Audit Report

VIC Rewards Smart Contract

Prepared by:

Ofnog Technologies Pvt. Ltd

Corporate Office:
516, Tower B4, Spaze
itech park Gurgaon
Haryana 122001 India
info@ofnog.com
www.ofnog.com
CIN: U72200HR2018PTC072925

Introduction

This is a technical audit for VIC token smart contract. This documents outlines our methodology, limitations and results for our security audit.

Token name – VIC (Vitality Coin) Rewards

Token Symbol - VIC

Decimals allowed - 6

Token Total Supply - 5,000,000,000

Synopsis

Overall, the code demonstrates high code quality standards adopted and effective use of concept and modularity. VIC smart contract development team demonstrated high technical capabilities, both in the design of the architecture and in the implementation.

Code Analysis

Besides, the results of the automated analysis, manual verification was also taken into account. The complete contract was manually analysed, every logic was checked and compared with the one described in the whitepaper. The manual analysis of code confirms that the Contract does not contain any serious susceptibility. No divergence was found between the logic in Smart Contract and the whitepaper.

Scope

This audit is into the technical and security aspects of the VIC smart contract. The key aim of this audit is to ensure that tokens to be distributed to the investors are secure and calculations of the amount is exact. The next aim of this audit is to ensure the implementation of token mechanism i.e. the Contract must follow all the ERC20 Standards. The audit of Smart Contract also checks the coded algorithms works as expected.

Ofnog Technologies is one of the parties that independently audited the VIC Smart Contract. This audit is purely technical and is not an investment advice. The scope of the audit is limited to the following source code file:

- **Filename:** VIC.sol
- **Github Link:** https://github.com/VIC/VICSmartContract/blob/master/root/VIC.sol
- **Commit Hash:** d0e33cf462e71afa336244039cea33b9b6c57f8e

