CSED211: Bomblab report / 20230784 jaewonyu 유재원 / bomb13

1. 개요

   본 과제는 바이너리를 분석하여 적당한 입력을 찾는 과정에서 바이너리 및 컴퓨터 시스템에 대한 이해를 높이며 리버스 엔지니어링을 경험하는데 그 의의가 있다.

2. 바이너리 분석

   이후 설명에서 사용할 '라인'이라는 단어를 사용하는데, 이는 해당 함수의 시작 instruction을 0이라 했을 때 몇 번째 instruction인지를 의미한다.

   a. phase_1
   sol: I am just a renegade hockey mom.

   먼저 전반적인 바이너리를 분석해보자. main 함수를 disass해 확인해 보면 각 phase가 다음과 같이 구성됨을 알 수 있다.

```
mov     $0x4024d8,%edi
callq   0x400b40 <puts@plt>
callq   0x401681 <read_line>
mov     %rax,%rdi
callq   0x400ef0 <phase_1>
callq   0x4017a7 <phase_defused>
```

   문자열을 출력하는 puts 함수는 분석할 필요가 없어보인다. read_line함수의 내용을 파악해야 할 것 같다. gdb를 이용해 동적으로 read_line 함수의 동작을 확인해보면 사용자의 입력을 받음을 알 수 있다. read_line 함수 실행 이후 rax 레지스터의 값을 참조하면 사용자의 입력을 확인할 수 있다. 따라서 phase_1 함수는 사용자의 입력의 주소를 인자로 받아 실행된다. 이는 앞으로의 phase 에서도 동일하게 적용된다. 사용자의 입력의 주소를 앞으로 user_input 이라 하자.

   다음은 phase_1함수의 disass결과 중 일부이다.

```
mov     $0x402560,%esi
callq   0x4012ee <strings_not_equal>
test    %eax,%eax
je      0x400f07 <phase_1+23>
callq   0x401554 <explode_bomb>
```

   strings_not_equal(user_input, 0x402560)을 실행한다. 함수의 이름으로 추측해보면 두 인자에 담긴 문자열이 다르면 1을 반환하는 함수일 것이라 추측된다. 그렇다면 0x402560주소에 담긴 문자열을 확인하고, 그 문자열을 입력으로 주면 해결될 것으로 보인다. 0x402560주소에 담긴 문자열을 확인해본 결과 "I am just a renegade hockey mom." 이라는 문자열이 담겨있었다. 그것을 입력으로 주니 phase_2 로 넘어가짐을 확인할 수 있었다.

   위 문제 해결 과정에서 strings_not_equal 함수의 내부 동작을 살펴보지 않았으나, 함수의 이름을 통해 예상한대로 동작하기에 내부 동작을 확인할 필요는 없을 것으로 보인다.

b. phase_2
sol: 1 2 4 8 16 32

함수 호출시 스택에 적당한 공간을 만드는 것을 확인할 수 있다. 그것의 주소를 buf라고 하면 read_six_numbers(user_input, buf)를 호출함을 확인할 수 있다. 해당 함수를 disass해서 확인해보면 sscanf(user_input, 0x403328, buf, buf+4, buf+8, buf+12, buf+16, buf+20)를 호출함을 확인할 수 있다. 0x403328주소의 값은 "%d %d %d %d %d %d" 라는 문자열이니 read_six_numbers 함수는 user_input의 값을 띄어쓰기 기준 6개의 숫자로 나누어 저장하는 함수임을 알 수 있다. buf를 int array라고 생각하면 될 것 같다.

이후 sscanf 의 반환값이 5보다 크면 점프, 그렇지 않다면 explode_bomb 함수가 실행됨을 확인할 수 있다. man sscanf를 통해 확인해보면 sscanf 함수의 반환값은 성공적으로 서식에 맞게 읽은 수임을 확인할 수 있다. 즉 숫자 6개(혹은 그 이상)를 입력해야 한다.

이후 buf[0] 이 1인지를 확인한 후, buf[1], buf[7](편의적인 표현으로, *buf[6]+4를 의미한다.)를 레지스터에 저장해둔 뒤 27번 라인으로 점프한다. 이후 buf내의 각 값들에 대해 각 값이 이전 값의 2배인지를 확인한다. 확인할 값이 buf[7] 이라면 반복을 종료하는 것이다.

