KNOWNSEC

Degree of secrecy ★ | Project team open

Smart Contract Audit Report

Security status

Safe

★ ★ ★ ★ ★

Principal tester: Knownsec blockchain security team

Version Summary

Content	Date	Version
Editing Document	2021/08/09	V1.0

Report Information

Title	Version	Document Number	Type
KFI Smart Contract Audit Report	V1.0	ce18e4a884924c7ca7681716195 7f3df	Open to project team

Copyright Notice

Table of Contents

1. Introduction

The effective test time of this report is from Since May 18, 2021 to May 31, 2021 . During this period, the security and standardization of **the smart contract code of the KFI** 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 types of attacks 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 3). **The smart contract code of the KFI** is comprehensively assessed as **SAFE**.

Results of this smart contract security audit： SAFE

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.

Report information of this audit:

Report Number： ce18e4a884924c7ca76817161957f3df

Report query address link：

https://attest.im/attestation/searchResult?qurey=ce18e4a884924c7ca76817161957f3df

Target information of the KFI audit:

Target information		
Project name	KFI	
Contract Address	KFIFarm	0xab2ae88D8698ddD65cE09Dd73f2ffe9481837242

Code type	Token code, OKExChain smart contract code
Code language	Solidity

Contract documents and hash:

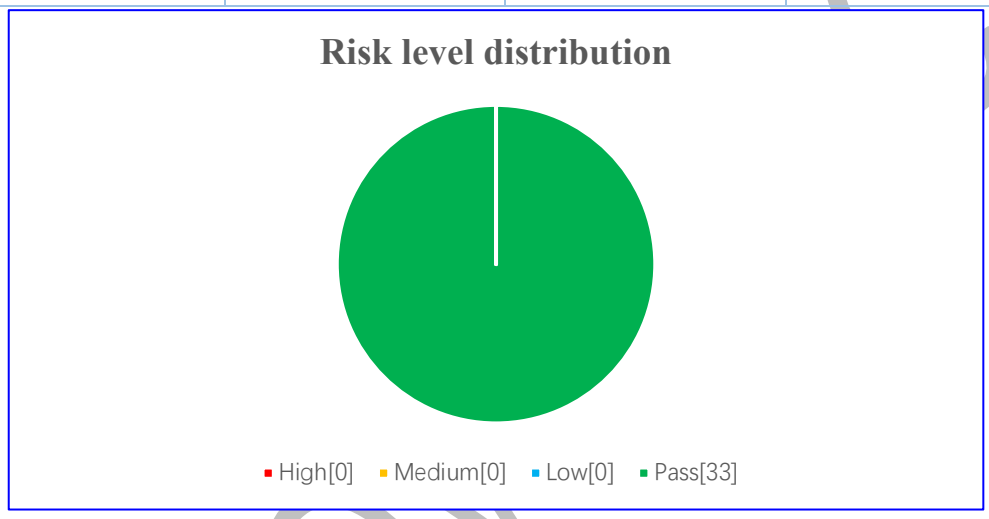
Contract documents	MD5
KFI.sol	A68BBC170431AD224B9E9667D7D2367A
KFIFarm.sol	F31B433A9C39CBFD329FA1039A7D411D
Strategy_KSwap_Core.sol	32D43671E969B844B63717250ED750E5

2. Code vulnerability analysis

2.1 Vulnerability Level Distribution

Vulnerability risk statistics by level：

Vulnerability risk level statistics table			
High	Medium	Low	Pass
0	0	0	33



2.2 Audit Result

Result of audit			
Audit Target	**Audit**	**Status**	**Audit Description**
Business security testing	**KFI token minting function**	Pass	After testing, there is no such safety vulnerability.
	KFIFarm contract mortgage mining function	Pass	After testing, there is no such safety vulnerability.
	KFIFarm contract withdrawal reward function	Pass	After testing, there is no such safety vulnerability.
	Strategy_KSwap_Core contract harvest reward re-investment function	Pass	After testing, there is no such safety vulnerability.
	Strategy_KSwap_Core contract repurchase function	Pass	After testing, there is no such safety vulnerability.
	Strategy_KSwap_Core contract parameter setting function	Pass	After testing, there is no such safety vulnerability.
Basic code vulnerability detection	**Compiler version security**	Pass	After testing, there is no such safety vulnerability.
	Redundant code	Pass	After testing, there is no such safety vulnerability.
	Use of safe arithmetic library	Pass	After testing, there is no such safety vulnerability.
	Not recommended encoding	Pass	After testing, there is no such safety vulnerability.

