Analysing Vulnerabilities in Smart Contracts

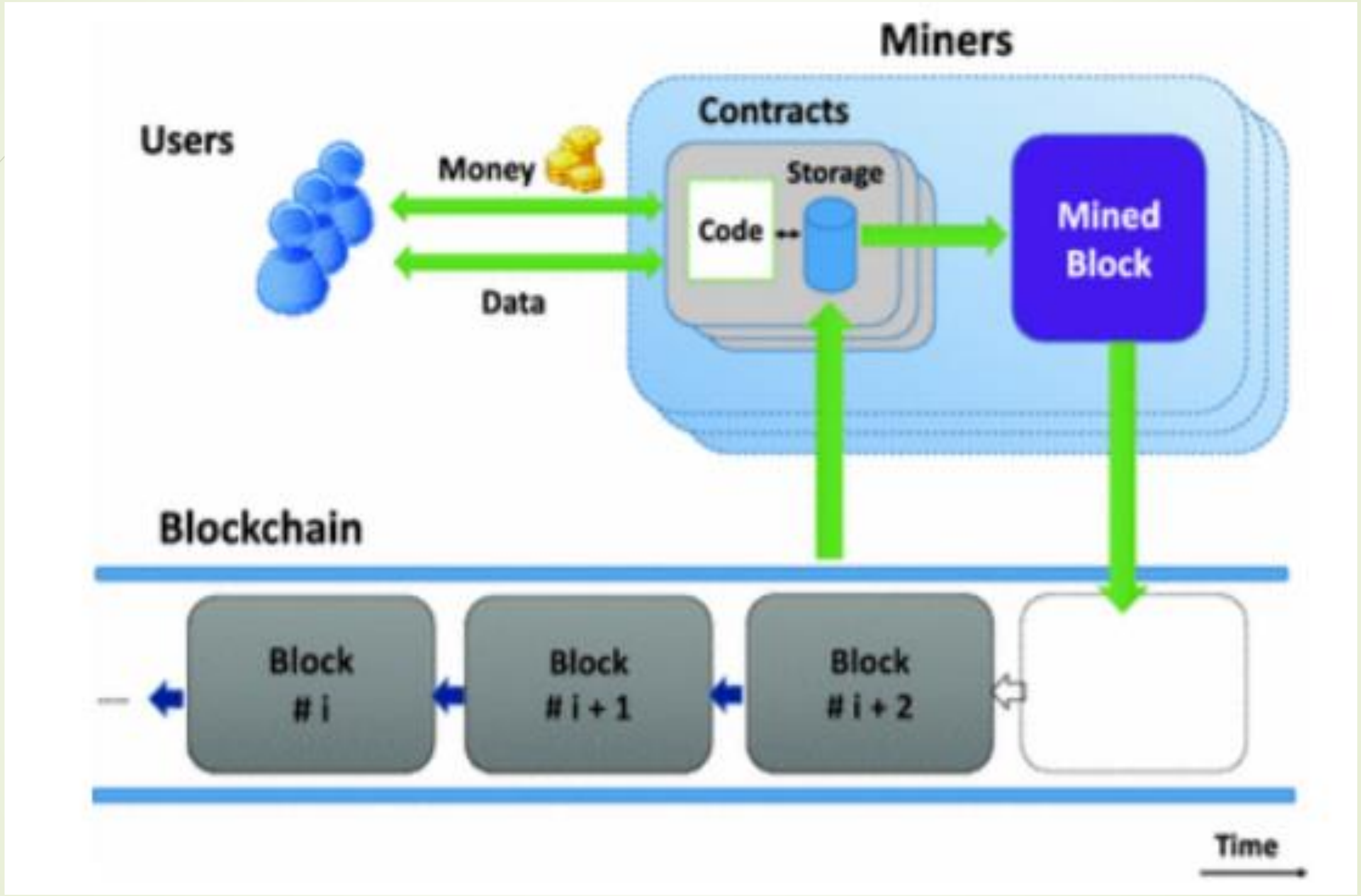
-By Shrutirupa Banerjiee

Topics

- Understanding Smart Contracts
- Security Issues in Smart Contracts

Understanding Smart Contracts

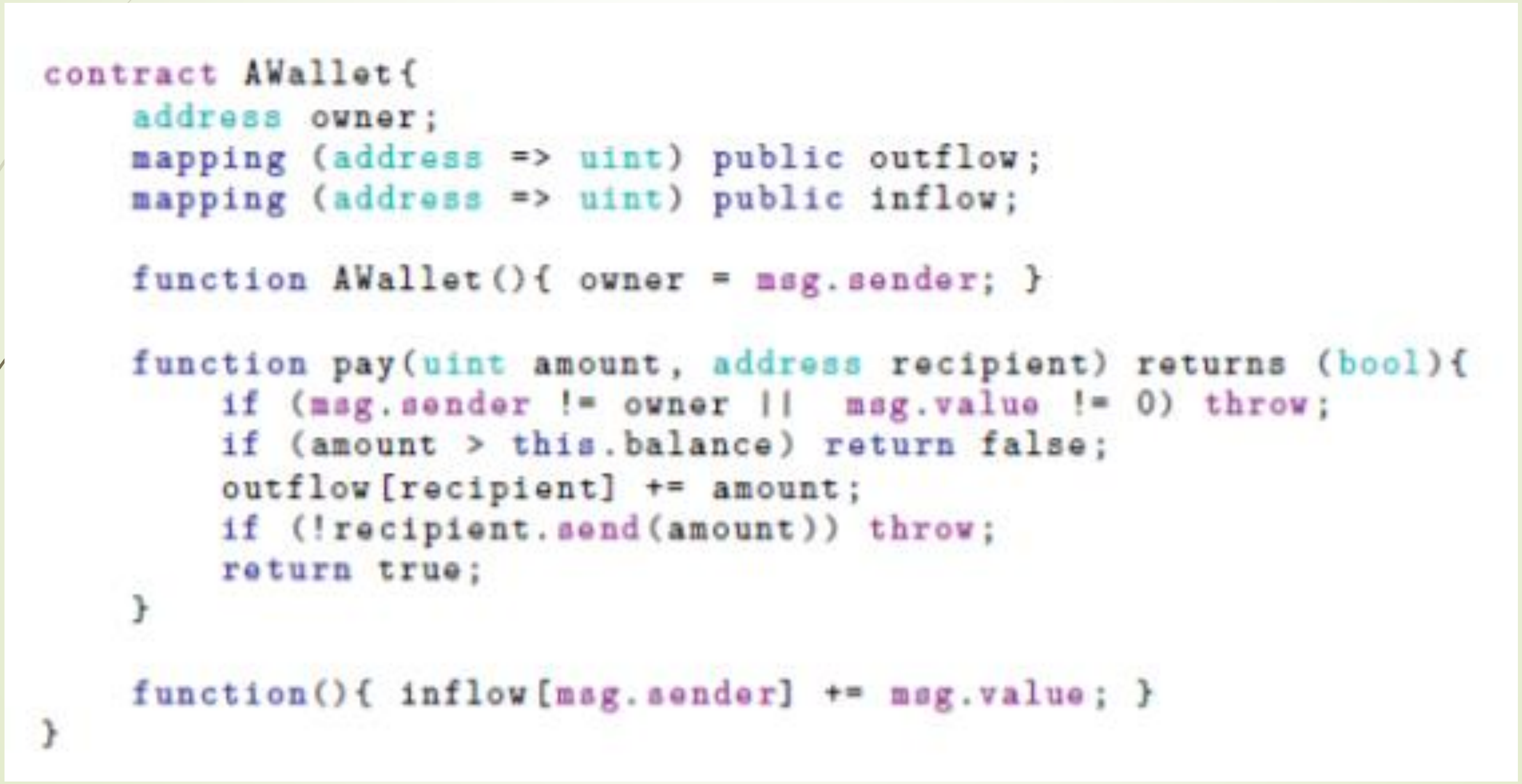
- Smart contracts are more alike with the concept of classes in object oriented programming
- Smart contracts represent self-autonomous and self-verifying agents stored in the blockchain.
- Once they are deployed in the blockchain they have their unique address which the users/clients can use to interact with and it is referred to as 'contract account'.
- Also, the code that is stored in blockchain after deployment is a low-level stack-based bytecode (EVM code) representative of the high-level programming language (a JavaScript like language) in which the smart contracts are initially written.
- As a result, it can be said that since the bytecode is publicly available from the blockchain, smart contracts' behaviour is completely predictable and its code can be inspected by every node in the network


