Heuristic Wave Follow

Sep 28, 2018 · 9 min read

Ethernaut GatekeeperOne Problem — 이더넛 13단계 문제 해설

원래 스팀잇에 연재하던게.... 스팀파워가 없어서 더이상 글을 올릴 수가 없다. 얼른 미디움으로 다 글들을 옮겨야 겠다.

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

The Ethernaut

by

Heuristic Wave

. . .

Gatekeeper One Problem

이번 문제는 문지기를 통과하는 문제이다. 우선 힌트를 살펴보자!

- Telephone 문제와 Token 문제를 잘 떠올려 보자.
- `msg.gas` 와 `gasleft()` 에 대해서 공부하자.

사실 이번 문제는 아직 부족한 나에게는 그동안 풀어왔던 문제들보다 훨씬 어려웠다. Privacy 문제도 어려웠다고 하지만, 이번문제는 풀다가 질려서 2주 이상 이더넛 문제풀이를 손 놓고 있었다. 구글링끝에 도움을 받은 블로그의 글들을 아래 링크로 남겨두었는데, 처음봐서는 쉽게 이해가 되지 않을 수도 있다는 나와같은 사람들이 있을 것 같아서 이번 포스팅에 더 공을 들여야 겠다.

코드 분석

```
pragma solidity ^0.4.18;

contract GatekeeperOne {

  address public entrant;

  modifier gateOne() {
    require(msg.sender != tx.origin);
    _;
  }

  modifier gateTwo() {
    require(msg.gas % 8191 == 0);
    _;
  }

  modifier gateThree(bytes8 _gateKey) {
    require(uint32(_gateKey) == uint16(_gateKey));
    require(uint32(_gateKey) != uint64(_gateKey));
    require(uint32(_gateKey) == uint16(tx.origin));
    _;
  }

  function enter(bytes8 _gateKey) public gateOne gateTwo
gateThree(_gateKey) returns (bool) {
    entrant = tx.origin;
    return true;
  }
}
```

enter라는 함수가 gate1, 2, 3을 상속받고 있다. 때문에 우리는 차례로 문지기를 통과 하여야 한다.

그러므로 우리는 각 modifier 함수에 대한 require문을 통과할 방법을 강구하면 해결된다.

첫 번째, gate는 Telephone문제에서 만난 것처럼 다른 컨트랙트를 만들어서 sender와 origin을 다르게 만들면 통과할 수 있다. (Telephone 풀이와 같기 때문에 여기서는 통과하겠다.)

두 번째는 가스비가 얼마나 소모되는지 정확하게 알고 있어야한다. msg.gas가 8191의 배수여야만 관문을 통과할 수 있기 때문에, 리믹스 디버거의 사용법을 잘 활용할 수 있다면 통과할 수 있다.

세 번째 관문은, 데이터 타입의 형변환에 대한 개념과 flag & mask에 대한 이해가 있어야 3개의 작은문(require 문)을 빠져나갈 수 있다.

이번 단계에서는 문제의 특성상 미리 답을 알려주고, 2번째와 3번째문을 통과하는 것을 설명하면서 동시에 솔루션 코드에 대한 해설을 진행하겠다.

먼저 아래의 솔루션 코드인 `Pass` 컨트랙트의 _target에 문제의 CA주소를 넣고 배포한다.

```
contract Pass {
  GatekeeperOne gk;
  bytes8 key = bytes8(tx.origin) & 0xFFFFFFFF0000FFFF;

  function Pass (address _target) public {
    gk = GatekeeperOne(_target);
  }

  function enter(uint gaslimit) public {
    gk.enter.gas(gaslimit)(key);
  }
}
```

우리는 2번문제를 풀기 위한 가스비를 측정의 감을 잡기 위해서는 먼저 JVM환경에서 실험한 뒤 롭슨에서 해보는게 좋을 것이다.

2번 Gate통과하기

`msg.gas` 는 최신문법에서 `gasleft()` 에 해당하는 현재 남아있는 가스량을 뜻한다.

조건에서 남은 가스량을 8191로 % 연산한것의 mod 가 0이여야 함으로 연산 직전의 남은 가스량이 얼마인지 알아야 한다.

리믹스에서는 중단점을 설정하고 중단점에서 멈추면서 각각의 상황을 디버거 창에서 확

인 할 수 있다.

먼저 앞서 배포한 Pass컨트랙트의 **enter**함수에 매개변수로 가스를 넉넉하게 500000을 넣어주고 호출해보자. **이때!! 아래 사진과 같이 msg.gas가 있는 13번째 라인을 더블클릭하여 중단점을 만들자!!** 분명 2번째 modifier를 통과하는 가스가 오십만이 아니기 때문에 revert에러가 발생할 것이다.

이후 우리는 남은 가스값을 확인하기위해서 디버거탭으로 이동하여 위 사진에서 노란색으로 활성화된 버튼 밑에있는 `Jump to the next breakpoint` 버튼을 눌러서 마지막 부분즈음에 위치하는 13라인으로 이동한다음 사진에서 노란색으로 활성화된 `Step over forward`을 클릭하여 opcode가 위 빨간 박스처럼 GAS라고 써있는 곳을 찾으면 % 연산이 일어나기 직전의 남은 가스량을 알 수 있다.

