi
  40110e:              e8 97 04 00 00        call   4015aa <read_six_numbers>
  401113:              4c 8d 6c 24 30        lea    0x30(%rsp),%r13
```

phase 6은 <read_six_numbers>로 6개의 숫자를 입력받는다. 0x30 + rsp부터 총 6개의 숫자가 순서대로 4바이트씩 차지하게 된다. 그 중 첫번째 원소의 위치를 r13에 복사한다.

```
  401118:              41 bc 00 00 00 00     mov    $0x0,%r12d
  40111e:     ,-------> 4c 89 ed             mov    %r13,%rbp
-----------------------------------------------------------------------
  401132:     |     '-> 41 83 c4 01          add    $0x1,%r12d
  401136:     |         41 83 fc 06          cmp    $0x6,%r12d
```

```
  40113a:    |     ,-- 75 07               jne    401143 <phase_6+0x44>
  40113c:    |     |   be 00 00 00 00      mov    $0x0,%esi
  401141: ,--|-----|-- eb 42               jmp    401185 <phase_6+0x86>
------------------------------------------------------------------------
  40115f: |  |        49 83 c5 04          add    $0x4,%r13
  401163: |  '-------- eb b9               jmp    40111e <phase_6+0x1f>
```

위의 긴 명령이 하나의 큰 덩어리이다. 우선, 바깥의 반복문과 안쪽의 반복문 총 2개의 반복 구조가 보인다. 바깥 반복문의 반복 횟수를 세는데는 r12d가 이용된다. 우선, 0x40111e와 0x40115f의 명령을 통해 r13이 4씩 더해지며, r13이 가리키는 값들이 1칸씩 옮겨가는 것을 확인할 수 있다. 이러한 변경 작업 후에는 r13을 rbp에 복사해서 사용하며, 이 값은 다음 r13 값을 받기 전까지는 변하지 않는다. 즉, 입력한 6개의 숫자를 첫 번째부터 하나씩 고정해놓고, 어떠한 연산을 수행함을 알 수 있다. r13을 옮길 때마다 r12d의 값도 하나씩 증가하며, 그 값이 0x6과 동일하면 0x401143로 jump하여 part 1 전체를 벗어나게 된다.

```
  401121:    |        41 8b 45 00          mov    0x0(%r13),%eax
  401125:    |        83 e8 01             sub    $0x1,%eax
  401128:    |        83 f8 05             cmp    $0x5,%eax
  40112b:    |     ,-- 76 05               jbe    401132 <phase_6+0x33>
  40112d:    |     |   e8 42 04 00 00      call   401574 <explode_bomb>
```

다음으로는 위의 명령의 중간에 위치한 이 코드를 살펴보자. r13이 가리키는 위치의 값을 eax로 복사하고, 1을 뺀 다음 5와 비교하고 있다. 계산한 값이 unsigned 비교인 jbe를 통해 비교되고 있으므로, 결과적으로 r13이 가리키는 위치의 값은 1부터 6 사이여야 함을 알 수 있다. 위에서 설명한 것처럼 0x40115f에서 r13이 가리키는 숫자를 한 칸씩 옮겨가고 있으니, 모든 숫자가 1부터 6 사이여야 한다.