즉 phase_2는 6개의 정수를 입력받아 1부터 시작해 공비가 2인 등비수열인지를 확인한다고 볼 수 있다.

c. phase_3
sol: 5 -944

sscanf(user_input, "%d %d", 0xc(%rsp), 0x8(%rsp))를 호출하고, 그 반환값이 2 이상일 때 통과함을 먼저 확인할 수 있다. 0xc(%rsp)의 값을 p1, 0x8(%rsp)의 값을 p2라 하자.

39번 라인부터 확인해 보면 만약 p1이 7보다 크다면 폭탄이 터진다. 그렇지 않다면 p1의 값을 eax에 저장하고, *0x4025c0(,%rax,8) 로 점프함을 확인할 수 있다. 이때 p1의 값에 따라 점프할 주소가 결정된다. 점프 테이블은 다음과 같다.

```
(gdb) x/16x 0x4025c0
0x4025c0:    0x00400f93    0x00000000    0x00400f8c    0x00000000
0x4025d0:    0x00400f9d    0x00000000    0x00400fa9    0x00000000
0x4025e0:    0x00400fb5    0x00000000    0x00400fc1    0x00000000
0x4025f0:    0x00400fcd    0x00000000    0x00400fd9    0x00000000
```

p1을 임의로 5로 골라보았다. 0x00400fc1(110번 라인)으로 점프하니, 이후 동작을 확인해보자.

```
<+110>:  mov   $0x0,%eax
<+115>:  sub   $0x3b0,%eax
<+120>:  jmp   0x400fd2 <phase_3+127>
<+127>:  add   $0x3b0,%eax
<+132>:  jmp   0x400fde <phase_3+139>
<+139>:  sub   $0x3b0,%eax
<+144>:  jmp   0x400fef <phase_3+156>
```

eax 를 0부터 시작해 944를 빼고 더하고 빼주고 156번 라인으로 점프함을 확인할 수 있다. 이후 p1이 5보다 크거나 p2가 eax보다 다를 경우 폭탄이 터짐을 확인할 수 있다. 위에서 eax의 결과는 -944이므로 p2를 -944로 넣어주면 phase_3가 해결됨을 확인할 수 있다.

d. phase_4
sol: 40 2 SecretPhase

solution 의 SecretPhase 부분은 일단은 무시하고 설명을 진행한다. phase_4 함수를 disass 해보면 마찬가지로 sscanf(user_input, "%d %d", 0x8(%rsp), 0xc(%rsp))를 호출한다. ‘format 에 맞게 파싱된 수’가 2가 아니면 터지는데, ‘SecretPhase’는 format 에 맞지 않아 무시되므로 괜찮다. 마찬가지로 각 파싱 결과를 p1, p2 라 하자. p2가 4 이하여야 폭탄이 터지지 않는다. p2를 임의로 2라 하고 분석을 해보자. func4(6, p2)를 호출하고, 그 결과가 p1과 다르면 터진다. func4함수를 살펴봐 func4(6, 2)의 결과를 구해 p1으로 넣으면 될 것으로 보인다. func4 함수를 분석해보자. 편의상 첫 인자를 argv[1], 두 번째 인자를 argv[2] 라고 하자.

```
<+4>:     mov    %edi,%ebx
<+6>:     test   %edi,%edi
<+8>:     jle    0x401034 <func4+46>
```

첫 인자가 0 이하라면 0을 반환한다.

```
<+10>:    mov    %esi,%ebp
<+12>:    mov    %esi,%eax
<+14>:    cmp    $0x1,%edi
<+17>:    je     0x401039 <func4+51>
```

첫 인자가 1이라면 argv[2]를 반환한다. rbx=argv[1], rbp=argv[2]로 설정한다.

```
<+19>:    lea    -0x1(%rdi),%edi
<+22>:    callq  0x401006 <func4>
```

func4(argv[1]-1, argv[2])를 호출한다.

```
<+27>:    lea    (%rax,%rbp,1),%r12d
```

r12 = func4(argv[1]-1, argv[2]) + argv[2]

```
<+31>:    lea    -0x2(%rbx),%edi
<+34>:    mov    %ebp,%esi
<+36>:    callq  0x401006 <func4>
```

func4(argv[1]-2, argv[2])를 호출

```
<+41>:    add    %r12d,%eax
<+44>:    jmp    0x401039 <func4+51>
```

func4(argv[1]-1, argv[2]) + argv[2] + func4(argv[1]-2, argv[2]) 를 반환한다.