500000(초기 가스값) — 499787(남은 가스량) + 2 (msg.gas에 해당하는 옵코드 가스 소모량) = 215

첫번째 사진에서 한번더 Step over을 하면 위 사진과 같이 연산에 해당하는 부분이 활성화되는데

500000–499785(msg.gas % 8191 직전의 남은 가스량) = 215 와 같은 방법으로 남은 가스를 구해도 된다.

그렇다면 우리는 2번을 통과하기 위해서 가스를 215 + 8191 * (넉넉한 만큼의 배수) 로 준다면 통과한 다는 것을 알수있다. 필자는 쉬운 계산을 위해 8191*100 + 215 = 819315 만큼의 양을 주었다.

3번 Gate통과하기

```
modifier gateThree(bytes8 _gateKey) {
    require(uint32(_gateKey) == uint16(_gateKey));
    require(uint32(_gateKey) != uint64(_gateKey));
    require(uint32(_gateKey) == uint16(tx.origin));
    _;
  }
```

우리는 이전 이더넷 문제를 풀면서 형변환과 데이터 손실에 관한 공부를 하였다. gateThree의 조건을 다시 확인해보면, 8바이트의 키를 넣어서 uint32와 16이 같아야하고 64와는 다르며 이것이 uint16의로 변환한 tx.origin과 동일해야한다. 비트연산에 대한 이해가 부족하면 해결하기 너무나 어려운 문제다. 아래에 공부를 할 수있는 링크를 달아 두었으니 flag와 mask에 대하여 잘 모른다면 링크를 통해 학습하자!!

16진수에서 마스킹을 할때 원하는 값을 살리기 위해서는 마스크 값에 F를 넣고 &연산을 한다.

```
bytes8 key = bytes8(tx.origin) & 0xFFFFFFFF0000FFFF;
```

따라서 solution에서 위와 같은 코드를 통하여 3번째 문을 통과하는 key를 만들 수 있다.

> 16진수 마스킹이 잘 이해가 되지 않을 수도 있기 때문에 짧은 16진수로 예시를 들어 보자면
>
> 0xA는 바이너리는 1010이고 0xF는 1111이다. 0xA & 0xF를 하면 결과값이 1010이나오고 이것은 0xA 본래의 모습과 같다.
>
> 필자의 CA (bytes20) : 0x344fbee17c2d215d364ec5943bc4a0c7030cfaa1 를 bytes8로 변환 하면 0x3bc4a0c7030cfaa1, 총 40자리의 CA주소 중에서 마지막 16자리만 보존된다.
>
> 조건을 bytes로 바꾸어 체크해보면, 아래와 같다.
>
> require(bytes4(_gateKey) == bytes2(_gateKey)); // 1번조건
> require(bytes4(_gateKey) != bytes8(_gateKey)); // 2번조건
>
> 그렇다면, 8바이트의 16진수를 다음과 같이 (A — 0000 | B — 0000 | C — 0000 | D — 0000) 4개씩 끊어 차례로 A, B, C, D라고 하자. bytes2로 주소를 변환하면 D만이 보존되고 나머지는 손실된다.
>
> 1번과 2번조건을 만족시키려면 C만을 손실시키면 됨으로 bytes8의 마스크는 0xFFFFFFFF0000FFFF가 되는 것이다.
>
> 이것을 통하여 3개의 require문을 통과할 수 있는 tx.orgin으로부터 만든 key를 얻을 수 있다.

여기까지 솔루션인 Pass 컨트랙트에 대한 설명이 끝났다.

이제, 롭슨 환경에서 Pass 컨트랙트에 Instance Address를 넣어 배포하고 enter 함수의 매개변수에 문제 2에서 구한 가스값을 넣고 호출하면 문제가 해결된다!!!

위 사진을 보면 주어진 문제의 주소가 나의 주소(from) 과 같아 진 것을 알 수 있다.

이번 문제에서 출제진이 가장 강조하고 싶은 문제는 힌트에도 언급된 이더리움의 가스비 책정 과정과 소모량에 대한 이해다.

한 문제를 풀기위해서 굉장히 많은 시간이 소요되었지만, 디버깅 수행과정, 형변환에 따른 데이터 손실, 비트연산에서의 flag와 mask, 16진수에관한 공부를 하며 빅엔디안과 리틀엔디안 등등의 다양한 공부를 할 수 있었다.

도움이 될만한 자료들

> 비트 연산자로 플래그 처리하기
>
> 16진수 비트 시프트와 마스킹 과정을 공부 할 수 있는 자료
>
> 16진수 변환기

필자가 도움을 받은 블로그

> https://medium.com/coinmonks/blockchain-ctf-write-up-ethernaut-gatekeeperone-level-49f8d0a0528b
>
> https://medium.com/coinmonks/ethernaut-lvl-13-gatekeeper-1-walkthrough-how-to-calculate-smart-contract-gas-consumption-and-eb4b042d3009

그럼, 다음번에는 14단계 Gatekeeper Two에서 만나요!

Ethernaut GatekeeperTwo Problem — 이더넛 14단계 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.

medium.com