SLOWMIST

Smart Contract Security Audit Report

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

Token Name :

LaqiraToken

The contract address :

https://bscscan.com/address/0xbc81ea817b579ec0334bca8e65e436b7cb540147

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 : 0X002203090004

Audit Date : 2022.03.07 - 2022.03.09

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that contains the Voting 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 can mint tokens arbitrarily through the mint function but the minted amount has an upper limit called _maxSupply.
2. The owner role can transfer any mistransferred BEP20 tokens to any address.

The source code:

LaqiraToken.sol

```
// SPDX-License-Identifier: MIT
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.0;

import "./VotingToken.sol";
import "./Ownable.sol";
import "./Pausable.sol";
import "./SafeMath.sol";


contract LaqiraToken is VotingToken, Ownable, Pausable {
    using SafeMath for uint256;
    mapping(address => uint256) private frosted;

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

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

    /**
     * @dev Gets the frosted balance of a specified address.
     * @param _owner is the address to query the frosted balance of.
     * @return uint256 representing the amount owned by the address which is frosted.
     */

    function frostedOf(address _owner) public view returns (uint256) {
        return frosted[_owner];
    }

    /**
     * @dev Gets the available balance of a specified address which is not frosted.
     * @param _owner is the address to query the available balance of.
     * @return uint256 representing the amount owned by the address which is not
frosted.
     */

    function availableBalance(address _owner) public view returns (uint256) {
        return _balances[_owner].sub(frosted[_owner]);
    }

    function _beforeTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal override {
        super._beforeTokenTransfer(from, to, amount);
        require(!paused(), "BEP20Pausable: token transfer while paused");
        require(_balances[from].sub(frosted[from]) >= amount, "LQR: not avaiable
balance");
    }

    /**
     * @dev Sets the values for {name}, {symbol}, {totalsupply} and {deciamls}.
     *
     * {name}, {symbol} and {decimals} are immutable: they can only be set once
during
```

```
     * construction. {totalsupply} may be changed by using mint and burn functions.
     */
    constructor(uint256 totalSupply_) {
        _name = "Laqira Token";
        _symbol = "LQR";
        _decimals = 18;
        _transferOwnership(_msgSender());
        _mint(_msgSender(), totalSupply_);
    }
    //SlowMist// The owner can mint tokens abitrarily through the mint function but the minted amount has a upper limit called _maxSupply
    function mint(address account, uint256 amount) public onlyOwner returns (bool) {
        _mint(account, amount);
        return true;
    }

    function burn(uint256 amount) public returns (bool) {
        _burn(_msgSender(), amount);
        return true;
    }

    function pause() public onlyOwner returns (bool) {
        _pause();
        return true;
    }

    function unpause() public onlyOwner returns (bool) {
        _unpause();
        return true;
    }

    /**
     * @dev transfer frosted tokens to a specified address
     * @param to is the address to which frosted tokens are transferred.
     * @param amount is the frosted amount which is transferred.
     */
    function frost(address to, uint256 amount) public onlyOwner returns (bool) {
        _frost(_msgSender(), to, amount);
        return true;
    }

    /**
     * @dev defrost frosted tokens of specified address
     * @param to is the address from which frosted tokens are defrosted.
     * @param amount is the frosted amount which is defrosted.
     */

    function defrost(address to, uint256 amount) public onlyOwner returns (bool) {
        _defrost(_msgSender(), to, amount);
        return true;
    }

    function _frost(address from, address to, uint256 amount) private {
        frosted[to] = frosted[to].add(amount);
        _transfer(from, to, amount);
        emit Frost(from ,to, amount);
    }

    function _defrost(address onBehalfOf, address to, uint256 amount) private {
        require(frosted[to] >= amount);
        frosted[to] = frosted[to].sub(amount);
        emit Defrost(onBehalfOf, to, amount);
    }
    //SlowMist// The owner role can transfer any mistransfered BEP20 tokens to any
address
    function transferAnyBEP20(address _tokenAddress, address _to, uint256 _amount)
public onlyOwner returns (bool) {
        IBEP20(_tokenAddress).transfer(_to, _amount);
        return true;
    }
}
```

