

DeFi 취약점 탐지와 정보보호(2) : DeFi 해킹 유형과 사례

2025. 9. 28. 성균관대학교 법학관

Table Of Contents

- 
1. EIPs/ERCs
 2. 실습 준비
 3. 해킹 유형과 사례

1. EIPs/ERCs

What are EIP and ERC?



EIP

- **Ethereum Improvement Proposal**의 약자 이더리움 개선 제안서를 의미
- 커뮤니티 주도의 이더리움 발전 메커니즘
- 제안 대상 :
 - 이더리움 스펙 변경
 - EVM 개선사항
 - 합의 알고리즘 업데이트 등
 - 클라이언트 소프트웨어 개선
- 절차 : GitHub 제출 → 커뮤니티 리뷰 → EIP 번호 할당 → 구현
- ex) EIP-1559 (가스비 개선), EIP-155 (하드포크)

ERC

- **Ethereum Request for Comment** 의 약자로, 기술 표준 정의문서인 RFC(Request for Comments)을 차용함
- 정의 대상:
 - 이더리움 토큰 및 스마트 컨트랙트 인터페이스
 - 컨트랙트 관련 메타데이터
- 주요 ERC 표준
 - 토큰 관련 표준 : **ERC-20**, ERC-721, ERC-777, ERC-1155
 - 인터페이스 관련 표준 : ERC-165
 - 계정 추상화 관련 표준 : **ERC-4337**, ERC-7579

EIP vs ERC

	EIP	ERC
목적	이더리움 프로토콜의 개선	토큰/컨트랙트 표준화
적용 범위	이더리움 전체	개별 프로젝트
제안자	이더리움 개발팀	개별 개발자/프로젝트

대표적인 EIP: EIP-712(서명)

The set of signable messages is extended from transactions and bytestrings $\mathbb{T} \cup \mathbb{B}^{8n}$ to also include structured data \mathbb{S} . The new set of signable messages is thus $\mathbb{T} \cup \mathbb{B}^{8n} \cup \mathbb{S}$. They are encoded to bytestrings suitable for hashing and signing as follows:

- `encode(transaction : \mathbb{T}) = RLP_encode(transaction)`
- `encode(message : \mathbb{B}^{8n}) = "\x19Ethereum Signed Message:\n" || len(message) || message` where `len(message)` is the *non-zero-padded* ascii-decimal encoding of the number of bytes in `message`.
- `encode(domainSeparator : \mathbb{B}^{256} , message : \mathbb{S}) = "\x19\x01" || domainSeparator || hashStruct(message)` where `domainSeparator` and `hashStruct(message)` are defined below.

대표적인 EIP: EIP-712(서명)

- 구조체에 대한 서명 방식을 규정([EIP-712](#))
 - TYPEHASH : 구조체 멤버의 자료형과 변수명을 CONCAT하여 해시

```
bytes32 constant MAIL_TYPEHASH = keccak256(  
    "Mail(address from,address to,string contents)");
```

- hashStruct : 실제 구조체의 데이터를 TYPEHASH와 함께 해시

```
function hashStruct(Mail memory mail) pure returns (bytes32 hash) {  
    return keccak256(abi.encode(  
        MAIL_TYPEHASH,  
        mail.from,  
        mail.to,  
        keccak256(mail.contents)  
    ));  
}
```


대표적인 EIP: EIP-712(서명)

- 구조체에 대한 서명 방식을 규정([EIP-712](#))
 - domainSeparator:
 - string name : DApp이나 프로토콜의 이름
 - string version : 어플리케이션 버전
 - uint256 chainId : 해당 컨트랙트가 배포된 [체인의 ID](#)
 - address verifyingContract : 해당 컨트랙트의 주소
 - bytes32 salt : 도메인 구분을 위한 마지막 수단. 만약 위에것이 다 같으면 이것을 사용하여 도메인을 구분