Miners
Contracts
Users
Money
Storage
Code
Data
Mined Block
Blockchain
Block # i
Block # i + 1
Block # i + 2
Time

Solidity

- **Solidity** is a contract-oriented programming language for writing smart contracts that run on the EVM.
- It is used for implementing smart contracts on various blockchain platforms.

Sample Code



```
contract AWallet{
    address owner;
    mapping (address => uint) public outflow;
    mapping (address => uint) public inflow;

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

    function pay(uint amount, address recipient) returns (bool){
        if (msg.sender != owner ||  msg.value != 0) throw;
        if (amount > this.balance) return false;
        outflow[recipient] += amount;
        if (!recipient.send(amount)) throw;
        return true;
    }

    function(){ inflow[msg.sender] += msg.value; }
}
```

What is an EVM???

- Ethereum Virtual Machine or the EVM is designed to be the runtime environment for the smart contracts.
- The Ethereum Virtual Machine possesses its own programming language, known as the ‘EVM bytecode’.
- When code is written in higher-level programming languages such as Ethereum’s contract-orientated language Solidity, this code can then be compiled to the EVM bytecode, so that the Ethereum Virtual Machine can understand what has been written.

Security Implications

- Miners can take advantage of how they order operations in the Smart Contracts to benefit themselves, provided Miners are also the owners or users of the Smart Contracts
- Methods publicly defined in a Smart Contract can be called by anyone
- A vulnerable Smart Contract once deployed, can not be undone

Lets Analyse the issue!!!

Access Specifiers

- Public
- Private
- Internal
- External

Exception Handling

- Execution logic runs out of gas
- The logic invokes throw
- Inconsistent methods that throw exceptions

Invalid Random Entropy Sources

- Block.number
- Block.gaslimit
- Block.blockhash(block.number)
- Block.timestamp

 and many more

Note: Miners can take advantage of this entropy because the above values can be manipulated

Sending Money through Smart Contracts

- Address.call.value()
- Address.send()
- Address.transfer()

Note: All the above methods trigger **fallback** function in the contract

Invoking Contracts Directly

- Call()
- delegateCall()
- Directly -> ExternalContract.methodToInvoke()

Fallback Functions

- Called when no method matches from an external request
- Called when users send money through send(), transfer(), call.value() method
- Need payable modifier to receive money

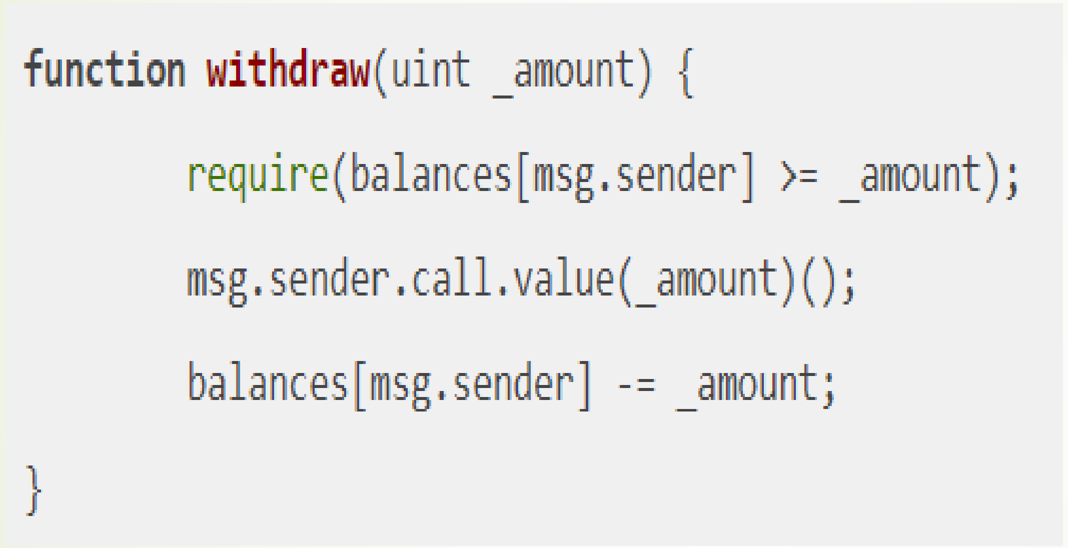
Here are some more!

