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 | 2021/09/22 | V1.1 |

# Report Information

| Title | Version | Document Number | Type |
| --- | --- | --- | --- |
| **DBL Miner Smart Contract Audit Report** | V1.1 | 62981450fbf645d294d293732fa874db | Open to project team |

# Copyright Notice

# Table of Contents

# 1. Introduction

The effective test time of this report is from **August 16, 2021** to **September 22, 2021**. During this period, the security and standardization of **the smart contract code of the DBL Miner** 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 4). **The smart contract code of the DBL Miner** 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：62981450fbf645d294d293732fa874db**

**Report query address link:**

https://attest.im/attestation/searchResult?qurey=62981450fbf645d294d293732fa874db

**Target information of the DBL Miner audit:**

| Target information | |
|---|---|
| **Project name** | DBL Miner |
| **Contract address** | 0xA316E8D4a6d36F36528b27419F6b3c0DFe7cD3d2 |

| | |
|---|---|
| **Code type** | Ethereum smart contract code |
| **Code language** | Solidity |

**Contract documents and hash:**

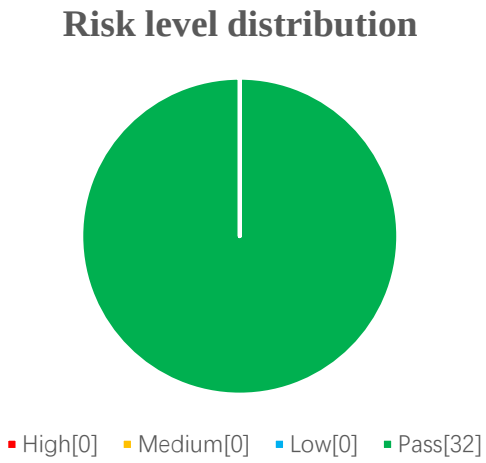
| Contract documents | MD5 |
|---|---|
| DBLendMine.sol | d9f6691e046b2889e2996a604599ddbb |

# 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 | 32 |



## 2.2 Audit Result

| Result of audit | | | |
|---|---|---|---|
| **Audit Target** | **Audit** | **Status** | **Audit Description** |
| **Business security testing** | **DBL and DEBI token inspection** | Pass | After testing, there is no such safety vulnerability. |
| | **DBLendToken contract voting related functions** | Pass | After testing, there is no such safety vulnerability. |
| | **Miner contract user pledge function** | Pass | After testing, there is no such safety vulnerability. |
| | **Miner contract user withdrawal function** | Pass | After testing, there is no such safety vulnerability. |
| | **Miner contract revenue calculation 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.oriigin 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. Analysis of code audit results

## 3.1. DBL and DEBI token inspection【PASS】

Perform security audits on the DBL and DEBI tokens in the contract to check whether they meet the ERC20 token standard.

**Audit analysis:**

DEBI tokens are developed based on the ERC20 token standard. The full name is DeerBit Token, or DEBI for short. The total issuance is 100000000000 * 1e8, with a precision of 8. It has functions such as transfer, authorized transfer, increase and decrease of authorization, and supports additional token issuance and token destruction.

DBL tokens are developed based on the ERC20 token standard. The full name is DBLend Line, or DBL for short. The total issuance is 2.100000 * 1e8, with a precision of 8. It has functions such as transfer and authorized transfer, and supports additional token issuance and token destruction.

**Recommendation：** nothing.

## 3.2. DBLendToken contract voting related functions【PASS】

Perform security audits on voting-related functions in the DBLendToken contract, check whether there is a check on parameters, whether there are logic design defects, etc.

