**CSED211 컴퓨터SW시스템개론**

**Lab Assignment #4: The Attack Lab:**

**Understanding Buffer Overflow Bugs**

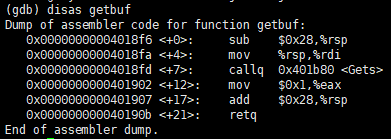
**20180551**

**컴퓨터공학과**

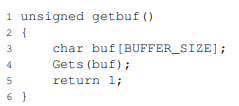
**이준석**

**[1번 문제]**

getbuf에서 버퍼 오버 플로우를 발생시켜서 touch1 함수를 호출하도록 하는 문제이다.

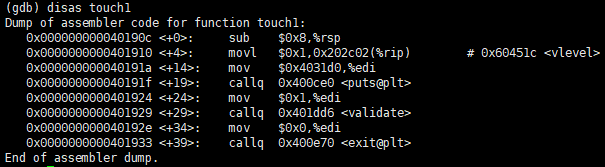


getbuf를 디스어셈블해보면 가장 처음 0x28 만큼 스택 메모리에 공간을 만드는 것을 확인할 수 있다.



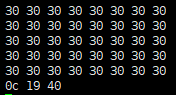
즉, 여기에서 BUFFER\_SIZE는 40 바이트만큼(0x28은 십진수로 40이므로)이라는 것을 알 수 있다.

따라서 버퍼 오버 플로우를 발생시켜 touch1을 호출하려면 40바이트만큼의 아무 값을 채워 넣고 touch1 함수의 시작 주소를 리틀 엔디안으로 적어 넣으면 된다.



touch1의 시작 주소는 0x40190c이다.

따라서 sol1.hex는 다음과 같아진다.



**[2번 문제]**

|  |
| --- |
| return address |
|  |
|  |
|  |
|  |
|  |

cookie에 들어있는 16진수와 같은 값을 인자로서 전달해야한다. 그러기 위해서 rdi에 cookie와 같은 16진수를 대입하는 과정이 필요하다.

<그림1>

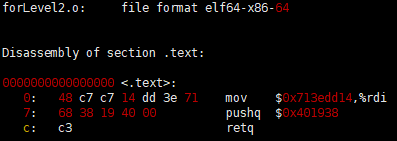
getbuf 함수에서의 대략적인 스택 메모리의 모습이다. 가장 아래에서부터 5칸이 buf배열의 자리이다. return address 자리에 가장 아래 칸의 주소를 저장하고, buf 배열 자리에 rdi에 cookie 값을 대입하는 어셈블리 코드와 touch2의 시작 주소를 스택 메모리에 넣는 어셈블리 코드와 ret가 쓰여져야 한다.

return address 자리에 그림1의 가장 아래 칸의 주소를 저장하는 이유는 getbuf가 종료될 때 우리가 실행시키고 싶은 어셈블리 코드의 시작 주소로 가서 그 코드를 실행하도록 만들기 위해서이다.

  
<그림2>

   
<그림3>

그림3은 cookie값이다. 우리가 실행시키고 싶은 어셈블리 코드는 그림2와 같다. 왜냐하면 cookie의 값을 rdi에 넣어야 touch2 함수의 인자로 cookie의 값을 활용할 수 있고 touch2를 스택 메모리에 push해야 우리가 이후에 ret명령어를 실행하면서 touch2 함수를 호출할 수 있기 때문이다.

  
<그림4>

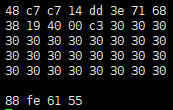
  
<그림5>

그림2에 대응하는 머신 코드는 그림4와 같다. 그림4에 있는 머신 코드를 buf에 대입해놓고 그 부분의 주소를 rsp를 통해 알아내서 그 주소를 return address 자리에 넣으면 된다.

우리의 머신 코드의 주소는 rsp – 0x28이므로 rsp를 확인해야한다.





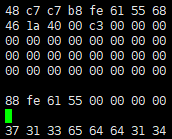
0x5561feb0 – 0x28 = 0x5561fe88 이다. 따라서 우리가 삽입한 머신 코드의 시작 주소는 0x5561fe88이 된다. 따라서 리틀 엔디안 형식으로 return address 자리에 넣어 주면 된다.

그림5가 우리가 최종적으로 입력해주어야 하는 것이다.

**[3번 문제]**

기본적인 골자는 2번 문제와 같다.

다만 쿠키 값과 같은 값의 문자열(아스키 코드로 쓰여짐)을 스택 메모리에 저장해두고 그것의 주소를 따로 push해주는 것이 차이점이다. 주소를 push하는 이유는 c언어로 나타내어진 코드를 확인해보면 char\*를 인자로 받고 있기 때문이다.

  
<그림 6>

  
<그림 7>

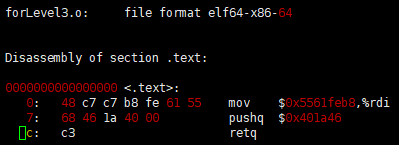
  
<그림 8>

그림 6에서 가장 아래 부분은 나의 쿠키 값을 아스키 코드 표를 참조하여 hex 값으로 나타낸 것이다.

그림 7은 내가 삽입할 어셈블리 코드를 보여준다.

그림 8은 내가 삽입할 어셈블리 코드를 그에 대응하는 바이너리 코드로 변환한 결과이다. 이 바이너리 코드를 스택 메모리에 삽입하면 된다.

그리고 다시 그림 6에서 밑에서 2번 째 줄 부분이 return address인데, 이것은 내가 삽입할 어셈블리 코드의 주소를 나타낸다.