BasicToken.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./SafeMath.sol";
import "./Context.sol";
import "./BEP20Basic.sol";

/**
 * @title Basic Token
 * @dev Basic version of BEP20 Standard Token, without transfer approvals.
 */
```

```
contract BasicToken is Context, BEP20Basic {
    using SafeMath for uint256;

    mapping(address => uint256) internal _balances;

    uint256 internal _totalSupply;
    uint8 internal _decimals;

    /**
     * @dev See {BEP20Basic-totalSupply}.
     */
    function totalSupply() public view override returns (uint256) {
        return _totalSupply;
    }

    /**
     * @dev Returns the number of decimals used to get its user representation.
     * For example, if `decimals` equals `2`, a balance of `505` tokens should
     * be displayed to a user as `5.05` (`505 / 10 ** 2`).
     *
     * Tokens usually opt for a value of 18, imitating the relationship between
     * Ether and Wei. This is the value {ERC20} uses, unless this function is
     * overridden;
     *
     * NOTE: This information is only used for _display_ purposes: it in
     * no way affects any of the arithmetic of the contract, including
     * {IERC20-balanceOf} and {IERC20-transfer}.
     */
    function decimals() public view returns (uint8) {
        return _decimals;
    }

    /**
     * @dev See {BEP20Basic-balanceOf}.
     */

    function balanceOf(address account) public view override returns (uint256) {
        return _balances[account];
    }

    /**
     * @dev See {BEP20Basic-transfer}.
     *
     * Requirements:
     *
     * - `recipient` cannot be the zero address.
     * - the caller must have a balance of at least `amount`.
     */
    function transfer(address recipient, uint256 amount) public override returns
(bool) {
        _transfer(_msgSender(), recipient, amount);
        //SlowMist// The return value conforms to the BEP20 specification
        return true;
    }

    /**
     * @dev Moves `amount` of tokens from `sender` to `recipient`.
     *
     * This internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `sender` cannot be the zero address.
     * - `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     */

    function _transfer(
        address sender,
        address recipient,
        uint256 amount
    ) internal {
        require(sender != address(0), "BEP20: transfer from the zero address");
        //SlowMist// This kind of check is very good, avoiding user mistake leading
to the loss of token during transfer
        require(recipient != address(0), "BEP20: transfer to the zero address");

        _beforeTokenTransfer(sender, recipient, amount);

        uint256 senderBalance = _balances[sender];
        require(senderBalance >= amount, "BEP20: transfer amount exceeds balance");
        _balances[sender] = senderBalance.sub(amount);
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);

        _afterTokenTransfer(sender, recipient, amount);
    }

    /**
     * @dev Hook that is called before any transfer of tokens. This includes
     * minting and burning.
     *
     * Calling conditions:
     *
     * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     * will be transferred to `to`.
     * - when `from` is zero, `amount` tokens will be minted for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens will be burned.
     * - `from` and `to` are never both zero.
     *
     * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].
     */
    function _beforeTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {}

    /**
     * @dev Hook that is called after any transfer of tokens. This includes
     * minting and burning.
     *
     * Calling conditions:
     *
     * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     * has been transferred to `to`.
     * - when `from` is zero, `amount` tokens have been minted for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens have been burned.
     * - `from` and `to` are never both zero.
     *
     * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].
     */
    function _afterTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {}

}
```

BEP20Basic.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

/**
 * @title BEP20Basic
 * @dev Simpler version of BEP20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/179
 */

interface BEP20Basic {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);

    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */

    function transfer(address recipient, uint256 amount) external returns (bool);

    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

}
```

Context.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

/**
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes calldata) {
        return msg.data;
    }
}
```

IBEP20.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./BEP20Basic.sol";
```

```
/**
 * @dev Interface of the BEP20 standard as defined in the EIP.
 */
interface IBEP20 is BEP20Basic {
    /**
     * @dev Returns the remaining number of tokens that `spender` will be
     * allowed to spend on behalf of `owner` through {transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are called.
     */
    function allowance(address owner, address spender) external view returns
(uint256);

    /**
     * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * IMPORTANT: 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
     *
     * Emits an {Approval} event.
     */
    function approve(address spender, uint256 amount) external returns (bool);

    /**
     * @dev Moves `amount` tokens from `sender` to `recipient` using the
     * allowance mechanism. `amount` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool);
    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);
}
```