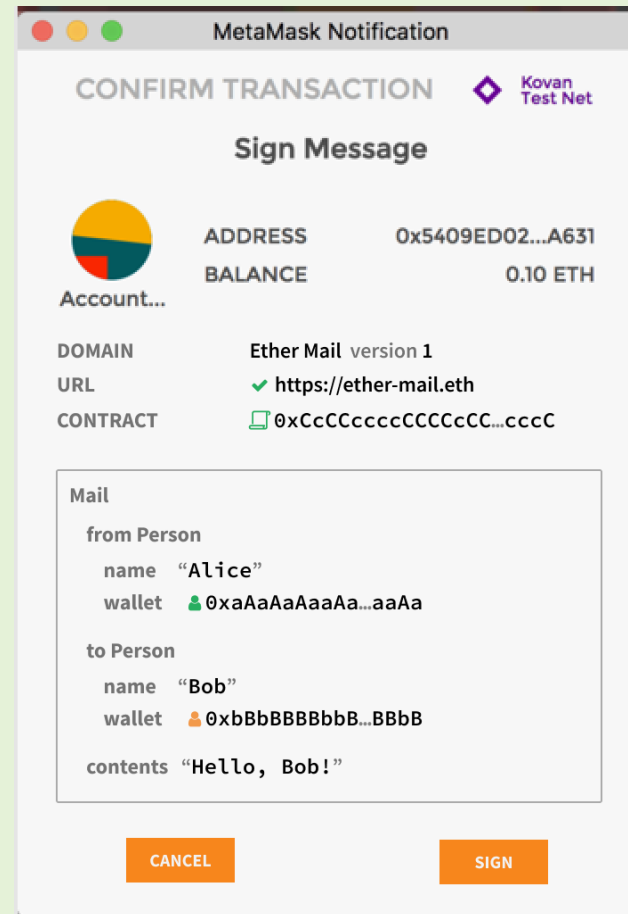
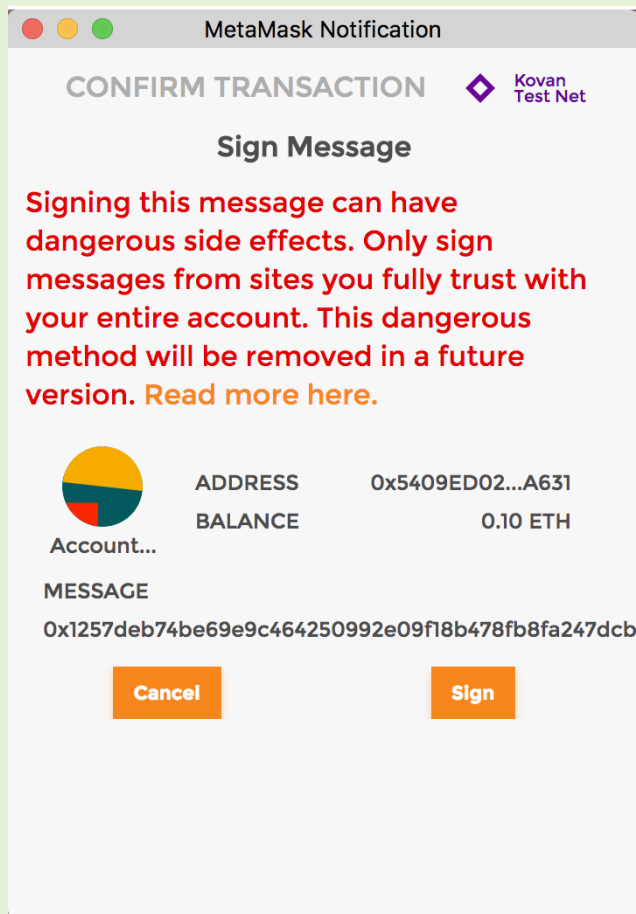
```
_DOMAIN_SEPARATOR = keccak256(  
    abi.encode(  
        keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)"),  
        keccak256(bytes(name)),  
        keccak256("1"),  
        block.chainid,  
        address(this)  
    )  
);
```

대표적인 EIP: EIP-712(서명)

- 구조체에 대한 서명 방식을 규정([EIP-712](#))
 - typedDataHash : H(Wx19Wx01 || domainSeparator || hashStruct)
 - 위 TypedDataHash에 ECDSA 서명을 한다.

```
function _toTypedDataHash (bytes32 _hashStruct) public returns (bytes32 _TypedDataHash){  
    return keccak256(  
        abi.encodePacked(  
            "\x19\x01",  
            _DOMAIN_SEPARATOR,  
            _hashStruct  
        )  
    );  
}
```

대표적인 EIP: EIP-712(서명)



대표적인 EIP: ERC-20

transfer

Transfers `_value` amount of tokens to address `_to`, and MUST fire the `Transfer` event. The function SHOULD `throw` if the message caller's account balance does not have enough tokens to spend.

Note Transfers of 0 values MUST be treated as normal transfers and fire the `Transfer` event.

```
function transfer(address _to, uint256 _value) public returns (bool success)
```



balanceOf

Returns the account balance of another account with address `_owner`.

```
function balanceOf(address _owner) public view returns (uint256 balance)
```



ERC-20

approve

Allows `_spender` to withdraw from your account multiple times, up to the `_value` amount. If this function is called again it overwrites the current allowance with `_value`.

NOTE: To prevent attack vectors like the one [described here](#) and discussed [here](#), clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to `0` before setting it to another value for the same spender. **THOUGH** The contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before

```
function approve(address _spender, uint256 _value) public returns (bool success)
```



transferFrom

Transfers `_value` amount of tokens from address `_from` to address `_to`, and MUST fire the `Transfer` event.

