System Programming
LAB 2

인천대학교

시스템 프로그래밍 | 지도교수 박문주

송병준 | 201701562

0.목차

System Programming

LAB 2

송병준
2019년 10월 3일

---

1. 과제

bomb94 폭탄 처리.

2. 환경

로컬
운영체제: macOS 10.14.6 (Darwin Kernel Version 18.7.0 x86_64)
프로세서: Intel Core i5
편집기: vim 8.0.1365
빌드: Apple LLVM version 10.0.1 (clang-1001.0.46.4)

원격
운영체제: Linux4.4.0-166-generic #195-Ubuntu x86_64 x86_64 GNU/Linux
프로세서: Intel(R) Xeon(R) CPU E3-1230 v3 @ 3.30GHz
편집기: vim 7.4.1689
빌드: gcc 5.4.0 20160609
디버그: gdb 7.11.1 & objdump 2.26.1

3. 수행

3.0. 사전 준비

어셈블리 추출

```
$ objdump -d bomb > dump.txt
```

스트링 추출

```
$ objdump -d bomd -j .rodata > strings.txt
```

3.1. Phase 1

입력한 문자열이 $0x402670$에 있는 문자열과 일치하지 않으면 폭발한다.

```
402668: 6e 65 2e 00 00 00 00 00 49 20 74 75 72 6e 65 64    ne.....I turned
402678: 20 74 68 65 20 6d 6f 6f 6e 20 69 6e 74 6f 20 73    the moon into s
402688: 6f 6d 65 74 68 69 6e 67 20 49 20 63 61 6c 6c 20    omething I call
402698: 61 20 44 65 61 74 68 20 53 74 61 72 2e 00 00 00    a Death Star…
```

해당 위치에 있는 문자열은 "I turned the moon into something I call a Death Star."이다. 마침표 포함.

3.2. Phase 2

먼저 정적으로 분석을 시도해본다.
주어진 어셈블리에서 logic 이해에 필요하지 않은 부분을 제거하고 주석을 추가하면 다음과 같다:

```
0000000000400f49 〈phase_2〉:
 400f4b:   48 83 ec 38              sub   $0x38,%rsp          # Allocate 0x38(56) bytes in stack,
                                                              which means 7 of 8 byte variables.
 400f5d:   31 c0                    xor   %eax,%eax           # Set %eax to zero.
 400f5f:   48 89 e6                 mov   %rsp,%rsi           # Save %rsp to %rsi.
 400f62:   e8 7a 07 00 00           callq 4016e1 〈read_numbers〉  # Call 〈read_numbers〉.
                                                              Pass the address of the array(may be) of 7 variables.
 400f67:   83 3c 24 00              cmpl  $0x0,(%rsp)         # Compare 0 and (%rsp).
 400f6b:   7f 05                    jg    400f72 〈phase_2+0x29〉  # If vars[0] is over zero, keep going.
 400f6d:   e8 39 07 00 00           callq 4016ab 〈explode_bomb〉  # Or die.
 400f72:   48 8d 6c 24 04           lea   0x4(%rsp),%rbp      # Save %rsp + 4 to %rbp.
 400f77:   bb 01 00 00 00           mov   $0x1,%ebx           # Save 1 to %ebx.
 400f7c:   eb 22                    jmp   400fa0 〈phase_2+0x57〉  # Go to 0x400fa0.

 Set %edx to 1, %eax to 0 and start loop below if %ebx 〉 0.
 In this case, %ebx is just set to 1 above. So get into the loop.

 〉〉〉 Loop1 begin
 400f7e:   01 d2                    add   %edx,%edx           # Save 2 * %edx to %edx. %edx += %edx.
 400f80:   83 c0 01                 add   $0x1,%eax           # Add 1 to %eax.
 400f83:   39 d8                    cmp   %ebx,%eax           # Compare %ebx and %eax.
 400f85:   75 f7                    jne   400f7e 〈phase_2+0x35〉  # If %ebx != %eax, go to 0x400f7e.
 〈〈〈 Loop1 end

 After the loop, %edx would be 1 〈〈 (%ebx − %eax).
 %ebx: 1
 %eax: 1
 %edx: 2

 400f87:   03 55 fc                 add   −0x4(%rbp),%edx     # Add vars[0] to %edx.
                                                              The %rbp has value of %rsp + 4,
                                                              so the %rbp − 4 means %rsp.
 400f8a:   39 55 00                 cmp   %edx,0x0(%rbp)      # Compare %edx and (%rbp).
 400f8d:   74 05                    je    400f94 〈phase_2+0x4b〉  # if %edx == (%rbp), keep going.
 400f8f:   e8 17 07 00 00           callq 4016ab 〈explode_bomb〉  # 〉〉 Or die.
 400f94:   83 c3 01                 add   $0x1,%ebx           # Add 1 to %ebx. Now 2.
 400f97:   48 83 c5 04              add   $0x4,%rbp           # Add 4 to %rbp. Now points &vars[1].
 400f9b:   83 fb 07                 cmp   $0x7,%ebx           # Compare 7 and %ebx.
 400f9e:   74 10                    je    400fb0 〈phase_2+0x67〉  # 〉〉 If 7 == %ebx, go to the end of the function.
 400fa0:   ba 01 00 00 00           mov   $0x1,%edx           # Save 1 to %edx.
 400fa5:   b8 00 00 00 00           mov   $0x0,%eax           # Save 0 to %eax.
 400faa:   85 db                    test  %ebx,%ebx           # Test %ebx.
 400fac:   7f d0                    jg    400f7e 〈phase_2+0x35〉  # If %ebx 〉 0, go to 0x400f7e and start loop.
 400fae:   eb d7                    jmp   400f87 〈phase_2+0x3e〉  # Go to 0x400f87.
 400fb0:   48 8b 44 24 28           mov   0x28(%rsp),%rax     # Ready to finish.
 400fc5:   48 83 c4 38              add   $0x38,%rsp          # Deallocate stack and return.
```

스택에 56바이트를 할당하고 현재 스택의 주소를 인자로 하여 read_numbers를 호출한다. 해당 함수는 $0x4029a1$에 위치한 포맷 스트링 "%d %d %d %d %d %d %d"을 가지고 stdin으로부터 7개의 숫자를 입력받는다. 이때 입력받은 숫자는 $rsp, $rsp + 4, $rsp + 8 …의 순서로 저장된다.

첫 번째 숫자는 0보다 커야 한다(0x400f67). 어셈블리에 표시된 Loop1은 비교할 숫자를 만드는 과정이다. 이렇게 만들어진 수는 $edx에 담긴다.

첫번째 숫자를 시작으로 반복문으로 진입해 다음 숫자들을 각각 $edx와 비교하는데, $edx는 매 반복마다 **두 배**가 된다(0x400f7e). Loop1을 거치면 $edx는 1 〈〈 ($ebx - $eax)의 값을 가지게 된다. 처음 실행시 $ebx에 1, $eax에 0이 할당되므로 **첫 번째 숫자는** 2이어야 한다.

그 후 두번째 숫자는 이전 숫자의 두 배가 되어야 하므로 4, 그 다음은 8, 16 … 128이다.
따라서 답은 2 4 8 16 32 64 128.

3.3. Phase 3

어셈블리를 보자.

```
0000000000400fcc <phase_3>:
  400fcc:   48 83 ec 18                 sub    $0x18,%rsp                  # Allocate 0x18(24) bytes in stack.
  400fd0:   64 48 8b 04 25 28 00        mov    %fs:0x28,%rax               # Save stack canary to %rax.
  400fd7:   00 00
  400fd9:   48 89 44 24 08              mov    %rax,0x8(%rsp)              # Save stack canary to stack with offset +8.
  400fde:   31 c0                       xor    %eax,%eax                   # Clear %rax.
  400fe0:   48 8d 4c 24 04              lea    0x4(%rsp),%rcx              # Save %rsp + 4 to %rcx.
  400fe5:   48 89 e2                    mov    %rsp,%rdx                   # Save %rsp to %rdx.
  400fe8:   be b0 29 40 00              mov    $0x4029b0,%esi              # Save address of "%d %d" to %rsi.
  400fed:   e8 4e fc ff ff              callq  400c40 <__isoc99_sscanf@plt> # Call scanf. Params are %rdi, %rsi, %rdx.
                                                                           %rsi is "%d %d", %rsi is %rsp.
  400ff2:   83 f8 01                    cmp    $0x1,%eax                   # Compare returned value.
  400ff5:   7f 05                       jg     400ffc <phase_3+0x30>       # If over two items, keep going.
  400ff7:   e8 af 06 00 00              callq  4016ab <explode_bomb>       # Or explode.
  400ffc:   8b 04 24                    mov    (%rsp),%eax                 # Save input[0] to %eax.

  Below is switch-case satement.
  Indices range from 0 to 7,
  so the input[0] must be in range from 43 to 50.

  400fff:   83 e8 2b                    sub    $0x2b,%eax                  # Subtract 0x2b(43) from $eax.
  401002:   83 f8 07                    cmp    $0x7,%eax                   # Compare unsigned %eax with 7.
  401005:   77 64                       ja     40106b <phase_3+0x9f>       # If over 7, jump and explode.
  401007:   89 c0                       mov    %eax,%eax                   # NOP
  401009:   ff 24 c5 e0 26 40 00        jmpq   *0x4026e0(,%rax,8)          # Jump! This is a switch-case statement!

  401010:   b8 88 00 00 00              mov    $0x88,%eax                  # >> 0
  401015:   eb 05                       jmp    40101c <phase_3+0x50>

