Smart Contract Security Audit Report

[2021]

The SlowMist Security Team received the team's application for smart contract security audit of the TouchCon on 2021.11.22. The following are the details and results of this smart contract security audit:

Token Name :

TouchCon

The contract address :

https://etherscan.io/address/0x549905519f9e06d55d7dfcd4d54817780f6b93e8

The audit items and results :

(Other unknown security vulnerabilities are not included in the audit responsibility scope)

NO.	Audit Items	Result
1	Replay Vulnerability	Passed
2	Denial of Service Vulnerability	Passed
3	Race Conditions Vulnerability	Passed
4	Authority Control Vulnerability	Passed
5	Integer Overflow and Underflow Vulnerability	Passed
6	Gas Optimization Audit	Passed
7	Design Logic Audit	Passed
8	Uninitialized Storage Pointers Vulnerability	Passed
9	Arithmetic Accuracy Deviation Vulnerability	Passed
10	"False top-up" Vulnerability	Passed
11	Malicious Event Log Audit	Passed
12	Scoping and Declarations Audit	Passed

NO.	Audit Items	Result
13	Safety Design Audit	Passed

Audit Result : Passed

Audit Number : 0X002111240001

Audit Date : 2021.11.22 - 2021.11.24

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that does not include the tokenVault part. The total amount of contract tokens can be changed, and owners can burn their own tokens and other users' tokens through the burnLockupListAmount function and the burnAdminAmount. Use the SafeMath security module, which is the recommended method. The contract has no issues with overflow and competition conditions.

During the audit, we found the following information:

1. The owner role can mint any number of tokens through the mint function, but here is the transaction for executing the finishMinting function by the owner role to stop minting:

   https://etherscan.io/tx/0x34188e14ae3bbd1fdb384545687adc3cf6db0ea99a5cdad18065f07a9b29ea3a

   In this transaction, the owner executing the finishMinting to set the mintingFinished value to true, and this state cannot be changed after it is set to true, and when the mintingFinished value is true, the contract can no longer mint tokens through the mint function.

2. The owner role can lock and unlock any user's account, and the owner role can burn any users' assets through the burnLockupListAmount function. Though communication with the project team, the ownership of the contract has been transferred to the black hole address, which making the contract can no longer burn any users' assets and lock other users' accounts. Here are the transactions for transferring the ownership of the owner and operator to the black hole address:

   https://etherscan.io/tx/0xc5e73ed4f064031642e40cfe5e30663f00b1e495cdc5dec1d46e4b9792a20785

   https://etherscan.io/tx/0xdf1cd6e43aab78b9a4ae201f3934bd1e008d985cef858fdde09449479b6bbb11

In the CheckLockupList modifier, only msg.sender is judged whether it is in the lockupList, and the _from parameter is not judged whether it is in the lockupList in the transferFrom function. If the value of the

3. passed _from parameter is in lockupList when the transferFrom function is called, msg.sender can still transfer the balance of the passed _from parameter.

   However, in all transactions about the contract, the project team has only locked one address and has been unlocked, and the ownership of the project team has been transferred to the black hole address, so this has no effect on the entire contract