```
  401143: |  |     '-> 44 89 e3            mov    %r12d,%ebx
  401146: |  |  ,----> 48 63 c3             movslq %ebx,%rax
  401149: |  |  |      8b 44 84 30          mov    0x30(%rsp,%rax,4),%eax
  40114d: |  |  |      39 45 00             cmp    %eax,0x0(%rbp)
  401150: |  |  |  ,-- 75 05                jne    401157 <phase_6+0x58>
  401152: |  |  |  |   e8 1d 04 00 00       call   401574 <explode_bomb>
  401157: |  |  |  '-> 83 c3 01             add    $0x1,%ebx
  40115a: |  |  |      83 fb 05             cmp    $0x5,%ebx
  40115d: |  |  '----- 7e e7                jle    401146 <phase_6+0x47>
```

이 코드를 살펴보면 현재 r12d의 값을 ebx에 옮기고, ebx를 하나씩 늘려가며 0x5가 되기 전까지 반복문을 돌리고 있다. 반복문 내부를 살펴보면 고정된 rbp를 ebx를 늘려가며 0x30 + rsp + rax*4와 같은지 확인하고 있다. 여기서 rax는 rbx와 동일한 값이다. rbx는 rbp가 가리키는 값 이후의 값들만을 나타내고, 둘이 동일하면 폭탄이 폭발하므로, 모든 rbp가 가리키는 값들은 자기자신 이후의 입력값들과 겹치지 않아야 한다.
따라서, 위의 전체 명령은 입력값이 1부터 6 사이의 정수이되, 모든 입력값이 겹치지 않아야 한다는 것을 시사한다.

2-6-2. Part 2

```
  401165: |  ,-------> 48 8b 52 08          mov    0x8(%rdx),%rdx
  401169: |  |         83 c0 01             add    $0x1,%eax
  40116c: |  |         39 c8                cmp    %ecx,%eax
  40116e: |  +-------- 75 f5                jne    401165 <phase_6+0x66>
  401170: |  |      ,-- eb 05                jmp    401177 <phase_6+0x78>
  401172: |  |  ,--|-> ba f0 42 60 00        mov    $0x6042f0,%edx
  401177: |  |  |  '-> 48 89 14 74          mov    %rdx,(%rsp,%rsi,2)
  40117b: |  |  |      48 83 c6 04          add    $0x4,%rsi
  40117f: |  |  |      48 83 fe 18          cmp    $0x18,%rsi
  401183: |  |  |  ,-- 74 15                je     40119a <phase_6+0x9b>
  401185: '--|--|--|-> 8b 4c 34 30          mov    0x30(%rsp,%rsi,1),%ecx
  401189:    |  |  |   83 f9 01             cmp    $0x1,%ecx
  40118c:    |  '--|-- 7e e4                jle    401172 <phase_6+0x73>
  40118e:    |     |   b8 01 00 00 00       mov    $0x1,%eax
  401193:    |     |   ba f0 42 60 00       mov    $0x6042f0,%edx
  401198:    '-----|-- eb cb                jmp    401165 <phase_6+0x66>
```

part 1에서 벗어나면 part 2의 0x401185로 이동하게 된다. 해당 위치에서는 우선 0x30 + rsp + rsi 위치의 값이 1 이하인지(즉, 정확히 1인지) 체크한다. jump 전에 esi를 0으로 만들었기에 0x30 + rsp의 위치를 계산하는 것이고, 이는 첫번째 입력값을 의미한다. 해당 값이 0이면, edx로 0x6042f0의 주소를 옮기고, 그 주소로 들어가서 값을 rsp + rsi * 2로 옮긴다. 반면에, 만약 해당 값이 0이 아니라면 eax를 1로 만들고, edx에 0x6042f0 주소를 넣는다. 그 후, eax를 늘려가며 ecx와 비교한다. ecx에 0x30 + rsp + rsi를 옮겨놨었기에, 해당 코드는 결과적으로 첫번째 입력 값이 eax와 똑같아질 때까지 반복된다. 이때, eax를 늘려가며 동시에 rdx + 0x8 위치의 값을 rdx로 다시 옮긴다.

