KNOWNSEC
Blockchain Lab

Smart Contract Audit Report

Version description

The revision	Date	Revised	Version
Write documentation	20220309	KNOWNSEC Blockchain Lab	V1.0

Document information

Title	Version	Document Number	Type
ROD Smart Contract Audit Report	V1.0	194661d2e69947a993571f862d6dbebc	Open to project team

Statement

KNOWNSEC Blockchain Lab only issues this report for facts that have occurred or existed before the issuance of this report, and assumes corresponding responsibilities for this. KNOWNSEC Blockchain Lab is unable to determine the security status of its smart contracts and is not responsible for the facts that will occur or exist in the future. The security audit analysis and other content made in this report are only based on the documents and information provided to us by the information provider as of the time this report is issued. KNOWNSEC Blockchain Lab 's assumption: There is no missing, tampered, deleted or concealed information. If the information provided is missing, tampered with, deleted, concealed or reflected in the actual situation, KNOWNSEC Blockchain Lab shall not be liable for any losses and adverse effects caused thereby.

Directory

1. Summarize

The effective test period of this report is from **March 5, 2022 to March 9, 2022**. During this period, **the RODToken token code security and standardization of ROD smart contracts** will be audited and used as the statistical basis for the report.

The scope of this smart contract security audit does not include external contract calls, new attack methods that may appear in the future, and code after contract upgrades or tampering. (With the development of the project, the smart contract may add a new pool , New functional modules, new external contract calls, etc.), does not include front-end security and server security.

In this audit report, engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (Chapter 6). **The smart contract code of the ROD** is comprehensively assessed as **PASS**.

Since the testing is under non-production environment, all codes are the latest version. In addition, the testing process is communicated with the relevant engineer, and testing operations are carried out under the controllable operational risk to avoid production during the testing process, such as: Operational risk, code security risk.

KNOWNSEC Attest information:

classification	information
report number	194661d2e69947a993571f862d6dbebc
report query link	https://attest.im/attestation/searchResult?qurey=194661d2e69947a993571f862d6dbebc

2. Item information

2.1. Item description

ROD is a digital token on the Binance Smart Chain, which meets the convenience of dual-currency storage and management. ROD and BSC-USDT can be freely exchanged. The new financial exchange mechanism builds a perfect ecology to meet the ultimate needs of the ecology. Completely anonymous transactions. Property freedom, a zero-knowledge transaction method, truly free to control your own property, convenient and safe.

2.2. The project's website

https://bscrod.github.io

2.3. White Paper

https://bscrod.github.io/whitePaper.pdf

2.4. Review version code

https://bscscan.com/address/0x9a829d93b956193bb3c28182e72d1052f3ec4893#code

2.5. Contract file and Hash/contract deployment address

The contract documents	MD5
ROD0303.sol	b5e7b0f176a4f8785f9b8eaa67d910ca

3. External visibility analysis

3.1. RODToken contracts

RODToken					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
name	public	False	---	---	---
symbol	public	False	---	---	---
decimals	public	False	---	---	---
totalSupply	public	False	override	---	---
balanceOf	public	False	override	---	---
transfer	public	True	override	---	---
allowance	public	False	override	---	---
approve	public	True	override	---	---
transferFrom	public	True	override	---	---
increaseAllowance	public	True	virtual	---	---
decreaseAllowance	public	True	virtual	---	---
isExcludedFromReward	public	False	---	---	---
totalFees	public	False	---	---	---
deliver	public	True	---	---	---
reflectionFromToken	public	False	---	---	---

tokenFromReflection	public	False	---	---	---
excludeFromReward	public	True	onlyOwner	---	---
includeInReward	external	True	onlyOwner	---	---
_transferBothExcluded	private	True	---	---	---
excludeFromFee	public	True	onlyOwner	---	---
includeInFee	public	True	onlyOwner	---	---
_reflectFee	private	True	---	---	---
_getValues	private	False	---	---	---
_getTValues	private	False	---	---	---
_getRValues	private	False	---	---	---
_getRate	private	False	---	---	---
_getCurrentSupply	private	False	---	---	---
_takeLiquidity	private	True	---	---	---
calculateTaxFee	private	False	---	---	---
calculateLiquidityFee	private	False	---	---	---
removeAllFee	private	True	---	---	---
restoreAllFee	private	True	---	---	---
isExcludedFromFee	public	False	---	---	---
_approve	private	True	---	---	---
_transfer	private	True	---	---	---

setMarketingWallet	external	True	onlyOwner	---	---
swapAndMarketing	private	True	lockTheSwap	---	---
swapTokensForUsdt	private	True	---	---	---
_tokenTransfer	private	True	---	---	---
_transferStandard	private	True	---	---	---
_transferToExcluded	private	True	---	---	---
_transferFromExcluded	private	True	---	---	---
saveStuckedToken	public	True	onlyOwner	---	---
sweep	external	True	onlyOwner	---	---

