Smart Contract Security Audit Report

[2021]

The SlowMist Security Team received the Sun team's application for smart contract security audit of the SunToken on 2021.05.24. The following are the details and results of this smart contract security audit:

Token Name :

SunToken

The contract address :

https://tronscan.org/#/contract/TSSMHYeV2uE9qYH95DqyoCuNCzEL1NvU3S

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 : 0x002105260001

Audit Date : 2021.05.24 - 2021.05.26

Audit Team : SlowMist Security Team

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

The source code:

BaseTRC20.sol

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.5.8;

import "./ITRC20.sol";
import "./Context.sol";
import "./SafeMath.sol";

contract BaseTRC20 is Context, ITRC20 {
    using SafeMath for uint;

    mapping (address => uint) private _balances;

    mapping (address => mapping (address => uint)) private _allowances;

    uint private _totalSupply;

    function totalSupply() public view returns (uint) {
        return _totalSupply;
    }
    function balanceOf(address account) public view returns (uint) {
        return _balances[account];
    }
    function transfer(address recipient, uint amount) public returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        //SlowMist// The return value conforms to the EIP20 specification
        return true;
    }
    function allowance(address owner, address spender) public view returns (uint) {
        return _allowances[owner][spender];
    }
    function approve(address spender, uint amount) public returns (bool) {
        _approve(_msgSender(), spender, amount);
        //SlowMist// The return value conforms to the EIP20 specification
        return true;
    }
    function transferFrom(address sender, address recipient, uint amount) public
returns (bool) {
        _transfer(sender, recipient, amount);
        _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount,
"TRC20: transfer amount exceeds allowance"));
        //SlowMist// The return value conforms to the EIP20 specification
        return true;
    }
    function increaseAllowance(address spender, uint addedValue) public returns
(bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()]
[spender].add(addedValue));
        return true;
    }
    function decreaseAllowance(address spender, uint subtractedValue) public returns
(bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()]
[spender].sub(subtractedValue, "TRC20: decreased allowance below zero"));
        return true;
    }
    function _transfer(address sender, address recipient, uint amount) internal {
        //SlowMist// This kind of check is very good, avoiding user mistake leading
to the loss of token during transfer
        require(sender != address(0), "TRC20: transfer from the zero address");
        require(recipient != address(0), "TRC20: transfer to the zero address");

        _balances[sender] = _balances[sender].sub(amount, "TRC20: transfer amount
exceeds balance");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }
    function _mint(address account, uint amount) internal {
        require(account != address(0), "TRC20: mint to the zero address");

        _totalSupply = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
        emit Transfer(address(0), account, amount);
    }
    function _burn(address account, uint amount) internal {
        require(account != address(0), "TRC20: burn from the zero address");

        _balances[account] = _balances[account].sub(amount, "TRC20: burn amount
exceeds balance");
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(account, address(0), amount);
    }
    function _approve(address owner, address spender, uint amount) internal {
        require(owner != address(0), "TRC20: approve from the zero address");
        //SlowMist// This kind of check is very good, avoiding user mistake leading
to approve errors
        require(spender != address(0), "TRC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }
}

contract TRC20Detailed is BaseTRC20 {
    string private _name;
    string private _symbol;
    uint8 private _decimals;


    constructor (string memory name, string memory symbol, uint8 decimals) public {
        _name = name;
        _symbol = symbol;
        _decimals = decimals;
    }

    function name() public view returns (string memory) {
        return _name;
    }
    function symbol() public view returns (string memory) {
        return _symbol;
    }
    function decimals() public view returns (uint8) {
        return _decimals;
    }
}
```

Context.sol

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.5.8;

contract Context {
    constructor () internal { }

    function _msgSender() internal view returns (address payable) {
        return msg.sender;
    }
}
```

ITRC20.sol

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.5.8;

contract TRC20Events {
    event Approval(address indexed src, address indexed guy, uint wad);
    event Transfer(address indexed src, address indexed dst, uint wad);
}

contract ITRC20 is TRC20Events {
    function totalSupply() public view returns (uint);
    function balanceOf(address guy) public view returns (uint);
    function allowance(address src, address guy) public view returns (uint);

    function approve(address guy, uint wad) public returns (bool);
    function transfer(address dst, uint wad) public returns (bool);
    function transferFrom(
        address src, address dst, uint wad
    ) public returns (bool);
}
```

SafeMath.sol

```
// SPDX-License-Identifier: MIT
//SlowMist// SafeMath security module is used, which is a recommend approach
pragma solidity ^0.5.8;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     *
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom
message on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure
returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     *
     * - Multiplication cannot 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-contracts/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");
        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with
custom message on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure
returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned
integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned
integer modulo),
     * Reverts with custom message when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure
returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}
```

SunToken.sol

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.5.8;

import "./Context.sol";
import "./ITRC20.sol";
import "./BaseTRC20.sol";

contract SunToken is ITRC20, TRC20Detailed {
    constructor(address gr) public TRC20Detailed("SUN TOKEN", "SUN", 18){
        require(gr != address(0), "invalid gr");
        _mint(gr, 19900730 * 1000 * 10 ** 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.