```
6042f0 8a010000 01000000 00436000 00000000  .........C`.....
604300 bb000000 02000000 10436000 00000000  .........C`.....
604310 cf020000 03000000 20436000 00000000  ........ C`.....
604320 37000000 04000000 30436000 00000000  7.......0C`.....
604330 5a020000 05000000 40436000 00000000  Z.......@C`.....
604340 6b010000 06000000 00000000 00000000  k...............
```

처음에 edx에 들어갔던 주소인 0x6042f0 부근을 살펴보면, 4바이트 숫자(특정 값) + 4바이트 숫자(index 역할) + 8바이트 주소로 구성된 것을 확인할 수 있다. 또한, 각 주소는 바로 다음 위치를 가리키고 있다. 이러한 구조는 단방향 연결리스트에 해당된다. 즉, rdx + 0x8을 rdx로 옮기는 것은 연결 리스트에서 각 노드와 연결된 다음 노드로 이동하는 것을 의미한다. 이러한 동작을 입력한 숫자와 같아질 때까지 eax를 늘려가며 반복하기에, 내가 입력한 숫자를 index로 하는 노드를 찾는 것이라 볼 수 있다. 마지막으로, 이러한 노드를 찾으면 그 주소 값을 rsp + rsi*2 위치로 옮긴다. 마지막으로 rsi를 4만큼 더하고 0x18과 비교하는데, 이는 입력값을 순차적으로 총 6번 처리(모든 입력값을 처리)하는 것을 나타낸다.
정리하면, 각 6개 입력값을 순차적으로 순회하며 만약 입력값이 1이면 연결리스트의 첫번째 노드의 주소를 스택에 옮기고, 입력값이 2 이상이면 그 위치까지 들어가서 주소를 옮기는 코드이다. 모든 처리가 끝나면 0x40119a로 이동한다.

```
  40119a:          '-> 48 8b 1c 24           mov    (%rsp),%rbx
  40119e:              48 8d 44 24 08        lea    0x8(%rsp),%rax
  4011a3:              48 8d 74 24 30        lea    0x30(%rsp),%rsi
  4011a8:              48 89 d9              mov    %rbx,%rcx
  4011ab:       ,----> 48 8b 10              mov    (%rax),%rdx
  4011ae:       |      48 89 51 08           mov    %rdx,0x8(%rcx)
  4011b2:       |      48 83 c0 08           add    $0x8,%rax
  4011b6:       |      48 39 f0              cmp    %rsi,%rax
  4011b9:       |  ,-- 74 05                 je     4011c0 <phase_6+0xc1>
  4011bb:       |  |   48 89 d1              mov    %rdx,%rcx
  4011be:       '--|-- eb eb                 jmp    4011ab <phase_6+0xac>
  4011c0:          '-> 48 c7 42 08 00 00 00  movq   $0x0,0x8(%rdx)
```

해당 명령은 우선 rbx에 rsp 값, 즉 연결리스트의 특정 노드의 값을 넣는다. 그 후, rax에 rsp + 8의 주소를 넣고 rcx에 다시 rbx의 값을 옮긴 다음, rsi에 rsp + 30의 주소를 넣는다. 그 후 rax를 8만큼씩 더해가며 그 값을 rsi와 비교하고 있다. 이때, 그 값의 주소를 rdx에 임시로 넣었다가 다시 rcx + 8 위치로 옮기고 있기 때문에, 결과적으로 우리가 스택에 넣어두었던 6개 노드의 포인터를 순차적으로 읽어들이며, 그 노드의 값을 덮어쓰는 코드라고 할 수 있다. 동시에, 값 바로 뒤에 주소를 넣음으로서 결과적으로 새로운 연결 리스트를 만드는 것이다. 이러한 과정이 끝나면 rdx 다음 위치에 0을 넣어 연결리스트의 마지막을 표기한다.