The source code:

```
/**
 *Submitted for verification at Etherscan.io on 2021-06-28
*/
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;

//SlowMist// SafeMath security module is used, which is a recommended approach
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);
        //SlowMist// It is recommended to replace "assert" with "require" to optimize
Gas
        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return a / b;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        assert(b <= a);
        //SlowMist// It is recommended to replace "assert" with "require" to optimize
Gas
        return a - b;
    }

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        assert(c >= a);
        //SlowMist// It is recommended to replace "assert" with "require" to optimize
Gas
        return c;
    }
}

contract ERC20Basic {
    function totalSupply() public view returns (uint256);
    function balanceOf(address who) public view returns (uint256);
    function transfer(address to, uint256 value) public returns (bool);
    event Transfer(address indexed from, address indexed to, uint256 value);
}

contract ERC20 is ERC20Basic {
    function allowance(address owner, address spender) public view returns (uint256);
    function transferFrom(address from, address to, uint256 value) public returns
(bool);
    function approve(address spender, uint256 value) public returns (bool);
    event Approval(address indexed owner, address indexed spender, uint256 value);
}

contract BasicToken is ERC20Basic {
    using SafeMath for uint256;

    mapping(address => uint256) balances;

    uint256 totalSupply_;

    function totalSupply() public view returns (uint256) {
        return totalSupply_;
    }
    function transfer(address _to, uint256 _value) public returns (bool) {
    //SlowMist// This kind of check is very good, avoiding user mistake leading to
the loss of token during transfer
        require(_to != address(0));
        require(_value <= balances[msg.sender]);

        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(_value);

        emit Transfer(msg.sender, _to, _value);
        //SlowMist// The return value conforms to the EIP20 specification
        return true;
    }

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

contract Ownable {

    address public owner;
    address public operator;

    event OwnershipTransferred(address indexed previousOwner, address indexed
newOwner);
    event OperatorTransferred(address indexed previousOperator, address indexed
newOperator);

    constructor() public {
        owner    = msg.sender;
        operator = msg.sender;
    }

    modifier onlyOwner() { require(msg.sender == owner); _; }
    modifier onlyOwnerOrOperator() { require(msg.sender == owner || msg.sender ==
operator); _; }

    function transferOwnership(address _newOwner) external onlyOwner {
        //SlowMist// This check is quite good in avoiding losing control of the
contract caused by user mistakes
        require(_newOwner != address(0));
        emit OwnershipTransferred(owner, _newOwner);
        owner = _newOwner;
    }

    function transferOperator(address _newOperator) external onlyOwner {
        require(_newOperator != address(0));
        emit OperatorTransferred(operator, _newOperator);
        operator = _newOperator;
    }
}

contract LockupList is Ownable {

    event Lock(address indexed LockedAddress);
    event Unlock(address indexed UnLockedAddress);

    mapping( address => bool ) public lockupList;

    modifier CheckLockupList { require(lockupList[msg.sender] != true); _; }

//SlowMist// The owner and operator can lock any user account through the
SetLockAddress function
    function SetLockAddress(address _lockAddress) external onlyOwnerOrOperator
returns (bool) {
        require(_lockAddress != address(0));
        require(_lockAddress != owner);
        require(lockupList[_lockAddress] != true);

        lockupList[_lockAddress] = true;

        emit Lock(_lockAddress);

        return true;
    }

//SlowMist// The owner can unlock any user account through the UnLockAddress function
    function UnLockAddress(address _unlockAddress) external onlyOwner returns (bool)
{
        require(lockupList[_unlockAddress] != false);

        lockupList[_unlockAddress] = false;

        emit Unlock(_unlockAddress);

        return true;
    }
```

```
}

contract Pausable is Ownable {
    event Pause();
    event Unpause();

    bool public paused = false;

    modifier whenNotPaused() { require(!paused); _; }
    modifier whenPaused() { require(paused); _; }
// SlowMist// Suspending all transactions upon major abnormalities is a recommended
approach.
    function pause() onlyOwnerOrOperator whenNotPaused public {
        paused = true;
        emit Pause();
    }

    function unpause() onlyOwner whenPaused public {
        paused = false;
        emit Unpause();
    }
}

contract StandardToken is ERC20, BasicToken {

    mapping (address => mapping (address => uint256)) internal allowed;

    function transferFrom(address _from, address _to, uint256 _value) public returns