	Reasonable use of require/assert	Pass	After testing, there is no such safety vulnerability.
	fallback function safety	Pass	After testing, there is no such safety vulnerability.
	tx.origin authentication	Pass	After testing, there is no such safety vulnerability.
	Owner permission control	Pass	After testing, there is no such safety vulnerability.
	Gas consumption detection	Pass	After testing, there is no such safety vulnerability.
	call injection attack	Pass	After testing, there is no such safety vulnerability.
	Low-level function safety	Pass	After testing, there is no such safety vulnerability.
	Vulnerability of additional token issuance	Pass	After testing, there is no such safety vulnerability.
	Access control defect detection	Pass	After testing, there is no such safety vulnerability.
	Numerical overflow detection	Pass	After testing, there is no such safety vulnerability.
	Arithmetic accuracy error	Pass	After testing, there is no such safety vulnerability.
	Wrong use of random number detection	Pass	After testing, there is no such safety vulnerability.
	Unsafe interface use	Pass	After testing, there is no such safety vulnerability.

	Variable coverage	Pass	After testing, there is no such safety vulnerability.
	Uninitialized storage pointer	Pass	After testing, there is no such safety vulnerability.
	Return value call verification	Pass	After testing, there is no such safety vulnerability.
	Transaction order dependency detection	Pass	After testing, there is no such safety vulnerability.
	Timestamp dependent attack	Pass	After testing, there is no such safety vulnerability.
	Denial of service attack detection	Pass	After testing, there is no such safety vulnerability.
	Fake recharge vulnerability detection	Pass	After testing, there is no such safety vulnerability.
	Reentry attack detection	Pass	After testing, there is no such safety vulnerability.
	Replay attack detection	Pass	After testing, there is no such safety vulnerability.
	Rearrangement attack detection	Pass	After testing, there is no such safety vulnerability.

3. Business security testing

3.1. KFI token minting function【PASS】

Audit analysis: The KFI contract implements KFI tokens based on the ERC20 protocol. The initial total amount of tokens is 0, and the minting limit is 100 million. The minting authority is controlled by the owner's authority, and the token accuracy is 18. After testing, the token can be used normally.

```
contract KFI is ERC20("KFI", "KFI"), Ownable {
        using SafeMath for uint256;
        uint256 maxSupply = 100000000e18;
        function mint(address _to, uint256 _amount) public onlyOwner {
            require(_amount.add(this.totalSupply())<=maxSupply,"maxSupply limit");
//knownsec// The total amount of coins required to not exceed the upper limit of coins
            _mint(_to, _amount); //knownsec// Call internal functions to mint coins
        }
    }
```

Recommendation： nothing.

3.2. KFIFarm contract mortgage mining function【PASS】

Audit analysis: The deposit function of the KFIFarm contract realizes the function of users collateralizing tokens for mining to obtain KFI tokens.

```
    function deposit(uint256 _pid, uint256 _wantAmt) public nonReentrant {
            updatePool(_pid); //knownsec// Update the pool

            PoolInfo storage pool = poolInfo[_pid];
            UserInfo storage user = userInfo[_pid][msg.sender];
            if (user.shares > 0) { //knownsec// If the user has a mortgage share
```

```
                uint256 pending = user.shares.mul(pool.accKfiPerShare).div(1e12).sub(user.rewardDebt); //knownsec// Calculate user's pending rewards
                if (pending > 0) {
                    safeKfiTransfer(msg.sender, pending); //knownsec// Transfer KFI tokens to users
                }
            }
            if (_wantAmt > 0) {
                pool.stakeToken.safeTransferFrom( //knownsec// Mortgage tokens into this contract
                    address(msg.sender),
                    address(this),
                    _wantAmt
                );
                pool.stakeToken.safeIncreaseAllowance(pool.strategy, _wantAmt); //knownsec// Add authorization to the strategic contract
                uint256 sharesAdded = IStrategy(poolInfo[_pid].strategy).deposit(//knownsec// Take the tokens pledged by the user to mortgage, and return the share corresponding to the mortgage amount
                    msg.sender,
                    _wantAmt
                );
                user.shares = user.shares.add(sharesAdded);
```

```
        }

        user.rewardDebt = user.shares.mul(pool.accKfiPerShare).div(1e12); //knownsec//
Update user's mortgage debt

        emit Deposit(msg.sender, _pid, _wantAmt); //knownsec// Trigger mortgage event

    }
```

Recommendation： nothing.