The `transferFrom` method is used for a withdraw workflow, allowing contracts to transfer tokens on your behalf. This can be used for example to allow a contract to transfer tokens on your behalf and/or to charge fees in sub-currencies. The function SHOULD `throw` unless the `_from` account has deliberately authorized the sender of the message via some mechanism.

Note Transfers of 0 values MUST be treated as normal transfers and fire the `Transfer` event.

```
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
```



ERC-20

approve

Allows `_spender` to withdraw from your account multiple times, up to the `_value` amount. If this function is called again it overwrites the current allowance with `_value`.

NOTE: To prevent attack vectors like the one [described here](#) and discussed [here](#), clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to `0` before setting it to another value for the same spender. **THOUGH** The contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before

```
function approve(address _spender, uint256 _value) public returns (bool success)
```



transferFrom

Transfers `_value` amount of tokens from address `_from` to address `_to`, and MUST fire the `Transfer` event.

The `transferFrom` method is used for a withdraw workflow, allowing contracts to transfer tokens on your behalf. This can be used for example to allow a contract to transfer tokens on your behalf and/or to charge fees in sub-currencies. The function SHOULD `throw` unless the `_from` account has deliberately authorized the sender of the message via some mechanism.

Note Transfers of 0 values MUST be treated as normal transfers and fire the `Transfer` event.

```
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
```



ERC-20

- 아래 3가지 토큰이 ERC-20 인터페이스를 갖추고 있는지 확인
 - [WETH](#)
 - [USDC](#)
 - [USDT](#)

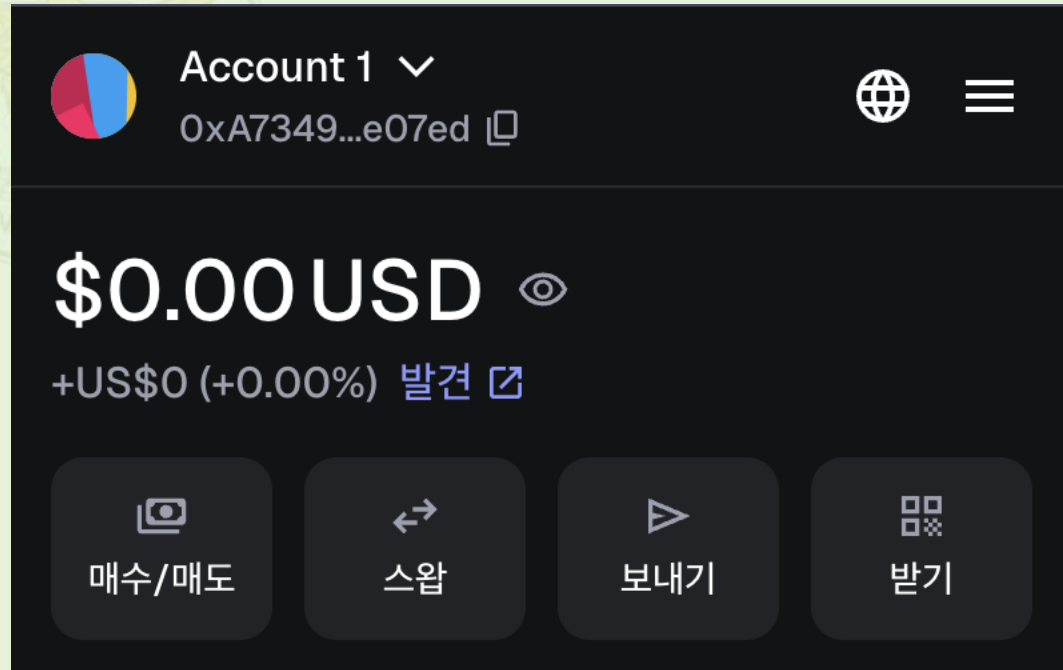


2. 실습 준비



실습 준비(1) – MetaMask 설치

1. Chrome 설치 : <https://www.google.com/chrome/>
2. Metamask Extension 설치 : <https://metamask.io/ko/download>
 1. 새 지갑 생성
 2. 비밀번호 복구 구문 사용
 3. 비밀번호 입력
 4. 비밀번호 복구 구문 컨펌
 5. 지갑 생성 완료