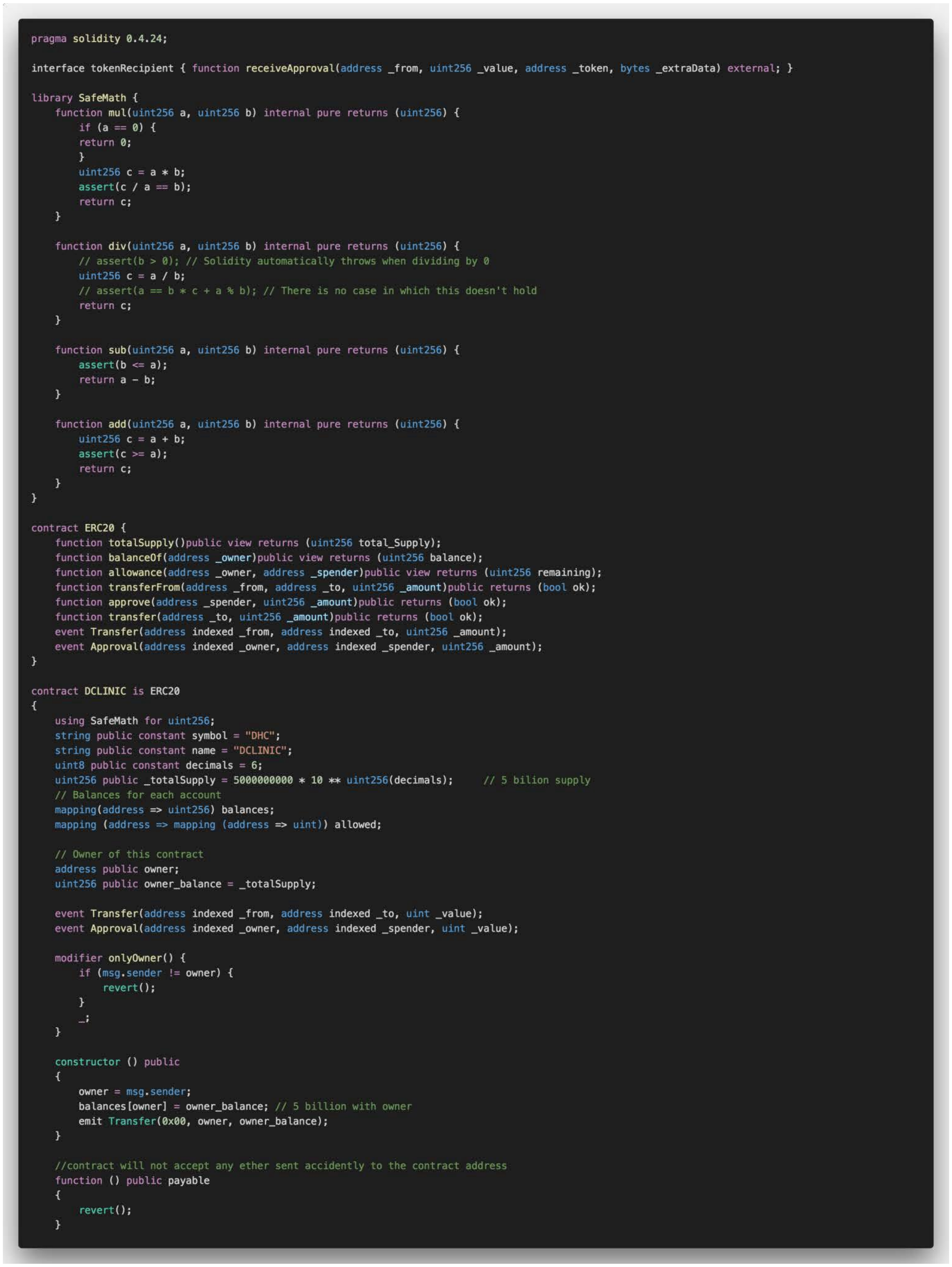
Traditional Way of Software Development

The code was provided to the auditors on Github. The codebase was properly version controlled.The code is written for Solidity version 0.4.24.

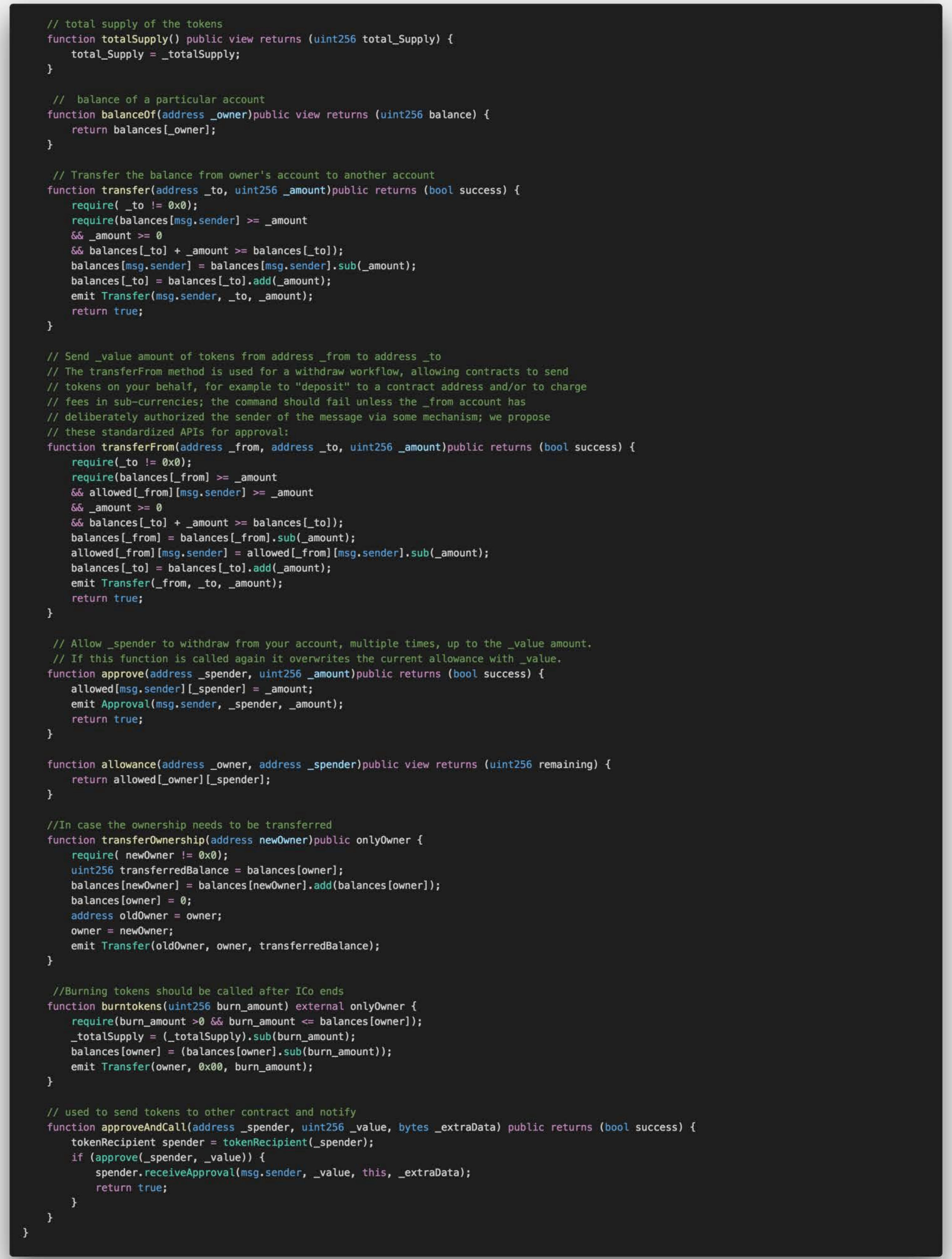
The codebase uses community administered high quality Open Zeppelin framework. This software development practices and components match the expected community standards.