4. Code vulnerability analysis

4.1. Summary description of the audit results

Audit results			
audit project	**audit content**	**condition**	**description**
Business security detection	Token function	Pass	After testing, there is no security issue.
	Whitelist account setting function	Pass	After testing, there is no security issue.
	Transfer function	Reminder	After detection, there is no security problem, but the event listener is wrong.
Code basic vulnerability detection	Compiler version security	Pass	After testing, there is no security issue.
	Redundant code	Pass	After testing, there is no security issue.
	Use of safe arithmetic library	Pass	After testing, there is no security issue.
	Not recommended encoding	Pass	After testing, there is no security issue.
	Reasonable use of require/assert	Pass	After testing, there is no security issue.
	fallback function safety	Pass	After testing, there is no security issue.
	tx.origin authentication	Pass	After testing, there is no security issue.
	Owner permission control	Pass	After testing, there is no security issue.
	Gas consumption detection	Pass	After testing, there is no security issue.
	call injection attack	Pass	After testing, there is no security issue.

	Low-level function safety	Pass	After testing, there is no security issue.
	Vulnerability of additional token issuance	Pass	After testing, there is no security issue.
	Access control defect detection	Pass	After testing, there is no security issue.
	Numerical overflow detection	Pass	After testing, there is no security issue.
	Arithmetic accuracy error	Pass	After testing, there is no security issue.
	Wrong use of random number detection	Pass	After testing, there is no security issue.
	Unsafe interface use	Pass	After testing, there is no security issue.
	Variable coverage	Pass	After testing, there is no security issue.
	Uninitialized storage pointer	Pass	After testing, there is no security issue.
	Return value call verification	Pass	After testing, there is no security issue.
	Transaction order dependency detection	Pass	After testing, there is no security issue.
	Timestamp dependent attack	Pass	After testing, there is no security issue.
	Denial of service attack detection	Pass	After testing, there is no security issue.
	Fake recharge vulnerability detection	Pass	After testing, there is no security issue.
	Reentry attack detection	Pass	After testing, there is no security issue.

	Replay attack detection	Pass	After testing, there is no security issue.
	Rearrangement attack detection	Pass	After testing, there is no security issue.

5. Business security detection

5.1. Token function【Pass】

Audit analysis: In the contract RODToken.sol, the token is inherited from the IERC20 contract class and conforms to the ERC20 token contract standard. After the audit, the function of the token is complete and the logical design is reasonable.

```
contract RODToken is Context, IERC20, Ownable {
    using SafeMath for uint256;
    using Address for address;

    mapping (address => uint256) private _rOwned;
    mapping (address => uint256) private _tOwned;
    mapping (address => mapping (address => uint256)) private _allowances;

    mapping (address => bool) private _isExcludedFromFee;
    mapping (address => bool) private _isExcluded;
    address[] private _excluded;

    uint256 private constant MAX = ~uint256(0);
    uint256 private _tTotal = 2 * 10**8 * 10**18;
    uint256 private _rTotal = (MAX - (MAX % _tTotal));
    uint256 private _tFeeTotal;

    string private _name = "ROD";// knownsec // token name
    string private _symbol = "ROD";// knownsec // token symbol
    uint8 private _decimals = 18;// knownsec // token Accuracy

    uint256 public _taxFee = 3;
    uint256 private _previousTaxFee = _taxFee;

    uint256 public _liquidityFee = 7;
```

```
uint256 private _previousLiquidityFee = _liquidityFee;

IUniswapV2Router02 public uniswapV2Router;
address public uniswapV2Pair;

bool inSwapAndLiquify;
bool public swapAndLiquifyEnabled = true;

uint256 private numTokensSellToAddToLiquidity = 1 * 10**18;
IERC20 public usdt = IERC20(0x55d398326f99059fF775485246999027B3197955);

address payable marketingWalletAddress =
0x3fbe0540b69aAF7883cCAf5E9b3B85E570Bb0003;
address payable deadAddress = 0x000000000000000000000000000000000000dEaD;

event MinTokensBeforeSwapUpdated(uint256 minTokensBeforeSwap);
event SwapAndLiquifyEnabledUpdated(bool enabled);
event SwapAndLiquify(
    uint256 tokensSwapped,
    uint256 ethReceived,
    uint256 tokensIntoLiqudity
);

modifier lockTheSwap {
    inSwapAndLiquify = true;
    _;
    inSwapAndLiquify = false;
}

constructor () public {// knownsec // constructor
    _rOwned[_msgSender()] = _rTotal;

    IUniswapV2Router02 _uniswapV2Router =
```

```
IUniswapV2Router02(0x10ED43C718714eb63d5aA57B78B54704E256024E);
        // Create a uniswap pair for this new token
        uniswapV2Pair =
IUniswapV2Factory(_uniswapV2Router.factory()).createPair(address(this), address(usdt));

        // set the rest of the contract variables
        uniswapV2Router = _uniswapV2Router;

        //exclude owner and this contract from fee
        _isExcludedFromFee[owner()] = true;
        _isExcludedFromFee[address(this)] = true;

        emit Transfer(address(0), _msgSender(), _tTotal);
    }
```

Security advice: None.

5.2. Whitelist account setting function【Pass】

Audit analysis: In the contract RODToken.sol, _isExcluded[account] is used to judge whether it is a whitelisted account, and the Owner can set whether the user is a whitelisted account by calling excludeFromReward and includeInReward. After the audit, the authority control is correct and the logic design is reasonable.

```
function excludeFromReward(address account) public onlyOwner() {// knownsec // Set up user whitelist
        require(!_isExcluded[account], "Account is already excluded");
        if(_rOwned[account] > 0) {
            _tOwned[account] = tokenFromReflection(_rOwned[account]);
        }
        _isExcluded[account] = true;
        _excluded.push(account);
```

```
    }

    function includeInReward(address account) external onlyOwner() {// knownsec // Cancel user whitelist
        require(_isExcluded[account], "Account is already excluded");
        for (uint256 i = 0; i < _excluded.length; i++) {
            if (_excluded[i] == account) {
                _excluded[i] = _excluded[_excluded.length - 1];
                _tOwned[account] = 0;
                _isExcluded[account] = false;
                _excluded.pop();
                break;
            }
        }
    }
```

Security advice: None.

5.3. Transfer function 【Reminder】

Audit analysis: In the contract RODToken.sol, the transfer account number and the transfer amount are checked before the transfer, whether it conforms to swapAndMarketing according to the conditions, and then whether the fee is charged, and finally the token transfer is carried out. When transferring, it determines whether the sender and receiver are whitelisted accounts, and uses different transfer functions according to the actual situation to complete the transfer after deducting the transfer fee and liquidity fee. After the audit, the authority control is correct and the function design is reasonable. In the swapAndMarketing() function, the function event should be monitored, but the contract is monitoring a non-existent function event

SwapAndLiquify.

```
function transfer(address recipient, uint256 amount) public override returns (bool) {// knownsec // transfer
        _transfer(_msgSender(), recipient, amount);
        return true;
    }
    .......
function _transfer(
        address from,
        address to,
        uint256 amount
    ) private {
        require(from != address(0), "ERC20: transfer from the zero address");
        require(to != address(0), "ERC20: transfer to the zero address");
        require(amount > 0, "Transfer amount must be greater than zero");

        // is the token balance of this contract address over the min number of
        // tokens that we need to initiate a swap + liquidity lock?
        // also, don't get caught in a circular liquidity event.
        // also, don't swap & liquify if sender is uniswap pair.
        uint256 contractTokenBalance = balanceOf(address(this));
        bool overMinTokenBalance = contractTokenBalance >=
numTokensSellToAddToLiquidity;// knownsec // Whether the token contract balance is greater than numTokensSellToAddToLiquidity
        if (
            overMinTokenBalance &&
            !inSwapAndLiquify &&
            from != uniswapV2Pair &&
            swapAndLiquifyEnabled
        ) {// knownsec // After meeting the conditions，swapAndMarketing
            swapAndMarketing(contractTokenBalance);// knownsec // Destroy 70% of the contractTokenBalance amount of tokens, and exchange the other 30% of tokens to usdt to
```

```
marketingWalletAddress
        }

        //indicates if fee should be deducted from transfer
        bool takeFee = true;

        //if any account belongs to _isExcludedFromFee account then remove the fee
        if(_isExcludedFromFee[from] || _isExcludedFromFee[to] || from ==
uniswapV2Pair){// knownsec // Determine whether it is a whitelisted account, if it is a whitelisted
account, no handling fee will be charged
            takeFee = false;
        }

        //transfer amount, it will take tax, burn, liquidity fee
        _tokenTransfer(from,to,amount,takeFee);// knownsec // transfer
    }
function swapAndMarketing(uint256 contractTokenBalance) private lockTheSwap {
        //5% destroy��2% marketing
        uint256 deadFee = contractTokenBalance.mul(71).div(100); //knownsec// 5% will be
destroyed and destroyed to the black hole address
        uint256 marketingFee = contractTokenBalance.mul(29).div(100); //knownsec// 2% for
marketing to support marketing campaigns and airdrops

        // split the contract balance into halves
        uint256 half = marketingFee;
        // swap tokens for ETH
        swapTokensForUsdt(half); // <- this breaks the ETH -> HATE swap when swap+liquify
is triggered //knownsec// Incoming Market Wallet

        // how much ETH did we just swap into?
        IERC20(address(this)).transfer(deadAddress, deadFee); //knownsec// destroy

        emit SwapAndLiquify(contractTokenBalance, deadFee, marketingFee); //knownsec// This
```

```
should listen to swapAndMarketing instead of SwapAndLiquify
    }
    ......
function _tokenTransfer(address sender, address recipient, uint256 amount,bool takeFee) private
{// knownsec // actual transfer
        if(!takeFee)
            removeAllFee();
        // knownsec // Select the transfer channel according to whether it is a whitelisted account
        if (_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferFromExcluded(sender, recipient, amount);
        } else if (!_isExcluded[sender] && _isExcluded[recipient]) {
            _transferToExcluded(sender, recipient, amount);
        } else if (!_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferStandard(sender, recipient, amount);
        } else if (_isExcluded[sender] && _isExcluded[recipient]) {
            _transferBothExcluded(sender, recipient, amount);
        } else {
            _transferStandard(sender, recipient, amount);
        }

        if(!takeFee)
            restoreAllFee();
    }
```