func4(6, 2)의 값을 구해야 하므로 func4(0, 2)부터 차례차례 구해주면 된다.

func4(0, 2) = 0, func4(1, 2) = 2, func4(2, 2) = 4, func4(3, 2) = 8,

func4(4, 2) = 14, func4(5, 2) = 24, func4(6, 2) = 40. 따라서 p1에 40, p2에 2를 넣으면 해결된다.

e. phase_5
sol: $"%))-

string_length(user_input)이 6이어야 한다. 함수의 이름으로 보아 user_input의 길이를 반환할 것이라 추측할 수 있다. 동적 디버깅을 통해 실제로 그렇게 동작함을 확인할 수 있으니, 추가적인 분석 없이 이후 동작을 분석하도록 하겠다.

```
<+19>:  mov     $0x0,%eax
<+24>:  mov     $0x0,%edx
<+29>:  movzbl (%rbx,%rax,1),%ecx
<+33>:  and     $0xf,%ecx
<+36>:  add     0x402600(,%rcx,4),%edx
<+43>:  add     $0x1,%rax
<+47>:  cmp     $0x6,%rax
<+51>:  jne     0x4010ac <phase_5+29>
```

user_input의 각 문자열과 0xf를 and 연산한 결과를 index로 사용해 0x402600주소의 table의 값을 쭉 더하고, 그 결과가 0x6이면 통과함을 확인할 수 있다. 임의로 아무 문자 6개를 가져온 뒤, table을 확인하여 계산해보고 글자들을 조금씩 조정하여 solution을 찾아냈다. man ascii만 참고하면 손으로도 어렵지 않게 계산할 수 있다. (또한 0xf와 and 연산은 mod 16과 같은 연산임을 이용하면 더욱 쉽게 계산할 수 있다.) 리버스 엔지니어링 경험이 있다면 자주 봤을 유형이다.

f. phase_6
sol: 1 6 2 5 3 4

read_six_numbers(user_input, 0x30(%rsp))를 호출한다. 0x30(%rsp)를 inputs[6] 이라고 이름짓자.

r13 = inputs
r12 = 0
rbp = inputs[0]

위의 동작을 차례대로 실행한다.

이후 동작을 요약하면 inputs의 각각의 숫자에 대하여, 6 이하임을 확인하고 그 이후의 숫자들과 대조해 같은 숫자가 있으면 폭발한다. 즉 inputs이 1~6에 대한 순열임을 확인하는 로직이다.

위의 요약을 풀어 서술해보겠다. r12는 검사 대상의 index이다. count라고 부르자. ebx는 대조 대상의 index이다. check라고 부르자.

inputs[0] 이 6 초과라면 터진다. count에 1을 더한게 6이라면, 즉 순열 여부 확인이 끝났다면 esi를 0으로 설정하고 134번 라인으로 점프한다. 그렇지 않다면 현재 count를 check에 저장하고, check를 1씩 더해가며 대조를 진행한다. check가 5 이하라면 해당 count에 대한 대조를 계속하고, check가 6이라면 inputs[1]를 rbp로 설정하고 위 작업을 반복한다. 순열 검사 로직에 대한 분석은 이쯤이면 충분한 것 같다. 이제 user_input 이 1 6 2 5 3 4 순서라 가정하고 분석을 이어가자. count가 6이 되면 134번 라인으로 점프한다. 이 때 esi는 0이다.

이하 약 5줄은 분석을 한 과정을 서술하였으나, 정돈된 해결 과정은 아니다. 정돈된 해결 과정은 0x6042f0주소의 값을 보는 것부터 시작한다.

inputs[0](= 1)의 값을 ecx에 넣고 1과 비교한다. 1 이하기에 115번 라인으로 점프한다.