IBEP677.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./IBEP20.sol";

/**
 * @title IBEP677 Token interface
 * @dev see https://github.com/ethereum/EIPs/issues/677
 */

interface IBEP677 is IBEP20 {
    function transferAndCall(address receiver, uint value, bytes memory data)
external returns (bool success);
    event Transfer(address indexed from, address indexed to, uint256 value, bytes
data);
}
```

IBEP677Receiver.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

/**
 * @title IBEP677 Receiving Contract interface
 * @dev see https://github.com/ethereum/EIPs/issues/677
 */

interface IBEP677Receiver {
    function onTokenTransfer(address _sender, uint _value, bytes memory _data)
external;
}
```

Math.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

/**
 * @dev Standard math utilities missing in the Solidity language.
 */


library Math {
    /**
     * @dev Returns the largest of two numbers.
     */
    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a >= b ? a : b;
    }

    /**
     * @dev Returns the smallest of two numbers.
     */
    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a < b ? a : b;
    }

    /**
     * @dev Returns the average of two numbers. The result is rounded towards
     * zero.
     */
    function average(uint256 a, uint256 b) internal pure returns (uint256) {
        // (a + b) / 2 can overflow.
        return (a & b) + (a ^ b) / 2;
    }

    /**
     * @dev Returns the ceiling of the division of two numbers.
     *
     * This differs from standard division with `/` in that it rounds up instead
     * of rounding down.
     */
    function ceilDiv(uint256 a, uint256 b) internal pure returns (uint256) {
        // (a + b - 1) / b can overflow on addition, so we distribute.
        return a / b + (a % b == 0 ? 0 : 1);
    }
}
```

Ownable.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./Context.sol";

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
abstract contract Ownable is Context {
    address private _owner;

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


    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }
    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(owner() == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public onlyOwner {
        _transferOwnership(address(0));
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public onlyOwner {
        //SlowMist// This check is quite good in avoiding losing control of the contract caused by user mistakes
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        _transferOwnership(newOwner);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Internal function without access restriction.
     */
    function _transferOwnership(address newOwner) internal {
        address oldOwner = _owner;
        _owner = newOwner;
        emit OwnershipTransferred(oldOwner, newOwner);
    }
}
```

Pausable.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./Context.sol";

/**
 * @dev Contract module which allows children to implement an emergency stop
 * mechanism that can be triggered by an authorized account.
 *
 * This module is used through inheritance. It will make available the
 * modifiers `whenNotPaused` and `whenPaused`, which can be applied to
 * the functions of your contract. Note that they will not be pausable by
 * simply including this module, only once the modifiers are put in place.
 */
abstract contract Pausable is Context {
    /**
     * @dev Emitted when the pause is triggered by `account`.
     */
    event Paused(address account);

    /**
     * @dev Emitted when the pause is lifted by `account`.
     */
    event Unpaused(address account);

    bool private _paused;

    /**
     * @dev Initializes the contract in unpaused state.
     */
    constructor() {
        _paused = false;
    }

    /**
     * @dev Returns true if the contract is paused, and false otherwise.
     */
    function paused() public view returns (bool) {
        return _paused;
    }

    /**
```

```
     * @dev Modifier to make a function callable only when the contract is not paused.
     *
     * Requirements:
     *
     * - The contract must not be paused.
     */
    modifier whenNotPaused() {
        require(!paused(), "Pausable: paused");
        _;
    }

    /**
     * @dev Modifier to make a function callable only when the contract is paused.
     *
     * Requirements:
     *
     * - The contract must be paused.
     */
    modifier whenPaused() {
        require(paused(), "Pausable: not paused");
        _;
    }

    /**
     * @dev Triggers stopped state.
     *
     * Requirements:
     *
     * - The contract must not be paused.
     */
    //SlowMist// Suspending all transactions upon major abnormalities is a recommended approach
    function _pause() internal whenNotPaused {
        _paused = true;
        emit Paused(_msgSender());
    }