VIC Smart Contract Address: 0xc9036aa3687f51265e97741e92d30dddfda76510

Solidity Code:



```
pragma solidity 0.4.24;

interface tokenRecipient { function receiveApproval(address _from, uint256 _value, address _token, bytes _extraData) external; }

library SafeMath {
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
        return 0;
        }
        uint256 c = a * b;
        assert(c / a == b);
        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        // assert(b > 0); // Solidity automatically throws when dividing by 0
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold
        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        assert(b <= a);
        return a - b;
    }

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        assert(c >= a);
        return c;
    }
}

contract ERC20 {
    function totalSupply()public view returns (uint256 total_Supply);
    function balanceOf(address _owner)public view returns (uint256 balance);
    function allowance(address _owner, address _spender)public view returns (uint256 remaining);
    function transferFrom(address _from, address _to, uint256 _amount)public returns (bool ok);
    function approve(address _spender, uint256 _amount)public returns (bool ok);
    function transfer(address _to, uint256 _amount)public returns (bool ok);
    event Transfer(address indexed _from, address indexed _to, uint256 _amount);
    event Approval(address indexed _owner, address indexed _spender, uint256 _amount);
}

contract DCLINIC is ERC20
{
    using SafeMath for uint256;
    string public constant symbol = "DHC";
    string public constant name = "DCLINIC";
    uint8 public constant decimals = 6;
    uint256 public _totalSupply = 5000000000 * 10 ** uint256(decimals);     // 5 bilion supply
    // Balances for each account
    mapping(address => uint256) balances;
    mapping (address => mapping (address => uint)) allowed;

    // Owner of this contract
    address public owner;
    uint256 public owner_balance = _totalSupply;

    event Transfer(address indexed _from, address indexed _to, uint _value);
    event Approval(address indexed _owner, address indexed _spender, uint _value);

    modifier onlyOwner() {
        if (msg.sender != owner) {
            revert();
        }
        _;
    }

    constructor () public
    {
        owner = msg.sender;
        balances[owner] = owner_balance; // 5 billion with owner
        emit Transfer(0x00, owner, owner_balance);
    }

    //contract will not accept any ether sent accidently to the contract address
    function () public payable
    {
        revert();
    }
```



```
    // total supply of the tokens
    function totalSupply() public view returns (uint256 total_Supply) {
        total_Supply = _totalSupply;
    }

     //  balance of a particular account
    function balanceOf(address _owner)public view returns (uint256 balance) {
        return balances[_owner];
    }

     // Transfer the balance from owner's account to another account
    function transfer(address _to, uint256 _amount)public returns (bool success) {
        require( _to != 0x0);
        require(balances[msg.sender] >= _amount
        && _amount >= 0
        && balances[_to] + _amount >= balances[_to]);
        balances[msg.sender] = balances[msg.sender].sub(_amount);
        balances[_to] = balances[_to].add(_amount);
        emit Transfer(msg.sender, _to, _amount);
        return true;
    }

    // Send _value amount of tokens from address _from to address _to
    // The transferFrom method is used for a withdraw workflow, allowing contracts to send
    // tokens on your behalf, for example to "deposit" to a contract address and/or to charge
    // fees in sub-currencies; the command should fail unless the _from account has
    // deliberately authorized the sender of the message via some mechanism; we propose
    // these standardized APIs for approval:
    function transferFrom(address _from, address _to, uint256 _amount)public returns (bool success) {
        require(_to != 0x0);
        require(balances[_from] >= _amount
        && allowed[_from][msg.sender] >= _amount
        && _amount >= 0
        && balances[_to] + _amount >= balances[_to]);
        balances[_from] = balances[_from].sub(_amount);
        allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_amount);
        balances[_to] = balances[_to].add(_amount);
        emit Transfer(_from, _to, _amount);
        return true;
    }

     // Allow _spender to withdraw from your account, multiple times, up to the _value amount.
     // If this function is called again it overwrites the current allowance with _value.
    function approve(address _spender, uint256 _amount)public returns (bool success) {
        allowed[msg.sender][_spender] = _amount;
        emit Approval(msg.sender, _spender, _amount);
        return true;
    }

    function allowance(address _owner, address _spender)public view returns (uint256 remaining) {
        return allowed[_owner][_spender];
    }

    //In case the ownership needs to be transferred
    function transferOwnership(address newOwner)public onlyOwner {
        require( newOwner != 0x0);
        uint256 transferredBalance = balances[owner];
        balances[newOwner] = balances[newOwner].add(balances[owner]);
        balances[owner] = 0;
        address oldOwner = owner;
        owner = newOwner;
        emit Transfer(oldOwner, owner, transferredBalance);
    }

     //Burning tokens should be called after ICo ends
    function burntokens(uint256 burn_amount) external onlyOwner {
        require(burn_amount >0 && burn_amount <= balances[owner]);
        _totalSupply = (_totalSupply).sub(burn_amount);
        balances[owner] = (balances[owner].sub(burn_amount));
        emit Transfer(owner, 0x00, burn_amount);
    }

    // used to send tokens to other contract and notify
    function approveAndCall(address _spender, uint256 _value, bytes _extraData) public returns (bool success) {
        tokenRecipient spender = tokenRecipient(_spender);
        if (approve(_spender, _value)) {
            spender.receiveApproval(msg.sender, _value, this, _extraData);
            return true;
        }
    }
}
```

