Smart Contract
Security Audit Report

The SlowMist Security Team received the team's application for smart contract security audit of the GOLFROCHAIN(GOLF) on 2022.03.07. The following are the details and results of this smart contract security audit:

Token Name :

GOLFROCHAIN(GOLF)

The contract address :

https://scope.klaytn.com/account/0x83c3b5a9a9d1f1438f2505ba972366eecfc4488e?tabId=contractCode

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
14	Non-privacy/Non-dark Coin Audit	Passed

Audit Result : Passed

Audit Number : 0X002203090002

Audit Date : 2022.03.07 - 2022.03.09

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that does not contain the tokenVault section. The total amount of contract tokens can be changed, users can burn their own tokens through the burn function. SafeMath security module is used, which is a recommended approach. The contract does not have the Overflow and the Race Conditions issue.

During the audit, we found the following information:

1. The owner role can lock all the transfers through the setLocked function and also can lock the specific user transactions through the lockAddress function.

The source code:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.4.24;

// ----------------------------------------------------------------------------
// 'GOLFROCHAIN'  Token Contract
//
// Name        : GOLFROCHAIN
// Symbol      : GOLF
// Total supply: 10 000 000 000
// Decimals    : 18
// ----------------------------------------------------------------------------

/**
```

```
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
//SlowMist// SafeMath security module is used, which is a recommended approach
library SafeMath {
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a * b;
    //SlowMist// It is recommended to replace "assert" with "require" to optimize Gas
    assert(a == 0 || 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) {
    //SlowMist// It is recommended to replace "assert" with "require" to optimize Gas
    assert(b <= a);
    return a - b;
  }

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

/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic authorization
control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
  address public owner;

  event OwnershipTransferred(address indexed previousOwner, address indexed
newOwner);
```

```
  /**
   * @dev The Ownable constructor sets the original `owner` of the contract to the
sender
   * account.
   */
  constructor () public {
    owner = msg.sender;
  }

  /**
   * @dev Throws if called by any account other than the owner.
   */
  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  /**
   * @dev Allows the current owner to transfer control of the contract to a newOwner.
   * @param newOwner The address to transfer ownership to.
   */
  function transferOwnership(address newOwner) onlyOwner public {
    //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;
  }
}

/**
 * @title ERC20Basic
 * @dev Simpler version of ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/179
 */
contract ERC20Basic {

  uint256 public totalSupply;
  function balanceOf(address who) public constant returns (uint256);
  function transfer(address to, uint256 value) public returns (bool);
  event Transfer(address indexed from, address indexed to, uint256 value);
}
```

```
/**
 * @title Basic token
 * @dev Basic version of StandardToken, with no allowances.
 */
contract BasicToken is ERC20Basic, Ownable {
  using SafeMath for uint256;

  mapping(address => uint256) balances;
  // allowedAddresses will be able to transfer even when locked
  // lockedAddresses will *not* be able to transfer even when *not locked*
  mapping(address => bool) public allowedAddresses;
  mapping(address => bool) public lockedAddresses;
  bool public locked = false;

  function allowAddress(address _addr, bool _allowed) public onlyOwner {
    require(_addr != owner);
    allowedAddresses[_addr] = _allowed;
  }

  function lockAddress(address _addr, bool _locked) public onlyOwner {
    require(_addr != owner);
    lockedAddresses[_addr] = _locked;
  }

  function setLocked(bool _locked) public onlyOwner {
    locked = _locked;
  }

  function canTransfer(address _addr) public constant returns (bool) {
    if(locked){
      if(!allowedAddresses[_addr]&&_addr!=owner) return false;
    }else if(lockedAddresses[_addr]) return false;

    return true;
  }

  /**
  * @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) {
    //SlowMist// This kind of check is very good, avoiding user mistake leading to
the loss of token during transfer
```

```
    require(_to != address(0));
    require(canTransfer(msg.sender));


    // SafeMath.sub will throw if there is not enough balance.
    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 KIP7 specification
    return true;
  }

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

/**
 * @title ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/20
 */
contract ERC20 is ERC20Basic {
  function allowance(address owner, address spender) public constant 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);
}


/**
 * @title Standard ERC20 token
 *
 * @dev Implementation of the basic standard token.
 * @dev https://github.com/ethereum/EIPs/issues/20
 * @dev Based on code by FirstBlood:
https://github.com/Firstbloodio/token/blob/master/smart_contract/FirstBloodToken.sol
 */
```

```
contract StandardToken is ERC20, BasicToken {

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

  /**
   * @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(_to != address(0));
    require(canTransfer(_from));

    uint256 _allowance = allowed[_from][msg.sender];

    // Check is not needed because sub(_allowance, _value) will already throw if this
condition is not met
    // require (_value <= _allowance);

    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(_value);
    allowed[_from][msg.sender] = _allowance.sub(_value);
    emit Transfer(_from, _to, _value);
    //SlowMist// The return value conforms to the KIP7 specification
    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) {
```

```
    allowed[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    //SlowMist// The return value conforms to the KIP7 specification
    return true;
  }

  /**
   * @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 constant returns
(uint256 remaining) {
    return allowed[_owner][_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
   */
  function increaseApproval (address _spender, uint _addedValue)
    public returns (bool success) {
    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, uint _subtractedValue)
    public returns (bool success) {
    uint 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;
  }
}
```

```
/**
 * @title Burnable Token
 * @dev Token that can be irreversibly burned (destroyed).
 */
contract BurnableToken is StandardToken {

    event Burn(address indexed burner, uint256 value);

    /**
     * @dev Burns a specific amount of tokens.
     * @param _value The amount of token to be burned.
     */
    function burn(uint256 _value) public {
        require(_value > 0);
        require(_value <= balances[msg.sender]);
        // no need to require value <= totalSupply, since that would imply the
        // sender's balance is greater than the totalSupply, which *should* be an
assertion failure

        address burner = msg.sender;
        balances[burner] = balances[burner].sub(_value);
        totalSupply = totalSupply.sub(_value);
        emit Burn(burner, _value);
        emit Transfer(burner, address(0), _value);
    }
}

contract GOLF is BurnableToken {

    string public constant name = "GOLFROCHAIN";
    string public constant symbol = "GOLF";
    uint public constant decimals = 18;
    // there is no problem in using * here instead of .mul()
    uint256 public constant initialSupply = 10000000000 * (10 ** uint256(decimals));

    // Constructors
    constructor () public {
        totalSupply = initialSupply;
        balances[msg.sender] = initialSupply; // Send all tokens to owner
        allowedAddresses[owner] = true;
        // owner is allowed to send tokens even when locked
        emit Transfer(address(0), owner, initialSupply);
    }
}
```

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.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist