Smart Contract Security Audit Report

[2021]

The SlowMist Security Team received the BORA team's application for smart contract security audit of the BORA Token on 2021.10.08. The following are the details and results of this smart contract security audit:

Token Name :

BORA Token

File name and hash (SHA256) :

31.BORAToken-klaytn-remix-20210914.sol.txt:

56ef104fe7c744aa0c4d22bb838740f91dfe05367b8fd5feba3c325b179c93c0

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

NO.	Audit Items	Result
12	Scoping and Declarations Audit	Passed
13	Safety Design Audit	Passed

Audit Result : Passed

Audit Number : 0X002110120002

Audit Date : 2021.10.08 - 2021.10.12

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that contains the tokenvault section. The total amount of tokens in the contract can be changed, the burner role can burn his 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 caller needs to have the three roles of Burner, Pauser, and whitelistAdmin at the same time to add a new admin.
2. Only WhitelistAdmin can add users' accounts in whitelist/blacklist through the addWhitelisted/addBlacklisted function.
3. Only WhitelistAdmin can remove users' accounts from whitelist/blacklist through the removeWhitelisted/removeBlacklisted function.
4. The _donor role in the LockToken can directly use the revoke function to withdraw the tokens in the contract without the need to lock the claim after the time when _revocable is true.

The source code:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.5.5;
```

```
contract Context {
    constructor () internal { }

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

    function _msgData() internal view returns (bytes memory) {
        this;
        return msg.data;
    }
}
//SlowMist// SafeMath security module is used, which is a recommended approach
library SafeMath {
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure
returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
            return 0;
        }

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

        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
```

```
        return div(a, b, "SafeMath: division by zero");
    }

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure
returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure
returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}

interface IERC20 {
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address recipient, uint256 amount) external returns (bool);
    function allowance(address owner, address spender) external view returns
(uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(address sender, address recipient, uint256 amount) external
returns (bool);

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
}

contract ERC20 is Context, IERC20 {
    using SafeMath for uint256;

    mapping (address => uint256) private _balances;
    mapping (address => mapping (address => uint256)) private _allowances;

    uint256 private _totalSupply;

    function totalSupply() public view returns (uint256) {
```

```
        return _totalSupply;
    }

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

    function transfer(address recipient, uint256 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 (uint256)
{
        return _allowances[owner][spender];
    }

    function approve(address spender, uint256 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, uint256 amount) public
returns (bool) {
        _transfer(sender, recipient, amount);
        _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount,
"ERC20: transfer amount exceeds allowance"));
        //SlowMist// The return value conforms to the EIP20 specification
        return true;
    }

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

    function decreaseAllowance(address spender, uint256 subtractedValue) public
returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()]
[spender].sub(subtractedValue, "ERC20: decreased allowance below zero"));
        return true;
```

```
    }

    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: 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), "ERC20: transfer to the zero address");

        _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount
exceeds balance");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }

    function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: 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, uint256 amount) internal {
        require(account != address(0), "ERC20: burn from the zero address");

        _balances[account] = _balances[account].sub(amount, "ERC20: burn amount
exceeds balance");
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(account, address(0), amount);
    }

    function _approve(address owner, address spender, uint256 amount) internal {
        require(owner != address(0), "ERC20: approve from the zero address");
        //SlowMist// This kind of check is very good, avoiding user mistake leading
to approve errors
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }
    //SlowMist// Because burnFrom() and transferFrom() share the allowed amount of
approve(), if the agent be evil, there is the possibility of malicious burn
    function _burnFrom(address account, uint256 amount) internal {
        _burn(account, amount);
        _approve(account, _msgSender(), _allowances[account]
```

```
[_msgSender()].sub(amount, "ERC20: burn amount exceeds allowance"));
    }
}

contract ERC20Detailed is IERC20 {
    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;
    }
}

library Roles {
    struct Role {
        mapping (address => bool) bearer;
    }

    function add(Role storage role, address account) internal {
        require(!has(role, account), "Roles: account already has role");
        role.bearer[account] = true;
    }

    function remove(Role storage role, address account) internal {
        require(has(role, account), "Roles: account does not have role");
        role.bearer[account] = false;
    }

    function has(Role storage role, address account) internal view returns (bool) {
        require(account != address(0), "Roles: account is the zero address");
```

```
        return role.bearer[account];
    }
}

contract BurnerRole is Context {
    using Roles for Roles.Role;

    event BurnerAdded(address indexed account);
    event BurnerRemoved(address indexed account);

    Roles.Role private _burners;

    constructor () internal {
        _addBurner(_msgSender());
    }

    modifier onlyBurner() {
        require(isBurner(_msgSender()), "BurnerRole: caller does not have the Burner
role");
        _;
    }

    function isBurner(address account) public view returns (bool) {
        return _burners.has(account);
    }
    //SlowMist// Only Burner can add a new burner through the addBurner function
    function addBurner(address account) public onlyBurner {
        _addBurner(account);
    }

    function renounceBurner() public {
        _removeBurner(_msgSender());
    }

    function _addBurner(address account) internal {
        _burners.add(account);
        emit BurnerAdded(account);
    }

    function _removeBurner(address account) internal {
        _burners.remove(account);
        emit BurnerRemoved(account);
    }
}
```

```
contract ERC20Burnable is ERC20, BurnerRole {

    function burn(uint256 amount) public {
        _burn(_msgSender(), amount);
    }

    function burnFrom(address account, uint256 amount) public {
        _burnFrom(account, amount);
    }
}

contract PauserRole is Context {
    using Roles for Roles.Role;

    event PauserAdded(address indexed account);
    event PauserRemoved(address indexed account);

    Roles.Role private _pausers;

    constructor () internal {
        _addPauser(_msgSender());
    }

    modifier onlyPauser() {
        require(isPauser(_msgSender()), "PauserRole: caller does not have the Pauser
role");
        _;
    }

    function isPauser(address account) public view returns (bool) {
        return _pausers.has(account);
    }
    //SlowMist// Only Pauser can add a new pauser through the addPauser function
    function addPauser(address account) public onlyPauser {
        _addPauser(account);
    }

    function renouncePauser() public {
        _removePauser(_msgSender());
    }

    function _addPauser(address account) internal {
        _pausers.add(account);
        emit PauserAdded(account);
    }
```

```
    function _removePauser(address account) internal {
        _pausers.remove(account);
        emit PauserRemoved(account);
    }
}

contract Pausable is Context, PauserRole {

    event Paused(address account);
    event Unpaused(address account);

    bool private _paused;

    constructor () internal {
        _paused = false;
    }

    function paused() public view returns (bool) {
        return _paused;
    }

    modifier whenNotPaused() {
        require(!_paused, "Pausable: paused");
        _;
    }

    modifier whenPaused() {
        require(_paused, "Pausable: not paused");
        _;
    }
    //SlowMist// Suspending all transactions upon major abnormalities is a
recommended approach.
    function pause() public onlyPauser whenNotPaused {
        _paused = true;
        emit Paused(_msgSender());
    }

    function unpause() public onlyPauser whenPaused {
        _paused = false;
        emit Unpaused(_msgSender());
    }
}

contract ERC20Pausable is ERC20, Pausable {
```

```
    function transfer(address to, uint256 value) public whenNotPaused returns (bool)
{
        return super.transfer(to, value);
    }

    function transferFrom(address from, address to, uint256 value) public
whenNotPaused returns (bool) {
        return super.transferFrom(from, to, value);
    }

    function approve(address spender, uint256 value) public whenNotPaused returns
(bool) {
        return super.approve(spender, value);
    }

    function increaseAllowance(address spender, uint256 addedValue) public
whenNotPaused returns (bool) {
        return super.increaseAllowance(spender, addedValue);
    }

    function decreaseAllowance(address spender, uint256 subtractedValue) public
whenNotPaused returns (bool) {
        return super.decreaseAllowance(spender, subtractedValue);
    }
}

contract Ownable is Context {
    address private _owner;