- Tx.origin
- Dynamic Libraries
- Transaction Order Dependence
- Timestamp Dependency

Some Famous Attacks

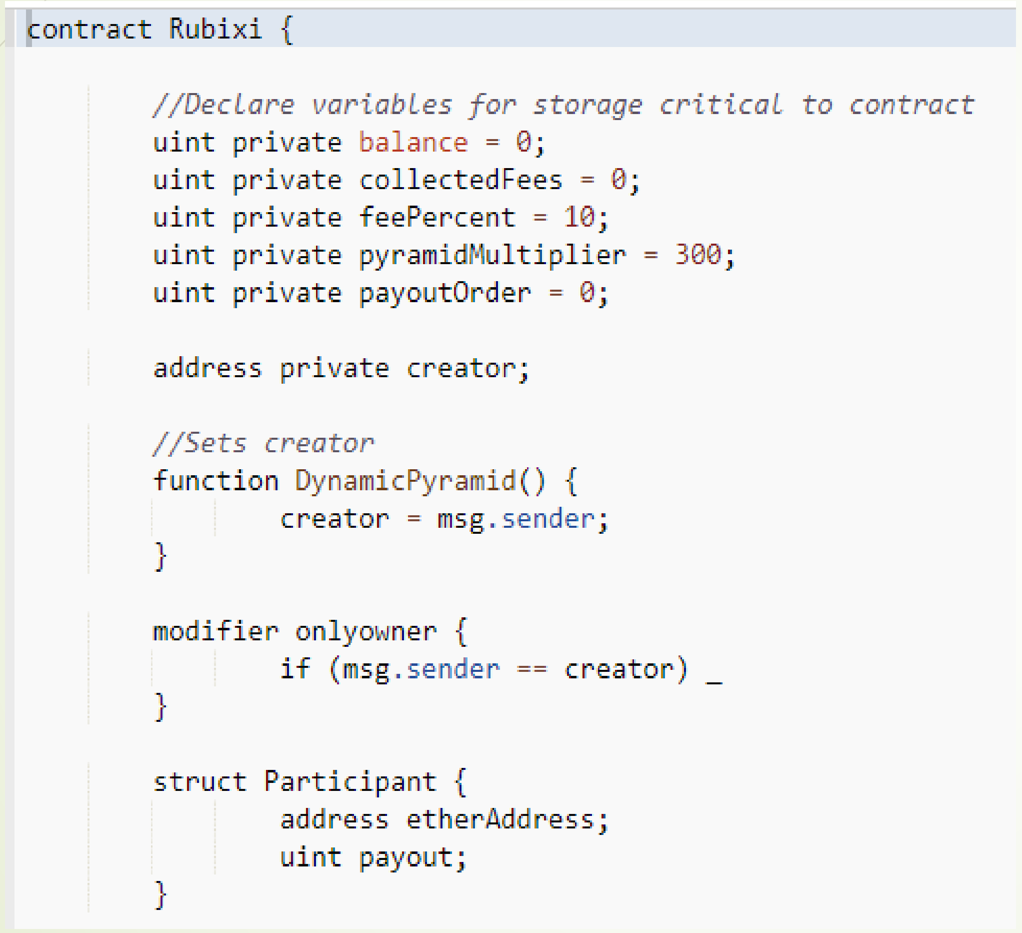
- Reentrancy
- Wrong Constructor Name
- Denial Of Service
- Integer Underflow/Overflow