3.3. KFIFarm contract withdrawing reward function【PASS】

Audit analysis: The withdraw and withdrawReward functions of the KFIFarm contract implement the functions of users withdrawing mortgage tokens (partial handling fees will be charged) and reward tokens KFI.

```
function withdrawReward(uint256 _pid) public nonReentrant { //knownsec// Withdraw only kfi
token rewards

        updatePool(_pid); //knownsec// Update pool

        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        uint256 totalStaked = IStrategy(poolInfo[_pid].strategy).totalStaked();//knownsec//
Get the number of collaterals in the strategy contract
        uint256 totalShared = IStrategy(poolInfo[_pid].strategy).totalShared();//knownsec//
Obtain the total mortgage share of the strategy contract

        require(user.shares > 0, "user.shares is 0");
        require(totalShared > 0, "sharesTotal is 0");

        // Withdraw pending KFI
        uint256                                pending                                =
```

```
user.shares.mul(pool.accKfiPerShare).div(1e12).sub(user.rewardDebt);
            if (pending > 0) {
                safeKfiTransfer(msg.sender, pending); //knownsec// Send user's KFI token reward
            }
            user.rewardDebt = user.shares.mul(pool.accKfiPerShare).div(1e12);
            emit TakeKfiReward(msg.sender, _pid, pending);
        }


function withdraw(uint256 _pid, uint256 _wantAmt) public nonReentrant { //knownsec// Withdraw rewards and mortgage tokens

            updatePool(_pid); //knownsec// Update mortgage pool

            PoolInfo storage pool = poolInfo[_pid];
            UserInfo storage user = userInfo[_pid][msg.sender];

            uint256 totalStaked = IStrategy(poolInfo[_pid].strategy).totalStaked();
            uint256 totalShared = IStrategy(poolInfo[_pid].strategy).totalShared();

            require(user.shares > 0, "user.shares is 0");
            require(totalShared > 0, "sharesTotal is 0");

            // Withdraw pending KFI
            uint256 pending =
user.shares.mul(pool.accKfiPerShare).div(1e12).sub(user.rewardDebt); //knownsec// Calculate pending rewards
            if (pending > 0) {
                safeKfiTransfer(msg.sender, pending); //knownsec// Send reward token KFI
            }

            // Withdraw want tokens
            uint256 amount = user.shares.mul(totalStaked).div(totalShared); //knownsec//
```

```
Calculate the strategy contract reward corresponding to the number of user mortgages
        if (_wantAmt > amount) {
            _wantAmt = amount;
        }
        if (_wantAmt > 0) {
            uint256 sharesRemoved = IStrategy(poolInfo[_pid].strategy).withdraw(msg.sender, _wantAmt); //knownsec// Call the strategy contract to withdraw mortgage tokens
            if (sharesRemoved > user.shares) {
                user.shares = 0;
            } else {
                user.shares = user.shares.sub(sharesRemoved);
            }
            uint256 stakeBalance = IERC20(pool.stakeToken).balanceOf(address(this)); //knownsec// Get the balance of mortgage tokens in this contract
            if (stakeBalance <= _wantAmt) {
                _wantAmt = stakeBalance;
            }
            pool.stakeToken.safeTransfer(address(msg.sender), _wantAmt); //knownsec// Get the balance of mortgage tokens in this contract }
        user.rewardDebt = user.shares.mul(pool.accKfiPerShare).div(1e12); //knownsec// Update user mortgage debt
        emit Withdraw(msg.sender, _pid, _wantAmt); //knownsec// Trigger user extraction event
    }
```

Recommendation： nothing.

