Ethereum Vulnerabilities Report

DSSSDF

No Institute Given

Abstract.

1 Introduction

Emerging blockchain technologies have shown great promise in offering decentralized services backed by a distributed ledger. The most prominent framework for smart contracts is Ethereum, whose capitalisation has reached 1 billion dollars since its launch in July 2015. In Ethereum, smart contracts are rendered as computer programs, written in a Turing-complete language. The consensus protocol of Ethereum, which specifies how the nodes of the peer-to-peer network extend the blockchain, has the goal of ensuring the correct execution of contracts.

However, the correctness of executions alone is not sufficient to make smart contracts secure. Indeed, several security vulnerabilities in Ethereum smart contracts have been discovered both by hands-on development experience, and by static analysis of all the contracts on the Ethereum blockchain. These vulnerabilities have been exploited by some real attacks on Ethereum contracts, causing losses of money. The most successful of these attacks managed to steal $\approx$ \$60M from a contract, but its effects were cancelled after an harshly debated revision of the blockchain.

There are several reasons which make the implementation of smart contracts particularly prone to errors in Ethereum. They are categorised into 3 levels: Solidity level, the EVM level, and the blockchain level, described by Atzei, Bartoletti, and Cimoli, 12 vulnerabilities among these components that pose risks to the contract owners are defined. After picking up from several sources, including the official documentation, research papers, and also Internet discussion forums, we conclude this report.

2 Vulnerabilities taxonomy

In this section, we list all the vulnerabilities indentified so far, and then give corresponding secure issues.

Level	Cause of vulnerability	Attacks
Solidity	Call to the unknown	1
	Gassless send	2
	Exception disorders	3
	Type casts	2
	Reentrancy	2
	Keeping secrets	2
EVM	Immutable bugs	2
	Ether lost in transfer	2
	Stack size limit	2
Blockchain	Transaction Ordering Dependence(TOD)	2
	Generating randomness	2
	Time constraints	2

2.1 Call to unknown

– **Explanation:**

This refers to any use of call, delegatecall, send, or direct call to another contract that results in execution of an unknown, possibly malicious, fallback function (the anonymous function on every smart contract invoked as a catch all).

– **Example:**

- The DAO Attack

```
contract SimpleDAO{
    mapping (address => uint) public credit;
    function donate(address to){credit[to] += msg.value;}
    function queryCredit(address to) returns (uint){
        return credit[to];
    }
    function withdraw(uint amount) {
        if (credit[msg.sender]>= amount) {
            msg.sender.call.value(amount)();
            credit[msg.sender]-=amount;
}}}

contract MalloryAttack{
    //attackerAddress is the attacker's address, should be substituted when test.
    SimpleDAO public dao = SimpleDAO(attackerAddress);
    address owner;
    function MalloryAttack(){owner = msg.sender;}
    function(){dao.withdraw(dao.queryCredit(this));}
    //recursion here when call withdraw without credit decrease.
    function getJackpot(){owner.send(this balance);}
}
```

- The Parity Wallet Hack

```
//the toy callee library.
contract ToyWalletLibrary {
    address owner;
    //****************attack here***************//
    function initWallet(address _owner) {
        owner = _owner;
    // ... more setup ...
    }

    function changeOwner(address _new_owner) external {
        if (msg.sender == owner) {
        owner = _new_owner;
    }}

    function () payable {
    // ... receive money, log events, ...
    }

    function withdraw(uint amount) external returns (bool success) {
        if (msg.sender == owner) {
            return owner.send(amount);
        } else {
            return false;
}}}

//the toy wallet calling the walletlibrary.
contract ToyWallet {
    address _walletLibrary;
    address owner;
    function ToyWallet(address _owner) {
        // replace the following line with _walletLibrary = new ToyWalletLibrary();
        // if you want to try to exploit this contract in Remix.
        _walletLibrary = <address of pre-deployed ToyWalletLibrary>;
        _walletLibrary.delegatecall(bytes4(sha3("initWallet(address)")), _owner);
    }
    function withdraw(uint amount) returns (bool success) {
        return _walletLibrary.delegatecall(bytes4(sha3("withdraw(uint)")), amount);
    }
    // fallback function gets called if no other function matches call
    function () payable {
        _walletLibrary.delegatecall(msg.data);
}}

contract WalletAttack{
    address wallet;
    // replace the following line with _walletLibrary = new ToyWalletLibrary();
    wallet = <address of pre-deployed ToyWallet>;
    function WalletAttack(){
        wallet.initWallet(this.address,"anynumber","anynumber");
    }
    function withdraw(uint amount){
        owner.send(amount);
}}
```

2.2 Gassless send

- **Explanation:**
  If the callee of a send (the Ethereum function to transfer ether) is a contract with a relatively expensive fallback function, then the amount of gas the caller is limited to $\approx$ 2300 units for sending ether to an address will be insufficient, and an out-of-gas exception will be thrown.
- **Example: King of the Ether Throne**

```
contract KotET{
    address public king;
    uint public claimPrice = 100;
    address owner;

    function KotET(){
        owner = msg.sender; king = msg.sender;
    }

    function sweepCommission(uint amount){
        owner.send(amount);
    }

    function(){
        if(msg.value < claimPrice) throw;
        uint compensation = calculateCompensation();
        //if the recipient is not a normal address, but a contract needing status updated, the send may fail because of
        king.send(compensation);
        king = msg.sender;
        claimPrice = calculateNewprice();
}}
```