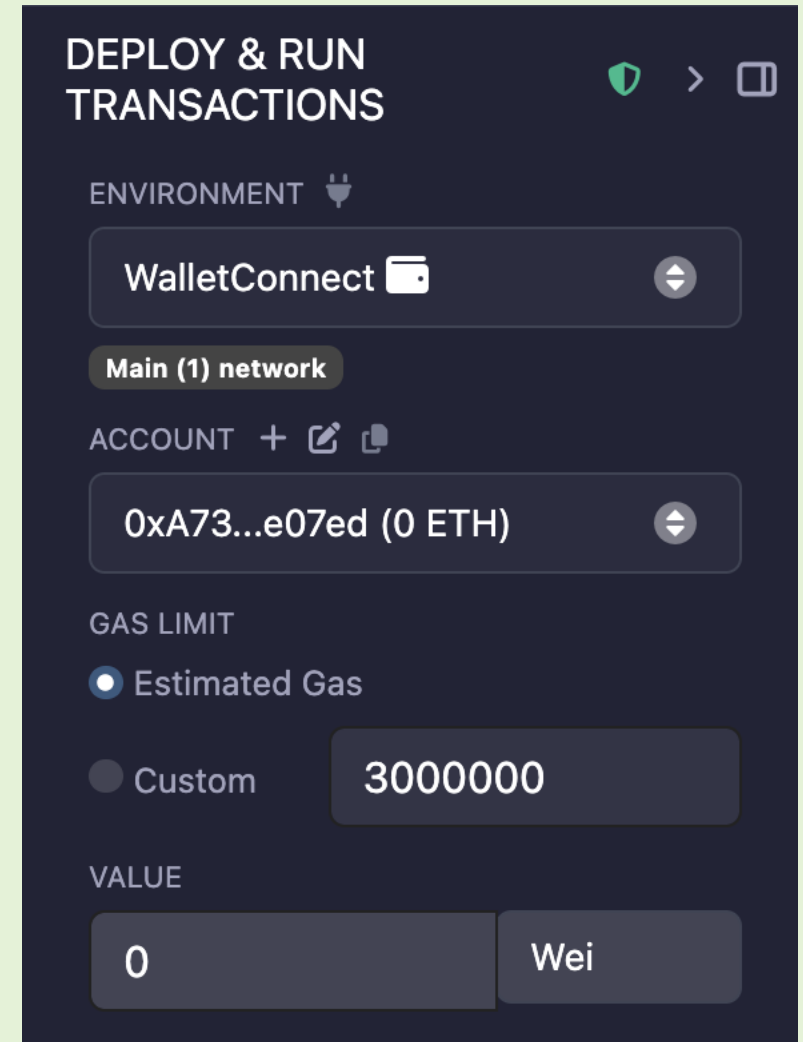


실습 준비(1) – 테스트넷 이더 긁어모으기

1. 테스트넷 이더를 무료 제공하는 서비스를 Faucet(수도꼭지) 라고 합니다.
2. [구글](#)에서 24시간마다 0.05 이더씩 받을 수 있습니다.
3. [POW 작업](#)을 실행하고 이더를 받을 수도 있습니다.
4. 0.5 이더 정도 있으면 충분합니다.