3.4. Strategy_KSwap_Core contract harvest reward re-investment function【PASS】

Audit analysis: The earn function of the Strategy_KSwap_Core contract realizes the function that the strategy contract takes the tokens pledged by the user to

other mining contracts for mortgage mining, and exchanges the rewards for swap into mortgage tokens to continue the mortgage mining.

```
    function earn() public virtual whenNotPaused {
        if (onlyGov) {    //knownsec//    Whether it can only manage the address call, can be changed, the default can be
            require(msg.sender == govAddress, "!gov");
        }

        _unfarm(0); //knownsec// Harvest reward tokens into this contract

        if (earnedAddress == woktAddress) {
            _wrapOKT();//knownsec// Replace OKT with WOKT
        }

        uint256 earnedAmt = IERC20(earnedAddress).balanceOf(address(this)); //knownsec// Get the balance of tokens harvested by this contract
        require(earnedAmt > 0, "earnedAmt is 0");
        uint256 stakeP1 = IPrice(priceAddr).coinPrice(earnedAddress); //knownsec// Calculate the reward token price
        uint256 earnUsd = earnedAmt.mul(stakeP1).div(1e18); //knownsec// Calculate the number of U rewards

        require(earnUsd > earnLimit, "earnUsdValueToo little income"); //knownsec// Require the harvest quantity to be greater than the minimum harvest quantity

        lastEarn = lastEarn.add(earnedAmt); //knownsec// Update the last harvested quantity
        earnedAmt = distributeFees(earnedAmt); //knownsec// Collect management fees and return the remaining harvest amount
        earnedAmt = buyBack(earnedAmt); //knownsec// Make a repurchase

        IERC20(earnedAddress).safeApprove(uniRouterAddress, 0);
        IERC20(earnedAddress).safeIncreaseAllowance(//knownsec// Then re-authorize the
```

```
remaining harvest amount to the Router
                    uniRouterAddress,
                    earnedAmt
                );
                if (isSingleToken) { //knownsec// Determine whether the mining pool is a single currency Defined in the constructor
                    if (earnedAddress != stakeToken) {
                        _safeSwap(
                            uniRouterAddress,
                            earnedAmt,
                            slippageFactor, //knownsec// 5% slippage
                            singleEarnToStakePath, //knownsec// Exchange it to mortgage tokens to this contract
                            address(this),
                            block.timestamp.add(600)
                        );
                    }
                } else {
                    if (earnedAddress != token0Address) {
                        _safeSwap(
                            uniRouterAddress,
                            earnedAmt.div(2), //knownsec// Half of the harvest
                            slippageFactor, //knownsec// 5% slippage
                            earnedToToken0Path, //knownsec// Change token0 to this contract
                            address(this), //knownsec// Send to this contract address after redemption
                            block.timestamp.add(600)
                        );
                    }

                    if (earnedAddress != token1Address) {
                        _safeSwap(
                            uniRouterAddress,
```

```
                earnedAmt.div(2),
                slippageFactor,
                earnedToToken1Path,
                address(this),
                block.timestamp.add(600)
            );
        }

        //knownsec// Add liquidity after exchange
        uint256 token0Amt = IERC20(token0Address).balanceOf(address(this));
//knownsec// Obtain two types of added liquidity token balances
        uint256 token1Amt = IERC20(token1Address).balanceOf(address(this));
        if (token0Amt > 0 && token1Amt > 0) {
            IERC20(token0Address).safeIncreaseAllowance(
                uniRouterAddress,
                token0Amt
            );
            IERC20(token1Address).safeIncreaseAllowance(
                uniRouterAddress,
                token1Amt
            );

            ISwapRouter(uniRouterAddress).addLiquidity(    //knownsec//    Add
liquidity after swap
                token0Address,
                token1Address,
                token0Amt,
                token1Amt,
                0,
                0,
                address(this),
                block.timestamp.add(600)
            );
```

```
                }
            }


            if (block.number - lastEarnBlock > 20) {    //knownsec//    If the block is greater than
20
                uint256        p1        =        IPrice(priceAddr).coinPrice(earnedAddress);
//knownsec// Get the price of reward tokens
                uint256 stakePrice = stakePrice(); //knownsec// Get the mortgage token price
                uint256    diffBlock    =    block.number.sub(lastEarnBlock);    //knownsec//
Calculate the blocks passed
                hourEarnUsd                                                                =
lastEarn.div(diffBlock).mul(hourBlockNum).mul(p1).div(1e18);  //knownsec// Number of tokens
harvested/blocks passed*number of blocks per hour*price of tokens harvested
                if(ERC20(earnedAddress).decimals() == 10){
                    hourEarnUsd = hourEarnUsd.mul(1e8); //knownsec// Update accuracy
                }
                stakeTokenUsd = totalStaked.mul(stakePrice).div(1e18); //knownsec// Amount
spent on total mortgage
                if(ERC20(stakeToken).decimals()==10){
                    stakeTokenUsd = totalStaked.mul(stakePrice).mul(1e8).div(1e18);
                }
                hourRate = hourEarnUsd.mul(1e18).div(stakeTokenUsd); //knownsec// Hourly
interest rate = hourly block generation value/total mortgage cost value
                lastEarn = 0;
                lastEarnBlock = block.number;
            }
            _farm(); //knownsec// Take the lp or mortgage tokens generated by increasing
liquidity to the mining pool for mortgage mining
        }
```

Recommendation： nothing.

