dasp top 10

1. Reentrancy (재진입)

**function** **withdraw**(uint \_amount) {

require(balances[msg.sender] >= \_amount);

msg.sender.call.value(\_amount)();

balances[msg.sender] -= \_amount;

}

(Msg.sender.call.value() 는 무슨의미일까?)->msg.sender(address)->call.value()는 현재버전에선 사용하지않음.->call{value}

Reentrancy가 발생하면 무한으로 발생하는지 아니면 1회만 작동할거 3회 작동한다던지?

동기 비동기는 뭘까? -> 비동기는 함수가 실행중에 다른함수를 실행하면 멈추는게 비동기. 다른함수 실행하면서 자기다음코드로 넘어가는것은 동기. 해결방법은 그냥 잔고업데이트를 송금 앞에두면됨.

예시

Withdraw()함수로 외부주소의 잔고를 추적하고 자금을 인출할수있음.

악의적인 계약은 withdraw()를 사용하여 전체 잔고를 인출하려고함.

피해를 받는 계약은 악의적인 계약에게 이더를전송하기위해

call.value(amount)() 라는 low level 함수를 악의적인 계약의 잔고를업데이트 하기 전에 사용함

악의적인 계약은 payable fallback() 함수로 자금을 받고 피해받는계약의 withdraw()함수를 다시 호출한다.

두번째 실행은 자금의 전송을 일으킴. 그러나 악의적인 계약의 잔고는 첫번째 인출에서 업데이트 되지 않음. 결과적으로 악의적인 계약은 두번째에서 모든 잔고를 인출해감.

1. Access control
2. **function** **initContract**() **public** {
3. owner = msg.sender;
4. }

함수를 호출한 사람이 주인으로 저장되게 만들어놨는데 public 이라 아무나 다쓸수있었음. 그래서 librariy 계약에 정의하는거로 변경함. 사용자는 delegateCall을 통해 libarary의 함수를 호출하여 초기화 한다. 근데 이 예제에서는 이 함수가 이미 초기화된것인지 확인하지않는다는것, 게다가 스마트컨트랙트라서 library를 초기화 하거나 없앨수 있다는것..

3.Arithmetic

3번예제는 if문을 평생 충족하지못하는것 아닌가?

4번 예제에서 왜 255이상의 숫자에 도달하지못하고 가스가 떨어지면 멈추는것인지?

**function** **withdraw**(uint \_amount) {

require(balances[msg.sender] - \_amount > 0);

msg.sender.transfer(\_amount);

balances[msg.sender] -= \_amount;

}

쉽게말해 오버플로 언더플로

Withdraw 함수는 잔고가 남아있는만큼 인출해주는 함수이다.

공격자가 피해자의 잔고보다 큰수를 인출하려고 시도함.

그럼 withdraw가 또 인출해주고 잔고는 -가 되므로 언더플로우로 어마어마한잔고가 생기게됨.

**function** **popArrayOfThings**() {

require(arrayOfThings.length >= 0);

arrayOfThings.length--;

}

이 예제는 배열의 길이가 uint로 표현된다는 사실로 인해 발생되는 off-by-one error이다.

function **votes**(**uint** postId, **uint** upvote, **uint** downvotes) {

**if** (upvote - downvote < 0) {

deletePost(postId)

}

}

이 예제는 uint 두개의 연산값이 uint인 첫번째 예의 변형이다.

**for** (**var** i = 0; i < somethingLarge; i ++) {

// ...

}

이 예제는 곧 사라질 var에 대한 예제인데 var은 가장 작은 자료형으로 변형되기때문에 uint8로 변경되고 만약 255보다 큰 숫자를 반복하길 원했다면 거기까지 도달할수가 없고 가스가 떨어지면 멈출것이다.

4.Unchecked Low Level Calls

**function** **withdraw**(uint256 \_amount) **public** {

require(balances[msg.sender] >= \_amount);

balances[msg.sender] -= \_amount;

etherLeft -= \_amount;

msg.sender.send(\_amount);

}

Call callcode delegatecall send같은 low level 함수는 에러가 나도 bool값만 false로 리턴하고 실행을 되돌리지않고 그냥 진행해버린다. 그래서 low level 함수의 return 값을 체크하지않으면 fail-opens나 원치않는오류들이 발생한다.