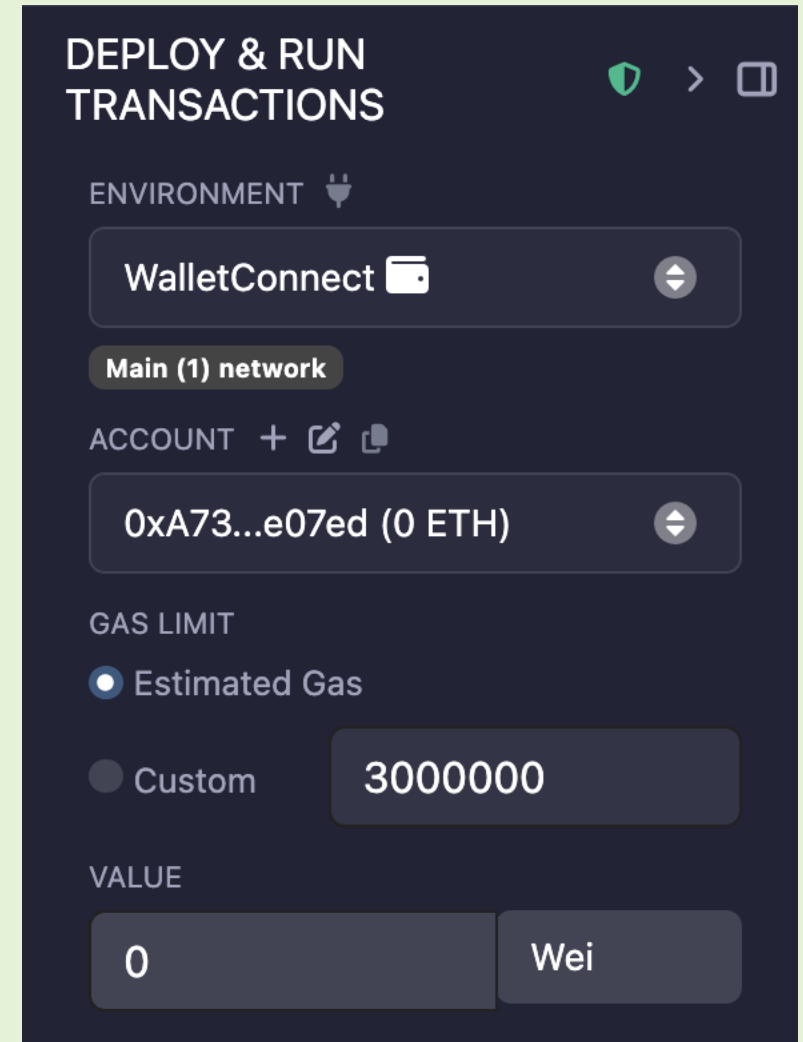
실습 준비(3) – Remix

1. REMIX IDE 접속
<https://remix.ethereum.org/>
2. Depoly & run transactions 선택
3. Environment 에서 WalletConnect 선택
4. 메타마스크 지갑 연결





실습 준비(4) – Ethernaut

1. Ethernaut 접속
<https://ethernaut.openzeppelin.com/>
2. 메타마스크 지갑 연결
3. Switch to Sepolia 선택
4. 메타마스크에서 네트워크 추가 컨펌





DEPLOY & RUN
TRANSACTIONS

ENVIRONMENT 

WalletConnect 

Main (1) network

ACCOUNT +  

0xA73...e07ed (0 ETH)

GAS LIMIT

☒ Estimated Gas

☐ Custom 3000000

VALUE

0 Wei

3. 해킹 유형과 사례

THE ETHEREUM DAO HACK_



FBI Warning

여기서 알려드리는 취약점을 실제 메인넷에서 실행하면 안됩니다.

여러분들의 집에 FBI가 찾아올 수도 있습니다.

조심하세요...

스마트 컨트랙트 취약점 유형

- Reentrancy
- Integer under/overflow
- Sensitive data exposure
- Randomness
- Transaction Ordering Dependence (TOD)
- Gas griefing
- Etc...

ABI 인코딩과 함수 선택자

- 함수 문자열의 keccak256 해시 값의 첫 4바이트
- ex) transfer(address to, uint256 amount)에 대한 시그니처를 만들어보자
 1. 함수 문자열 s 작성: "transfer(address,uint256)"
 2. H(s) 계산
 3. 앞 4바이트를 취함 : xa9059cbb
- balanceOf(address owner) 시그니처 만들어보기
- 이미 계산해놓은 [테이블](#)도 있음 => Etherscan은 이 데이터를 이용해서 보여줌
- 4byte 만 사용하는 것으로 발생 가능한 문제가 있음. 무엇일까요?

ABI 인코딩과 함수 선택자

- [illegible]

취약점 유형(1): 재진입 공격

- 특정 컨트랙트의 **상태변화(storage 입출력)** 이 이루어지기 전에 해당 컨트랙트에 재진입하여 동작을 수행하여 공격하는 형태
 1. 공격자는 자신이 원하는 공격 로직을 구현
 2. 피해 대상 컨트랙트가 공격 컨트랙트로 ETH 송신 or 함수 호출을 유도
 3. 공격 컨트랙트는 피해 대상 컨트랙트의 특정 함수를 재호출
 4. 위 과정을 원하는 만큼 반복하면 됨

취약점 유형(1): 재진입 공격

- 오른쪽 Victim_1 컨트랙트는 어느 부분이 문제가 될까요?

```
1  contract Victim_1 {
2      mapping(address => uint256) private balances;