(bool) {
        require(_to != address(0));
        require(_value <= balances[_from]);
        require(_value <= allowed[_from][msg.sender]);

        //SlowMist// The _from parameter is not judged whether it is in the
lockupList
        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(_value);
        allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);

        emit Transfer(_from, _to, _value);
        //SlowMist// The return value conforms to the EIP20 specification

        return true;
    }
    function approve(address _spender, uint256 _value) public returns (bool) {
        allowed[msg.sender][_spender] = _value;

        emit Approval(msg.sender, _spender, _value);
        //SlowMist// The return value conforms to the EIP20 specification
        return true;
    }

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

    function increaseApproval(address _spender, uint256 _addedValue) public returns
(bool) {
        allowed[msg.sender][_spender] = (allowed[msg.sender]
[_spender].add(_addedValue));

        emit Approval(msg.sender, _spender, allowed[msg.sender][_spender]);

        return true;
    }

    function decreaseApproval(address _spender, uint256 _subtractedValue) public
returns (bool) {
        uint256 oldValue = allowed[msg.sender][_spender];

        if (_subtractedValue > oldValue) {
        allowed[msg.sender][_spender] = 0;
        } else {
        allowed[msg.sender][_spender] = oldValue.sub(_subtractedValue);
        }

        emit Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
        return true;
    }
}

contract BurnableToken is StandardToken, Ownable {

    event BurnAdminAmount(address indexed burner, uint256 value);
    event BurnLockupListAmount(address indexed burner, uint256 value);
//SlowMist// The owner role can call burnAdminAmount function to burn his own balance
```

```
and totalsupply
    function burnAdminAmount(uint256 _value) onlyOwner public {
        require(_value <= balances[msg.sender]);

        balances[msg.sender] = balances[msg.sender].sub(_value);
        totalSupply_ = totalSupply_.sub(_value);

        emit BurnAdminAmount(msg.sender, _value);
        emit Transfer(msg.sender, address(0), _value);
    }
//SlowMist// The owner role can burn any users' assets through the
burnLockupListAmount function
    function burnLockupListAmount(address _user, uint256 _value) onlyOwner public {
        require(_value <= balances[_user]);

        balances[_user] = balances[_user].sub(_value);
        totalSupply_ = totalSupply_.sub(_value);

        emit BurnLockupListAmount(_user, _value);
        emit Transfer(_user, address(0), _value);
    }
}

contract MintableToken is StandardToken, Ownable {
    event Mint(address indexed to, uint256 amount);
    event MintFinished();

    bool public mintingFinished = false;

    modifier canMint() { require(!mintingFinished); _; }
    modifier cantMint() { require(mintingFinished); _; }

//SlowMist// The mint function does not set an upper limit,and the owner role can
mint tokens to any accounts through the mint function when the mintingFinished value
is false
    function mint(address _to, uint256 _amount) onlyOwner canMint public returns
(bool) {
        totalSupply_ = totalSupply_.add(_amount);
        balances[_to] = balances[_to].add(_amount);

        emit Mint(_to, _amount);
        emit Transfer(address(0), _to, _amount);

        return true;
    }
//SlowMist// The owner role can execute the finishMInting function to set the
mintingFinished value to true, and this state cannot be changed after it is set to
true, and when the mintingFinished value is true,the contract can no longer mint
tokens through the mint function
    function finishMinting() onlyOwner canMint public returns (bool) {
        mintingFinished = true;
        emit MintFinished();
        return true;
    }

}

contract PausableToken is StandardToken, Pausable, LockupList {

    function transfer(address _to, uint256 _value) public whenNotPaused
CheckLockupList returns (bool) {
        return super.transfer(_to, _value);
    }

    function transferFrom(address _from, address _to, uint256 _value) public
whenNotPaused CheckLockupList returns (bool) {
        return super.transferFrom(_from, _to, _value);
    }

    function approve(address _spender, uint256 _value) public whenNotPaused
CheckLockupList returns (bool) {
        return super.approve(_spender, _value);
    }

    function increaseApproval(address _spender, uint _addedValue) public
whenNotPaused CheckLockupList returns (bool success) {
        return super.increaseApproval(_spender, _addedValue);
    }

    function decreaseApproval(address _spender, uint _subtractedValue) public
whenNotPaused CheckLockupList returns (bool success) {
        return super.decreaseApproval(_spender, _subtractedValue);
    }
}

contract TouchCon is PausableToken, MintableToken, BurnableToken {
    string public name = "TouchCon";
    string public symbol = "TOC";
    uint256 public decimals = 18;
}
```

Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.