이 예제코드에서 보여주려는것은 send의 리턴값을 받지않아서 etherleft가 부정확한 값을 가지고있다는것을 보여주는거같은데 맞는것인가? 저 코드에서 무엇이 문제인것인가?

5.Denial of Services

**function** **becomePresident**() **payable** {

require(msg.value >= price); // must pay the price to become president

president.transfer(price); // we pay the previous president

president = msg.sender; // we crown the new president

price = price \* 2; // we double the price to become president

}

**function** **selectNextWinners**(uint256 \_largestWinner) {

**for**(uint256 i = 0; i < largestWinner, i++) {

// heavy code

}

largestWinner = \_largestWinner;

}

인위적으로 가스비용을 늘림, 액세스컨트롤 방해해서 프라이빗 요소에 접근, 혼동과 과실 활용 등 여러가지 방법으로 발생함.

1.옥션 컨트랙트는 사용자가 다른자산에 입찰할수있는 계약이다.

2.입찰하려면 희망하는 가격의 이더를 포함해bid(uint object ) 함수를 호출한다. 그러면 옥션 컨트랙트는 물건의 주인이 입찰을 수락하거나 입찰자가 취소할때까지 escrow에 이더를 저장하고있는다. 이것은 옥션 컨트랙트가 무조건 해결되지않은 입찰과 이더를 모두 저장해놓고 있어야한다.

3.옥션 컨트랙트는 계약에서 출금하는 함수withdraw(uint amount) 를 포함하고있다. 함수가 하드코딩된 주소로 송금할때 개발자는 이 함수를 퍼블릭으로 바꾸기로 한다.

4.공격자는 공격가능성을 보고 계약 관리자의 모든 자금을 향해 함수를 호출한다. 이것은 에스크로 의 약속을 파괴하고 보류중인 입찰을 차단함.

게임 계약의 함수는 이전 계약에 공개적으로 뇌물을주면 대통령이 될수있음. 근데 이전 대통령이 스마트컨트랙트고 지불을 되돌리면 권력 이전이 실패하고 악의적인 스마트 계약이 대통령으로 계속 남을것이다 독재느낌

두번째 예는 for문이 안그래도 가스비로 감당하기 힘든 숫자를 넣어서 멈추게만드는것이다.

6.Bad Randomness

스마트컨트랙트는 게임용 랜덤요소로 block.number를 사용함

공격자가 최근 블록넘버가 승자인지 확인하는 컨트랙트를 생성함 그러면 승리를 하기위해 첫번째 컨트랙트를 호출함. 호출이 같은 트랜잭션에서 나온것이기때문에 블록넘버는 서로 같은 값을 저장한다.

공격자는 이길때까지 계속 호출하면됨.

uint256 **private** seed;

function **play**() **public** payable {

require(msg.**value** >= 1 ether);

iteration++;

**uint** randomNumber = **uint**(keccak256(seed + iteration));

**if** (randomNumber % 2 == 0) {

msg.sender.transfer(**this**.balance);

}

}

Private seed는 iteration과 같이 사용되고 keccak256해쉬함수는 호출자가 이겼는지 확인기위해 사용됨.

Seed가 private여도 특정시간의 트랜잭션을 통해 설정되고 그로인해 블록체인안에서는 확인할수있게된다.

**function** **play**() **public** **payable** {

require(msg.value >= 1 ether);

**if** (block.blockhash(blockNumber) % 2 == 0) {

msg.sender.transfer(**this**.balance);

}

}

Block.blockhash는 난수를 생성하기위해 사용된다. 이 해쉬는 blockNumber가 최근 block.number이면 알수 없으므로 0으로 설정한다. 과거에 블록넘버가 256이상으로 설정되었으면 항상 0으로 되어있다. 만약 오래되지않은 이전블록넘버로 돼있으면 다른 스마트컨트랙트도 같은 숫자에 접근할수있고 동일한 트랜잭션의 일부로 게임 컨트랙트를 호출할수있다.