Overview

The project has only one file, the VIC.sol file which contains 172 lines of Solidity code. All the functions and state variables are well commented using the Natspec documentation for the functions which is good to understand quickly how everything is supposed to work.

Testing

Primary checks followed during testing of Smart Contract is to see that if code :

- We check the Smart Contracts Logic and compare it with one described in Whitepaper.
- The contract code should follow the Conditions and logic as per user request.
- We deploy the Contract and run the Tests.
- We make sure that Contract does not lose any money/Ether.

Vulnerabilities Check

Smart Contract was scanned for commonly known and more specific vulnerabilities.

Following are the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- **TimeStamp Dependence:** The timestamp of the block can be manipulated by the miner, and so should not be used for critical components of the contract. *Block numbers* and *average block time* can be used to estimate time (suggested). VIC smart contract does not have any timestamp dependence in its code.

- **Gas Limit and Loops:** Loops that do not have a fixed number of iterations, hence due to normal operation, the number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. VIC smart contract is free from the gas limit check as the contract code does not contain any loop in its code.

- **Compiler Version:** Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs. VIC smart contract is locked to a specific compiler version of 0.4.24 which is good coding practice.

- **ERC20 Standards:** VIC smart contract follows all the universal ERC20 coding standards and implements all its functions and events in the contract code.

- **Redundant fallback function:** The standard execution cost of a fallback function should be less than 2300 gas, VIC smart contract code does not have any fallback function, hence it is free from this vulnerability.

- **Unchecked math:** Need to guard uint overflow or security flaws by implementing the proper maths logic checks. The VIC smart contract uses the popular SafeMath library for critical operations to avoid arithmetic over or underflow and safeguard against unwanted behaviour. In particular the balances variable is updated using the safemath operation.

- **Exception disorder:** When an exception is thrown, it cannot be caught: the execution stops, the fee is lost. The irregularity in how exceptions are handled may affect the security of contracts.