3
4      function deposit() external payable {
5          if (msg.value > 0) {
6              balances[msg.sender] += msg.value;
7          }
8          else {
9              revert("nope!");
10         }
11     }
12
13     function withdraw(uint256 amount) external {
14         uint256 memory currentBalance = balances[msg.sender];
15         if (currentBalance >= amount) {
16             msg.sender.call(""){value: amount};
17             balances[msg.sender] -= amount;
18         }
19         else {
20             revert("you don't have enough balance");
21         }
22     }
23 }
```

취약점 유형(1): 재진입 공격

- 16행에서 자금을 전송한 이후, 17행에서 요청자의 잔고 정보에서 요청한 만큼의 balance를 차감하는 형태로 되어 있음
- 이런 경우를 CEI 패턴을 충족하지 못하였다고 하며, 재진입 공격의 대상이 됨
 - Check : 체크하고
 - Effects : 효과를 반영하고
 - Interaction : 상호작용하여야 함

```
1  contract Victim_1 {
2      mapping(address => uint256) private balances;
3
4      function deposit() external payable {
5          if (msg.value > 0) {
6              balances[msg.sender] += msg.value;
7          }
8          else {
9              revert("nope!");
10         }
11     }
12
13     function withdraw(uint256 amount) external {
14         uint256 memory currentBalance = balances[msg.sender];
15         if (currentBalance >= amount) {
16             msg.sender.call(""){value: amount};
17             balances[msg.sender] -= amount;
18         }
19         else {
20             revert("you don't have enough balance");
21         }
22     }
23 }
```