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

    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    function owner() public view returns (address) {
        return _owner;
    }

    modifier onlyOwner() {
        require(isOwner(), "Ownable: caller is not the owner");
        _;
```

```
    }

    function isOwner() public view returns (bool) {
        return _msgSender() == _owner;
    }

    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    function transferOwnership(address newOwner) public onlyOwner {
        _transferOwnership(newOwner);
    }

    function _transferOwnership(address newOwner) internal {
        //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");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

contract WhitelistAdminRole is Context {
    using Roles for Roles.Role;

    event WhitelistAdminAdded(address indexed account);
    event WhitelistAdminRemoved(address indexed account);

    Roles.Role private _whitelistAdmins;

    constructor () internal {
        _addWhitelistAdmin(_msgSender());
    }

    modifier onlyWhitelistAdmin() {
        require(isWhitelistAdmin(_msgSender()), "WhitelistAdminRole: caller does not
have the WhitelistAdmin role");
        _;
    }

    function isWhitelistAdmin(address account) public view returns (bool) {
        return _whitelistAdmins.has(account);
    }
```

```
    //SlowMist// Only WhitelistAdmin can add a new WhitelistAdmin through the
addWhitelistAdmin function
    function addWhitelistAdmin(address account) public onlyWhitelistAdmin {
        _addWhitelistAdmin(account);
    }

    function renounceWhitelistAdmin() public {
        _removeWhitelistAdmin(_msgSender());
    }

    function _addWhitelistAdmin(address account) internal {
        _whitelistAdmins.add(account);
        emit WhitelistAdminAdded(account);
    }

    function _removeWhitelistAdmin(address account) internal {
        _whitelistAdmins.remove(account);
        emit WhitelistAdminRemoved(account);
    }
}

contract WhitelistedRole is Context, WhitelistAdminRole {
    using Roles for Roles.Role;

    event WhitelistedAdded(address indexed account);
    event WhitelistedRemoved(address indexed account);

    Roles.Role private _whitelisteds;

    modifier onlyWhitelisted() {
        require(isWhitelisted(_msgSender()), "WhitelistedRole: caller does not have
the Whitelisted role");
        _;
    }

    function isWhitelisted(address account) public view returns (bool) {
        return _whitelisteds.has(account);
    }
    //SlowMist// Only WhitelistAdmin can add users' accounts in whitelist through the
addWhitelisted function
    function addWhitelisted(address account) public onlyWhitelistAdmin {
        _addWhitelisted(account);
    }
    //SlowMist// Only WhitelistAdmin can remove users' accounts from whitelist
through the removeWhitelisted function
```

```
    function removeWhitelisted(address account) public onlyWhitelistAdmin {
        _removeWhitelisted(account);
    }

    function renounceWhitelisted() public {
        _removeWhitelisted(_msgSender());
    }

    function _addWhitelisted(address account) internal {
        _whitelisteds.add(account);
        emit WhitelistedAdded(account);
    }

    function _removeWhitelisted(address account) internal {
        _whitelisteds.remove(account);
        emit WhitelistedRemoved(account);
    }
}

contract BlacklistedRole is WhitelistAdminRole {
    using Roles for Roles.Role;

    event BlacklistedAdded(address indexed account);
    event BlacklistedRemoved(address indexed account);

    Roles.Role private _blacklisteds;

    modifier notBlacklisted() {
        require(!isBlacklisted(msg.sender), "BlacklistedRole: caller has the
Blacklisted role");
        _;
    }

    function isBlacklisted(address account) public view returns (bool) {
        return _blacklisteds.has(account);
    }
    //SlowMist// Only WhitelistAdmin can add users' accounts in blacklist through the
addBlacklisted function
    function addBlacklisted(address account) public onlyWhitelistAdmin {
        _addBlacklisted(account);
    }
    //SlowMist// Only WhitelistAdmin can remove users' accounts from blacklist
through the removeBlacklisted function
    function removeBlacklisted(address account) public onlyWhitelistAdmin {
        _removeBlacklisted(account);
```