- **Unsafe type Inference:** It is not always necessary to explicitly specify the type of a variable, the compiler automatically infers it from the type of the first expression that is assigned to the variable.

- **Reentrancy:** The reentrancy attack consists of the recursively calling a method to extract ether from a contract if user is not updating the balance of the sender before sending the ether.
  In VIC smart contract calls to external functions happen after any changes to state variables in the contract so the contract is not vulnerable to a reentrancy exploit. The VIC smart contract does not have any vulnerabilities against reentrancy attack.

- **DoS with (Unexpected) Throw:** The Contract code can be vulnerable to the Call Depth Attack! So instead, code should have a pull payment system instead of push. The VIC smart contract does not implement any payment related scenario thus it is not vulnerable to this attack.

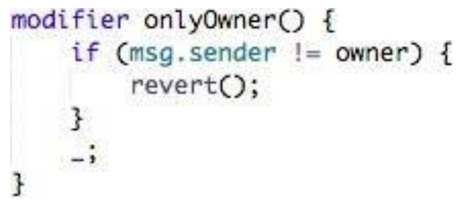
- **DoS with Block Gas Limit:** In a contract by paying out to everyone at once, contract risk running into the block gas limit. Each Ethereum block can process a certain maximum amount of computation. If one try to go over that, the transaction will fail. Therefore again push over pull payment is suggested to remove the above issue.

- **Explicit Visibility in functions and state variables:** Explicit visibility in the function and state variables are provided. Visibility like external, internal, private and public is used and defined properly.

Features in Smart Contract

1. **Ownable -** The smart contract and the tokens implemented it are owned by a particular entity (ethereum address). To use those tokens the owner will have to sign with his private key. As we deploy the smart contract on Ethereum blockchain, initially all the tokens will be owned by the owner of the smart contract i.e. the entity offering the crowdsale.

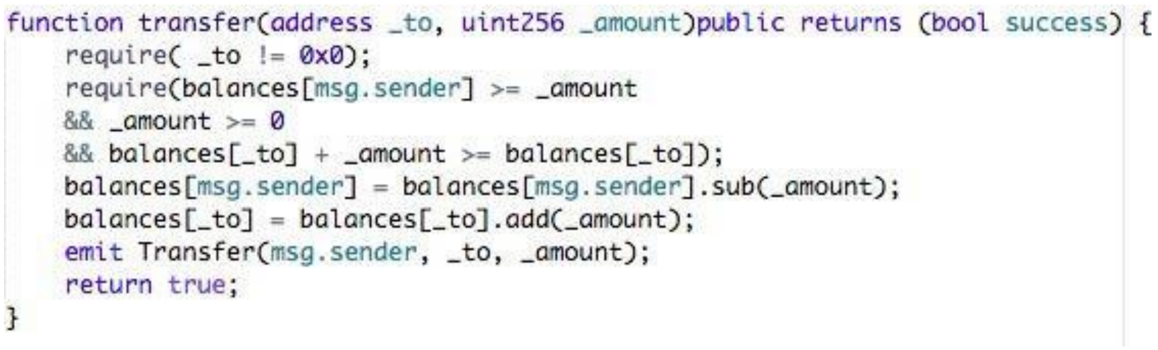
   Code: Line 63 - 68

```
modifier onlyOwner() {
    if (msg.sender != owner) {
        revert();
    }
    _;
}
```

2. **Transferrable -** The tokens can be transferred from one entity to other(like to exchanges for trading). This transfer can be made by using any ethereum wallet which supports ERC20 token standard, for eg. MyEtherWallet, Mist Etc.

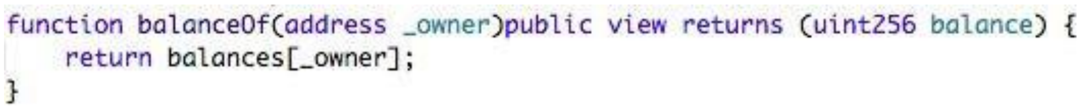
Code: Line 95 - 104



```
function transfer(address _to, uint256 _amount)public returns (bool success) {
    require( _to != 0x0);
    require(balances[msg.sender] >= _amount
    && _amount >= 0
    && balances[_to] + _amount >= balances[_to]);
    balances[msg.sender] = balances[msg.sender].sub(_amount);
    balances[_to] = balances[_to].add(_amount);
    emit Transfer(msg.sender, _to, _amount);
    return true;
}
```

3. **Viewable Tokens -** As you already may be aware with the transparent nature of blockchain, all the tokens holders and their exact balance is made clearly visible, on Ethereum blockchain explorers like Etherscan and Ethplorer. Their are functions which are implemented to return informations like token balance of any particular token holder, the token allowance amount of any particular Token holder, which he has allowed to any other entity(Ethereum address).