```
  4011c8:              bd 05 00 00 00        mov    $0x5,%ebp
  4011cd:       ,----> 48 8b 43 08           mov    0x8(%rbx),%rax
  4011d1:       |      8b 00                 mov    (%rax),%eax
  4011d3:       |      39 03                 cmp    %eax,(%rbx)
  4011d5:       |  ,-- 7d 05                 jge    4011dc <phase_6+0xdd>
  4011d7:       |  |   e8 98 03 00 00        call   401574 <explode_bomb>
  4011dc:       |  '-> 48 8b 5b 08           mov    0x8(%rbx),%rbx
  4011e0:       |      83 ed 01              sub    $0x1,%ebp
  4011e3:       '----- 75 e8                 jne    4011cd <phase_6+0xce>
  4011e5:              48 83 c4 58           add    $0x58,%rsp
```

최종적으로 이렇게 재연결한 연결리스트를 다시 순차적으로 읽어들인다. 이때, 그 다음 값을 rax, 현재 값을 rbx가 가리키도록 하고 둘을 jge로 비교하고 있기 때문에, 재연결된 연결리스트의 값이 내림차순이 되어야 함을 알 수 있다. 이제 다시 원본 연결리스트를 보면, 값 0x18A,0xBB,0x2CF,0x37,0x25A,0x16B(10진수로는 394, 187, 719, 55, 602, 363)이 순서대로 저장되어있음을 확인할 수 있다. 즉, 이를 내림차순으로 만드는 순서 6개가 폭탄이 터지지 않게 하는 최종 답이 된다.
따라서, 전체 흐름에서 폭탄을 터지지 않게 하는 입력값은 **3 5 1 6 2 4**이다.

2-7. Phase Secret

2-7-1. Entrance

In phase 4, use input

```
13 3 DrJisungPark
```

secret phase의 존재성은 0x40122e 위치의 <secret_phase>로 인해 쉽게 알 수 있다. 그러나, 위의 6개 답을 순차적으로 입력하면 bomb이 그대로 종료되기 때문에, 특정한 동작을 수행해야 입장할 수 있음을 알 수 있다.

```
401470: ,----> 48 63 05 25 33 20 00  movslq 0x203325(%rip),%rax
401477: |      48 8d 3c 80           lea    (%rax,%rax,4),%rdi
40147b: |      48 c1 e7 04           shl    $0x4,%rdi
40147f: |      48 81 c7 c0 47 60 00  add    $0x6047c0,%rdi
401486: |      48 8b 15 13 33 20 00  mov    0x203313(%rip),%rdx        # 6047a0 <infile>
40148d: |      be 50 00 00 00        mov    $0x50,%esi
401492: |      e8 19 f7 ff ff        call   400bb0 <fgets@plt>
```

<skip>의 일부인 위 코드를 보자. 참고로, 모든 문자열 입력은 <fgets@plt>가 유일하게 호출되는 skip이 처리한다. 여기서 0x203325 + rip의 계산된 결과는 항상 0x60479c이며, objdump는 <num_input_strings>라는 레이블로 표현했다. 즉, 이 값은 어떠한 정수인데, 위 코드는 그 정수를 rax로 옮긴 후, 5배하고, 4만큼 왼쪽 shift한다. 그리고 그것을 0x6047c0과 더해서 최종적으로 rdi로 넣는다. cppreference에 의하면 fgets는 char*, int, FILE* 값을 순서대로 받아들인다. 여기서 FILE*은 파일포인터로 입력 스트림의 위치를 나타낸다. bomb은 문자열 입력과 파일을 이용한 입력을 동시에 지원하며, 파일을 이용한 입력은 infile이라는 전역변수를 통해, 키보드를 이용한 cli의 입력은 stdin 스트림의 경로로 이동하여 읽어들인다. 여기서 중요한 것은, fgets를 이용해 고정된 길이의 입력을 받는다는 것이며 이 고정된 길이는 앞에서 <num_input_strings>를 연산하며 최종적으로 곱한 상수인 80이다. 즉, fgets는 0x6047c0 부터 시작하여 80칸씩 입력 스트림에서 값을 가져온다. 이제, <num_input_strings>의 정확한 역할을 찾아보자.

```
readelf -S ./bomb18
```

위의 명령어를 이용해 각 섹션 헤더의 위치를 확인하였고, <num_input_strings>의 위치인 0x60479c는 0x604780 부터 0x6d0 크기를 가진 .bss 영역내에 존재하는 것을 확인하였다. linux manual에 의하면, 아래와 같다.