0x6042f0를 edx에 넣고, (%rsp,%rsi,2)에 넣는다. rsi가 0이므로 rsp에 0x6042f0를 넣는다. rsi를 4 더하고, 24과 비교해 같다면 155번 라인으로 점프한다. 이 경우 점프하지 않는다.

inputs[1](= 6) 의 값을 ecx에 넣고 이제는 115번 라인으로 점프하지 않는다.

eax에 1을 넣고, edx에 0x6042f0를 넣는다. 102라인으로 점프한다.

edx + 8을 참조해 rdx에 넣는다. rdx에는 0x00604300이 들어간다. eax에 1을 더하고(= 2) ecx (6)랑 비교한다. 다르므로 102번 라인으로 점프한다.

슬슬 코드를 한 줄 씩 읽어가며 분석하기 어려우므로 0x6042f0 주소의 값을 보며 분석해보자.

```
0x6042f0 <node1>:    0x0000005d    0x00000001    0x00604300    0x00000000
0x604300 <node2>:    0x00000209    0x00000002    0x00604310    0x00000000
0x604310 <node3>:    0x0000035c    0x00000003    0x00604320    0x00000000
0x604320 <node4>:    0x0000037e    0x00000004    0x00604330    0x00000000
0x604330 <node5>:    0x00000251    0x00000005    0x00604340    0x00000000
0x604340 <node6>:    0x00000112    0x00000006    0x00000000    0x00000000
```

node를 발견할 수 있었다. 각 노드는 16바이트로, 첫 4바이트에 값, 이후 4바이트에 이름, 이후 8바이트에 다음 node의 주소를 갖는 linked list라고 볼 수 있겠다. 이제 이 linked list에 대한 개념을 가지고 위에서 했던 분석을 다시 해보자.

rsi의 값은 4씩 변하며, mov 0x30(%rsp,%rsi,1),%ecx 동작에서 inputs의 값을 ecx에 넣음을 확인할 수 있다. 즉 rsi는 inputs의 index의 의미를 가짐을 확인할 수 있다.

inputs의 값을 순서대로 ecx에 넣으며 특정 로직을 반복한다. 124~132번 라인을 보면 rsi가 24, 즉 6번 반복하면 155번 라인으로 점프하며 반복이 끝남을 확인할 수 있다. 만약 inputs[index]의 값이 1일 때와 그 외의 처리 로직이 다름을 확인할 수 있다.

만약 inputs[index]가 1이라면 (%rsp,%rsi,2) 에 node1의 주소를 저장한다..

그렇지 않다면 eax를 1, edx를 node1의 주소로 설정한 후 rdx를 rdx+8의 값으로 설정한다. 이는 다음 node의 주소임을 우리는 알아냈다. eax를 1 더해 ecx와 비교하여 다르면 102번, 같으면 120번으로 점프한다. 즉 우리의 input번째 node로 건너간 뒤에 120번으로 점프한다. 이후 아까와 같이 (%rsp,%rsi,2) 주소에 건너간 node의 주소를 넣어주는 것이다.

즉 rsp, rsp+8, …에다가 inputs 의 값 순서대로 노드의 주소를 적어둔 것이다. 1 6 2 5 3 4 라는 우리의 입력에 따르면 rsp 부터 그 구조는 다음과 같다. 아래의 결과는 gdb로도 확인할 수 있다.

```
rsp: node1    rsp+8: node6    rsp+16: node2    rsp+24: node5    rsp+32: node3    rsp+40: node4
```

이후 동작은 linked list의 연결 순서를 우리의 permutation에 따라 변경해준다.

```
<+155>:    mov     (%rsp),%rbx
<+159>:    lea     0x8(%rsp),%rax
<+164>:    lea     0x30(%rsp),%rsi
<+169>:    mov     %rbx,%rcx
```

이후 반복 로직에서 rax가 8씩 더해지다가 rsi와 같아지면 반복을 종료한다.

rcx에 현재 처리중인 노드, rax에 그 다음 노드에 대한 정보가 담긴다고 보면 된다.

```
<+172>:    mov     (%rax),%rdx
<+175>:    mov     %rdx,0x8(%rcx)
<+179>:    add     $0x8,%rax
<+183>:    cmp     %rsi,%rax
<+186>:    je      0x401192 <phase_6+193>
```

```
<+188>:    mov      %rdx,%rcx
<+191>:    jmp      0x40117d <phase_6+172>
<+193>:    movq     $0x0,0x8(%rdx)
```

먼저 그 다음 노드의 주소를 현재 처리중인 노드의 다음 주소로 넣는다. 이후 rax도 8 더해주고 rcx도 다음 노드로 넘겨주며 반복한다. 언급했듯 rax가 rsi가 되면 종료한다. 마무리로 가장 마지막 노드의 다음 노드는 0을 가리키게 해주자.