그림6과 같이 입력을 주었을 때 결과적으로 문제가 풀리게 된다.

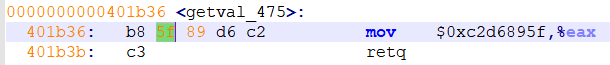
**[4번 문제]**

2번 문제를 ROP 방식을 이용해서 푸는 문제이다.

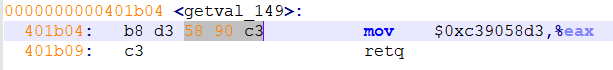
4번 문제와 5번 문제를 풀기 위해서는 start\_farm부터 end\_farm까지의 코드를 바이너리 코드로 표시할 필요가 있다. 우리가 이용할 수 있는 gadget을 찾아야 하기 때문이다.

readme를 읽어보면 2개의 gadget을 이용해서 풀 수 있다고 나와 있다. 그리고 cookie를 스택 메모리 어떤 부분에 저장해 놓고 그것을 rdi에 옮겨야 한다는 것을 알 수 있다.

즉, 우선적으로 알아볼 것은 popq %rdi에 대응하는 gadget이 있는지 알아보아야 한다.

  
<그림 9>

바이너리 코드 전체를 펼쳐 놓고 찾아보면 popq %rdi에 대응하는 5f가 단 하나 존재함을 알 수 있다. 그것이 바로 그림 9인데, 이것은 ret에 대응하는 c3가 바로 뒤에 붙어 있지 않아서 사용할 수 없다. 따라서 어떤 레지스터에 pop을 해놓고 그 값을 mov를 통해서 rdi로 옮겨야 한다는 것을 알 수 있다. 그리고 그래서 2개의 gadget을 이용해서 푸는 것이었다는 사실 또한 알 수 있다.

  
<그림 10>

58 90 c3를 찾을 수 있다. 90은 아무 것도 하지 않는 명령어라고 할 수 있으므로 중간에 끼어 있어도 괜찮다. 58 c3는 popq %rax 와 ret에 대응된다. 따라서 movq %rax, %rdi( 혹은 movl %eax, %edi )를 찾아야 한다.

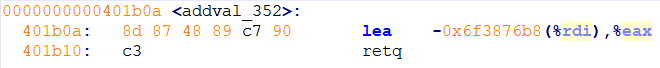
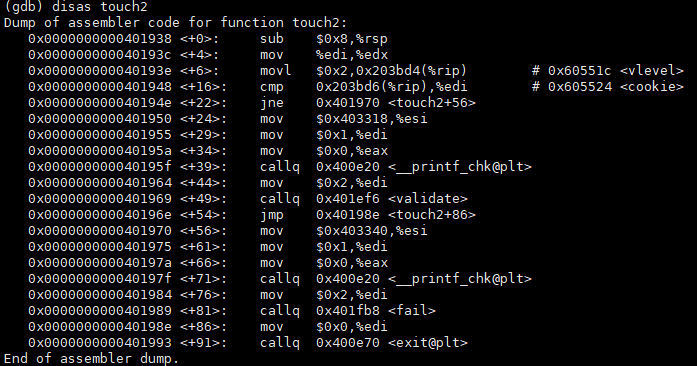
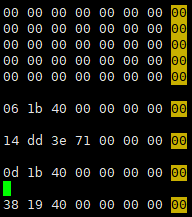
  
<그림 11>

그림 11을 보면, 89 c7 90 c3(movl %eax, %edi)를 찾을 수 있다.

  
<그림 12>

마지막으로 rtarget에서 touch2의 시작 주소가 0x401938라는 것을 알 수 있다.

이 4가지를 종합하여 hex 파일을 작성해주면 그림 13과 같다.

  
<그림 13>

**[5번 문제]**

3번 문제를 ROP를 이용하여 푸는 문제이다.

결과적으로 char\*를 인자로 넘겨받으므로 cookie를 저장한 곳의 주소가 필요하게 된다. 주소를 다루는 명령어는 lea 종류이다.

  
<그림 14>

start\_farm부터 end\_farm까지 찾아보면,

rdi와 rsi를 더해서 rax에 넣어주는 명령어가 gadget으로 존재함을 알 수 있다.

그 rax를 rdi에 다시 넣어주면 인자로 cookie의 주소를 넘길 수 있게 된다.

결국, rax에 넣을 ( rdi + rsi )는 어떻게 구성해야 하는가에 관한 문제만이 남게 된다.

rdi는 rsp라고 하고 rsi를 rsp를 넣는 시점에서 rsp와 cookie 사이의 메모리 상의 거리라고 생각하면 된다는 것을 알 수 있다.

|  |
| --- |
| cookie |
| touch3의 시작 주소 |
| rax -> rdi |
| lea |
| rax -> rdi |
| rsp -> rax |
| edx -> esi |
| ecx -> edx |
| eax -> ecx |
| 0x20 |
| pop eax |
| 40byte만큼의 padding |

<그림 15>

그림 15는 그것을 바탕으로 스택 메모리를 어떻게 채워야 할지 그림으로 나타낸 것이다.

cookie 부분이 High address이고, padding 부분이 Row address이다.

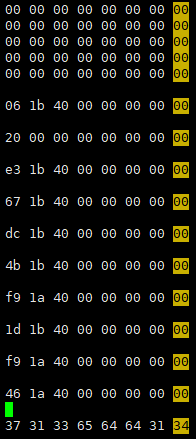
  
<그림 16>

그림 16은 그림 15의 각 칸에 해당하는 gadget들을 찾아서 그것을 바탕으로 작성한 hex파일이다.

이것으로 5번 문제를 해결할 수 있다.