```
This section holds uninitialized data that contributes to
the program's memory image. By definition, the system
initializes the data with zeros when the program begins to
run.
```

즉, 모든 bss 영역은 정의에 의해 0으로 초기화된다. <num_input_strings>이 0에서 시작하는 변수인 것을 확인하였다.

```
4016ac:    |   8b 05 ea 30 20 00     mov    0x2030ea(%rip),%eax
4016b2:    |   8d 50 01              lea    0x1(%rax),%edx
4016b5:    |   89 15 e1 30 20 00     mov    %edx,0x2030e1(%rip)
```

위는 <read_line>의 일부이다. 이를 확인해보면 해당 함수가 실행될 때 마다 <num_input_strings> 값이 1이 증가하는 것을 확인할 수 있다. 즉, <num_input_strings>는 <read_line>가 실행된 횟수를 정수로 기록한다.

```
401720:        83 3d 75 30 20 00 06  cmpl   $0x6,0x203075(%rip)
401727: ,----- 75 6d                 jne    401796 <phase_defused+0x84>
401729: |      4c 8d 44 24 10        lea    0x10(%rsp),%r8
40172e: |      48 8d 4c 24 08        lea    0x8(%rsp),%rcx
401733: |      48 8d 54 24 0c        lea    0xc(%rsp),%rdx
401738: |      be ff 27 40 00        mov    $0x4027ff,%esi
40173d: |      bf b0 48 60 00        mov    $0x6048b0,%edi
```

이제, <phase_defused>를 보자. 우선, <num_input_strings>인 0x203075 + rip과 6을 비교한다. 문제가 6개이므로, 이는 모든 문제를 다 풀었는지 체크하는 것이다. 만약 그렇다면, rdi에는 0x6048b0를 옮기고, rsi에는 0x4027ff를 옮긴다. 0x6048b0 - 0x6047c0 = 0xF0(10진수로 240)이고, 각 입력값에는 80자씩 사용하기 때문에, 결과적으로 이는 정확히 4번째 입력값의 시작점 위치가 된다. 즉, 최종적으로 secret phase는 4번째 입력값을 재사용하는 것이라 해석할 수 있다. esi에 옮긴 0x4027ff 위치의 텍스트는 %d %d %s이기에, 4번째 입력값의 뒷부분에 특정한 문자열을 넣어야 함을 알 수 있다.

```
401751: |  |   be 08 28 40 00        mov    $0x402808,%esi
401756: |  |   48 8d 7c 24 10        lea    0x10(%rsp),%rdi
40175b: |  |   e8 ae fb ff ff        call   40130e <strings_not_equal>
```

입력된 문자열은 0x402808 위치의 문자열과 비교된다. 해당 위치에 있는 문자열은 **DrJisungPark**이다. 기존 4번째 입력값은 **13 3**이었고, %d %d를 sscanf의 인자로 넣어 값을 레지스터로 옮겼다. sscanf는 esi에 입력된 파싱 문자열을 제외한 나머지 문자열은 그냥 무시하기 때문에, 4번째 입력값 뒤에 새로운 문자열을 추가해도 phase 4의 어떠한 조건도 위배하지 않는다.
따라서, secret phase에 입장하기 위한 올바른 입력값은 **13 3 DrJisungPark**이다.

2-7-2. Solve

Answer:

```
35
```

```
40122f:     e8 b8 03 00 00        call   4015ec <read_line>
401234:     ba 0a 00 00 00        mov    $0xa,%edx
401239:     be 00 00 00 00        mov    $0x0,%esi
40123e:     48 89 c7              mov    %rax,%rdi
401241:     e8 ba f9 ff ff        call   400c00 <strtol@plt>
401246:     48 89 c3              mov    %rax,%rbx
401249:     8d 40 ff              lea    -0x1(%rax),%eax
40124c:     3d e8 03 00 00        cmp    $0x3e8,%eax
401251: ,-- 76 05                 jbe    401258 <secret_phase+0x2a>
401253: |   e8 1c 03 00 00        call   401574 <explode_bomb>
```

secret phase는 <read_line>으로 한 줄을 읽어들이고, strtol을 실행하는데 cppreference에 의하면 이 함수는 주어진 문자열을 숫자로 변환한다. 즉, 입력값이 문자열로 받아들여지고 그것이 숫자 형태로 변환된다는 것을 알 수 있다. 결과 숫자인 rax는 rbx로 옮겨지고, 1을 뺀 값을 0x3e8과 비교한다. 만약, 값이 그것보다 크면 폭탄이 터지기 때문에, 입력값은 1과 1000 사이여야 한다는 것을 알 수 있다.