Reentrancy

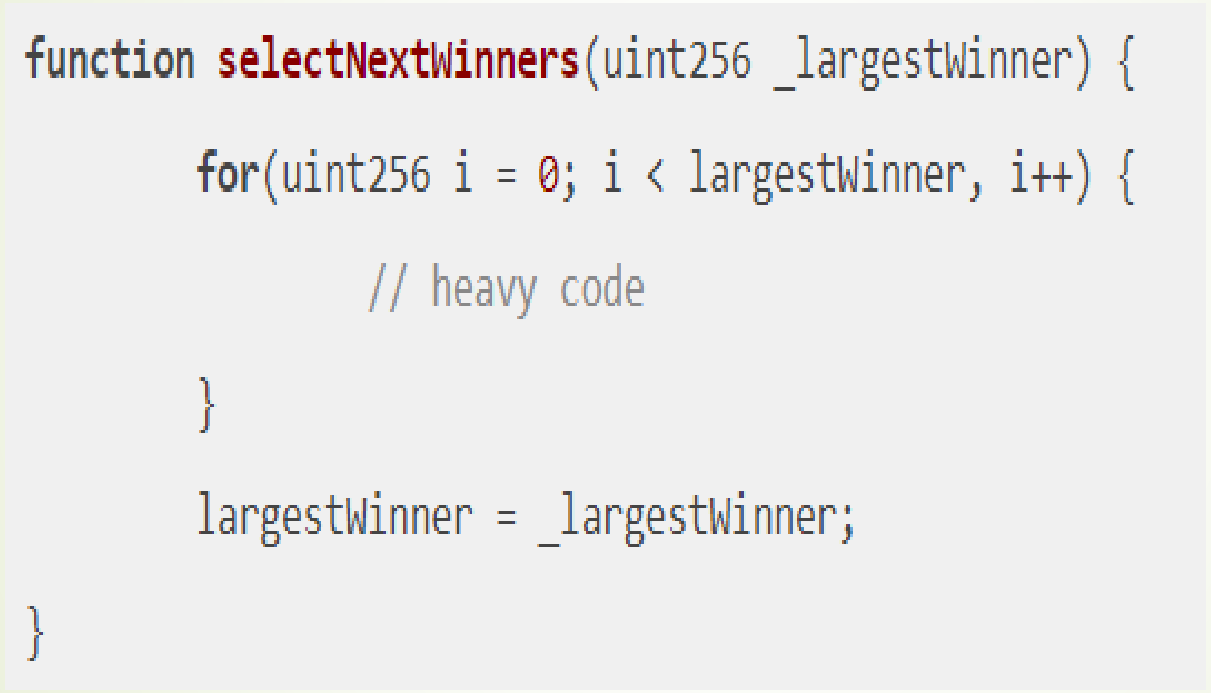


```
function withdraw(uint _amount) {

        require(balances[msg.sender] >= _amount);

        msg.sender.call.value(_amount)();

        balances[msg.sender] -= _amount;

}
```

