Smart Contract Security Audit Report

The SlowMist Security Team received the Crypto.com team's application for smart contract security audit of the ERC900 module on Sep. 04, 2020. The following are the details and results of this smart contract security audit:

Project name :

ERC900 module

File name and HASH(SHA256) :

audit-slowmist.zip:

f5745c5dee96ce5137ed263058e41266cbf80bb968092937ddb1c1b998f4776a

The audit items and results :

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

No.	Audit Items	Audit Subclass	Audit Subclass Result
1	Overflow Audit	-	Passed
2	Race Conditions Audit	-	Passed
3	Authority Control Audit	Permission vulnerability audit	Passed
		Excessive auditing authority	Passed
4	Safety Design Audit	Zeppelin module safe use	Passed
		Compiler version security	Passed
		Hard-coded address security	Passed
		Fallback function safe use	Passed
		Show coding security	Passed
		Function return value security	Passed
		Call function security	Passed
5	Denial of Service Audit	-	Passed
6	Gas Optimization Audit	-	Passed
7	Design Logic Audit	-	Passed
8	“False Deposit” vulnerability Audit	-	Passed
9	Malicious Event Log Audit	-	Passed

10	Scoping and Declarations Audit	-	Passed
11	Replay Attack Audit	ECDSA's Signature Replay Audit	Passed
12	Uninitialized Storage Pointers Audit	-	Passed
13	Arithmetic Accuracy Deviation Audit	-	Passed

Audit Result : **Passed**

Audit Number : 0X002009070001

Audit Date : Sep. 07, 2020

Audit Team : SlowMist Security Team

(Statement : SlowMist only issues this report based on the fact that has occurred or existed before the report is issued, and bears the corresponding responsibility in this regard. For the facts occur or exist later after the report, SlowMist cannot judge the security status of its smart contract. SlowMist is not responsible for it. The security audit analysis and other contents of this report are based on the documents and materials provided by the information provider to SlowMist as of the date of this report (referred to as "the provided information"). SlowMist assumes that: there has been no information missing, tampered, deleted, or concealed. If the information provided has been missed, modified, deleted, concealed or reflected and is inconsistent with the actual situation, SlowMist will not bear any responsibility for the resulting loss and adverse effects. SlowMist will not bear any responsibility for the background or other circumstances of the project.)

Summary: This is a contract that contains the ERC900 module section. The contract does not have the Overflow and the Race Conditions issue. The comprehensive evaluation contract is no risk.

The source code:

ERC900.sol:

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


/**
 * @title ERC900 Simple Staking Interface
 * @dev See https://github.com/ethereum/EIPs/blob/master/EIPS/eip-900.md
 */
contract ERC900 {
  event Staked(address indexed user, uint256 amount, uint256 total, bytes data);
  event Unstaked(address indexed user, uint256 amount, uint256 total, bytes data);

  function stake(uint256 amount, bytes data) public;
```

```
  function stakeFor(address user, uint256 amount, bytes data) public;
  function unstake(uint256 amount, bytes data) public;
  function totalStakedFor(address addr) public view returns (uint256);
  function totalStaked() public view returns (uint256);
  function token() public view returns (address);
  function supportsHistory() public pure returns (bool);

  // NOTE: Not implementing the optional functions
  // function lastStakedFor(address addr) public view returns (uint256);
  // function totalStakedForAt(address addr, uint256 blockNumber) public view returns (uint256);
  // function totalStakedAt(uint256 blockNumber) public view returns (uint256);
}
```

ERC900BasicStakeContract.sol:

```
/* solium-disable security/no-block-members */

//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;

import "openzeppelin-solidity/contracts/token/ERC20/ERC20.sol";
import "openzeppelin-solidity/contracts/math/SafeMath.sol";

import "./ERC900.sol";



/**
 * @title ERC900 Simple Staking Interface basic implementation
 * @dev See https://github.com/ethereum/EIPs/blob/master/EIPS/eip-900.md
 */
contract ERC900BasicStakeContract is ERC900 {
  // @TODO: deploy this separately so we don't have to deploy it multiple times for each contract
  using SafeMath for uint256;

  // Token used for staking
  ERC20 stakingToken;

  // The default duration of stake lock-in (in seconds)
  uint256 public defaultLockInDuration;

  // To save on gas, rather than create a separate mapping for totalStakedFor & personalStakes,
  //  both data structures are stored in a single mapping for a given addresses.
```

```
    //
    // It's possible to have a non-existing personalStakes, but have tokens in totalStakedFor
    //   if other users are staking on behalf of a given address.
    mapping (address => StakeContract) public stakeHolders;

    // Struct for personal stakes (i.e., stakes made by this address)
    // unlockedTimestamp - when the stake unlocks (in seconds since Unix epoch)
    // actualAmount - the amount of tokens in the stake
    // stakedFor - the address the stake was staked for
    struct Stake {
      uint256 unlockedTimestamp;
      uint256 actualAmount;
      address stakedFor;
    }