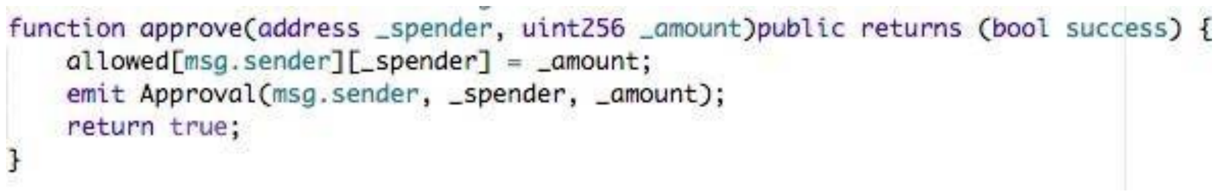
Code: Line 90 - 92



```
function balanceOf(address _owner)public view returns (uint256 balance) {
    return balances[_owner];
}
```

4. **Approvable -** Any Token holder if he wills, can approve some other address, who will on his behalf transfer the approved amount of tokens from token holder¶s to others.

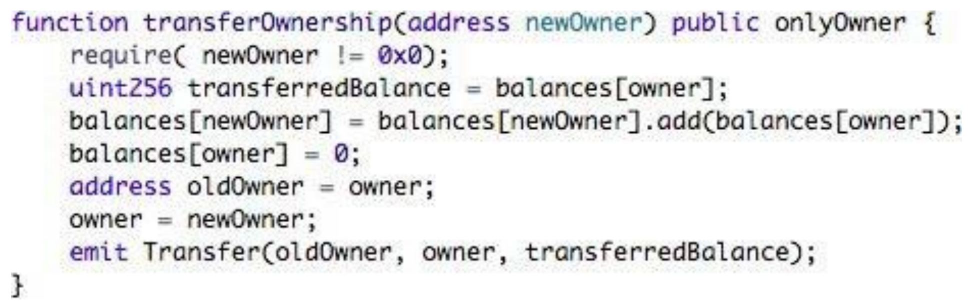
Code: Line 131 - 135



```
function approve(address _spender, uint256 _amount)public returns (bool success) {
    allowed[msg.sender][_spender] = _amount;
    emit Approval(msg.sender, _spender, _amount);
    return true;
}
```

5. **Transfer Ownership -**Ownership of the smart contract can be transferred to a new ethereum address(entity), this can be done only by the current owner of the smart contract.

Code: **Line 142 - 150**



```
function transferOwnership(address newOwner) public onlyOwner {
    require( newOwner != 0x0);
    uint256 transferredBalance = balances[owner];
    balances[newOwner] = balances[newOwner].add(balances[owner]);
    balances[owner] = 0;
    address oldOwner = owner;
    owner = newOwner;
    emit Transfer(oldOwner, owner, transferredBalance);
}
```

Risk

The VIC Smart Contract has no risk of losing any amounts of ethers in case of external attack or a bug, as contract does not takes any kind of funds from the user. If anyone tries to send any amount of ether to the contract address, the transaction will cancel itself and no ether comes to the contracts.

The flow of tokens from this VIC contract can be controlled using a script running on the backend and visually through Etherscan.io. By using Etherscan.io working of code can be verified which will lead the compiled code getting matched with the bytecode of deployed smart contract in the blockchain.

Therefore, there is no anomalous gap in the VIC smart contract, tokens will be distributed to all the investors as per the amount paid by them during Pre-ICO/ ICO. As all the funds are held with owner's address thus he will be distributing all the tokens, so any possible losses due to flaws in the VIC smart contract is not possible to occur.

Conclusion

In this report, we have concluded about the security of VIC Smart Contract. The smart contract has been analysed under different facets. Code quality is very good, and well modularised. We found that VIC smart contract adapts a very good coding practice and have clean, documented code. Smart Contract logic was checked and compared with the one described in the whitepaper. No discrepancies were found.