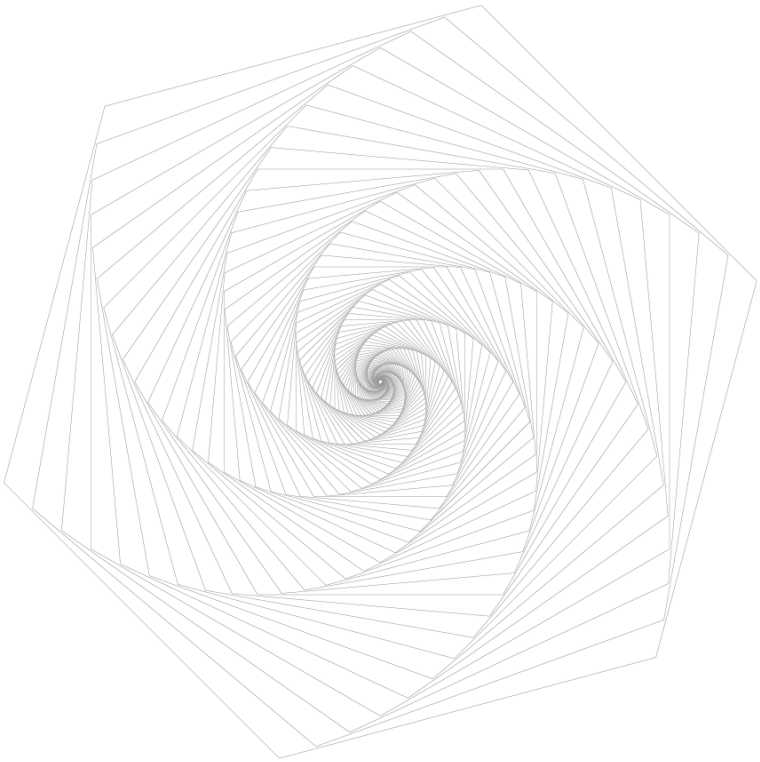
KNOWNSEC
Blockchain Lab

Smart Contract Audit Report



Version description

The revision	Date	Revised	Version
Write documentation	20220125	KNOWNSEC Blockchain Lab	V1.0

Document information

Title	Version	Document Number	Type
Blade Smart Contract Audit Report	V1.0	188a1d55d39f41da91e1478582b744f3	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 **January 20, 2022 to January 25, 2022.** During this period, the security and standardization of **the token code of the Blade smart contract** 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 Blade** 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	188a1d55d39f41da91e1478582b744f3
report query link	https://attest.im/attestation/searchResult?qurey=188a1d55d39f41da91e1478582b744f3

2. Item information

2.1. Item description

Blade Warrior is an immersive, fantasy, NFT blockchain game that makes money from games.

2.2. The project's website

https://www.blade.game/

2.3. White Paper

https://docs.blade.game/

2.4. Review version code

farm.zip

2.5. Contract file and Hash/contract deployment address

The contract documents	MD5
Ownable.sol	A9A5F037FF6EB170C61159489A2E59A5
SafeMath.sol	9D9111B542083B32ED6A981CD25587B9
IERC20.sol	A2B7C205E38A51FD414D256C8C294FEE
SafeERC20.sol	45646FB3F276A48B747A11C86E45EEDC
Context.sol	B184DAA83F6127830D8E8D657E6152D5
EnumerableSet.sol	CD0E1995CA83B2F2B80884427338499E
ReentrancyGuard.sol	10FA3498FD03560DDDB3D70A7CA7DDA4
IPriceOracle.sol	5549F4B8301CD9E579C9B8DB130A4665

MasterGardener.sol	B584C594A81BC54B4D30868F0F127024
PriceOracle.sol	047A112DAFEEF41C3FEB294D59D7369E
WithAdminRole.sol	1CAD70C6D88698874E9ED0547F0A0C5F
WithPriceOracle.sol	662FA6C993DBB18A6550AF08F2625E87