```
    }

    function _addBlacklisted(address account) internal {
        _blacklisteds.add(account);
        emit BlacklistedAdded(account);
    }

    function _removeBlacklisted(address account) internal {
        _blacklisteds.remove(account);
        emit BlacklistedRemoved(account);
    }
}

library Address {

    function isContract(address account) internal view returns (bool) {
        bytes32 codehash;
        bytes32 accountHash =
0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;

        assembly { codehash := extcodehash(account) }
        return (codehash != accountHash && codehash != 0x0);
    }

    function toPayable(address account) internal pure returns (address payable) {
        return address(uint160(account));
    }

    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        (bool success, ) = recipient.call.value(amount)("");
        require(success, "Address: unable to send value, recipient may have
reverted");
    }
}

library SafeERC20 {
    using SafeMath for uint256;
    using Address for address;

    function safeTransfer(IERC20 token, address to, uint256 value) internal {
        callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to,
value));
    }
```

```
    function safeTransferFrom(IERC20 token, address from, address to, uint256 value)
internal {
        callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector,
from, to, value));
    }

    function safeApprove(IERC20 token, address spender, uint256 value) internal {
        require((value == 0) || (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
spender, value));
    }

    function safeIncreaseAllowance(IERC20 token, address spender, uint256 value)
internal {
        uint256 newAllowance = token.allowance(address(this), spender).add(value);
        callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
spender, newAllowance));
    }

    function safeDecreaseAllowance(IERC20 token, address spender, uint256 value)
internal {
        uint256 newAllowance = token.allowance(address(this), spender).sub(value,
"SafeERC20: decreased allowance below zero");
        callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
spender, newAllowance));
    }

    function callOptionalReturn(IERC20 token, bytes memory data) private {
        require(address(token).isContract(), "SafeERC20: call to non-contract");

        (bool success, bytes memory returndata) = address(token).call(data);
        require(success, "SafeERC20: low-level call failed");

        if (returndata.length > 0) {
            require(abi.decode(returndata, (bool)), "SafeERC20: ERC20 operation did
not succeed");
        }
    }
}
```

//SlowMist// The _donor role in the LockToken can directly use the revoke function to withdraw the tokens in the contract without the need to lock the claim after time when _revocable is true.

```
contract LockedToken {
    using SafeERC20 for IERC20;

    IERC20 private _token;

    address private _donor;
    address private _beneficiary;
    uint256 private _releaseTime;
    bool public _revocable;

    event Claim(address beneficiary, uint256 amount, uint256 releaseTime);
    event Revoke(address donor, uint256 amount);

    constructor (IERC20 token, address donor, address beneficiary, uint256
releaseTime, bool revocable) public {
        require(donor != address(0), "LockedToken: donor is zero address");
        require(beneficiary != address(0), "LockedToken: beneficiary is zero
address");
        require(releaseTime > block.timestamp, "LockedToken: release time is before
current time");

        _token = IERC20(token);
        _donor = donor;
        _beneficiary = beneficiary;
        _releaseTime = releaseTime;
        _revocable = revocable;
    }

    function token() public view returns (IERC20) {
        return _token;
    }

    function donor() public view returns (address) {
        return _donor;
    }

    function beneficiary() public view returns (address) {
        return _beneficiary;
    }

    function releaseTime() public view returns (uint256) {
        return _releaseTime;
    }

    function revocable() public view returns (bool) {
```

```
        return _revocable;
    }

    function balanceOf() public view returns (uint256) {
        return _token.balanceOf(address(this));
    }

    function revoke() public {
        require(_revocable, "LockedToken: tokens are not revocable");
        require(msg.sender == _donor, "LockedToken: only donor can revoke");

        uint amount = _token.balanceOf(address(this));
        require(amount > 0, "LockedToken: no tokens to revoke");