    // Struct for all stake metadata at a particular address
    // totalStakedFor - the number of tokens staked for this address
    // personalStakeIndex - the index in the personalStakes array.
    // personalStakes - append only array of stakes made by this address
    // exists - whether or not there are stakes that involve this address
    struct StakeContract {
      uint256 totalStakedFor;

      uint256 personalStakeIndex;

      Stake[] personalStakes;

      bool exists;
    }

    /**
     * @dev Modifier that checks that this contract can transfer tokens from the
     *   balance in the stakingToken contract for the given address.
     * @dev This modifier also transfers the tokens.
     * @param _address address to transfer tokens from
     * @param _amount uint256 the number of tokens
     */
    modifier canStake(address _address, uint256 _amount) {
      require(
        stakingToken.transferFrom(_address, this, _amount),
        "Stake required");
```

```
    _;
}

/**
 * @dev Constructor function
 * @param _stakingToken ERC20 The address of the token contract used for staking
 */
constructor(ERC20 _stakingToken) public {
    stakingToken = _stakingToken;
}

/**
 * @dev Returns the timestamps for when active personal stakes for an address will unlock
 * @dev These accessors functions are needed until https://github.com/ethereum/web3.js/issues/1241 is solved
 * @param _address address that created the stakes
 * @return uint256[] array of timestamps
 */
function getPersonalStakeUnlockedTimestamps(address _address) external view returns (uint256[]) {
    uint256[] memory timestamps;
    (timestamps,,) = getPersonalStakes(_address);

    return timestamps;
}

/**
 * @dev Returns the stake actualAmount for active personal stakes for an address
 * @dev These accessors functions are needed until https://github.com/ethereum/web3.js/issues/1241 is solved
 * @param _address address that created the stakes
 * @return uint256[] array of actualAmounts
 */
function getPersonalStakeActualAmounts(address _address) external view returns (uint256[]) {
    uint256[] memory actualAmounts;
    (,actualAmounts,) = getPersonalStakes(_address);

    return actualAmounts;
}

/**
 * @dev Returns the addresses that each personal stake was created for by an address
 * @dev These accessors functions are needed until https://github.com/ethereum/web3.js/issues/1241 is solved
 * @param _address address that created the stakes
 * @return address[] array of amounts
```

```
    */
  function getPersonalStakeForAddresses(address _address) external view returns (address[]) {
    address[] memory stakedFor;
    (,,stakedFor) = getPersonalStakes(_address);

    return stakedFor;
  }

  /**
   * @notice Stakes a certain amount of tokens, this MUST transfer the given amount from the user
   * @notice MUST trigger Staked event
   * @param _amount uint256 the amount of tokens to stake
   * @param _data bytes optional data to include in the Stake event
   */
  function stake(uint256 _amount, bytes _data) public {
    createStake(
      msg.sender,
      _amount,
      defaultLockInDuration,
      _data);
  }

  /**
   * @notice Stakes a certain amount of tokens, this MUST transfer the given amount from the caller
   * @notice MUST trigger Staked event
   * @param _user address the address the tokens are staked for
   * @param _amount uint256 the amount of tokens to stake
   * @param _data bytes optional data to include in the Stake event
   */
  function stakeFor(address _user, uint256 _amount, bytes _data) public {
    createStake(
      _user,
      _amount,
      defaultLockInDuration,
      _data);
  }

  /**
   * @notice Unstakes a certain amount of tokens, this SHOULD return the given amount of tokens to the user, if unstaking is
currently not possible the function MUST revert
   * @notice MUST trigger Unstaked event
   * @dev Unstaking tokens is an atomic operation—either all of the tokens in a stake, or none of the tokens.
```

```
    * @dev Users can only unstake a single stake at a time, it is must be their oldest active stake. Upon releasing that stake, the tokens will be
    *  transferred back to their account, and their personalStakeIndex will increment to the next active stake.
    * @param _amount uint256 the amount of tokens to unstake
    * @param _data bytes optional data to include in the Unstake event
    */
  function unstake(uint256 _amount, bytes _data) public {
    withdrawStake(
      _amount,
      _data);
  }

  /**
    * @notice Returns the current total of tokens staked for an address
    * @param _address address The address to query
    * @return uint256 The number of tokens staked for the given address
    */
  function totalStakedFor(address _address) public view returns (uint256) {
    return stakeHolders[_address].totalStakedFor;
  }

  /**
    * @notice Returns the current total of tokens staked
    * @return uint256 The number of tokens staked in the contract
    */
  function totalStaked() public view returns (uint256) {
    return stakingToken.balanceOf(this);
  }