  401017:   b8 00 00 00 00              mov    $0x0,%eax                   # >> 1
  40101c:   2d 4c 02 00 00              sub    $0x24c,%eax

  401021:   eb 05                       jmp    401028 <phase_3+0x5c>
  401023:   b8 00 00 00 00              mov    $0x0,%eax                   # >> 2
  401028:   05 e9 02 00 00              add    $0x2e9,%eax
  40102d:   eb 05                       jmp    401034 <phase_3+0x68>

  40102f:   b8 00 00 00 00              mov    $0x0,%eax                   # >> 3
  401034:   2d 00 02 00 00              sub    $0x200,%eax
  401039:   eb 05                       jmp    401040 <phase_3+0x74>

  40103b:   b8 00 00 00 00              mov    $0x0,%eax                   # >> 4
  401040:   05 00 02 00 00              add    $0x200,%eax
  401045:   eb 05                       jmp    40104c <phase_3+0x80>

  401047:   b8 00 00 00 00              mov    $0x0,%eax                   # >> 5
  40104c:   2d 00 02 00 00              sub    $0x200,%eax
  401051:   eb 05                       jmp    401058 <phase_3+0x8c>

  401053:   b8 00 00 00 00              mov    $0x0,%eax                   # >> 6
  401058:   05 00 02 00 00              add    $0x200,%eax
  40105d:   eb 05                       jmp    401064 <phase_3+0x98>

  40105f:   b8 00 00 00 00              mov    $0x0,%eax                   # >> 7
  401064:   2d 00 02 00 00              sub    $0x200,%eax
  401069:   eb 0a                       jmp    401075 <phase_3+0xa9>
  40106b:   e8 3b 06 00 00              callq  4016ab <explode_bomb>
  401070:   b8 00 00 00 00              mov    $0x0,%eax                   # Save 0 to %eax.
```

```
401075:   3b 44 24 04          cmp    0x4(%rsp),%eax          # Compare %eax with input[1].
401079:   74 05                je     401080 <phase_3+0xb4>   # If input[1] == 0, safe.
40107b:   e8 2b 06 00 00       callq  4016ab <explode_bomb>   # Or die.
401080:   48 8b 44 24 08       mov    0x8(%rsp),%rax          # Restore stack canary.
401085:   64 48 33 04 25 28 00 xor    %fs:0x28,%rax           # Compare.
```

0x401009 명령을 보니 0x4026e0에 점프 테이블이 존재하는 것을 알 수 있다.
gdb를 동원하여 이를 캐보았다.

```
(gdb) x/8gx 0x4026e0
0x4026e0:   0x0000000000401010   0x0000000000401017
0x4026f0:   0x0000000000401023   0x000000000040102f
0x402700:   0x000000000040103b   0x0000000000401047
0x402710:   0x0000000000401053   0x000000000040105f
```

위의 어셈블리에는 이 테이블이 가리키는 곳을 해당 테이블의 인덱스로 표시해 놓았다.

Phase 3이 시작되면 0x4029b0에 위치한 포맷 스트링 "%d %d"을 이용해 stdin으로부터 두 개의 숫자를 입력받는다. 이중 첫번째 숫자에서 43을 뺀 값이 점프테이블의 인덱스로 사용된다.

이때 43을 뺀 값이 음수가 되면 안 되고 7을 넘어서도 안 되므로, 첫 번째 숫자가 가질 수 있는 값은 **43 이상 50 이하**이다.

switch 문을 빠져나오면, $eax가 두 번째 숫자와 같은지 비교한다. 같아야 한다. 7개의 경우 중 답이 될 수 있는 경우가 적어도 하나 이상 존재할 것이다. 7개밖에 되지 않으니 계산해보자.

입력받은 첫번째 숫자	$eax
43	136 − 588 + 745 − 512 + 512 − 512 + 512 − 512 = −219
44	−588 + 745 − 512 + 512 − 512 + 512 − 512 = −355
45	745 − 512 + 512 − 512 + 512 − 512 = 233
46	−512 + 512 − 512 + 512 − 512 = 512
47	512 − 512 + 512 − 512 = 0
48	−512 + 512 − 512 = −512
49	+512 − 512 = 0
50	−512

모두 답이 될 수 있다. 그래도 하나만 선택하자면 숫자가 예쁜 것으로: **47 0**.

3.4. Phase 4

어셈블리를 보자.

```
00000000004010d5 <phase_4>:
 4010d5:   48 83 ec 18          sub    $0x18,%rsp              # Allocate 24 bytes.
 4010d9:   64 48 8b 04 25 28 00 mov    %fs:0x28,%rax           # Save stack canary to %rax.
 4010e0:   00 00
 4010e2:   48 89 44 24 08       mov    %rax,0x8(%rsp)          # Save %rax(value of stack canary) to (%rsp + 8).
 4010e7:   31 c0                xor    %eax,%eax               # Clear %rax.
 4010e9:   48 89 e1             mov    %rsp,%rcx               # Save stack pointer to %rcx. &input[1].
 4010ec:   48 8d 54 24 04       lea    0x4(%rsp),%rdx          # Save stack pointer + 4 to %rdx. &input[0].
 4010f1:   be b0 29 40 00       mov    $0x4029b0,%esi          # Save address of "%d %d" to %rsi.
 4010f6:   e8 45 fb ff ff       callq  400c40 <__isoc99_sscanf@plt> # Call sscanf. Read two integers.
                                                               Input numbers from %rsp + 4.
 4010fb:   83 f8 02             cmp    $0x2,%eax               # Compare %rax with 2.
 4010fe:   75 0b                jne    40110b <phase_4+0x36>   # If %rax != 2, jump and explode.
 401100:   8b 04 24             mov    (%rsp),%eax             # Save (%rsp) to %rax.
 401103:   83 e8 03             sub    $0x3,%eax               # %rax -= 3.
 401106:   83 f8 02             cmp    $0x2,%eax               # Compare %rax with 2.
 401109:   76 05                jbe    401110 <phase_4+0x3b>   # If %rax <= 2, keep going.
 40110b:   e8 9b 05 00 00       callq  4016ab <explode_bomb>   # Or explode.
 401110:   8b 34 24             mov    (%rsp),%esi             # Save (%rsp)(input[1]) to %rsi.
 401113:   bf 08 00 00 00       mov    $0x8,%edi               # Save 8 to %rdi.
 401118:   e8 7d ff ff ff       callq  40109a <func4>          # Call func4. Params: 8, *%rsp.
 40111d:   3b 44 24 04          cmp    0x4(%rsp),%eax          # Compare %rax and (%rsp + 4)(input[0])
 401121:   74 05                je     401128 <phase_4+0x53>   # If %rax == input[0], keep going.
 401123:   e8 83 05 00 00       callq  4016ab <explode_bomb>   # Or explode.
 401128:   48 8b 44 24 08       mov    0x8(%rsp),%rax          # Restore stack canary.
 40112d:   64 48 33 04 25 28 00 xor    %fs:0x28,%rax           # Check canary.
 401134:   00 00
 401136:   74 05                je     40113d <phase_4+0x68>   # If ok, finish function.
 401138:   e8 53 fa ff ff       callq  400b90 <__stack_chk_fail@plt>
 40113d:   48 83 c4 18          add    $0x18,%rsp
 401141:   c3
```

포맷 스트링 "%d %d"을 가지고 숫자를 두 개 받는다. 이때 순서에 유의한다. 첫번째는 $rsp + 4에, 두 번째는 $rsp에 담긴다.

첫 번째 숫자는 **3 이상 5 이하**이어야 한다(0x401106). 8과 두 번째 숫자를 인자로 하여 func4를 호출하는데, 이때 반환값이 첫 번째 숫자와 같아야 한다. 즉, **func4(8, input[1]) == input[0] 이 참**이 되어야 한다.

func4는 어셈블리를 분석하는 대신 brute force 방법을 사용하였다. 그 이유는 다음과 같다.

- 함수 자체는 블랙박스로 놓고 입력과 출력만 분석하여도 문제를 풀 수 있다.
- 실행 파일을 확보한 상황에서 이를 실행하지 못할 이유가 없다.
- 폭탄은 어차피 터지지 않으므로 기회는 무한하다.
- 이미 훌륭한 디버거가 존재한다.
- 인간은 CPU가 아니다.
- 쉽고 빠르다.

gdb로 실행해본 결과는 다음과 같다.

```
(gdb) call func4(0, 1)
$1 = 0
(gdb) call func4(1, 1)
$2 = 1
(gdb) call func4(2, 1)
$3 = 2
(gdb) call func4(3, 1)
$4 = 4
(gdb) call func4(4, 1)
$5 = 7
(gdb) call func4(5, 1)
$6 = 12
(gdb) call func4(6, 1)
$7 = 20
(gdb) call func4(7, 1)
$8 = 33
(gdb) call func4(8, 1)
$9 = 54
(gdb) call func4(9, 1)
$10 = 88
(gdb) call func4(10, 1)
$11 = 143
```

0, 1, 2, 4, 7, 12, 20…은 피보나치 - 1 수열이다. 즉 func4는 아래와 같이 정의된다.

$$fun4(x, y) = (Fibonacci(x + 1) - 1) * y.$$

첫번째 인자 8은 고정되어 있으며 func4(8, 1)은 54이다. func4(8, 2)는 108, func4(8, 3)은 162이다. 따라서 input[1] 〉= 3 && input[1] 〈= 5 && input[0] == 54 * input[1] 이 참이 되도록 하면 된다.

가능한 답은 다음과 같다:
162 3
216 4
270 5

하나만 고르자면 숫자가 예쁜 270 5.

3.5. Phase 5

어셈블리는 간단하지만 조금 까다로운 문제다.

```
0000000000401142 <phase_5>:
  401142:   53                      push   %rbx                    # Save %rbx to stack.
  401143:   48 89 fb                mov    %rdi,%rbx               # Save %rdi to %rbx.
                                                                   In this case it is the address of input string.
  401146:   e8 6e 02 00 00          callq  4013b9 <string_length>  # Measure string length.
  40114b:   83 f8 06                cmp    $0x6,%eax               # Compare returned value with 6.
  40114e:   74 05                   je     401155 <phase_5+0x13>   # If strlen == 6, keep going.
  401150:   e8 56 05 00 00          callq  4016ab <explode_bomb>   # Or explode.
  401155:   48 89 d8                mov    %rbx,%rax               # Save address of input string to %rax.
  401158:   48 8d 7b 06             lea    0x6(%rbx),%rdi          # Save address of input string + 1 to %rdi.
  40115c:   b9 00 00 00 00          mov    $0x0,%ecx               # Save 0 to %ecx.

  Until the end of the string, add *(0x402720 + (4 * (str[i] & 0xf)))
  Total of them must be 0x21(33).

  >>> Loop1 begin
  401161:   0f b6 10                movzbl (%rax),%edx             # Save currently pointing character of
                                                                   input string to %rdx.
  401164:   83 e2 0f                and    $0xf,%edx               # Leave only nibble..
  401167:   03 0c 95 20 27 40 00    add    0x402720(,%rdx,4),%ecx  # Add 0x402720(,%rdx,4) to %rcx.
  40116e:   48 83 c0 01             add    $0x1,%rax               # Add 1 to %rax.
  401172:   48 39 f8                cmp    %rdi,%rax               # Compare %rax with %rdi.
  401175:   75 ea                   jne    401161 <phase_5+0x1f>   # If %rax != %rdi, go to loop begin.
  <<< Loop1 end

  401177:   83 f9 21                cmp    $0x21,%ecx              # Compare %rcx with 0x21(33).
  40117a:   74 05                   je     401181 <phase_5+0x3f>   # If %ecx == 0x21(33), finish function.
  40117c:   e8 2a 05 00 00          callq  4016ab <explode_bomb>
  401181:   5b                      pop    %rbx
  401182:   c3                      retq
```

한 줄의 문자열을 입력받는데, **길이는 6**이어야 한다. 그런 다음 문자열을 이루는 각 문자들에 하위 4비트 마스크를 취한 값을 바이트 오프셋으로 하여 0x402720에 위치한 테이블에 접근하여 그 값을 $ecx에 더한다.

테이블의 내용은 다음과 같다.

```
402720:   02 00 00 00 0a 00 00 00 06 00 00 00 01 00 00 00   ................
402730:   0c 00 00 00 10 00 00 00 09 00 00 00 03 00 00 00   ................
402740:   04 00 00 00 07 00 00 00 0e 00 00 00 05 00 00 00   ................
402750:   0b 00 00 00 08 00 00 00 0f 00 00 00 0d 00 00 00   ................
```

n이 입력 문자열의 i번째 문자에 0x0f 마스크를 취한 값일 때 n에 대한 테이블 대응은 다음과 같다:

```
n       word

0x0:    0x02
0x1:    0x0a
0x2:    0x06
0x3:    0x01

0x4:    0x0c
0x5:    0x10
0x6:    0x09
0x7:    0x03

0x8:    0x04
0x9:    0x07
0xa:    0x0e
0xb:    0x05

0xc:    0x0b
0xd:    0x08
0xe:    0x0f
0xf:    0x0d
```

잠시 코틀린의 힘을 빌려서 요약하자면

```
inputString.length == 6 &&
inputString.map { eachChar ->
        arrayOf(
                0x02, 0x0a, 0x06, 0x01,
                0x0c, 0x10, 0x09, 0x03,
                0x04, 0x07, 0x0e, 0x05,
                0x0b, 0x08, 0x0f, 0x0d
        )[eachChar.toInt() & 0x0f]
}.sum() == 0x21
```

인 것이다!

합이 33(0x21)이 되도록 하는 적절한 조합을 찾아보았다.

값	인덱스
10	1
7	9
6	2
4	8
4	8
2	0
합계 33	

대략 이 정도면 적절할 듯 싶다.

아스키 테이블에서 적절한 문자들을 0x30번대부터 0x70번대까지 5쌍 찾아 이중에서 선택하였다.

	0x00	0x01	0x02	0x08	0x08	0x09
0x30	0	1	2	8	8	9
0x40	@	A	B	H	H	I
0x50	P	Q	R	X	X	Y
0x60	‘	a	b	h	h	i
0x70	p	q	r	x	x	y

선택한 답은 pabxxi이다.

3.6. Phase 6

Phase 6은 어셈블리가 너무 복잡하여 전체 분석은 포기하였다. 대신 직접 실행하면서 분석하였다. 폭탄이 터지면 안되니까(이미 한번 터져서) 뇌관을 제거한 다음에 스텝마다 스택을 분석하였다.

먼저 stdin에서 7개의 숫자를 받은 뒤, 이 숫자들에 **중복이 없으며 지정된 범위(1부터 7) 내**에 있는지 확인한다. 그리고 그 숫자들 각각에 8의 보수를 취한다.

그런 다음 0x6042f0에 위치한 노드들의 주소를 $rsp + 0x20을 목적지 시작 주소로 하여 하나씩 옮기는데, 이때 사용자가 입력한 순서대로 올라간다.

0x6042f0가 가리키는 곳에는 노드들의 정보가 있다.

```
(gdb) x/28wx 0x6042f0
0x6042f0 <node1>: 0x0000033e    0x00000001    0x00604300    0x00000000
0x604300 <node2>: 0x00000147    0x00000002    0x00604310    0x00000000
0x604310 <node3>: 0x000002df    0x00000003    0x00604320    0x00000000
0x604320 <node4>: 0x00000059    0x00000004    0x00604330    0x00000000
0x604330 <node5>: 0x000002f9    0x00000005    0x00604340    0x00000000
0x604340 <node6>: 0x0000004e    0x00000006    0x00604350    0x00000000
0x604350 <node7>: 0x000003ba    0x00000007    0x00000000    0x00000000
```

처음 4바이트는 노드의 값, 세번째 4바이트는 다음 노드의 주소이다.

노드의 주소가 스택에 순서대로 모두 올라가면, 그 순서대로 리스트를 정렬한다. 스택은 리스트를 주어진 순서대로 정렬하기 위한 임시 저장소였던 것이다.

그런 다음 **내림차순으로 정렬**되었는지 확인한다. 즉, 주어진 입력이 노드를 오름차순으로 정렬하도록 해야 하는 것이다. 이때 입력한 숫자들의 **8의 보수로 반전**되는 것에 유의한다.

표로 정리해 보았다.

노드 번호	값	값에 따른 내림차순 순서	정렬된 노드 번호	8의 보수를 취한 값
1	0x33e	2	7	1
2	0x147	5	1	7
3	0x2df	4	5	3
4	0x059	6	3	5
5	0x2f9	3	2	6
6	0x04e	7	4	4
7	0x3ba	1	6	2

답은 1 7 3 5 6 4 2.

3.7. Secret Phase

Secret phase로 가는 길은 〈phase_defused〉에 있다.

```
000000000040184b <phase_defused>:
  40184b:   48 83 ec 78                 sub    $0x78,%rsp
  40184f:   64 48 8b 04 25 28 00        mov    %fs:0x28,%rax
  401856:   00 00
  401858:   48 89 44 24 68              mov    %rax,0x68(%rsp)
  40185d:   31 c0                       xor    %eax,%eax
  40185f:   bf 01 00 00 00              mov    $0x1,%edi
  401864:   e8 38 fd ff ff              callq  4015a1 <send_msg>
  401869:   83 3d 5c 2f 20 00 06        cmpl   $0x6,0x202f5c(%rip)        # 6047cc <num_input_strings>
  401870:   75 6d                       jne    4018df <phase_defused+0x94>

  401872:   4c 8d 44 24 10              lea    0x10(%rsp),%r8
  401877:   48 8d 4c 24 0c              lea    0xc(%rsp),%rcx             # p3 is %rsp + 12.
  40187c:   48 8d 54 24 08              lea    0x8(%rsp),%rdx             # p2 is %rsp + 8.
  401881:   be fa 29 40 00              mov    $0x4029fa,%esi             # p1 is "%d %d %s".
  401886:   bf d0 48 60 00              mov    $0x6048d0,%edi             # p0 is infile?
  40188b:   b8 00 00 00 00              mov    $0x0,%eax
  401890:   e8 ab f3 ff ff              callq  400c40 <__isoc99_sscanf@plt>
  401895:   83 f8 03                    cmp    $0x3,%eax
  401898:   75 31                       jne    4018cb <phase_defused+0x80>  # If input count not 3,
                                                                          just pass the secret phase.
  40189a:   be 03 2a 40 00              mov    $0x402a03,%esi             # "NoOneKnowsMeBomb"
  40189f:   48 8d 7c 24 10              lea    0x10(%rsp),%rdi            # String at %rsp + 16.
  4018a4:   e8 2e fb ff ff              callq  4013d7 <strings_not_equal> # Test string.
  4018a9:   85 c0                       test   %eax,%eax
  4018ab:   75 1e                       jne    4018cb <phase_defused+0x80>  # If not matched "NoOneKnowsMeBomb",
                                                                          just pass the secret phase.
  4018ad:   bf 58 28 40 00              mov    $0x402858,%edi             # "Curses! ..blahblah"
  4018b2:   e8 b9 f2 ff ff              callq  400b70 <puts@plt>
  4018b7:   bf 80 28 40 00              mov    $0x402880,%edi
  4018bc:   e8 af f2 ff ff              callq  400b70 <puts@plt>
  4018c1:   b8 00 00 00 00              mov    $0x0,%eax
  4018c6:   e8 23 fa ff ff              callq  4012ee <secret_phase>      # Get into the secret phase!
  4018cb:   bf b8 28 40 00              mov    $0x4028b8,%edi             # "Congratulations!.. blahblah"
  4018d0:   e8 9b f2 ff ff              callq  400b70 <puts@plt>
  4018d5:   bf e8 28 40 00              mov    $0x4028e8,%edi             # "Your instructor ...blahblah"
  4018da:   e8 91 f2 ff ff              callq  400b70 <puts@plt>

  4018df:   48 8b 44 24 68              mov    0x68(%rsp),%rax
  4018e4:   64 48 33 04 25 28 00        xor    %fs:0x28,%rax
  4018eb:   00 00
  4018ed:   74 05                       je     4018f4 <phase_defused+0xa9>
  4018ef:   e8 9c f2 ff ff              callq  400b90 <__stack_chk_fail@plt>
  4018f4:   48 83 c4 78                 add    $0x78,%rsp
  4018f8:   c3                          retq
```

Secret phase를 호출하는 명령은 0x4018c6에 위치하는데, phase 6까지 도달하지 않으면 실행되지 않는다. 0x401872부터는 phase 6을 풀어야만 도달할 수 있기 때문이다.