예제 이해가 잘 되지않음 왜 2로 나눈나머지를 연산하는거지?

7.Front Running

채굴자가 외부소유주소를 대신하여 코드실행하면 가스비를통해 보상받으므로 사용자는 더 높은 가스비를 내고 거래를 빨리채굴할수있음. 이더리움블록체인은 공개되어있으므로 보류중인 거래내용 확인이 가능.

특정사용자가 퍼즐이나 비밀에 대한 솔루션을 공개하는경우 악의적인 사용자가 솔루션을 훔쳐 더 높은 수수료로 거래를 복사하여 원래꺼를 선점할수있다.

스마트컨트랙트가 RSA숫자 하나를 공개한다.(소수1 ,소수2)

submitSolution이라는 public 함수를 올바른 소수 두개와 같이 호출하면 보상을 준다.

앨리스가 RSA숫자를 제출한다.

누군가가 채굴 대기중인 앨리스의 정답을 포함한 트랜잭션을 보고 더 높은 비용으로 제출해버린다.

두번째 트랜잭션이 더비싸기때문에 선택되고 공격자가 보상을 받는다.

8.Time Manipulation

채굴자가 계약에 대한 지분을 보유하고있으면 채굴중인 블록에 적절한 타임스탬프를 선택하여 이득을취할수있다.

현재 시간에 대한 정보는 block.timestamp 또는 now를 사용하는데 이 값들은 채굴자들로부터 온다. 트랜잭션의 채굴자는 채굴이 일어난 시간을 보고해야한다. 좋은 스마트 컨트랙트는 그 보고된 시간에 크게 의존하지않아야한다. Block.timestamp 는 가끔 bad Randomness에서 랜덤생성할때 잘못사용되기도한다.

어떤 게임이 자정이후 가장빠른사람에게 지불을한다.

악의적인 채굴자가 게임을 이기려는 시도를 포함하고 자정으로 timestamp를 설정한다.

자정 아주 잠깐전에 채굴을 끝내고 진짜 현재시간은 자정에 아주 가깝다.네트워크에 있는 다른 노드들이 이 블럭을 연결하기로 선택함.

**function** **play**() **public** {

require(now > 1521763200 && neverPlayed == true);

neverPlayed = false;

msg.sender.transfer(1500 ether);

}

이 함수는 특정날짜 이후에 오는호출만 허용함. 채굴자는 자신의 block.timestamp에 영향을 끼칠수있기때문에 미래에 설정된 블록 타임스탬프를 사용하여 트랜잭션이 포함된 블록을 채굴할수 있음.

쉽게말해 block.timestamp를 수정해서 시간조건을 맞춰서 미리 보상을 타간다는 것인가? Block.timestamp는 뭐때문에 시간조정이 가능한지? 그리고 그 block에서 컨트랙트를 만들면 그 시간에서는 now가 block.timestamp값이랑 동일해지는것인지?

9.Short Addresses

교환api에는 받는사람의 주소와 양을 거래하는 함수가 있음

그런다음 api가 스마트컨트랙트의 transfer(address \_to, uint256 \_amount)함수를 (20바이트로 예상되는 주소와 연장된 12개의 0바이트로 해서 합쳐진 32바이트와 함께)이용한다

Bob은 alice에게 20토큰을 전송해달라고 요청한다. 그때 bob은 악의적으로 후행 0을 제거한다.

Alice는 짧아진 19바이트 bob의 주소로 교환 api를 사용한다.

Api는 12 0바이트를 채워서 31바이트로 만든다. \_amount 에서 1바이트를 효과적으로 훔친다.

결과적으로 EVM은 코드를 실행하고 데이터가 제대로 패딩되지 않았고 부족한 바이트를 \_amount의 끝부분에 더할것이다. 생각한거보다 256배 많은 토큰을 보내게 될것임.

10. Unknown Unknowns

솔리디티는 초기형이고 아직 부족한게많음 공격자들은 매번 새 공격도구를 개발함 그러니까 공부열심히해라~

이더리움에서 안전한 난수를 만들어내는것은 불가능? 노드들끼리 트랜잭션을 하고 안하고 뭐~~ cryptozombie 챕터4 레슨4