    /**
     * @dev Returns to normal state.
     *
     * Requirements:
     *
     * - The contract must be paused.
     */
    function _unpause() internal whenPaused {
        _paused = false;
        emit Unpaused(_msgSender());
    }
}
```

SafeMath.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// CAUTION
// This version of SafeMath should only be used with Solidity 0.8 or later,
// because it relies on the compiler's built in overflow checks.

/**
 * @dev Wrappers over Solidity's arithmetic operations.
 *
 * NOTE: `SafeMath` is generally not needed starting with Solidity 0.8, since the
compiler
 * now has built in overflow checking.
 */
//SlowMist// SafeMath security module is used, which is a recommended approach
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, with an overflow flag.
     *
     * _Available since v3.4._
     */
    function tryAdd(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            uint256 c = a + b;
            if (c < a) return (false, 0);
            return (true, c);
        }
    }

    /**
     * @dev Returns the substraction of two unsigned integers, with an overflow flag.
     *
     * _Available since v3.4._
     */
    function trySub(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            if (b > a) return (false, 0);
            return (true, a - b);
        }
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, with an overflow
flag.
     *
     * _Available since v3.4._
     */
    function tryMul(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            // 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 (true, 0);
            uint256 c = a * b;
            if (c / a != b) return (false, 0);
            return (true, c);
        }
    }

    /**
     * @dev Returns the division of two unsigned integers, with a division by zero
flag.
     *
     * _Available since v3.4._
     */
    function tryDiv(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        unchecked {
            if (b == 0) return (false, 0);
            return (true, a / b);
        }
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers, with a division
by zero flag.
```

```
 *
 * _Available since v3.4._
 */
function tryMod(uint256 a, uint256 b) internal pure returns (bool, uint256) {
    unchecked {
        if (b == 0) return (false, 0);
        return (true, a % b);
    }
}

/**
 * @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) {
    return a + b;
}

/**
 * @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 a - b;
}

/**
 * @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) {
        return a * b;
    }

    /**
     * @dev Returns the integer division of two unsigned integers, reverting on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator.
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return a / b;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned
integer modulo),
     * reverting 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 a % b;
    }
}
```

SmartToken.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./StandardToken.sol";
import "./IBEP677.sol";
import "./IBEP677Receiver.sol";

/**
 * @title Smart Token
 * @dev Enhanced Standard Token, with "transfer and call" possibility.
 */


contract SmartToken is StandardToken, IBEP677 {
    /**
     * @dev Current token cannot be transferred to the token contract based on
follwing override modification.
     */

    function _beforeTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual override {
        super._beforeTokenTransfer(from, to, amount);
        require(validRecipient(to), "LQR: recipient cannot be Laqira token address");
    }

    /**
     * @dev transfer token to a contract address with additional data if the
recipient is a contract.
     * @param _to address to transfer to.
     * @param _value amount to be transferred.
     * @param _data extra data to be passed to the receiving contract.
     */

    function transferAndCall(address _to, uint256 _value, bytes memory _data) public
override returns (bool success) {
        _transfer(_msgSender(), _to, _value);
        emit Transfer(_msgSender(), _to, _value, _data);
        if (isContract(_to)) {
            contractFallback(_to, _value, _data);
        }
        return true;
    }

    /**
     * @dev Returns true if `account` is a contract.
     *
     * [IMPORTANT]
     * ====
     * It is unsafe to assume that an address for which this function returns
     * false is an externally-owned account (EOA) and not a contract.
     *
     * Among others, `isContract` will return false for the following
     * types of addresses:
     *
     *  - an externally-owned account
     *  - a contract in construction
     *  - an address where a contract will be created
     *  - an address where a contract lived, but was destroyed
     * ====
     */
    function isContract(address account) internal view returns (bool) {
        // This method relies on extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
        assembly {
            size := extcodesize(account)
        }
        return size > 0;
    }

    function validRecipient(address _recipient) internal view returns (bool) {
        return _recipient != address(this);
    }

