Audit Report

DCLINIC 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 DCLINIC token smart contract. This documents outlines our methodology, limitations and results for our security audit.

Token name - DCLINIC

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. DCLINIC 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 DCLINIC 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 DCLINIC 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 - dClinic.sol
- **Github Link:** https://github.com/dclinicpteltd/smartContract/blob/master/VIC%20-%20dClinic
- **Commit Hash:** 8111a327156f62a52614f726f676d7f0a275fda9

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.

DCLINIC Smart Contract Address: 0xe667539c2e470f2da38bf1ff5d154f3af37739c76510

Solidity Code:

```
pragma solidity ^0.4.24;

/**
 * @title SafeMath
 * @dev Math operations with safety checks that revert on error
 */
library SafeMath {

  /**
  * @dev Multiplies two numbers, reverts on overflow.
  */
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
    // benefit is lost if 'b' is also tested.
    // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522
    if (a == 0) {
      return 0;
    }

    uint256 c = a * b;
    require(c / a == b);

    return c;
  }

  /**
  * @dev Integer division of two numbers truncating the quotient, reverts on division by zero.
  */
  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b > 0); // Solidity only automatically asserts 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;
  }

  /**
  * @dev Subtracts two numbers, reverts on overflow (i.e. if subtrahend is greater than minuend).
  */
  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a);
    uint256 c = a - b;

    return c;
  }

  /**
```

```
   * @dev Adds two numbers, reverts on overflow.
   */
  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    require(c >= a);

    return c;
  }

  /**
   * @dev Divides two numbers and returns the remainder (unsigned integer modulo),
   * reverts when dividing by zero.
   */
  function mod(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b != 0);
    return a % b;
  }
}

/**
 * @title ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/20
 */
interface IERC20 {
  function totalSupply() external view returns (uint256);

  function balanceOf(address who) external view returns (uint256);

  function allowance(address owner, address spender)
    external view returns (uint256);

  function transfer(address to, uint256 value) external returns (bool);

  function approve(address spender, uint256 value)
    external returns (bool);

  function transferFrom(address from, address to, uint256 value)
    external returns (bool);

  event Transfer(
    address indexed from,
    address indexed to,
    uint256 value
  );

  event Approval(
    address indexed owner,
```

```
    address indexed spender,
    uint256 value
  );
}

contract ERC20 is IERC20 {
  using SafeMath for uint256;

  mapping (address => uint256) private _balances;

  mapping (address => mapping (address => uint256)) private _allowed;

  uint256 private _totalSupply;
  string private _name;
  string private _symbol;
  uint8 private _decimals;

  constructor(string name, string symbol, uint8 decimals, uint256 totalSupply) public {
    _name = name;
    _symbol = symbol;
    _decimals = decimals;
    _totalSupply = totalSupply;
    _balances[msg.sender] = _balances[msg.sender].add(_totalSupply);
    emit Transfer(address(0), msg.sender, totalSupply);
  }

  /**
   * @return the name of the token.
   */
  function name() public view returns(string) {
    return _name;
  }

  /**
   * @return the symbol of the token.
   */
  function symbol() public view returns(string) {
    return _symbol;
  }

  /**
   * @return the number of decimals of the token.
   */
  function decimals() public view returns(uint8) {
    return _decimals;
  }

  /**
```

```
/**
* @dev Total number of tokens in existence
*/
function totalSupply() public view returns (uint256) {
  return _totalSupply;
}

/**
* @dev Gets the balance of the specified address.
* @param owner The address to query the balance of.
* @return An uint256 representing the amount owned by the passed address.
*/
function balanceOf(address owner) public view returns (uint256) {
  return _balances[owner];
}

/**
 * @dev Function to check the amount of tokens that an owner allowed to a spender.
 * @param owner address The address which owns the funds.
 * @param spender address The address which will spend the funds.
 * @return A uint256 specifying the amount of tokens still available for the spender.
 */
function allowance(
  address owner,
  address spender
 )
  public
  view
  returns (uint256)
{
  return _allowed[owner][spender];
}

/**
* @dev Transfer token for a specified address
* @param to The address to transfer to.
* @param value The amount to be transferred.
*/
function transfer(address to, uint256 value) public returns (bool) {
  _transfer(msg.sender, to, value);
  return true;
}

/**
 * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
 * Beware that changing an allowance with this method brings the risk that someone may use both the old
 * and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this
 * race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards:
```

```
 * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
 * @param spender The address which will spend the funds.
 * @param value The amount of tokens to be spent.
 */
function approve(address spender, uint256 value) public returns (bool) {
  require(spender != address(0));

  _allowed[msg.sender][spender] = value;
  emit Approval(msg.sender, spender, value);
  return true;
}

/**
 * @dev Transfer tokens from one address to another
 * @param from address The address which you want to send tokens from
 * @param to address The address which you want to transfer to
 * @param value uint256 the amount of tokens to be transferred
 */
function transferFrom(
  address from,
  address to,
  uint256 value
)
  public
  returns (bool)
{
  require(value <= _allowed[from][msg.sender]);

  _allowed[from][msg.sender] = _allowed[from][msg.sender].sub(value);
  _transfer(from, to, value);
  return true;
}

/**
 * @dev Increase the amount of tokens that an owner allowed to a spender.
 * approve should be called when allowed_[_spender] == 0. To increment
 * allowed value is better to use this function to avoid 2 calls (and wait until
 * the first transaction is mined)
 * From MonolithDAO Token.sol
 * @param spender The address which will spend the funds.
 * @param addedValue The amount of tokens to increase the allowance by.
 */
function increaseAllowance(
  address spender,
  uint256 addedValue
)
  public
  returns (bool)
```

```
  {
    require(spender != address(0));

    _allowed[msg.sender][spender] = (
      _allowed[msg.sender][spender].add(addedValue));
    emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
    return true;
  }

  /**
   * @dev Decrease the amount of tokens that an owner allowed to a spender.
   * approve should be called when allowed_[_spender] == 0. To decrement
   * allowed value is better to use this function to avoid 2 calls (and wait unt
   * the first transaction is mined)
   * From MonolithDAO Token.sol
   * @param spender The address which will spend the funds.
   * @param subtractedValue The amount of tokens to decrease the allowance by.
   */
  function decreaseAllowance(
    address spender,
    uint256 subtractedValue
  )
    public
    returns (bool)
  {
    require(spender != address(0));

    _allowed[msg.sender][spender] = (
      _allowed[msg.sender][spender].sub(subtractedValue));
    emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
    return true;
  }

  /**
  * @dev Transfer token for a specified addresses
  * @param from The address to transfer from.
  * @param to The address to transfer to.
  * @param value The amount to be transferred.
  */
  function _transfer(address from, address to, uint256 value) internal {
    require(value <= _balances[from]);
    require(to != address(0));

    _balances[from] = _balances[from].sub(value);
    _balances[to] = _balances[to].add(value);
    emit Transfer(from, to, value);
  }
}
```