        _token.safeTransfer(_donor, amount);
        emit Revoke(_donor, amount);
    }

    function claim() public {
        require(block.timestamp >= _releaseTime, "LockedToken: current time is before
release time");

        uint amount = _token.balanceOf(address(this));
        require(amount > 0, "LockedToken: no tokens to claim");

        _token.safeTransfer(_beneficiary, amount);
        emit Claim(_beneficiary, amount, _releaseTime);
    }
}

contract BoraToken is ERC20, ERC20Detailed, ERC20Burnable, ERC20Pausable, Ownable,
BlacklistedRole {

    event Lock(address token, address beneficiary, uint256 amount, uint256
releaseTime);
    event Burn(address to, uint256 amount, uint256 totalSupply);
    event AdminAdded(address indexed account);
    event AdminRemoved(address indexed account);
    event OwnershipTransferred(address indexed previousOwner, address indexed
newOwner);

    constructor(uint256 initialSupply) ERC20Detailed("BORA", "BORA", 18) public {
        _mint(msg.sender, initialSupply);
    }
```

```
    function transfer(address to, uint256 value) public notBlacklisted returns (bool)
{
        return super.transfer(to, value);
    }

    function transferFrom(address sender, address recipient, uint256 amount) public
notBlacklisted returns (bool) {
        require(!isBlacklisted(sender), "BoraToken: sender has the Blacklisted
role");
        return super.transferFrom(sender, recipient, amount);
    }

    function lock(address donor, address beneficiary, uint256 amount, uint256
duration, bool revocable) public onlyPauser returns (LockedToken) {
        uint256 releaseTime = now.add(duration.mul(1 days));
        LockedToken lockedToken = new LockedToken(this, donor, beneficiary,
releaseTime, revocable);
        transfer(address(lockedToken), amount);
        emit Lock(address(lockedToken), beneficiary, lockedToken.balanceOf(),
releaseTime);
        return lockedToken;
    }

    function burn(uint256 amount) public onlyBurner {
        ERC20Burnable.burn(amount);
        emit Burn(msg.sender, amount, totalSupply());
    }

    function burnFrom(address account, uint256 amount) public onlyBurner {
        ERC20Burnable.burnFrom(account, amount);
        emit Burn(account, amount, totalSupply());
    }
    //SlowMist// The caller needs to have the three roles of Burner, Pauser and
whitelistAdmin at the same time to add a new admin
    function addAdmin(address newAdmin) public onlyBurner onlyPauser
onlyWhitelistAdmin {
        require(newAdmin != address(0), "BoraToken: add admin of the zero address");
        if ( (!isBurner(newAdmin))){
            super.addBurner(newAdmin);
        }
        if ( !isPauser(newAdmin)){
            super.addPauser(newAdmin);
        }
        if ( !isWhitelistAdmin(newAdmin)){
            super.addWhitelistAdmin(newAdmin);
```

```
        }
        emit AdminAdded(newAdmin);
    }

    function renounceAdmin() public onlyBurner onlyPauser onlyWhitelistAdmin {
        require(!isOwner(), "BoraToken: owner should be admin");
        super.renounceBurner();
        super.renouncePauser();
        super.renounceWhitelistAdmin();
        emit AdminRemoved(msg.sender);
    }

    function transferOwnership(address newOwner) public onlyOwner {
        addAdmin(newOwner);
        Ownable.transferOwnership(newOwner);
    }

    function renounceOwnership() public onlyOwner {
        revert("BoraToken: renounceOwnership is disabled");
    }

    function renounceBurner() public onlyBurner {
        require(!isOwner(), "BoraToken: owner should have the Burner role");
        super.renounceBurner();
    }

    function renouncePauser() public onlyPauser {
        require(!isOwner(), "BoraToken: owner should have the Pauser role");
        super.renouncePauser();
    }

    function renounceWhitelistAdmin() public onlyWhitelistAdmin {
        require(!isOwner(), "BoraToken: owner should have the WhitelistAdmin role");
        super.renounceWhitelistAdmin();
    }
}
```

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