**Audit analysis:** The voting rights transfer function _moveDelegates and the voting signature verification function delegateBySig of the DBLendToken contract are designed reasonably and correctly, and no related security risks have been found.

```
function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
//knownsec    //  Voting rights transfer
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                uint256 srcRepNum = numCheckpoints[srcRep];
```

```
                    uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum -
1].votes : 0;
                    uint256 srcRepNew = srcRepOld - amount;
                    _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
                   //knownsec //  Minus the voting power of the old proxy
                }

                if (dstRep != address(0)) {
                    uint256 dstRepNum = numCheckpoints[dstRep];
                    uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum -
1].votes : 0;
                    uint256 dstRepNew = dstRepOld + amount;
                    _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
                    //knownsec //  Increase voting rights for new agents
                }
            }
        }

        function _writeCheckpoint(address delegatee, uint256 nCheckpoints, uint256 oldVotes,
uint256 newVotes) internal {
            uint32 blockNumber = safe32(block.number, "DBL::_writeCheckpoint: block number
exceeds 32 bits");

            if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock ==
blockNumber) {
                checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
            } else {
                checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);
                numCheckpoints[delegatee] = nCheckpoints + 1;
            }
           //knownsec //  Update related status based on conditions
            emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
            //knownsec //  Log transfer events
```

```
        }
    function delegateBySig(address delegatee, uint nonce, uint expiry, uint8 v, bytes32 r, bytes32 s)
public returns (address){
    //knownsec //  Voting signature verification
        bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH,
keccak256(bytes(name())), getChainId(), address(this)));
        bytes32 structHash = keccak256(abi.encode(DELEGATION_TYPEHASH, delegatee, nonce,
expiry));
        bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator, structHash));
        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "DBL::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "DBL::delegateBySig: invalid nonce");
        require(block.timestamp <= expiry, "DBL::delegateBySig: signature expired");
        _delegate(signatory, delegatee);
        return signatory;
    }
```

**Recommendation：** nothing.

## 3.3. Miner contract user pledge function【PASS】

Perform security audits on the user pledge function in the Miner contract, check whether there is a check of parameters, whether there is an integer overflow, whether there are logic design defects, etc.

**Audit analysis:** The logic design of the user pledge function is reasonable and correct, and no related security risks are found.

```
    function stake(uint256 _amount) external {
    //knownsec //  Pledge function
            // Pausing is a very serious situation - we revert to sound the alarms
            require(!rewardGuardianPaused && !stakeGuardianPaused, "STAKE_OR_REWARD
IS PAUSED");
        //knownsec The pledge and feedback functions are not suspended
```

```
        require(_amount > 0,"AMOUNT MUST > 0");
        require(DEBIToken(DIBI).balanceOf(msg.sender) >= _amount, "INSUFFICIENT
DIBI");          //knownsec //  The number of DIBI tokens held by the user must be greater than or
equal to the number of tokens used for mortgage mining
        STAKER storage staker = stakers[msg.sender];
        _calPerDibiSharedDbl();    //knownsec //  Function used to calculate output
        uint256 earned = 0;
        if(staker.stakeBalance>0){
            earned = ( perDibiSharedDbl * (staker.stakeBalance) - staker.harvestDebt ) / 1e18;
            //knownsec //  Calculate user benefits
            if(earned>0 && earned!=2**256-1){
                staker.rewardPending = staker.rewardPending + earned;
                // DBLendToken(DBL).mint(msg.sender,earned);
            }
            staker.lastHarvestBlock = block.number;
        }

        DEBIToken(DIBI).transferFrom(msg.sender,address(this),_amount);
        if(totalStaked==0) latestBlockNum=block.number;
        totalStaked += _amount;

        staker.stakeBalance += _amount;
        staker.harvestDebt = perDibiSharedDbl * staker.stakeBalance;
        emit Stake(msg.sender, _amount,earned);
        //knownsec //  Record mortgage events
    }
```

**Recommendation：** nothing.

## 3.4. Miner contract user withdrawal function【PASS】

Perform security audits on the logic design of the user's withdrawal function in the Miner contract, check whether there is a check on the parameters, whether there are logic design defects, etc.

**Audit analysis:** The logic design of the user's withdrawal function is reasonable and correct, and no related security risks have been found.

```
    function withdraw(uint256 _amount) public {
    //knownsec //  User withdrawal function
                require(stakers[msg.sender].stakeBalance >= _amount,"DBL: INSUFFICIENT DIBI");
                STAKER storage staker = stakers[msg.sender];
                if(block.number > staker.lastHarvestBlock){
                    //knownsec //  Calculate revenue and change related status
                    _calPerDibiSharedDbl();
                    uint256 earned = ( perDibiSharedDbl * (staker.stakeBalance) -
staker.harvestDebt ) / 1e18;
                    if(earned>0 && earned!=2**256-1){
                        staker.rewardPending = staker.rewardPending + earned;
                    }
                    staker.lastHarvestBlock = block.number;
                }

                staker.stakeBalance -= _amount;
                staker.harvestDebt = staker.stakeBalance * perDibiSharedDbl;
                totalStaked -= _amount;
                DEBIToken(DIBI).transfer(msg.sender, _amount);
                //knownsec //  Send DIBI tokens
            }
```

**Recommendation：** nothing.

## 3.5. Miner contract revenue calculation function【PASS】

Perform a security audit on the revenue calculation function in the Miner contract to check whether the parameters are checked, whether there is a logical design defect, whether it conforms to the predetermined design model, etc.