Overview

The project has only one file, the VIC - dClinic.sol file which contains 285 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). DCLINIC 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. DCLINIC 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. DCLINIC smart contract is locked to a specific compiler version of 0.4.24 which is good coding practice.

- **ERC20 Standards:** DCLINIC 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, DCLINIC 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 DCLINIC 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 DCLINIC 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 DCLINIC 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 DCLINIC 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.

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 277 – 284

```
/**
* @dev Transfer token for a specified addresses
* @param from The address to transfer from.
* @param to The address to transfer to.
* @param value The amount to be transferred.
*/
function _transfer(address from, address to, uint256 value) internal {
  require(value <= _balances[from]);
  require(to != address(0));

  _balances[from] = _balances[from].sub(value);
  _balances[to] = _balances[to].add(value);
  emit Transfer(from, to, value);
}
```

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 149 – 156

```
/**
* @dev Gets the balance of the specified address.
* @param owner The address to query the balance of.
* @return An uint256 representing the amount owned by the passed
address.
*/
function balanceOf(address owner) public view returns (uint256) {
  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 185 – 200

```
/**
 * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
 * Beware that changing an allowance with this method brings the risk that someone may use both the old
 * and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this
 * race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards:
 * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
 * @param spender The address which will spend the funds.
 * @param value The amount of tokens to be spent.
 */
function approve(address spender, uint256 value) public returns (bool) {
  require(spender != address(0));

  _allowed[msg.sender][spender] = value;
  emit Approval(msg.sender, spender, value);
  return true;
}
```

Risk

The DCLINIC 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 DCLINIC 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 DCLINIC 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 DCLINIC smart contract is not possible to occur.

Conclusion

In this report, we have concluded about the security of DCLINIC Smart Contract. The smart contract has been analysed under different facets. Code quality is very good, and well modularised. We found that DCLINIC 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.