Security advice: None.

6. Code basic vulnerability detection

6.1. Compiler version security【Pass】

Check to see if a secure compiler version is used in the contract code implementation.

Detection results: After detection, the smart contract code has developed a compiler version of 0.6.12 or more, there is no security issue.

Security advice: None.

6.2. Redundant code【Pass】

Check that the contract code implementation contains redundant code.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.3. Use of safe arithmetic library【Pass】

Check to see if the SafeMath security abacus library is used in the contract code implementation.

Detection results: The SafeMath security abacus library has been detected in the smart contract code and there is no such security issue.

Security advice: None.

6.4. Not recommended encoding【Pass】

Check the contract code implementation for officially uns recommended or deprecated coding methods.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.5. Reasonable use of require/assert【Pass】

Check the reasonableness of the use of require and assert statements in contract code implementations.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.6. Fallback function safety【Pass】

Check that the fallback function is used correctly in the contract code implementation.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.7. tx.origin authentication【Pass】

tx.origin is a global variable of Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts makes contracts vulnerable to phishing-like attacks.z

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.8. Owner permission control【Pass】

Check that theowner in the contract code implementation has excessive permissions. For example, modify other account balances at will, and so on.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.9. Gas consumption detection【Pass】

Check that the consumption of gas exceeds the maximum block limit.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.10. call injection attack【Pass】

When a call function is called, strict permission control should be exercised, or the function called by call calls should be written directly to call calls.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.11. Low-level function safety【Pass】

Check the contract code implementation for security vulnerabilities in the use of call/delegatecall

The execution context of the call function is in the contract being called, while the execution context of the delegatecall function is in the contract in which the function is currently called.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.12. Vulnerability of additional token issuance【Pass】

Check to see if there are functions in the token contract that might increase the total token volume after the token total is initialized.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.13. Access control defect detection 【Pass】

Different functions in the contract should set reasonable permissions, check whether the functions in the contract correctly use pubic, private and other keywords for visibility modification, check whether the contract is properly defined and use modifier access restrictions on key functions, to avoid problems caused by overstepping the authority.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.14. Numerical overflow detection 【Pass】

The arithmetic problem in smart contracts is the integer overflow and integer overflow, with Solidity able to handle up to 256 digits (2^256-1), and a maximum number increase of 1 will overflow to get 0. Similarly, when the number is an unsigned type, 0 minus 1 overflows to get the maximum numeric value.