3.5. Strategy_KSwap_Core contract repurchase function 【PASS】

Audit analysis: The buyBack function of the Strategy_KSwap_Core contract enables the project party to exchange part of the reward tokens (controlled by the owner) into KFI tokens for repurchase.

```
function buyBack(uint256 _earnedAmt) internal virtual returns (uint256) {//knownsec// Buy back KFI tokens

        if (buyBackRate <= 0) {

            return _earnedAmt;

        }

        uint256 buyBackAmt = _earnedAmt.mul(buyBackRate).div(buyBackRateMax); //knownsec// Calculate the repurchase quantity

        if (earnedAddress == kfiAddress) {

            IERC20(earnedAddress).safeTransfer(buyBackAddress, buyBackAmt); //knownsec// Transfer the repurchase amount to the repurchase address

        } else {

            IERC20(earnedAddress).safeIncreaseAllowance( //knownsec// Authorize Router

                uniRouterAddress,

                buyBackAmt

            );

            if(buyUsdt){

                _safeSwap(

                    uniRouterAddress,
```

```
                    buyBackAmt,
                    slippageFactor, //knownsec// Allow 5% slippage
                    earnedToUsdtPath, //knownsec// Change to USDT
                    buyBackAddress,
                    block.timestamp.add(600)
                );
            }else{
                _safeSwap(
                    uniRouterAddress,
                    buyBackAmt,
                    slippageFactor, //knownsec// Allow 5% slippage
                    earnedToKfiPath, //knownsec// Change to KFI
                    buyBackAddress, //knownsec// Send the exchanged tokens to the repurchase address
                    block.timestamp.add(600)
                );
            }
        }
        return _earnedAmt.sub(buyBackAmt); //knownsec// Return reward tokens-the number of tokens repurchased
    }
```

Recommendation： nothing.

3.6. Strategy_KSwap_Core contract parameter setting function【PASS】

Audit analysis: The onlyAllowGov permission of Strategy_KSwap_Core contract has the function of changing mining parameters at any time, including mining address, mining handling fee, repurchase address, swap exchange path and many other modification permissions.