```
401258: '-> 89 de                 mov    %ebx,%esi
40125a:     bf 10 41 60 00        mov    $0x604110,%edi
40125f:     e8 8c ff ff ff        call   4011f0 <fun7>
401264:     83 f8 06              cmp    $0x6,%eax
401267: ,-- 74 05                 je     40126e <secret_phase+0x40>
401269: |   e8 06 03 00 00        call   401574 <explode_bomb>
40126e: '-> bf 40 25 40 00        mov    $0x402540,%edi
```

그 후, 함수 <fun7>를 실행하며, 결과를 6과 비교한다. 즉, 함수의 출력 결과가 6이어야 한다. 함수의 입력에는 0x604110 주소값과 우리가 입력한 숫자를 순서대로 입력한다.

```
4011f4:    |         48 85 ff             test   %rdi,%rdi
4011f7: ,--|-------- 74 2b                je     401224 <fun7+0x34>
4011f9: |  |         8b 17                mov    (%rdi),%edx
4011fb: |  |         39 f2                cmp    %esi,%edx
4011fd: |  |     ,-- 7e 0d                jle    40120c <fun7+0x1c>
4011ff: |  |     |   48 8b 7f 08          mov    0x8(%rdi),%rdi
401203: |  +-----|-- e8 e8 ff ff ff       call   4011f0 <fun7>
401208: |  |     |   01 c0                add    %eax,%eax
40120a: |  |  ,--|-- eb 1d                jmp    401229 <fun7+0x39>
40120c: |  |  |  '-> b8 00 00 00 00       mov    $0x0,%eax
401211: |  |  |      39 f2                cmp    %esi,%edx
401213: |  |  +----- 74 14                je     401229 <fun7+0x39>
401215: |  |  |      48 8b 7f 10          mov    0x10(%rdi),%rdi
401219: |  '--|----- e8 d2 ff ff ff       call   4011f0 <fun7>
40121e: |     |      8d 44 00 01          lea    0x1(%rax,%rax,1),%eax
401222: |     +----- eb 05                jmp    401229 <fun7+0x39>
401224: '-----|----> b8 ff ff ff ff       mov    $0xffffffff,%eax
```

우선 함수 <fun7>은 자기자신을 실행하기 때문에 재귀함수이다. 초반부를 살펴보면 rdi 값이 0이면, eax에 -1을 넣고 반환함을 알 수 있다. 만약 rdi가 0이 아니라면 그 내부 값을 edx로 옮긴다. 그리고 그것을 우리가 입력한 숫자와 비교하는데, edx가 크면 rdi + 0x8을 다시 rdi로 옮기고 <fun7>을 실행한다. 그리고 그 함수의 결과인 eax를 2배하고 이를 반환한다. 만약, edx가 작거나 같으면 eax에 0을 넣는다. 만약, edx와 esi가 같으면 그대로 0을 반환하고, 아니라면 rdi + 0x10을 rdi로 옮기고 <fun7>을 실행한다. 그리고 그 반환값인 rax를 2배하고 1을 더한 뒤 반환한다. 이 구조는 우리가 phase 4에서 본 적이 있다. 그러나, phase 4는 입력값을 다루는 것에 반해, 이 함수는 외부 주소에 접근하고 그 값과 입력값을 비교하며 eax를 변화시키고 있다. 이제, 함수의 초기 입력값인 주소 0x604110를 살펴보자.