Wrong Constructor Name

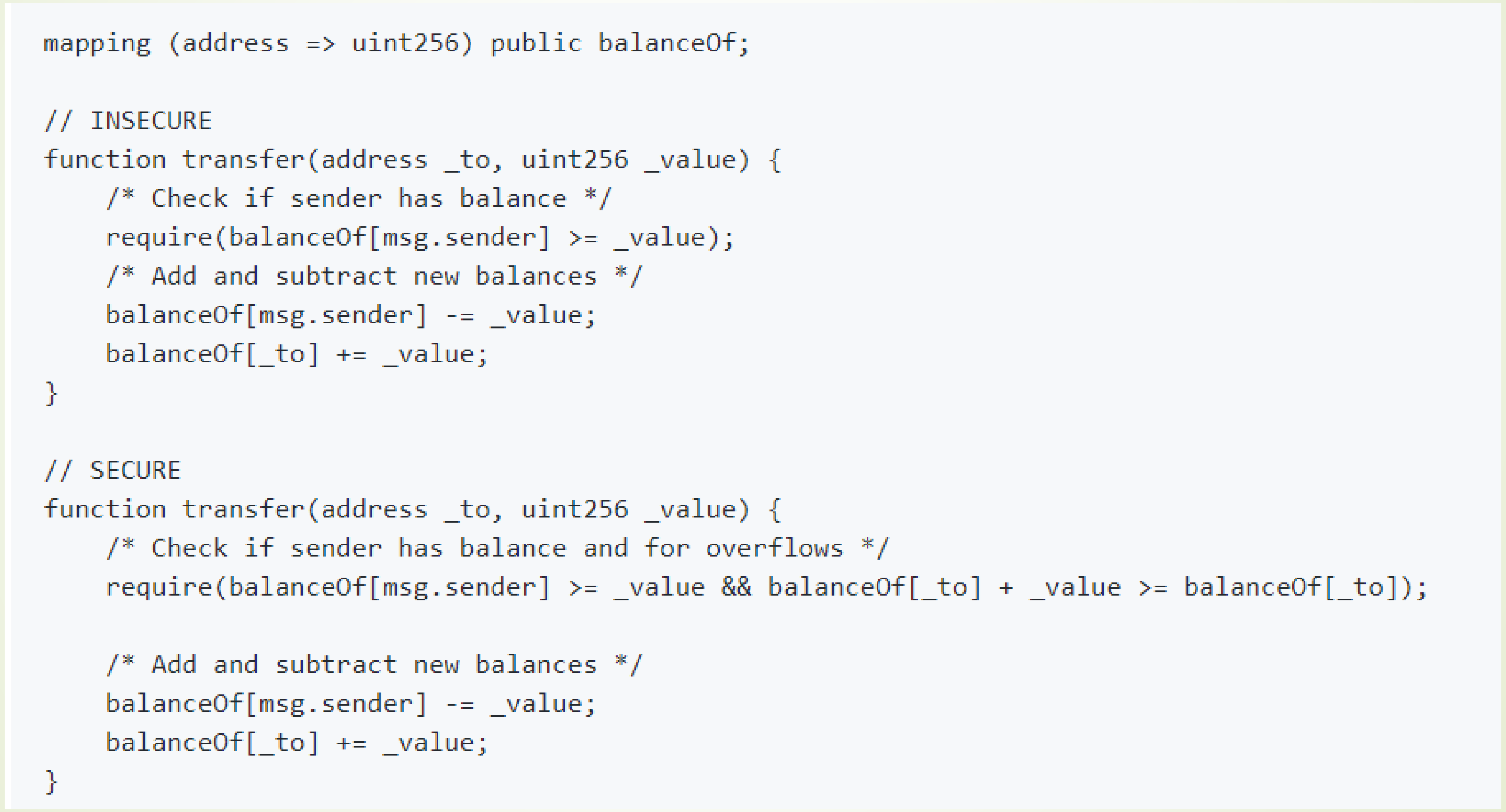

```
contract Rubixi {

        //Declare variables for storage critical to contract
        uint private balance = 0;
        uint private collectedFees = 0;
        uint private feePercent = 10;
        uint private pyramidMultiplier = 300;
        uint private payoutOrder = 0;

        address private creator;

        //Sets creator
        function DynamicPyramid() {
                creator = msg.sender;
        }

        modifier onlyowner {
                if (msg.sender == creator) _
        }

        struct Participant {
                address etherAddress;
                uint payout;
        }
```

Denial Of Service



```
function selectNextWinners(uint256 _largestWinner) {
        for(uint256 i = 0; i < largestWinner, i++) {
                // heavy code
        }
        largestWinner = _largestWinner;
}
```

Integer Underflow/Overflow



```
mapping (address => uint256) public balanceOf;

// INSECURE
function transfer(address _to, uint256 _value) {
    /* Check if sender has balance */
    require(balanceOf[msg.sender] >= _value);
    /* Add and subtract new balances */
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
}

// SECURE
function transfer(address _to, uint256 _value) {
    /* Check if sender has balance and for overflows */
    require(balanceOf[msg.sender] >= _value && balanceOf[_to] + _value >= balanceOf[_to]);

    /* Add and subtract new balances */
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
}
```

Any Questions???

References

- https://brage.bibsys.no/xmlui/bitstream/handle/11250/2479191/18400_FULLTEXT.pdf?sequence=1
- https://www.youtube.com/watch?reload=9&v=apCGPh7tKhw
- https://eprint.iacr.org/2016/1007.pdf

- Thank you