Remind: Changes in parameters may result in changes in mining rewards.

```
function setHourBlockNum(uint256 _hourBlockNum) public virtual onlyAllowGov {
        hourBlockNum = _hourBlockNum;
        emit SetHourBlockNum(_hourBlockNum);
    }//knownsec// Set the number of blocks per hour

    function setStakeToken(address _stakeToken) public virtual onlyAllowGov {
        stakeToken = _stakeToken;
        emit UpdateAddress(_stakeToken);
    }//knownsec// Set mortgage token address

    function setToken0(address _token0) public virtual onlyAllowGov {
        token0Address = _token0;
        emit UpdateAddress(_token0);
    }
    function setToken1(address _token1) public virtual onlyAllowGov {
        token1Address = _token1;
```

```
        emit UpdateAddress(_token1);
    }

    function setRewardsAddress(address _rewardsAddress) public virtual onlyAllowGov {
        rewardsAddress = _rewardsAddress;
        emit UpdateAddress(_rewardsAddress);
    }//knownsec// Set up the handling fee collection address

    function setBuyBackAddress(address _buyBackAddress) public virtual onlyAllowGov {
        buyBackAddress = _buyBackAddress;
        emit UpdateAddress(_buyBackAddress);
    }//knownsec// Set up the repurchase KFI token address

    function setStrategyFarmAddr(address _strategyFarmAddr) public virtual
onlyAllowGov {
        strategyFarmAddr = _strategyFarmAddr;
        emit UpdateAddress(_strategyFarmAddr);
    }//knownsec// Set strategy contract mining address

    function setUniRouterAddress(address _uniRouterAddress) public virtual onlyAllowGov
{
        uniRouterAddress = _uniRouterAddress;
        emit UpdateAddress(_uniRouterAddress);
```

```
    }//knownsec// Set the router contract address of uniswap

    function setEarnedAddress(address _earnedAddress) public virtual onlyAllowGov {

        earnedAddress = _earnedAddress;

        emit UpdateAddress(_earnedAddress);

    }//knownsec// Set the reward token address

    function setKfiAddress(address _kfiAddress) public virtual onlyAllowGov {

        kfiAddress = _kfiAddress;

        emit UpdateAddress(_kfiAddress);

    }//knownsec// Set up KFI token address


    function setPriceAddrAddress(address _priceAddr) public virtual onlyAllowGov {

        priceAddr = _priceAddr;

        emit UpdateAddress(_priceAddr);

    }

    function setGov(address _govAddress) public virtual onlyAllowGov {

        govAddress = _govAddress;

        emit UpdateAddress(_govAddress);

    }


    function setEarnedToToken0Path(address[] memory _address) public virtual
onlyAllowGov {

        earnedToToken0Path = _address;
```

```
        emit UpdatePath(_address);

    }//knownsec// Set the token exchange path array

    function setEarnedToToken1Path(address[] memory _address) public virtual onlyAllowGov {

        earnedToToken1Path = _address;

        emit UpdatePath(_address);

    }

    function setEarnedToUsdtPath(address[] memory _address) public virtual onlyAllowGov {

        earnedToUsdtPath = _address;

        emit UpdatePath(_address);

    }

    function setEarnedToKfiPath(address[] memory _address) public virtual onlyAllowGov {

        earnedToKfiPath = _address;

        emit UpdatePath(_address);

    }

    function setSingleEarnToStakePath(address[] memory _address) public virtual onlyAllowGov {

        singleEarnToStakePath = _address;

        emit UpdatePath(_address);

    }
```

```
    function setOnlyGov(bool _onlyGov) public virtual onlyAllowGov {
        onlyGov = _onlyGov;
        emit SetOnlyGov(_onlyGov);
    }//knownsec// Set the administrator address

    function setBuyUsdt(bool _buyUsdt) public virtual onlyAllowGov {
        buyUsdt = _buyUsdt;
        emit SetBuyUsdt(_buyUsdt);
    }
```

Recommendation： nothing.

4. Basic code vulnerability detection

4.1. Compiler version security【PASS】

Check whether a safe compiler version is used in the contract code implementation.

Audit result: After testing, the smart contract code has formulated the compiler version 0.6.0 within the major version, and there is no such security problem.

Recommendation： nothing.

4.2. Redundant code【PASS】

Check whether the contract code implementation contains redundant code.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.3. Use of safe arithmetic library【PASS】

Check whether the SafeMath safe arithmetic library is used in the contract code implementation.

Audit result: After testing, the SafeMath safe arithmetic library has been used in the smart contract code, and there is no such security problem.

Recommendation： nothing.

4.4. Not recommended encoding【PASS】

Check whether there is an encoding method that is not officially recommended or abandoned in the contract code implementation

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.5. Reasonable use of require/assert【PASS】

Check the rationality of the use of require and assert statements in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.6. Fallback function safety【PASS】

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

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.7. 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 a smart contract makes the contract vulnerable to attacks like phishing.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.8. Owner permission control【PASS】

Check whether the owner in the contract code implementation has excessive authority. For example, arbitrarily modify other account balances, etc.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.9. Gas consumption detection【PASS】

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

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.10. call injection attack【PASS】

When the call function is called, strict permission control should be done, or the function called by the call should be written dead.

Audit result: After testing, the smart contract does not use the call function, and this vulnerability does not exist.

Recommendation： nothing.

4.11. Low-level function safety【PASS】

Check whether there are security vulnerabilities in the use of low-level functions (call/delegatecall) in the contract code implementation

The execution context of the call function is in the called contract; the execution context of the delegatecall function is in the contract that currently calls the function.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.12. Vulnerability of additional token issuance【PASS】

Check whether there is a function that may increase the total amount of tokens in the token contract after initializing the total amount of tokens.

Audit result: After testing, the smart contract code has the function of issuing additional tokens, but it is approved due to the need for additional token issuance due to business needs.

Recommendation： nothing.

4.13. Access control defect detection【PASS】

Different functions in the contract should set reasonable permissions.