  /**
    * @notice Address of the token being used by the staking interface
    * @return address The address of the ERC20 token used for staking
    */
  function token() public view returns (address) {
    return stakingToken;
  }

  /**
    * @notice MUST return true if the optional history functions are implemented, otherwise false
    * @dev Since we don't implement the optional interface, this always returns false
    * @return bool Whether or not the optional history functions are implemented
    */
```

```
function supportsHistory() public pure returns (bool) {
    return false;
}

/**
 * @dev Helper function to get specific properties of all of the personal stakes created by an address
 * @param _address address The address to query
 * @return (uint256[], uint256[], address[])
 *   timestamps array, actualAmounts array, stakedFor array
 */
function getPersonalStakes(
    address _address
)
    public
    view
    returns(uint256[], uint256[], address[])
{
    StakeContract storage stakeContract = stakeHolders[_address];

    uint256 arraySize = stakeContract.personalStakes.length - stakeContract.personalStakeIndex;
    uint256[] memory unlockedTimestamps = new uint256[](arraySize);
    uint256[] memory actualAmounts = new uint256[](arraySize);
    address[] memory stakedFor = new address[](arraySize);

    for (uint256 i = stakeContract.personalStakeIndex; i < stakeContract.personalStakes.length; i++) {
        uint256 index = i - stakeContract.personalStakeIndex;
        unlockedTimestamps[index] = stakeContract.personalStakes[i].unlockedTimestamp;
        actualAmounts[index] = stakeContract.personalStakes[i].actualAmount;
        stakedFor[index] = stakeContract.personalStakes[i].stakedFor;
    }

    return (
        unlockedTimestamps,
        actualAmounts,
        stakedFor
    );
}

/**
 * @dev Helper function to create stakes for a given address
 * @param _address address The address the stake is being created for
 * @param _amount uint256 The number of tokens being staked
```

```
  * @param _lockInDuration uint256 The duration to lock the tokens for
  * @param _data bytes optional data to include in the Stake event
  */
```

//SlowMist// Stake logic

```
function createStake(
    address _address,
    uint256 _amount,
    uint256 _lockInDuration,
    bytes _data
)
    internal
    canStake(msg.sender, _amount)
{
    require(
        _amount > 0,
        "Stake amount has to be greater than 0!");
    if (!stakeHolders[msg.sender].exists) {
        stakeHolders[msg.sender].exists = true;
    }

    stakeHolders[_address].totalStakedFor = stakeHolders[_address].totalStakedFor.add(_amount);
    stakeHolders[msg.sender].personalStakes.push(
        Stake(
            block.timestamp.add(_lockInDuration),
            _amount,
            _address)
        );

    emit Staked(
        _address,
        _amount,
        totalStakedFor(_address),
        _data);
}

/**
  * @dev Helper function to withdraw stakes for the msg.sender
  * @param _amount uint256 The amount to withdraw. MUST match the stake amount for the
  *   stake at personalStakeIndex.
  * @param _data bytes optional data to include in the Unstake event
  */
```

//SlowMist// Withdraw logic

```
  function withdrawStake(
    uint256 _amount,
    bytes _data
  )
    internal
  {
    Stake storage personalStake =
stakeHolders[msg.sender].personalStakes[stakeHolders[msg.sender].personalStakeIndex];

    // Check that the current stake has unlocked & matches the unstake amount
    require(
      personalStake.unlockedTimestamp <= block.timestamp,
      "The current stake hasn't unlocked yet");

    require(
      personalStake.actualAmount == _amount,
      "The unstake amount does not match the current stake");

    // Transfer the staked tokens from this contract back to the sender
    // Notice that we are using transfer instead of transferFrom here, so
    //   no approval is needed beforehand.
    require(
      stakingToken.transfer(msg.sender, _amount),
      "Unable to withdraw stake");

    stakeHolders[personalStake.stakedFor].totalStakedFor = stakeHolders[personalStake.stakedFor]
      .totalStakedFor.sub(personalStake.actualAmount);

    personalStake.actualAmount = 0;
    stakeHolders[msg.sender].personalStakeIndex++;

    emit Unstaked(
      personalStake.stakedFor,
      _amount,
      totalStakedFor(personalStake.stakedFor),
      _data);
  }
}
```

BasicStakingContract.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.4.24;


import "./ERC900BasicStakeContract.sol";



/**
 * @title BasicStakingContract
 */
contract BasicStakingContract is ERC900BasicStakeContract {
  /**
   * @dev Constructor function
   * @param _stakingToken ERC20 The address of the token used for staking
   * @param _lockInDuration uint256 The duration (in seconds) that stakes are required to be locked for
   */
  constructor(
    ERC20 _stakingToken,
    uint256 _lockInDuration
  )
    public
    ERC900BasicStakeContract(_stakingToken)
  {
    defaultLockInDuration = _lockInDuration;
  }
}
```

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

Github
https://github.com/slowmist