    function contractFallback(address _to, uint _value, bytes memory _data) private {
        IBEP677Receiver receiver = IBEP677Receiver(_to);
        receiver.onTokenTransfer(_msgSender(), _value, _data);
    }
}
```

StandardToken.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./BasicToken.sol";
import "./IBEP20.sol";
import "./SafeMath.sol";


contract StandardToken is BasicToken, IBEP20 {
    /**
     * Libraries can be seen as implicit base contracts of the contracts that use
them.
     * They will not be explicitly visible in the inheritance hierarchy.
     */
    using SafeMath for uint256;
    mapping(address => mapping(address => uint256)) private _allowances;

    string internal _name;
    string internal _symbol;

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

    /**
     * @dev Returns the symbol of the token, usually a shorter version of the
     * name.
     */
    function symbol() public view returns (string memory) {
        return _symbol;
    }

    /**
     * @dev See {IBEP20-allowance}.
     */
    function allowance(address owner, address spender) public view override returns
(uint256) {
        return _allowances[owner][spender];
    }

    /**
     * @dev See {IBEP20-approve}.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public override returns (bool)
{
        _approve(_msgSender(), spender, amount);
        //SlowMist// The return value conforms to the BEP20 specification
        return true;
    }

    /**
     * @dev See {IBEP20-transferFrom}.
     *
     * Emits an {Approval} event indicating the updated allowance. This is not
     * required by the EIP. See the note at the beginning of {BEP20}.
     *
     * Requirements:
     *
     * - `sender` and `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     * - the caller must have allowance for ``sender``'s tokens of at least
     * `amount`.
     */

    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) public override returns (bool) {
        _transfer(sender, recipient, amount);

        uint256 currentAllowance = _allowances[sender][_msgSender()];
        require(currentAllowance >= amount, "BEP20: transfer amount exceeds
allowance");
        _approve(sender, _msgSender(), currentAllowance.sub(amount));
        //SlowMist// The return value conforms to the BEP20 specification
        return true;
    }

    /**
     * @dev Atomically increases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IBEP20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */

    function increaseAllowance(address spender, uint256 addedValue) public returns
(bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()]
[spender].add(addedValue));
        return true;
    }

    /**
     * @dev Atomically decreases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IBEP20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */

    function decreaseAllowance(address spender, uint256 subtractedValue) public
returns (bool) {
        uint256 currentAllowance = _allowances[_msgSender()][spender];
        require(currentAllowance >= subtractedValue, "BEP20: decreased allowance
below zero");
        _approve(_msgSender(), spender, currentAllowance.sub(subtractedValue));
        return true;
```

```
}

/** @dev Creates `amount` tokens and assigns them to `account`, increasing
 * the total supply.
 *
 * Emits a {Transfer} event with `from` set to the zero address.
 *
 * Requirements:
 *
 * - `account` cannot be the zero address.
 */

function _mint(address account, uint256 amount) internal virtual {
    require(account != address(0), "BEP20: mint to the zero address");
    _totalSupply = _totalSupply.add(amount);
    _balances[account] = _balances[account].add(amount);
    emit Transfer(address(0), account, amount);
}

/**
 * @dev Destroys `amount` tokens from `account`, reducing the
 * total supply.
 *
 * Emits a {Transfer} event with `to` set to the zero address.
 *
 * Requirements:
 *
 * - `account` cannot be the zero address.
 * - `account` must have at least `amount` tokens.
 */

function _burn(address account, uint256 amount) internal virtual {
    require(account != address(0), "BEP20: burn from the zero address");
    uint256 accountBalance = _balances[account];
    require(accountBalance >= amount, "BEP20: burn amount exceeds balance");
    _balances[account] = accountBalance.sub(amount);
    _totalSupply = _totalSupply.sub(amount);
    emit Transfer(account, address(0), amount);
}

/**
 * @dev Sets `amount` as the allowance of `spender` over the `owner` s tokens.
 *
 * This internal function is equivalent to `approve`, and can be used to
```

```
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(
        address owner,
        address spender,
        uint256 amount
    ) internal {
        require(owner != address(0), "BEP20: approve from the zero address");
        //SlowMist// This kind of check is very good, avoiding user mistake leading
to approve errors
        require(spender != address(0), "BEP20: approve to the zero address");

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


}
```