이제 사용자의 입력에 따라서 linked list를 재배열하였다. 다음 로직을 분석해보자.

```
<+201>:    mov      $0x5,%ebp
<+206>:    mov      0x8(%rbx),%rax
<+210>:    mov      (%rax),%eax
<+212>:    cmp      %eax,(%rbx)
<+214>:    jle      0x4011ae <phase_6+221>
<+216>:    callq    0x401554 <explode_bomb>
<+221>:    mov      0x8(%rbx),%rbx
<+225>:    sub      $0x1,%ebp
<+228>:    jne      0x40119f <phase_6+206>
```

ebp를 5로 설정해주고, 5씩 빼가며 반복문을 돌린다. 1 빼줬을 때 0이 될 때까지 돌아간다. 총 5번 돌아감을 어렵지 않게 확인할 수 있다. rbx는 155번 라인에 의해 첫 노드를 가리킨다.

eax를 다음 노드의 값으로 설정하고, rbx의 값하고 비교한다.

(cmp 다음노드값 이번노드값) 에서 이번노드값이 더 작거나 같아야 폭탄이 터지지 않는다. 이를 반복한다. 즉 linked list가 오름차순이면 문제가 해결됨을 확인할 수 있다. 즉 우리가 원하는 input은 linked list를 오름차순으로 정렬시키는 input이었던 것이다. phase_6을 해결했다.

g. secret_phase
sol: 35

phase_defused 함수를 재미삼아 disass 해보면 신기한 부분을 발견할 수 있다.

```
<+14>:     cmpl     $0x6,0x202fe0(%rip)        # 0x60479c <num_input_strings>
<+21>:     jne      0x40182b <phase_defused+132>
(중략)
<+38>:     mov      $0x40337e,%esi
<+43>:     mov      $0x6048b0,%edi
<+48>:     mov      $0x0,%eax
<+53>:     callq    0x400c30 <__isoc99_sscanf@plt>
<+58>:     cmp      $0x3,%eax
<+61>:     jne      0x401817 <phase_defused+112>
<+63>:     mov      $0x403387,%esi
<+68>:     lea      0x10(%rsp),%rdi
<+73>:     callq    0x4012ee <strings_not_equal>
<+78>:     test     %eax,%eax
<+80>:     jne      0x401817 <phase_defused+112>
(중략)
<+102>:    mov      $0x0,%eax
<+107>:    callq    0x401200 <secret_phase>
```

phase가 넘어가는 사이에 이런 은밀하고 비밀스러운 검증 과정이 있음을 확인할 수 있다.

num_input_strings의 경우 gdb를 이용해 심심할때마다 값을 찍어 본 결과 read_line시마다 1씩 증가함을 확인할 수 있다. 즉 38번 라인은 6페이즈가 끝났을 때 발동된다. 0x40337e 주소에는

"%d %d %s"가, 0x6048b0 주소에는 phase_4 의 입력이 들어있음을 이 순간에 gdb를 이용해 확인할 수 있다. 만약 sscanf에 의해 성공적으로 읽힌게 3개라면 63번 라인이 실행된다. 0x403387 주소에 담긴 문자열이 "SecretPhase"이다. 0x10(%rsp)가 sscanf에 의해 읽힌 3번째 문자열일 것이라는 의심은 합리적이다. 또한 그것을 동적 디버깅을 통해 확인할 수 있다. 이제 secret_phase 함수에 진입하자.

```
<+1>:    callq   0x401681 <read_line>
<+6>:    mov     $0xa,%edx
<+11>:   mov     $0x0,%esi
<+16>:   mov     %rax,%rdi
<+19>:   callq   0x400c00 <strtol@plt>
<+24>:   mov     %rax,%rbx
<+27>:   lea     -0x1(%rax),%eax
<+30>:   cmp     $0x3e8,%eax
<+35>:   jbe     0x40122a <secret_phase+42>
<+37>:   callq   0x401554 <explode_bomb>
```