2.3 Exception disorders

- **Explanation:**
  This refers to unchecked send errors or called contracts that throw exceptions. The effect is that the calling contract transaction is entirely reverted and all gas is lost. There are 2 different behaviours, which depend on how contracts call each others, directly invoke or via call.
- **Example:**

```
contract Alice{ function ping(uint) returns (uint)}
//the following direct invoke will return x=0, if there is an exception in function ping. All the gas allocated by the o
contract Bob{ function x=0; function pong(Alice c){x=1; c.ping(42); x=2;}
//the following call will return x=2, if there is an exception in function ping. The call just returns false, and the ga
contract John{ function x=0; function pong(Alice c){x=1; c.call.ping(42); x=2;}
```

2.4 Type casts

- **Explanation:**
  If the arguments to a direct call from one contract to a function of another contract is incorrectly typed, or the address of the called contract is incorrect, either nothing will happen, or the wrong code including the fallback function will execute. In neither case is an exception thrown, and the caller is not made aware.
- **Example: No facts this moment**

2.5 Reentrancy

- **Explanation:**
  In some cases, a contracts fallback function allows it to re-enter a caller function before it has terminated. This can result in the repeated execution of functions intended only to be executed once per transaction.
- **Example: the DAO Attack**

2.6 Keeping secrets

- **Explanation:**
  By the public nature of the blockchain, contract fields marked private are not guaranteed to remain secret to set a private field, a contract owner must broadcast a transaction. Cryptographic protocols are required to guarantee that fields are not visible to anyone mining or inspecting the blockchain.
- **Example: plotted odds and evens(no actual fact)**

```
contract OddsAndEvens{
    struct Player{ address addr; uint number;}
    Player[2] private players;
    uint8 tot = 0 ; address owner;

    function OddsAndEvens(){ owner = msg.sender;}

    function play(uint number){
        if(msg.value != 1 ether) throw;
        players[tot] = Player(msg.sender, number);
        tot++;
        if(tot == 2) andTheWinnerIs();
    }
    //the adversary infers the first player's bet by inspecting the blockchain transaction where he joined the game.
    function andTheWinnerIs() private{
        uint n = player[0].number + players[1].number;
        players[n%2].addr.send(1800 finney);
        delete players;
        tot = 0;
    }

    function getProfit(){
        owner.send(this.balance);
}}
```

2.7 Immutable bugs

- **Explanation:**
  When a contract is added to the blockchain, there is no way to edit it. If a contract is found to be defective, there is often nothing to be done short of killing the contract(assuming measures were taken by the contract creator to make this possible).
- **Example: Rubixi**

```
//the name is changed from DynamicPyramid to Rubixi, but the name of constructor is omitted,so users would invoke Dynami
contract Rubixi{
    address private owner;
    function DynamicPyramid(){ owner = msg.sender; }
    function collectAllFees(){owner.send(collectedFees);}
}
```

2.8 Ether lost in transfer

- **Explanation:**
  If ether is sent to an orphan address that doesnt actually belong to any user or contract, that ether will be lost and cannot be retrieved.
- **Example:**

2.9 Stack size limit

- **Explanation:**
  A transactions call stack grows with each contract invocation, and once the stack is 1,024 frames tall, an exception is thrown. Changes to gas rules and certain instruction costs have resolved this vulnerabilityin the current enviromnent.
  Note: This cause of vulnerability has been addressed by an hard-fork of the Ethereum blockchain. The fork changed the cost of several EVM instructions, and redefined the way to compute the gas consumption of call and delegatecall. After the fork, a caller can allocate at most 63/64 of its gas: since, currently, the gas limit per block is 4,7M units, this implies that the maximum reachable depth of the call stack is always less than 1024.
- **Example: Governmental**

```
contract Governmental{
    address public owner;
    address public lastInvestor;
    uint public jackpot = 1 ether;
    uint public lastInvestmentTimestamp;
    uint public ONE_MINUTE = 1 minutes;

    function Governmental(){
        owner = msg.sender;
        if(msg.value < 1 ether) throw;
    }

    function invest(){
        if(msg.value < jackpot/2) throw;
        lastInvestor = msg.sender;
        jackpot += msg.value/2;
        lastInvestmentTimestamp = block.timestamp;
    }
    //this timestamp dependance may be attacked by the player who is also a miner.
    //also the miner can manipulate which transaction to be included in the block.
    function restInvestment(){
        if(block.timestamp < lastInvestmentTimestamp + ONE_MINUTE) throw;
        lastINvestor.send(jackpot);
        owner.send(this.balance - 1 ether);
        lastInvestor = 0;
        jackpot = 1 ether;
        lastInvestmentTimestamp = 0;
}}
```

2.10 Transaction Ordering Dependence (TOD)

- **Explanation:**
  This occurs when the assumed state of the blockchain is not the blockchains actual state when a transaction is executed. The order in which transactions are mined can have adverse effects on the execution of any given transaction. This bug is said to be present in up to %15.8 of all contracts on the blockchain.
- **Example: the MarketPlace(together with the Governmental)**

```
contract MarketPlace{
    uint public price;
    uint public stock;
    /.../
    // 2.the owner update the price later but may be mined first which leads a failure trade for the buyer.
    function updatePrice(uint _price){
        if(msg.sender == owner)
            price = _price;
    }
    // 1.the buyer order the stock first according to the price he see but may be mined later.
    function buy(uint quant) returns (uint){
        if(msg.value < quant * price || quant > stock) throw;
        stock -= quant;
}}
```

2.11 Generating randomness

- **Explanation:**
  Many contracts that require random numbers use the hash or the timestamp of some block yet-to-appear in the blockchain. A malicious miner could arrange their block to bias the outcome of this random number generation.
- **Example:**

2.12 Time constraints

- **Explanation:** Some Dapps use timestamps to generate random numbers. However, the clock in Ethereum is set by the local clocks of its miners. So, such Dapps can be influenced with slight adjustments to miners reported times.
- **Example:**

3 Verification with Symbolic Exection