VotingToken.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./SmartToken.sol";
import "./Math.sol";
import "./SafeMath.sol";
/**
 * @dev Extension of BEP20 to support voting and delegation. This version supports
token supply up to 2 ** 96 - 1.
 *
 * This extension keeps a history (checkpoints) of each account's vote power. Vote
power can be delegated either
 * by calling the {delegate} function directly, or by providing a signature to be
```

```
used with {delegateBySig}. Voting
 * power can be queried through the public accessors {getVotes} and {getPastVotes}.
 *
 * By default, token balance does not account for voting power. This makes transfers
cheaper. Acquiring vote power
 * requires token holders to delegate to themselves in order to activate checkpoints
and have their voting power
 * tracked.
 */


contract VotingToken is SmartToken {
    using SafeMath for uint256;
    struct Checkpoint {
        uint32 fromBlock;
        uint96 votes;
    }

    struct Delegatee {
        address _delegatee;
        uint96 votes;
    }

    /// @notice The EIP-712 typehash for the contract's domain
    bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string
name,uint256 chainId,address verifyingContract)");

    /// @notice The EIP-712 typehash for the delegation struct used by the contract
    bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address
delegatee,uint256 nonce,uint256 expiry)");

    /// @notice A record of states for signing / validating signatures
    mapping (address => uint) public nonces;

    mapping(address => Delegatee) private _delegates;
    mapping(address => Checkpoint[]) private _checkpoints;
    Checkpoint[] private _totalSupplyCheckpoints;

    /**
     * @dev Emitted when an account changes their delegate.
     */
    event DelegateeChanged(address indexed delegator, address indexed fromDelegate,
address indexed toDelegate);
```

```
    /**
     * @dev Emitted when a token transfer or delegate change results in changes to an
account's voting power.
     */
    event DelegateVotesChanged(address indexed delegate, uint256 previousBalance,
uint256 newBalance);


    /**
     * @dev Get the `pos`-th checkpoint for `account`.
     */
    function checkpoints(address account, uint32 pos) public view returns (Checkpoint
memory) {
        return _checkpoints[account][pos];
    }

    /**
     * @dev Get number of checkpoints for `account`.
     */
    function numCheckpoints(address account) public view returns (uint256) {
        return _checkpoints[account].length;
    }

    /**
     * @dev Get the address `account` is currently delegating to.
     */
    function delegates(address account) public view returns (address) {
        return _delegates[account]._delegatee;
    }

    /**
     * @dev Gets the current votes balance for `account`
     */
    function getVotes(address account) public view returns (uint96) {
        uint256 pos = _checkpoints[account].length;
        return pos == 0 ? 0 : _checkpoints[account][pos.sub(1)].votes;
    }

    /**
     * @dev Retrieve the number of votes for `account` at the end of `blockNumber`.
     *
     * Requirements:
     *
     * - `blockNumber` must have been already mined
     */
    function getPastVotes(address account, uint256 blockNumber) public view returns (uint96) {
        require(blockNumber < block.number, "BEP20Votes: block not yet mined");
        return _checkpointsLookup(_checkpoints[account], blockNumber);
    }

    /**
     * @dev Retrieve the `totalSupply` at the end of `blockNumber`. Note, this value is the sum of all balances.
     * It is but NOT the sum of all the delegated votes!
     *
     * Requirements:
     *
     * - `blockNumber` must have been already mined
     */
    function getPastTotalSupply(uint256 blockNumber) public view returns (uint96) {
        require(blockNumber < block.number, "BEP20Votes: block not yet mined");
        return _checkpointsLookup(_totalSupplyCheckpoints, blockNumber);
    }

    /**
     * @dev Lookup a value in a list of (sorted) checkpoints.
     */
    function _checkpointsLookup(Checkpoint[] storage ckpts, uint256 blockNumber) private view returns (uint96) {
        // We run a binary search to look for the earliest checkpoint taken after `blockNumber`.
        //
        // During the loop, the index of the wanted checkpoint remains in the range [low-1, high).
        // With each iteration, either `low` or `high` is moved towards the middle of the range to maintain the invariant.
        // - If the middle checkpoint is after `blockNumber`, we look in [low, mid)
        // - If the middle checkpoint is before or equal to `blockNumber`, we look in [mid+1, high)
        // Once we reach a single value (when low == high), we've found the right checkpoint at the index high-1, if not
        // out of bounds (in which case we're looking too far in the past and the result is 0).
        // Note that if the latest checkpoint available is exactly for `blockNumber`, we end up with an index that is
        // past the end of the array, so we technically don't find a checkpoint after
```

```
`blockNumber`, but it works out
        // the same.
        uint256 high = ckpts.length;
        uint256 low = 0;
        while (low < high) {
            uint256 mid = Math.average(low, high);
            if (ckpts[mid].fromBlock > blockNumber) {
                high = mid;
            } else {
                low = mid.add(1);
            }
        }

        return high == 0 ? 0 : ckpts[high.sub(1)].votes;
    }

    /**
     * @dev Delegate votes from the sender to `delegatee`.
     */
    function delegate(address delegatee) public {
        _delegate(_msgSender(), delegatee);
    }

    /**
     * @dev Remove previous delegatee and set it to zero address. After receiving
more tokens, token owner needs to
     * delegates once more to update voting powers. If source and destination
delegatee be the same (means token owner
     * wants to delegate the same address and update vote powers of the same
address), voting powers will not be updated.
     * In such case, token owner should call resetDelegate function and then delegate
to the address again.
     */
    function resetDelegate() public {
        _delegate(_msgSender(), address(0));
    }

    /**
     * @dev Delegates votes from signer to `delegatee`
     * @notice Delegates votes from signer to `delegatee`
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
```

```
 * @param r Half of the ECDSA signature pair
 * @param s Half of the ECDSA signature pair
 */
function delegateBySig(
    address delegatee,
    uint nonce,
    uint expiry,
    uint8 v,
    bytes32 r,
    bytes32 s
)
    public
{
    require(block.timestamp <= expiry, "BEP20Votes: signature expired");
    bytes32 domainSeparator = keccak256(
        abi.encode(
            DOMAIN_TYPEHASH,
            keccak256(bytes(name())),
            getChainId(),
            address(this)
        )
    );

    bytes32 structHash = keccak256(
        abi.encode(
            DELEGATION_TYPEHASH,
            delegatee,
            nonce,
            expiry
        )
    );

    bytes32 digest = keccak256(
        abi.encodePacked(
            "\x19\x01",
            domainSeparator,
            structHash
        )
    );

    address signer = ecrecover(digest, v, r, s);
    require(signer != address(0), "LQR::delegateBySig: invalid signature");
    require(nonce == nonces[signer]++, "LQR::delegateBySig: invalid nonce");
    return _delegate(signer, delegatee);
    }

    /**
     * @dev Maximum token supply is limited to 10 ** 10 units in order to avoid
inflation and overflow in voting mechanism.
     */
    function _maxSupply() internal pure returns (uint96) {
        return 10 ** 28;
    }

    /**
     * @dev Snapshots the totalSupply after it has been increased.
     */
    function _mint(address account, uint256 amount) internal override {
        super._mint(account, amount);
        require(totalSupply() <= _maxSupply(), "BEP20Votes: total supply risks
overflowing votes");

        _writeCheckpoint(_totalSupplyCheckpoints, _add, amount);
    }

    /**
     * @dev Snapshots the totalSupply after it has been decreased.
     */
    function _burn(address account, uint256 amount) internal override {
        super._burn(account, amount);
        if (delegates(account) != address(0)) {
            uint256 currentBalance = balanceOf(account);
            uint96 delegateeVotePower = _delegates[account].votes;
            if (currentBalance < delegateeVotePower) {
                uint256 diff = castTo256(delegateeVotePower).sub(currentBalance);
                _moveVotingPower(delegates(account), address(0), diff,
currentBalance);
                _delegates[account].votes = safeCastTo96(currentBalance,
"LQR::_writeCheckpoint: number exceeds 96 bits");
            }
        }