read_line 함수를 호출 후 strtol(%rax, 0, 10)을 호출한다. 사용자의 입력을 10진수로 해석해 long 형으로 바꾸어 rax레지스터에 저장할 것이다. 그것을 rbx 레지스터로 옮기고, rax – 1이 0x3e8보다 작아야 한다. long형 사용자 입력을 user_input이라 하자.

```
<+42>:   mov     %ebx,%esi
<+44>:   mov     $0x604110,%edi
<+49>:   callq   0x4011c2 <fun7>
<+54>:   cmp     $0x6,%eax
<+57>:   je      0x401240 <secret_phase+64>
<+59>:   callq   0x401554 <explode_bomb>
```

이후 fun7(0x604110, user_input)을 실행하고, 그 결과가 6이라면 통과할 수 있다. fun7 함수를 분석해 func7의 결과가 6이 나오도록 하는 입력을 찾으면 됨을 확인하였다. 이제 func7 함수를 disass하여 분석해보도록 하겠다.

```
<+4>:    test    %rdi,%rdi
<+7>:    je      0x4011f6 <fun7+52>
```

만약 rdi가 0이라면 52번 라인으로 가 0xffffffff (-1로 보아도 좋다.)를 반환한다.

```
<+9>:    mov     (%rdi),%edx
<+11>:   cmp     %esi,%edx
<+13>:   jle     0x4011de <fun7+28>
```

rdi를 참조해 그 값을 edx에 넣고, user_input과 비교한다. 참조된 값이 user_input보다 더 작거나 같다면 28번 라인으로 점프한다. 크다면 15번 라인부터 쭉 진행한다.

```
<+15>:   mov     0x8(%rdi),%rdi
<+19>:   callq   0x4011c2 <fun7>
<+24>:   add     %eax,%eax
<+26>:   jmp     0x4011fb <fun7+57>
```

fun7(rdi +8, user_input) 을 호출하고, 그 결과를 두배해 반환한다.

```
<+28>:   mov     $0x0,%eax
<+33>:   cmp     %esi,%edx
<+35>:   je      0x4011fb <fun7+57>
```

참조된 값이 user_input과 같다면 0을 반환한다.

```
<+37>:   mov     0x10(%rdi),%rdi
<+41>:   callq   0x4011c2 <fun7>
<+46>:   lea     0x1(%rax,%rax,1),%eax
<+50>:   jmp     0x4011fb <fun7+57>
```

```
<+52>:    mov    $0xffffffff,%eax
<+57>:    add    $0x8,%rsp
```
이제 참조된 값이 더 작은 경우이다. fun7(rdi + 16) * 2 + 1을 반환한다.

데이터 구조를 들었다면 위 과정이 BST에서 node를 찾는 알고리즘이라는 것을 어렵지 않게 생각해낼 수 있을것이다. 따라서 최초로 fun7 함수를 호출할 때 첫 번째 인자로 들어간 0x604110라는 주소는 BST의 head의 주소임을 쉽게 생각할 수 있다.

0x604110 주소를 확인해 보면 BST의 node는 첫 8바이트는 value, 그다음 8바이트는 left, 그다음 8바이트는 right, 그다음 8바이트는 null인 24바이트짜리 구조체임을 확인할 수 있다.

BST를 직접 노트에 그려 6이 나오도록 하는 결과를 찾아보았다. 6은 짝수이므로 왼쪽으로 내려가고, 3은 홀수이므로 오른쪽으로 내려간다. 1도 홀수이므로 오른쪽으로 한 번 내려가니 그 값은 0x23, 즉 35였다. 이로써 모든 폭탄 해제를 완료하였다.

3. 결론

리버스 엔지니어링을 통해 바이너리를 분석하며, 그 과정에서 디버깅 도구에 친숙해지고 시스템에 대한 이해를 높일 수 있었다. 특히 모든 바이너리를 분석하기 보다는 상황에 따라서 특정 루프문 혹은 함수의 내용을 예상해보고, 그 예상한 내용을 동적 디버깅으로 확인해 봄으로써 정적 분석만 이용하는 것 보다 훨씬 수월한 분석이 가능함도 확인할 수 있었던, 재미있는 과제였다.