**Audit analysis:** The logic design of the revenue calculation function is reasonable and correct, conforms to the predetermined design model, and no relevant security risks are found.

The maximum benefit of reward tokens depends on the number of DBL tokens in the DBL token contract. If the number of DBL tokens in the DBL token contract is less than the user's due reward tokens, it may affect the user's reward token income.

```
    function harvest() public returns(uint256 earned){
    //knownsec //  Income calculation function
            require(stakers[msg.sender].stakeBalance > 0 || stakers[msg.sender].rewardPending > 0,
"DBL: INSUFFICIENT STAKE OR REWARD");
    //knownsec //  The user has a pledge or the income is not zero
            require(block.number > stakers[msg.sender].lastHarvestBlock,"DBL: REPEAT
HARVEST");

            _calPerDibiSharedDbl();
            STAKER storage staker = stakers[msg.sender];
            earned = ( staker.stakeBalance * perDibiSharedDbl - staker.harvestDebt ) / 1e18;
            if(staker.rewardPending >0 || earned>0){
                staker.lastHarvestBlock = block.number;
                staker.harvestDebt = staker.stakeBalance * perDibiSharedDbl;
                earned = staker.rewardPending + earned;
                staker.rewardPending = 0;
                uint256 dblbalance = DBLendToken(DBL).balanceOf(DBL); //knownsec // Get
the number of DBL tokens under the token contract
                if(earned>dblbalance){
                    earned = dblbalance; //knownsec // The maximum number of reward tokens is
the number of DBL tokens in the token contract
                }
```

```
                DBLendToken(DBL).mint(msg.sender,earned);    //knownsec //  Send revenue to
users
                emit Harvest(msg.sender, earned);     //knownsec Log event

            }
        }
    function _calPerDibiIncDbl() internal view returns (uint256 incDblPerDIBI){
    //knownsec //  Calculate the profit function per coin
            incDblPerDIBI = 0;

            if(rewardGuardianPaused) return incDblPerDIBI;

            if(block.number > latestBlockNum && totalStaked>0){
                uint256 current = currentYearDibi();
                uint256 increaseDbl = ( block.number - latestBlockNum ) * current /
blocksPerYear ;
                incDblPerDIBI = increaseDbl / totalStaked;

            }

            return incDblPerDIBI;
            //knownsec //  Return per coin
        }
```

**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 compiler version 0.8.0 is formulated in the smart contract code, 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 is not used in the smart contract code. Since the compiler version will perform arithmetic overflow detection after 0.8.0, 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 require statement is reasonably used in the smart contract code, and there is no such security problem.

**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 fallback function is not defined in the smart contract code, and there is no such security problem.

**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 tx.origin global variable is not used in the smart contract code, and there is no such security problem.

**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 smart contract code reasonably restricts the owner's authority, and there is no such security problem.

**Recommendation：** nothing.

## 4.9. Gas consumption detection【PASS】

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

**Audit result:** After testing, there is no circular call in the smart contract code, and there is no such security problem.

**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 call/delegatecall function is not used in the smart contract code to perform operations, and there is no such security problem.

**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 because liquid mining requires additional tokens, it is approved.

**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, keywords are reasonably used in the smart contract code to modify the function, and there is no such security problem.

**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, block.number and block.timestamp are correctly used in the smart contract code, and there is no such security 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 uses the structure correctly, and there is no such 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 ETH 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 ETH sending failure.

**Audit result:** After testing, the smart contract code does not use transfer (), send (), call.value () and other currency transfer methods, and there is no such security problem.

**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 key functions that rely on the timestamp in the smart contract code are used correctly, and there is no such security problem.

**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 key judgments in the smart contract code are very strict, and there is no such security problem.

**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 ETH. When the **call.value()** function to send ETH occurs before the actual reduction of the sender's account balance, There is a risk of reentry attacks.

**Audit results：** After testing, the call function is not used in the smart contract code, and there is no such security problem.

**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 testing, there are no related vulnerabilities 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 access loopholes that can cause contract accounts to lose ETH or tokens, etc;<br>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.;<br>Vulnerabilities that can cause the token contract to not work properly, such as: denial of service vulnerability caused by sending ETH to malicious addresses, and denial of service vulnerability caused by exhaustion of energy. |
| **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 ETH or tokens to trigger, vulnerabilities where attackers cannot directly profit after triggering value overflow, and the transaction sequence triggered by specifying high energy 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 |