        _writeCheckpoint(_totalSupplyCheckpoints, _subtract, amount);
    }

    /**
    * @dev Move voting power when tokens are transferred.
    *
    * Emits a {DelegateVotesChanged} event.
     */
    function _afterTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal override {
        super._afterTokenTransfer(from, to, amount);
        if (delegates(from) != address(0)) {
            uint256 currentBalance = balanceOf(from);
            uint96 delegateeVotePower = _delegates[from].votes;
            if (currentBalance < delegateeVotePower) {
                uint256 diff = castTo256(delegateeVotePower).sub(currentBalance);
                _moveVotingPower(delegates(from), address(0), diff, currentBalance);
                _delegates[from].votes = safeCastTo96(currentBalance,
"LQR::_writeCheckpoint: number exceeds 96 bits");
            }
        }
    }

    /**
     * @dev Change delegation for `delegator` to `delegatee`.
     *
     * Emits events {DelegateeChanged} and {DelegateVotesChanged}.
     */
    function _delegate(address delegator, address delegatee) internal {
        address currentDelegate = delegates(delegator);
        uint256 currentVotePower = castTo256(_delegates[delegator].votes);
        uint256 delegatorBalance = balanceOf(delegator);
        if (currentDelegate != delegatee) {
            _delegates[delegator]._delegatee = delegatee;
            _delegates[delegator].votes = delegatee == address(0) ? 0 :
safeCastTo96(delegatorBalance, "LQR::_writeCheckpoint: number exceeds 96 bits");
            emit DelegateeChanged(delegator, currentDelegate, delegatee);
        }

        _moveVotingPower(currentDelegate, delegatee, currentVotePower,
delegatorBalance);
    }

    /**
    * @dev The function returns the chain id in which token contract
     */
    function getChainId() internal view returns (uint) {
        uint256 chainId;
        assembly { chainId := chainid() }
        return chainId;
    }

    function safeCastTo96(uint n, string memory errorMessage) internal pure returns
(uint96) {
        require(n < 2**96, errorMessage);
        return uint96(n);
    }

    function safeCastTo32(uint n, string memory errorMessage) internal pure returns
(uint32) {
        require(n < 2**32, errorMessage);
        return uint32(n);
    }

    function castTo256(uint96 n) internal pure returns (uint256) {
        return uint256(n);
    }

    function _moveVotingPower(
        address src,
        address dst,
        uint256 transferredVote,
        uint256 amount
    ) private {
        if (src != dst && amount > 0) {
            if (src != address(0)) {
                (uint256 oldWeight, uint256 newWeight) =
_writeCheckpoint(_checkpoints[src], _subtract, transferredVote);
                emit DelegateVotesChanged(src, oldWeight, newWeight);
            }

            if (dst != address(0)) {
                (uint256 oldWeight, uint256 newWeight) =
_writeCheckpoint(_checkpoints[dst], _add, amount);
                emit DelegateVotesChanged(dst, oldWeight, newWeight);
            }
        }
    }

    function _writeCheckpoint(
        Checkpoint[] storage ckpts,
        function(uint256, uint256) view returns (uint256) op,
        uint256 delta
    ) private returns (uint256 oldWeight, uint256 newWeight) {
        uint256 pos = ckpts.length;
        oldWeight = pos == 0 ? 0 : castTo256(ckpts[pos.sub(1)].votes);
        newWeight = op(oldWeight, delta);

        uint32 blockNumber = safeCastTo32(block.number, "LQR::_writeCheckpoint: block
number exceeds 32 bits");
        if (pos > 0 && ckpts[pos.sub(1)].fromBlock == blockNumber) {
            ckpts[pos.sub(1)].votes = safeCastTo96(newWeight, "LQR::_writeCheckpoint:
number exceeds 96 bits");
        } else {
            ckpts.push(Checkpoint({fromBlock: blockNumber, votes:
safeCastTo96(newWeight, "LQR::_writeCheckpoint: number exceeds 96 bits")}));
        }
    }

    function _add(uint256 a, uint256 b) private pure returns (uint256) {
        return a + b;
    }

    function _subtract(uint256 a, uint256 b) private pure returns (uint256) {
        return a - b;
    }
}
```

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.

SLOWMIST

Official Website

www.slowmist.com

✉

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist