KNOWNSEC

Degree of secrecy ★ | Project team open

Smart Contract Audit Report

Security status

Medium

Principal tester: Knownsec blockchain security team

Version Summary

Content	Date	Version
Editing Document	2021/10/13	V1.0

Report Information

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

Copyright Notice

Table of Contents

1. Introduction

The effective test time of this report is from October 8, 2021 to October 13, 2021. During this period, the security and standardization of the **METASEER Code of pledge mining and ore pool operation contract of smart contract,** were 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 3). **The smart contract code of the METASEER** is comprehensively assessed as **Medium**.

Results of this smart contract security audit: Medium

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:

Report query address link:

Target information of the GOAT audit:

Target information	
Project name	MBNB
Entry name	METASEER
Token address	https://testnet.bscscan.com/address/0xb19ed5f2693bca3361f

	70e090fcfe1d35ac98f5a
Code type	BSC smart contract code, pledge mining contract, ore pool operation contract
Code language	Solidity

Contract documents and hash:

Contract documents	MD5
OptionTrade.sol	A9D0BD4B9E64C9967B826D6C48101DF8

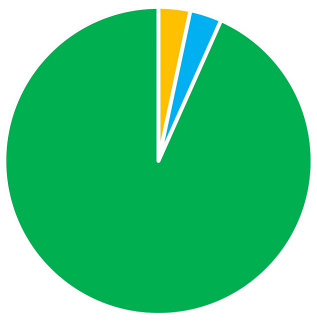
2. Code vulnerability analysis

2.1 Vulnerability Level Distribution

Vulnerability risk statistics by level:

Vulnerability risk level statistics table			
High	Medium	Low	Pass
0	1	1	28

Risk level distribution



2.2 Audit Result

Result of audit			
Audit Target	**Audit**	**Status**	**Audit Description**
Business security testing	**Pledge withdrawal function**	Pass	After testing, there is no such safety vulnerability.
	Token sending function	Medium	After detection, it may lead to malicious withdrawal in the contract.
	Income distribution function	Low	It is detected that the owner permission is too large.
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	warn	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. Pledge withdrawal function 【PASS】

Audit analysis: Conduct security audit on the logic of pledge withdrawal function (insake / outsake) in the contract. Its purpose is to mortgage BMB to obtain liquidity tokens and withdraw mortgage. The method permission is external. The function call is reasonable and the logic is normal.

```
    function InStake() external payable{      //knownsec    External pledge method
        uint256 MBNBPrice = 0;          //knownsec Initialization mbnb price is 0
        if(this.totalSupply() > 0 && (address(this).balance).mul(10**18) > 0){
            MBNBPrice                                                              =
(((address(this).balance).sub(msg.value)).mul(10**18))/this.totalSupply();   //knownsecProportion
of new balance
        }



        uint256 payMBNB = 0;
        if(this.totalSupply() == 0 || MBNBPrice == 0){
            payMBNB = msg.value/minimumMBNBPrice; //knownsec Calculate payment token
        }else{
            payMBNB = msg.value/MBNBPrice; //knownsec Mbnb price exists
        }

        if(payMBNB>0){
            payMBNB   =   payMBNB.mul(10**18);   //knownsec   Multiplication   accuracy
calculation
        }

        _mint(msg.sender, payMBNB); //knownsec to mint
        // emit Withdraw(msg.sender, amount, burn);
    }
    function OutStake(uint256 MBNBAmount) external { //knownsec Withdrawal pledge
        require(MBNBAmount <= balanceOf(msg.sender), "Pool: Insufficient MBNB");
        //knownsec Sorry, your credit is running low
        uint256 MBNBPrice = 0;
        if(this.totalSupply() > 0 && (address(this).balance).mul(10**18) > 0){ //knownsec
Existing balance and supply
            MBNBPrice = (address(this).balance).mul(10**18)/this.totalSupply(); //knownsec
Calculate currency price
        }



        uint256 payBNB = 0;
        if(this.totalSupply() == 0 || MBNBPrice == 0){
            payBNB = MBNBAmount * minimumMBNBPrice;
        }else{
            payBNB = MBNBAmount * MBNBPrice;
        }



        if(payBNB>0){
            payBNB = payBNB.div(10**18);
        }


        _burn(msg.sender, MBNBAmount); //konwnsec Burn coins
        sendBNBDividends(msg.sender,payBNB); //knownsec send out   bnb
    }
```

Recommendation: nothing.

3.2. Token sending function 【Medium】

Audit analysis: Check the token sending function indirectly called in the contract. It is found that sendbnbdivideds and sendethdivideds will send all tokens in the contract when send fails (returns false), while send fails in many cases, such as insufficient balance, gas depletion, recursion depth up to 1024, etc. If the interaction object is a contract, it may be maliciously used to extract all tokens in the contract..

```
 function sendBNBDividends(address receiver,uint amount ) private {
            if (!address(uint160(receiver)).send(amount))  {//knownsec It is recommended to add the check of amount and balance here
                return address(uint160(receiver)).transfer(address(this).balance);
            }
    }
        …
        function sendETHDividends(address receiver,uint amount ) private {//knownsec Private currency transfer
            if (!address(uint160(receiver)).send(amount)) { //knownsec send Return false and send balance directly
                return address(uint160(receiver)).transfer(address(this).balance);
            }
    }
```

Recommendation: When interacting with other contracts, it is recommended to verify whether the changed address is a contract, or check and judge the relationship between balance and amount here and use transfer to prevent the risk of wrong withdrawal of all tokens or re-entry.

3.3. Income distribution function 【Low】

Audit analysis: After checking the revenue distribution function in the contract, it is found that the functions of profitdistribution and insuredreturn are the same. At the same time, the owner can freely distribute the funds in all contracts and has great authority.

```
        function ProfitDistribution(address[] memory user,uint[] memory amount) public
```

```
onlyOwner { //knownsec Owner Available profit distribution with large authority
                  for(uint i=0;i<user.length;i++){
                        sendETHDividends(user[i],amount[i]);
                  }
      }

            function  InsuredRefund(address[]  memory  user,uint[]  memory  amount)  public
onlyOwner {//knownsec Owne    rAvailable profit distribution with large authority
                  for(uint i=0;i<user.length;i++){
                        sendETHDividends(user[i],amount[i]);
                  }
      }
```

Recommendation: The revenue design rules are extracted by the user to prevent other risks such as the disclosure of the owner's private key. At the same time, wrong currency withdrawal may cause damage to the user's rights and interests.

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 specifies that the compiler version is greater than or equal to 0.6.0, and there is no such security issue.

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【warn】

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

Audit result: After detection, the security problem exists in the smart contract code, and the owner can arbitrarily allocate the funds in the contract. For details, refer to 3.3.

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 have this vulnerability.

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, there is a coin code in the smart contract code but there is no such security problem.

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, the security problem does not exist in the smart contract code.

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 security problem does not exist in the smart contract code.

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 BNB 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 BNB 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】

Re-entry vulnerability is the most famous Ethereum smart contract vulnerability, which once led to the fork of Ethereum (The DAO hack).

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

Audit results: After auditing, the vulnerability 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 have this vulnerability.

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 (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 BNB 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 BNB 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 BNB 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