3. External visibility analysis

3.1. MasterGardener contracts

MasterGardener					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
initialize	public	True	initializer	---	---
currentDayth	public	False	---	---	---
dayth	public	False	---	---	---
currentWeekth	public	False	---	---	---
currentLockPercentage	public	False	---	---	---
weekth	public	False	---	---	---
poolLength	external	False	---	---	---
add	public	True	restrictednonDuplicated(_lpToken)	---	---
fixTotalAllocPoint	public	True	restricted	---	---
set	public	True	restricted	---	---
addRewardToken	public	False	---	---	---
getRewardTokenInfo	public	True	---	---	---
updatePool	public	True	---	---	---
getMultiplier	public	False	---	---	---
pendingReward	public	True	---	---	---

claimReward	public	True	onlyNonContract nonReentrant	---	---
_harvest	internal	True	---	---	---
_sendReward	internal	True	---	---	---
unLockReward	public	True	---	---	---
getLockReward	public	False	---	---	---
getPoolInfo	public	False	---	---	---
lpLimit	public	False	---	---	---
getAprValues	public	True	restricted	---	---
getLockday	public	True	restricted	---	---
getLocdWithdraw	public	False	---	---	---
getBlockNum	public	False	---	---	---
pendingCherry	public	False	---	---	---
claimCherryReward	public	True	restricted	---	---
init	public	True	notInitialized	---	---
deposit	public	False	---	---	---
withdrawFor	public	True	restricted	---	---
setMutiply	public	False	---	---	---

4. Code vulnerability analysis

4.1. Summary description of the audit results

Audit results			
audit project	**audit content**	**condition**	**description**
Business security detection	MasterGardener retracement function	Pass	After testing, there is no security issue.
	MasterGardener pledge interest earning function	Pass	After testing, there is no security issue.
	MasterGardener contract permission control function	Pass	After testing, there is no security issue.
	MasterGardener contract initialization function	Reminder	After testing, there is no security problem, and only prompts are given.
	PriceOracle Contract Price Prediction Function	Reminder	After testing, there is no security problem, and only prompts are given.
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. MasterGardener retracement function【Pass】

Audit analysis: perform security audit on the withdrawal function in the MasterGardener contract; withdraw implements security measures such as anti-reentrancy and safemath in the specified pid pool through the LP withdrawal of funds and the reward function. After the audit, the logic design is reasonable, and no security problems were found.

```
    function withdraw(uint256 _pid, uint256 _amount)
    public
    nonReentrant
    onlyNonContract
{// knownsec //The amount specified by the specified pool can be withdrawn. The non-contract account can be executed.
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];

    require(user.amount >= _amount, "over deposited");
    uint256 day = _getDays();
    if (isLock) {// knownsec // lock check
     require(day > lockDays, "lp_is_locked");
    }

    updatePool(_pid);// knownsec // update the pool
    _harvest(_pid);// knownsec // receive rewards

    if (_amount > 0) {
      LpInterface(address(liquidityaddr)).withdraw(pool.pid, _amount);// knownsec // liquidityaddr retrace
      uint256 _userWithdrawAmount = _amount;
```

```
        if (isWithdraw) {
            if (day <= lockWithdrawDays) {
                _userWithdrawAmount = _amount.mul(withdrawPercent).div(100);
            }
        }
        IERC20(pool.lpToken).safeTransfer(
            address(msg.sender),// knownsec// transfer
            _userWithdrawAmount
        );
        lpSupply = lpSupply.sub(_amount); //
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accGovTokenPerShare);
        emit Withdraw(msg.sender, _pid, _amount);
    }
}
```

Security advice: None.

5.2. MasterGardener pledge interest earning function【Pass】

Audit analysis: Conduct a security audit on the deposit staking and interest-generating function in the MasterGardener contract. The function of this function is to allow users to pledge their lpTokens and generate rewards. After audit, the logic design of this function is reasonable, and no security problems were found.

```
    function deposit(uint256 _pid, uint256 _amount)
    public
    nonReentrant
    onlyNonContract
{// knownsec // Pledge designated pool lp and earn interest EGG
    require(
        _amount > 0,
```

```
            "CND 0"
        );

        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount + _amount <= lpLimit(), "out of 200 limit");// knownsec // lp limit
200 ether

        // When a user deposits, we need to update the pool and harvest beforehand,
        // since the rates will change.
        updatePool(_pid);// knownsec // update the pool
        _harvest(_pid);// knownsec // receive rewards

        IERC20(pool.lpToken).safeTransferFrom(// knownsec // transfer
            address(msg.sender),
            address(this),
            _amount
        );

        bool isSuccess = pool.lpToken.approve(liquidityaddr, _amount);// knownsec // Approved
liquidity pool
        require(isSuccess, "approve failed");// knownsec// require examination
        LpInterface(address(liquidityaddr)).deposit(pool.pid,  _amount);// knownsec // pledge
liquidityaddr

        if (isLock) {
            IERC20(certificateaddr).safeTransfer(msg.sender, certificateNums);
        }
        lpSupply = lpSupply.add(_amount); //
        _depositeBlockNumber[msg.sender] = block.number;
        if (user.amount == 0) {
            user.rewardDebtAtBlock = block.number;
        }
```

```
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accGovTokenPerShare);// Reward Update
        emit Deposit(msg.sender, _pid, _amount);
    }
```

Security advice: None.

5.3. MasterGardener contract permission control function 【Pass】

Audit analysis: Audit the function call permissions in the MasterGardener contract. The modification of various important functions such as state variables is controlled by the restricted modifier. After auditing, this decorator is implemented to check if msg.sender is GAME_ADMIN, there is no security issue.

```
    function addRewardToken(string memory name, address rewardToken)
    external
    // uint256 price
    restricted// knownsec // Added reward token modifier restricted
{
    require(rewardToken != address(0), "token is null");

    uint256 len = rewardTokenInfos.length;// knownsec // list length
    for (uint256 i = 0; i < len; i++) {// knownsec // High gas consumption, other check methods can be used
        require(
            rewardToken != rewardTokenInfos[i].rewardToken,// knownsec
            "already add"
        );
    }

    RewardTokenInfo storage rt = rewardTokenInfos.push();// knownsec // renew
```

```
        rt.name = name;
        rt.rewardToken = rewardToken;
        // rt.price = price;
        emit NewRewardToken(len, rewardToken);
    }
...
        modifier restricted() {
            require(hasRole(GAME_ADMIN, msg.sender), "NGA");
            _;
        }
```

Security advice: None.

5.4. MasterGardener contract initialization function 【Reminder】

Audit analysis: Security audit of the initialization function in the MasterGardener contract. After auditing, there is no permission control for the function, so it may be preempted and initialized by someone during deployment. But this question is not a security question, so it is only a hint.

```
        function init(
            address _liquidityaddr,
            uint256 _rewardPerBlock,
            uint256 _startBlock,
            uint256[] memory _rewardMultiplier
        ) public notInitialized {// knownsec // Initialization  Can only be initialized once Uncontrolled permissions
            // super.initOwner();
            _INITIALIZED_ = true;
            liquidityaddr = _liquidityaddr;
            REWARD_PER_BLOCK = _rewardPerBlock;
```

```
    START_BLOCK = _startBlock;
    REWARD_MULTIPLIER = _rewardMultiplier;
    for (uint256 i = 0; i < REWARD_MULTIPLIER.length - 1; i++) {
        uint256 halvingAtBlock = _halvingAfterBlock
            .mul(i + 1)
            .add(_startBlock)
            .add(1);
        HALVING_AT_BLOCK.push(halvingAtBlock);
    }
    FINISH_BONUS_AT_BLOCK = _halvingAfterBlock
        .mul(REWARD_MULTIPLIER.length - 1)
        .add(_startBlock);
    HALVING_AT_BLOCK.push(uint256(-1));
    //    init PERCENT_LOCK_BONUS_REWARD
    for (uint256 i = 0; i < REWARD_MULTIPLIER.length - 1; i++) {
        uint256 unlockPercentage = initPercent.add(incrPercent.mul(i));
        uint256 lockPercentage = unlockPercentage >= 100
            ? 0
            : 100 - unlockPercentage;
        PERCENT_LOCK_BONUS_REWARD.push(lockPercentage);
    }
    PERCENT_LOCK_BONUS_REWARD.push(0);
}
```

Security advice: None.

5.5. PriceOracle Contract Price Prediction Function 【Reminder】

Audit Analysis: Conduct a security audit of the price oracle function in the PriceOracle contract. The oracle machine judges the price of the balance through an unsafe transaction in the acquisition pool, which is an unsafe price prediction. After

communicating with the project party, it is only used to obtain the price expression of apr. After the audit, only getAprValues is called in the code, and there is no security problem at present, so this is only a reminder, and subsequent development should pay attention to the usage scenarios of the oracle.

```
        function usdtReserved() public view override returns (uint256 usdtAmount) {
            usdtAmount = _usdtContract.balanceOf(_cheLpContract);// knownsec // cheLp's
USDT
        }

        function bladeReserved() public view override returns (uint256 blade) {
            blade = _bladeContract.balanceOf(_cheLpContract);// knownsec // chelp's blade
        }

        //  xxxx usdt gwei/blade
        function usdtBladePrice() public view override returns (uint256 price) {
            price = (usdtReserved() * 1000000000) / bladeReserved();
        }

        // xxxx  blade gwei/usdt
        function bladeUsdtPrice() public view override returns (uint256 price) {
            price = (bladeReserved() * 1000000000) / usdtReserved();
        }
```

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 0.6.0 , 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 mortgage token assets	deposit	**SAFE**
Users mortgage platform currency assets	---	**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