Integer overflows and underflows are not a new type of vulnerability, but they are particularly dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if the likelihood is not anticipated, which can affect the reliability and safety of the program.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.15. Arithmetic accuracy error【Pass】

Solidity has a data structure design similar to that of a normal programming language, such as variables, constants, arrays, functions, structures, and so on, and there is a big difference between Solidity and a normal programming language - Solidity does not have floating-point patterns, and all of Solidity's numerical operations result in integers, without the occurrence of decimals, and without allowing the definition of decimal type data. Numerical operations in contracts are essential, and numerical operations are designed to cause relative errors, such as sibling operations: 5/2 x 10 x 20, and 5 x 10/2 x 25, resulting in errors, which can be greater and more obvious when the data is larger.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.16. Incorrect use of random numbers【Pass】

Random numbers may be required in smart contracts, and while the functions and variables provided by Solidity can access significantly unpredictable values, such as block.number and block.timestamp, they are usually either more public than they seem, or are influenced by miners, i.e. these random numbers are somewhat predictable, so malicious users can often copy it and rely on its unpredictability to attack the feature.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.17. Unsafe interface usage 【Pass】

Check the contract code implementation for unsafe external interfaces, which can be controlled, which can cause the execution environment to be switched and control contract execution arbitrary code.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.18. Variable coverage 【Pass】

Check the contract code implementation for security issues caused by variable overrides.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.19. Uninitialized storage pointer 【Pass】

A special data structure is allowed in solidity as a strut structure, while local variables within the function are stored by default using stage or memory.

The existence of store (memory) and memory (memory) is two different concepts, solidity allows pointers to point to an uninitialized reference, while uninitialized local stage causes variables to point to other stored variables, resulting in variable overrides, and even more serious consequences, and should avoid initializing the task variable in the function during development.

Detection results: After detection, the smart contract code does not have the problem.

Security advice: None.

6.20. Return value call verification【Pass】

This issue occurs mostly in smart contracts related to currency transfers, so it is also known as silent failed sending or unchecked sending.

In Solidity, there are transfer methods such as transfer(), send(), call.value(), which can be used to send tokens to an address, the difference being: transfer send failure will be throw, and state rollback; Call.value returns false when it fails to send, and passing all available gas calls (which can be restricted by incoming gas_value parameters) does not effectively prevent reentration attacks.

If the return values of the send and call.value transfer functions above are not checked in the code, the contract continues to execute the subsequent code, possibly with unexpected results due to token delivery failures.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.21. Transaction order dependency【Pass】

Because miners always get gas fees through code that represents an externally owned address (EOA), users can specify higher fees to trade faster. Since blockchain is public, everyone can see the contents of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transactions at a higher cost to preempt the original solution.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.22. Timestamp dependency attack【Pass】

Block timestamps typically use miners' local time, which can fluctuate over a range of about 900 seconds, and when other nodes accept a new chunk, they only need to verify that the timestamp is later than the previous chunk and has a local time error of less than 900 seconds. A miner can profit from setting the timestamp of a block to meet as much of his condition as possible.

Check the contract code implementation for key timestamp-dependent features.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.23. Denial of service attack【Pass】

Smart contracts that are subject to this type of attack may never return to normal operation. There can be many reasons for smart contract denial of service, including malicious behavior as a transaction receiver, the exhaustion of gas caused by the artificial addition of the gas required for computing functionality, the misuse of access control to access the private component of smart contracts, the exploitation of confusion and negligence, and so on.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.24. Fake recharge vulnerability【Pass】

The transfer function of the token contract checks the balance of the transfer initiator (msg.sender) in the if way, when the balances < value enters the else logic part and return false, and ultimately does not throw an exception, we think that only if/else is a gentle way of judging in a sensitive function scenario such as transfer is a less rigorous way of coding.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.25. Reentry attack detection 【Pass】

The call.value() function in Solidity consumes all the gas it receives when it is used to send tokens, and there is a risk of re-entry attacks when the call to the call tokens occurs before the balance of the sender's account is actually reduced.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.26. Replay attack detection 【Pass】

If the requirements of delegate management are involved in the contract, attention should be paid to the non-reusability of validation to avoid replay attacks

In the asset management system, there are often cases of entrustment management, the principal will be the assets to the trustee management, the principal to pay a certain fee to the trustee. This business scenario is also common in smart contracts.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.27. Rearrangement attack detection 【Pass】

A reflow attack is an attempt by a miner or other party to "compete" with a smart contract participant by inserting their information into a list or mapping, giving an attacker the opportunity to store their information in a contract.

Detection results: After detection, there are no related vulnerabilities in the smart contract code.

Security advice: None.

7. Appendix A: Security Assessment of Contract Fund Management

Contract fund management		
The type of asset in the contract	**The function is involved**	**Security risks**
User transfers token assets	transfer、swapAndMarketing	**SAFE**

Check the security of the management of **digital currency assets** transferred by users in the business logic of the contract. Observe whether there are security risks that may cause the loss of customer funds, such as **incorrect recording, incorrect transfer, and backdoor** withdrawal of the **digital currency assets** transferred into the contract.

KNOWNSEC
Blockchain Lab

Official Website
www.knownseclab.com

E-mail
blockchain@knownsec.com

WeChat Official Account