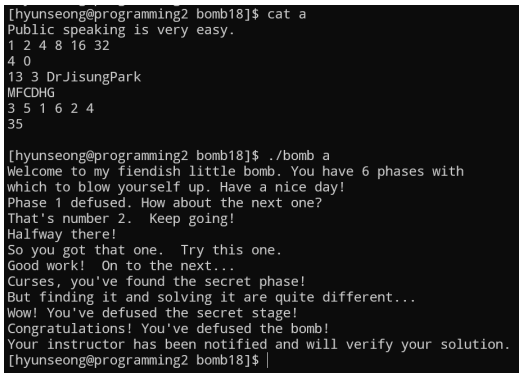
```
 604110 24000000 00000000 30416000 00000000  $.......0A`.....
 604120 50416000 00000000 00000000 00000000  PA`.............
 604130 08000000 00000000 b0416000 00000000  .........A`.....
 604140 70416000 00000000 00000000 00000000  pA`.............
 604150 32000000 00000000 90416000 00000000  2........A`.....
 604160 d0416000 00000000 00000000 00000000  .A`.............
 604170 16000000 00000000 90426000 00000000  .........B`.....
 604180 50426000 00000000 00000000 00000000  PB`.............
--------------------------------------------------------------------------
 604250 23000000 00000000 00000000 00000000  #...............
```

해당 주소를 보면, 4바이트 숫자와 4칸 패딩(혹은 8바이트 숫자), 주소 값 2개, 그리고 8바이트 패딩이 연속적으로 나열된 것을 확인할 수 있다. 위의 edx와 esi 비교에서 값이 큰지 작은지에 따라 +0x8이나 +0x10(10진수로 16) 연산을 수행하기 때문에, 값이 존재하고 값이 크고 작은지의 유무에 따라 왼쪽과 오른쪽으로 분기하는 binary tree 구조임을 유추해볼 수 있다.
원하는 반환값 6=2*(2*(2*(0)+1)+1))이기 때문에, 우리는 비교값에 비해 입력값이 작고, 크고, 크도록 설정해야 한다.

- 첫번째 재귀 호출(edx > esi): 숫자가 0x604110 위치의 0x24보다 작아야 한다. 0x604130로 분기한다.
- 첫번째 재귀 호출(edx < esi): 숫자가 0x604130 위치의 0x08보다 커야 한다. 0x604170로 분기한다.
- 첫번째 재귀 호출(edx < esi): 숫자가 0x604170 위치의 0x16보다 커야 한다. 0x604250로 분기한다.
- 마지막 재귀 호출(edx == esi): 숫자가 0x604250 위치의 0x23와 같아야 한다. 즉, 숫자는 0x23이어야 한다.

최종적으로 찾은 값인 0x23은 10진수로 35이다.
따라서, 전체 흐름에서 폭탄을 터지지 않게 하는 입력값은 **35**이다.

3. Result



왼쪽의 스크린샷은 포스텍 Programming2 서버에서 bomb이 실행된 모습을 나타낸다. 이 보고서를 빌어 어떠한 규칙도 어기지 않고 모든 문제를 적법한 방법으로 해결하였음을 알리며, 각 문제는 전부 objdump를 이용한 정적 분석으로 해결하였기에 정확히 1번의 실행만을 필요로 하였다.

4. References

[1] linux command **objdump**: https://man7.org/linux/man-pages/man1/objdump.1.html
[2] linux command **readelf**: https://man7.org/linux/man-pages/man1/readelf.1.html
[3] section header **.bss**: https://man7.org/linux/man-pages/man5/elf.5.html
[4] cpp function **sscanf**: https://en.cppreference.com/w/c/io/fscanf
[5] cpp function **fgets**: https://en.cppreference.com/w/c/io/fgets
[6] stream **stdin**: https://en.cppreference.com/w/cpp/io/c/std_streams
[7] cpp function **strtol**: https://en.cppreference.com/w/c/string/byte/strtol