취약점 유형(1): 재진입 공격

- Ethernaut 10번 풀어보기
 - REMIX IDE 이용
 - 컨트랙트의 모든 자금을 탈취하면 성공

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.12;

import "openzeppelin-contracts-06/math/SafeMath.sol";

contract Reentrance {
    using SafeMath for uint256;

    mapping(address => uint256) public balances;

    function donate(address _to) public payable {
        balances[_to] = balances[_to].add(msg.value);
    }

    function balanceOf(address _who) public view returns (uint256 balance) {
        return balances[_who];
    }

    function withdraw(uint256 _amount) public {
        if (balances[msg.sender] >= _amount) {
            (bool result,) = msg.sender.call{value: _amount}("");
            if (result) {
                _amount;
            }
            balances[msg.sender] -= _amount;
        }
    }

    receive() external payable {}
}
```

취약점 유형(1): 재진입 공격 실제 사례

The DAO Hack

- The DAO는 2016년 ICO를 통해 설립된 DAO(Decentralized Autonomous Organization)으로, 투자를 통해 받은 자금을 어느 기업에 투자할지 탈중앙적으로 결정하는 투자 DAO였다. 크라우드펀딩에 참여하여 ETH를 지불한 사람들에게는 DAO 토큰을 주어서...
- 당시 ETH 발행량의 14%가 도난당함
- 롤백 해주어야 하나?
 - 해주어야 한다 => ETH
 - 해주면 안된다. code is law => ETC

취약점 유형(1): 재진입 공격 실제 사례(1)

The DAO Hack

```
function splitDAO(  
    uint _proposalID,  
    address _newCurator  
) noEther onlyTokenholders returns (bool _success) {  
    ...  
    // Move ether and assign new Tokens  
    uint fundsToBeMoved =  
        (balances[msg.sender] * p.splitData[0].splitBalance) /  
        p.splitData[0].totalSupply;  
    if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender) == false)  
        throw;  
    ...  
    // Burn DAO Tokens  
    Transfer(msg.sender, 0, balances[msg.sender]);  
    withdrawRewardFor(msg.sender); // be nice, and get his rewards  
    totalSupply -= balances[msg.sender];  
    balances[msg.sender] = 0;  
    paidOut[msg.sender] = 0;  
    return true;  
}
```

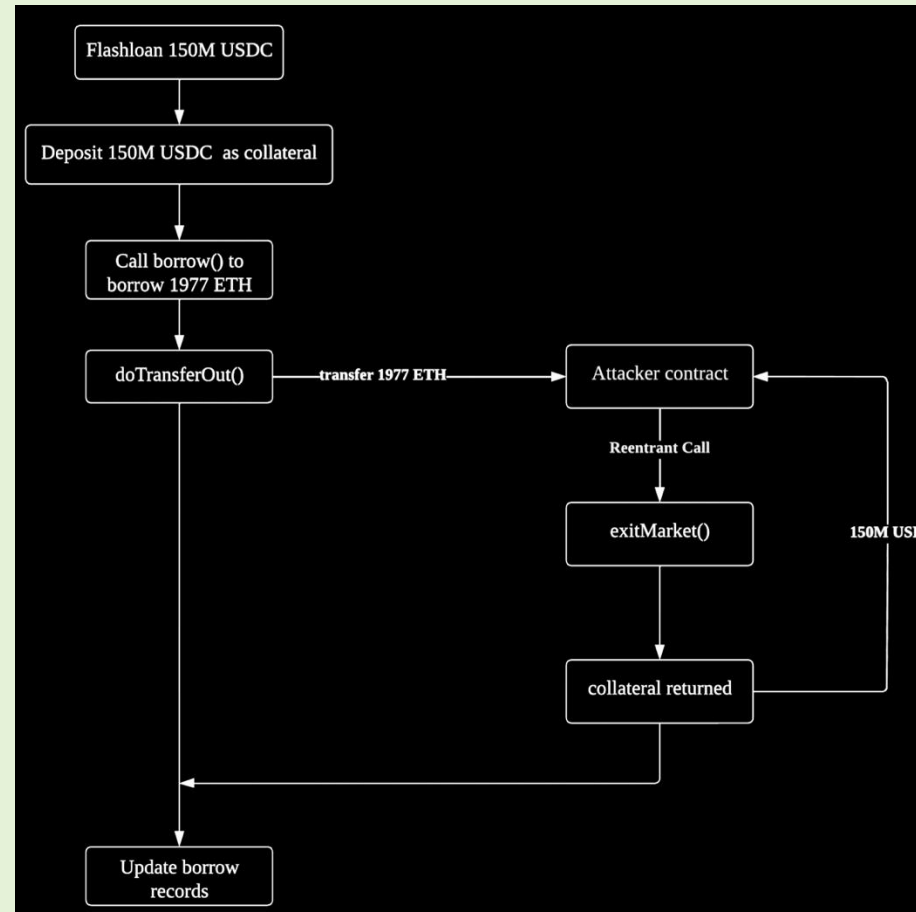
취약점 유형(1): 재진입 공격 실제 사례(2)

Fei-rari 파이낸스

- 유명한 대출 프로토콜인 [컴파운드\(Compound Finance\)](#) 프로젝트를 포크해서 만듦
- 계정 생성 => 토큰 예치 => 대출 => 상환 => 계정 삭제 순으로 정보를 관리
- 계정 삭제를 하기 위해서는 현재 대출 잔고가 없음을 체크함
- 이더를 대출할 때 대출 잔액을 업데이트하는 코드에 CEI 패턴이 적용X
- 털린 자산 : 6,037.8139071514 eth, 20,251,603.11559831 fei, 14,278,990.684390573 dai, 1,948,952.17886651 lUSD, 10,055,556.328173 usdc...
- **Total estimated value: \$79,749,026**

취약점 유형(1): 재진입 공격 실제 사례(2)

Fei-rari 파이낸스



취약점 유형(1): 재진입 공격 실제 사례(3)

Cream Finance

- 이 사건 역시 CEI 패턴을 지키지 않아 발생하였음([공격 TX](#))
- 해당 프로토콜은 감사를 받은 상태였으나, 감사를 받은 이후로 담보 대상 토큰을 추가하여 공격당하게 됨
 - 추가한 토큰은 AMP 토큰
 - 토큰 표준들(ERC-20 등)은 대부분 토큰 수신자의 주소에서 무엇인가를 실행하는 기능(callback)을 두고 있지 않음
 - 하지만 콜백이 존재하는 토큰 표준이 존재하는데(ERC-777), 해당 표준 기반의 AMP 토큰을 담보로 지원하게 되자 공격자는 이를 활용하였음
- 털린 자산 : 2804 ETH, 462,079,976 AMP
- **Total estimated value: \$18,800,000**

취약점 유형(1): 재진입 공격 실제 사례(3)

Cream Finance