Check whether each function in the contract correctly uses keywords such as public and private for visibility modification, check whether the contract is correctly defined and use modifier to restrict access to key functions to avoid problems caused by unauthorized access.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.14. Numerical overflow detection【PASS】

The arithmetic problems in smart contracts refer to integer overflow and integer underflow.

Solidity can handle up to 256-bit numbers (2^256-1). If the maximum number increases by 1, it will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum digital value.

Integer overflow and underflow are not a new type of vulnerability, but they are especially dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if the possibility is not expected, which may affect the reliability and safety of the program.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.15. Arithmetic accuracy error【PASS】

As a programming language, Solidity has data structure design similar to ordinary programming languages, such as variables, constants, functions, arrays, functions, structures, etc. There is also a big difference between Solidity and ordinary programming languages-Solidity does not float Point type, and all the numerical calculation results of Solidity will only be integers, there will be no decimals, and it is not allowed to define decimal type data. Numerical calculations in the contract are indispensable, and the design of numerical calculations may cause relative errors. For example, the same level of calculations: 5/2*10=20, and 5*10/2=25, resulting in errors, which are larger in data The error will be larger and more obvious.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.16. Incorrect use of random numbers【PASS】

Smart contracts may need to use random numbers. Although the functions and variables provided by Solidity can access values that are obviously unpredictable, such as block.number and block.timestamp, they are usually more public than they

appear or are affected by miners. These random numbers are predictable to a certain extent, so malicious users can usually copy it and rely on its unpredictability to attack the function.

Audit result: After testing, there is a function of using pseudo-random numbers in the smart contract code, and there are risks such as being affected by miners, but the project party chose to ignore this problem.

Recommendation： nothing.

4.17. Unsafe interface usage【PASS】

Check whether unsafe interfaces are used in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.18. Variable coverage【PASS】

Check whether there are security issues caused by variable coverage in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.19. Uninitialized storage pointer【PASS】

In solidity, a special data structure is allowed to be a struct structure, and the local variables in the function are stored in storage or memory by default.

The existence of storage (memory) and memory (memory) are two different concepts. Solidity allows pointers to point to an uninitialized reference, while uninitialized local storage will cause variables to point to other storage variables, leading to variable coverage, or even more serious As a consequence, you should avoid initializing struct variables in functions during development.

Audit result: After testing, the smart contract code does not have this problem.

Recommendation： nothing.

4.20. Return value call verification【PASS】

This problem mostly occurs in smart contracts related to currency transfer, so it is also called silent failed delivery or unchecked delivery.

In Solidity, there are transfer(), send(), call.value() and other currency transfer methods, which can all be used to send OKT to an address. The difference is: When the transfer fails, it will be thrown and the state will be rolled back; Only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when send fails; only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when call.value fails to be sent; all available gas will be passed for calling

(can be Limit by passing in gas_value parameters), which cannot effectively prevent reentry attacks.

If the return value of the above send and call.value transfer functions is not checked in the code, the contract will continue to execute the following code, which may lead to unexpected results due to OKT sending failure.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.21. Transaction order dependency【PASS】

Since miners always get gas fees through codes that represent externally owned addresses (EOA), users can specify higher fees for faster transactions. Since the Ethereum blockchain is public, everyone can see the content 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 transaction at a higher fee to preempt the original solution.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.22. Timestamp dependency attack【PASS】

The timestamp of the data block usually uses the local time of the miner, and this time can fluctuate in the range of about 900 seconds. When other nodes accept a new block, it only needs to verify whether the timestamp is later than the previous block and The error with local time is within 900 seconds. A miner can profit from it by setting the timestamp of the block to satisfy the conditions that are beneficial to him as much as possible.

Check whether there are key functions that depend on the timestamp in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.23. Denial of service attack【PASS】

In the world of Ethereum, denial of service is fatal, and a smart contract that has suffered this type of attack may never be able to return to its normal working state. There may be many reasons for the denial of service of the smart contract, including malicious behavior as the transaction recipient, artificially increasing the gas required for computing functions to cause gas exhaustion, abusing access control to access the private component of the smart contract, using confusion and negligence, etc. Wait.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.24. Fake recharge vulnerability【PASS】

The transfer function of the token contract uses the if judgment method to check the balance of the transfer initiator (msg.sender). When balances[msg.sender] <value, enter the else logic part and return false, and finally no exception is thrown. We believe that only if/else this kind of gentle judgment method is an imprecise coding method in sensitive function scenarios such as transfer.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.25. Reentry attack detection【PASS】

The **call.value()** function in Solidity consumes all the gas it receives when it is used to send OKT. When the **call.value()** function to send OKT occurs before the actual reduction of the sender's account balance, There is a risk of reentry attacks.

Audit results： After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.26. Replay attack detection【PASS】

If the contract involves the need for entrusted management, attention should be paid to the non-reusability of verification to avoid replay attacks

In the asset management system, there are often cases of entrusted management. The principal assigns assets to the trustee for management, and the principal pays a certain fee to the trustee. This business scenario is also common in smart contracts.

Audit results： After testing, the smart contract does not use the call function, and this vulnerability does not exist.

Recommendation： nothing.

4.27. Rearrangement attack detection【PASS】

A rearrangement attack refers to a miner or other party trying to "compete" with smart contract participants by inserting their own information into a list or mapping, so that the attacker has the opportunity to store their own information in the contract. in.

Audit results： After auditing, the vulnerability does not exist in the smart contract code.

Recommendation： nothing.

5. Appendix A：Vulnerability rating standard

Smart contract vulnerability rating standards	
Level	**Level Description**
High	Vulnerabilities that can directly cause the loss of token contracts or user funds, such as: value overflow loopholes that can cause the value of tokens to zero, fake recharge loopholes that can cause exchanges to lose tokens, and can cause contract accounts to lose OKT or tokens. Access loopholes, etc.; Vulnerabilities that can cause loss of ownership of token contracts, such as: access control defects of key functions, call injection leading to bypassing of access control of key functions, etc.; Vulnerabilities that can cause the token contract to not work properly, such as: denial of service vulnerability caused by sending OKT to malicious addresses, and denial of service vulnerability caused by exhaustion of gas.
Medium	High-risk vulnerabilities that require specific addresses to trigger, such as value overflow vulnerabilities that can be triggered by token contract owners; access control defects for non-critical functions, and logical design defects that cannot cause direct capital losses, etc.
Low	Vulnerabilities that are difficult to be triggered, vulnerabilities with limited damage after triggering, such as value overflow vulnerabilities that require a large amount of OKT or tokens to trigger, vulnerabilities where attackers cannot directly profit after triggering value overflow, and the transaction sequence triggered by specifying high gas depends on the risk.

6. Appendix B：Introduction to auditing tools

6.1. Manticore

Manticore is a symbolic execution tool for analyzing binary files and smart contracts. Manticore includes a symbolic Ethereum Virtual Machine (EVM), an EVM disassembler/assembler and a convenient interface for automatic compilation and analysis of Solidity. It also integrates Ethersplay, Bit of Traits of Bits visual disassembler for EVM bytecode, used for visual analysis. Like binary files, Manticore provides a simple command line interface and a Python for analyzing EVM bytecode API.

6.2. Oyente

Oyente is a smart contract analysis tool. Oyente can be used to detect common bugs in smart contracts, such as reentrancy, transaction sequencing dependencies, etc. More convenient, Oyente's design is modular, so this allows advanced users to implement and Insert their own detection logic to check the custom attributes in their contract.

6.3. securify.sh

Securify can verify common security issues of Ethereum smart contracts, such as disordered transactions and lack of input verification. It analyzes all possible execution paths of the program while fully automated. In addition, Securify also has a

specific language for specifying vulnerabilities, which makes Securify can keep an eye on current security and other reliability issues at any time.

6.4. Echidna

Echidna is a Haskell library designed for fuzzing EVM code.

6.5. MAIAN

MAIAN is an automated tool for finding vulnerabilities in Ethereum smart contracts. Maian processes the bytecode of the contract and tries to establish a series of transactions to find and confirm the error.

6.6. ethersplay

ethersplay is an EVM disassembler, which contains relevant analysis tools.

6.7. ida-evm

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

6.8. Remix-ide

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

6.9. Knownsec Penetration Tester Special Toolkit

Pen-Tester tools collection is created by KnownSec team. It contains plenty of Pen-Testing tools such as automatic testing tool, scripting tool, Self-developed tools etc.

KNOWNSEC

Beijing KnownSec Information Technology Co., Ltd.

Advisory telephone	+86(10)400 060 9587
E-mail	sec@knownsec.com
Website	www.knownsec.com
Address	wangjing soho T2-B2509,Chaoyang District, Beijing