만약 phase 6까지 해결한 경우, 0x6048d0에 위치한 문자열을 가지고 phase 4를 다시 푸는데, 이때 포맷 스트링으로 “%d %d”을 사용하지 않고 “%d %d %s”를 사용한다. 이때 해당 위치에는 phase 4를 풀 때에 입력했던 문자열이 저장되어 있다.

```
(gdb) x/24bc 0x6048d0
0x6048d0 <input_strings+240>:  50 '2'  55 '7'  48 '0'  32 ' '  53 '5'  32 ' '  78 'N'  111 'o'
0x6048d8 <input_strings+248>:  79 'O'  110 'n' 101 'e' 75 'K'  110 'n' 111 'o' 119 'w' 115 's'
0x6048e0 <input_strings+256>:  77 'M'  101 'e' 66 'B'  111 'o' 109 'm' 98 'b'  0 '₩000'       0 '₩000'
```

(이미 답을 알고 적어넣은 상태에서 dump했기 때문에 두 숫자 다음에 문자열이 들어 있는 모습이다. 원래는 아직 모른다.)

즉, secret phase에 도달하려면 phase 4의 답에 어떤 문자열을 하나 추가한 뒤 phase 6까지 풀어야 하는 것이다. 그 문자열은 0x402a03에 위치한 "NoOneKnowsMeBomb"이다.

이렇게 도달하고 나면 "잘 찾았네^^ 하지만 찾는거랑 푸는건 얘기가 다르지^^" 라며 약을 올린다.

아래는 secret phase의 어셈블리이다.

```
00000000004012ee <secret_phase>:
  4012ee:   53                      push   %rbx
  4012ef:   e8 31 04 00 00          callq  401725 <read_line>
  4012f4:   ba 0a 00 00 00          mov    $0xa,%edx          # base.
  4012f9:   be 00 00 00 00          mov    $0x0,%esi          # endptr.
  4012fe:   48 89 c7                mov    %rax,%rdi          # str.
  401301:   e8 1a f9 ff ff          callq  400c20 <strtol@plt>
  401306:   48 89 c3                mov    %rax,%rbx          # The result.
  401309:   8d 40 ff                lea    -0x1(%rax),%eax    # Subtract 1 from %rax.
  40130c:   3d e8 03 00 00          cmp    $0x3e8,%eax        # Compare %rax with 0x3e8.
  401311:   76 05                   jbe    401318 <secret_phase+0x2a>  # If 0 <= %rax <= 0x3e8, keep going.
  401313:   e8 93 03 00 00          callq  4016ab <explode_bomb>       # Or explode.
  401318:   89 de                   mov    %ebx,%esi          # The number as p1.
  40131a:   bf 10 41 60 00          mov    $0x604110,%edi     # Address of 0x0000000000000024 as p0.
  40131f:   e8 8c ff ff ff          callq  4012b0 <fun7>      # Call fun7.
  401324:   85 c0                   test   %eax,%eax          # Test the result.
  401326:   74 05                   je     40132d <secret_phase+0x3f>  # If the returned value is zero, keep going.
  401328:   e8 7e 03 00 00          callq  4016ab <explode_bomb>       # Or explode.
  40132d:   bf a8 26 40 00          mov    $0x4026a8,%edi     # "Wow! ...blahblah"
  401332:   e8 39 f8 ff ff          callq  400b70 <puts@plt>
  401337:   e8 0f 05 00 00          callq  40184b <phase_defused>
  40133c:   5b                      pop    %rbx
  40133d:   c3                      retq
```

일단 숫자를 하나 입력받는데, 문자열로 받은 다음에 strtol을 이용해 정수로 바꾼다. 이 수는 **1 이상 1001 이하**이어야 한다. 그런 다음 0x604110과 입력받은 수를 인자로 하여 fun7을 호출한다. fun7의 반환값이 0이면 된다. 약이 올랐기 때문에 fun7의 어셈블리를 분석하지는 않을 것이다.

대신 brute force 방법을 사용하였다. 시도한 결과 아주 간단하게 원하는 값(0)을 얻어낼 수 있었다.

```
(gdb) call fun7(0x604110, 0)
$1 = -16
(gdb) call fun7(0x604110, 1)
$2 = 0
(gdb) call fun7(0x604110, 2)
$3 = -8
(gdb) call fun7(0x604110, 3)
$4 = -8
```

답은 1.

4. 마무리

재미난 것 알려주셔서 감사합니다. 😊