```
address recipientImplementation;  
recipientImplementation = interfaceAddr(_to, AMP_TOKENS_RECIPIENT);
```

```
if (recipientImplementation != address(0)) {  
    IAmpTokensRecipient(recipientImplementation).tokensReceived(  
        msg.sig,  
        _toPartition,  
        _operator,  
        _from,  
        _to,  
        _value,  
        _data,  
        _operatorData  
    );  
}
```

```
}
```


취약점 유형(1): 재진입 공격 실제 사례(3)

Cream Finance

```
506      /*
507      * We invoke doTransferOut for the borrower and the borrowAmount.
508      * Note: The cToken must handle variations between ERC-20 and ETH underlying.
509      * On success, the cToken borrowAmount less of cash.
510      * doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
511      */
512      doTransferOut(borrower, borrowAmount, isNative);
513
514      /* We write the previously calculated values into storage */
515      accountBorrows[borrower].principal = vars.accountBorrowsNew;
516      accountBorrows[borrower].interestIndex = borrowIndex;
517      totalBorrows = vars.totalBorrowsNew;
518
519      /* We emit a Borrow event */
520      emit Borrow(borrower, borrowAmount, vars.accountBorrowsNew, vars.totalBorrowsNew);
521
```

취약점 유형(2): 민감 데이터 노출

- 이더리움 네트워크에 저장하는 데이터는 전부 조회할 수 있음
 - 주소 : 슬롯 번호 이용
- 즉 비밀 키 같은 데이터는 체인에 업로드하면 안됨

취약점 유형(2): 민감 데이터 노출

- Ethernaut 12번 풀어보기
 - REMIX IDE 혹은 이더스캔 이
용
 - key키를 전달하여 문제 컨트랙
트를 unlock 하면 해결할 수
있음

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

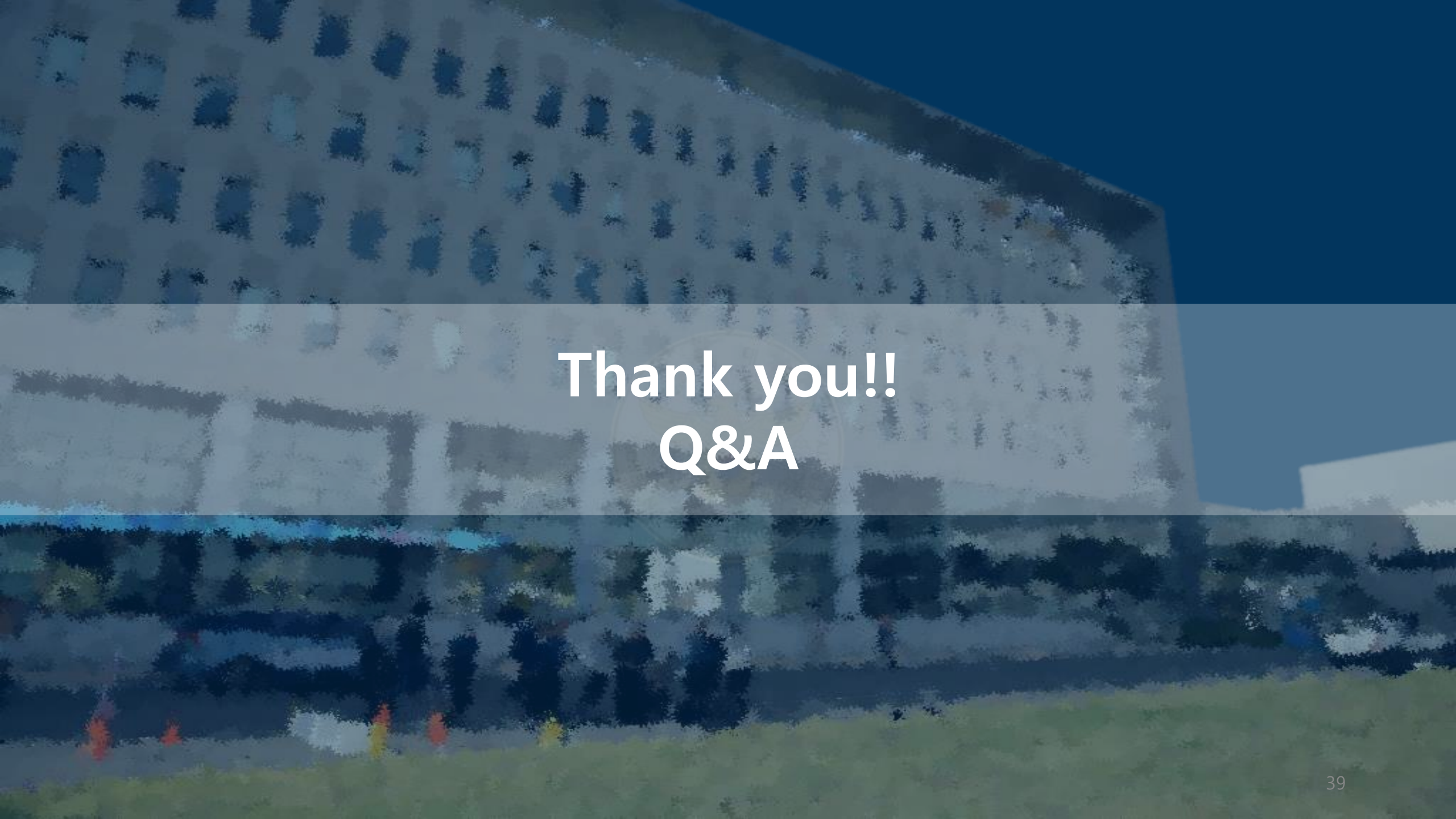
contract Privacy {
    bool public locked = true;
    uint256 public ID = block.timestamp;
    uint8 private flattening = 10;
    uint8 private denomination = 255;
    uint16 private awkwardness = uint16(block.timestamp);
    bytes32[3] private data;

    constructor(bytes32[3] memory _data) {
        data = _data;
    }

    function unlock(bytes16 _key) public {
        require(_key == bytes16(data[2]));
        locked = false;
    }

    /*
    A bunch of super advanced solidity algorithms...

    ,*!^*,*!^*,*!^*,*!^*,*!^*,*!^*
    ,*!^*,*!^*,*!^*,*!^*,*!^*,*!^*,
    *,*!^*,*!^*,*!^*,*!^*,*!^*,*!^*,*!^* ,---/V\
    `*,*!^*,*!^*,*!^*,*!^*,*!^*,*!^*,*!^* ~|_(0.0)
    ^*,*!^*,*!^*,*!^*,*!^*,*!^*,*!^*,*!^*,*!^*,*!^*,*!^*,*!^*,*!^* UU UU
    */
}
```



Thank you!!
Q&A