KNOWNSEC

Degree of secrecy ★ | Project team open

Smart Contract Audit Report

Security status

Safe

★ ★ ★ ★ ★

Principal tester: **Knownsec Blockchain Security Team**

Version Summary

Content	Date	Revised	Version
Editing Document	20210909	Knownsec Blockchain Security Team	V1.0

Report Information

Title	Version	Document Number	Type
OOE Smart Contract Audit Report	V1.0	6a7e922c0ed74292b77c20cf6de2d4ef	Open to project team

Copyright Notice

Table of Contents

1. Introduction

The effective test time of this report is from From **August 20, 2021 to August 26, 2021.** During this period, the security and standardization of **the smart contract code of the OOE** 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 3). **The smart contract code of the OOE** 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：6a7e922c0ed74292b77c20cf6de2d4ef

Report query address link:

https://attest.im/attestation/searchResult?qurey=6a7e922c0ed74292b77c20cf6de2d4ef

Target information of the OOE audit:

Entry		Description
Contract address	MultiReward	0xb55bC8473f84DcF03F7be699D6B44622970aC7e8

	OpenOceanExchange	0xeD85325119cCFc6aCB16FA931bAC6378B76e4615
	OpenOceanExchangeProxy	0x6352a56caadC4F1E25CD6c75970Fa768A3304e64
Code type	AVAX smart contract code	
The code language	Solidity	

Contract documents and hash:

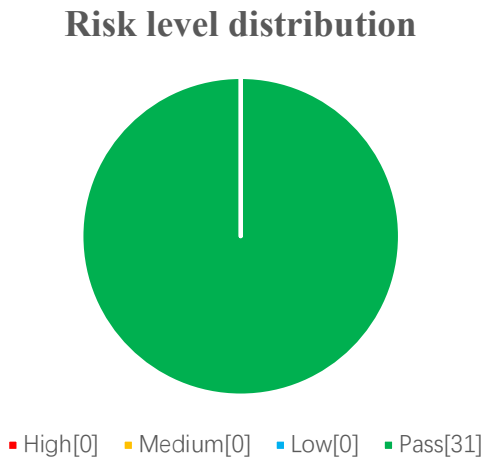
Contract documents	MD5
MultiReward.sol	447B13417D34B92F83B0DCAAA55BD5C7
OpenOceanExchange.sol	84B8B7B5AA13A2B2E11F816F66C15D91
RevertReasonParser.sol	154B8F724AB79FE8487070A685C209B0
UniversalERC20.sol	9C24DBF34A61F8EFF58A7F9EBE68D0B6

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	31



2.2 Audit Result

Result of audit			
Audit Target	**Audit**	**Status**	**Audit Description**
Business security testing	**MultiReward contract mortgage mining function**	Pass	After testing, there is no such safety vulnerability.
	MultiReward Contract Withdrawal Mortgage Token Feature	Pass	After testing, there is no such safety vulnerability.
	MultiReward contract harvest reward token feature	Pass	After testing, there is no such safety vulnerability.
	OpenOcean Exchange Contract Redemption Arbitrage Feature	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. Business security detection

3.1. MultiReward contract mortgage mining function【PASS】

Audit Analysis: The demosit function of the MultiReward contract implements the function of the user mortgage token to mine in the contract. Project can set up a variety of reward tokens, to achieve the mortgage of a token to obtain a variety of reward token functions.

```
    function deposit(uint amount) public {
        // update index
        for (uint i = 1; i <= rewardsId; i++) {
            updateRewardIndex(i); //knownsec// Update your reward currency information
            distributeReward(i, msg.sender); //knownsec// Send the previous block reward to the user
        }
        uint balance = stakeToken.balanceOf(address(msg.sender)); //knownsec// Get the user's mortgage token balance
        if (amount > balance) {
            amount = balance; //knownsec// If insufficient, the number of mortgages is updated to the balance
        }
        // update state
        userStaked[msg.sender] += amount; //knownsec// Update the number of user
```

```
mortgage tokens
        totalStaked += amount; //knownsec// Update the total number of mortgage tokens
        // transfer asset
        stakeToken.safeTransferFrom(msg.sender, address(this), amount); //knownsec// The
user transfers the mortgage token into this contract
        emit Deposit(msg.sender, amount, block.number); //knownsec// Trigger a mortgage
event
    }
```

Recommendation：nothing.

3.2. MultiReward Contract Withdrawal Mortgage Token Feature【PASS】

Audit Analysis: The withdraw function of the MultiReward contract enables the user to withdraw the tokens of their mortgage while reaping the various reward tokens obtained from the mortgage.

```
    function withdraw(uint amount) public {
        // update index
        for (uint i = 1; i <= rewardsId; i++) { //knownsec// Update your reward information
and send your reward tokens
            updateRewardIndex(i);
            distributeReward(i, msg.sender);
```

```
        }
        if (amount > userStaked[msg.sender]) { //knownsec// If the number of replacement coins is greater than the number of collaterals, it is updated to the number of mortgage tokens
            amount = userStaked[msg.sender];
        }
        uint balance = stakeToken.balanceOf(address(this)); //knownsec// Get the mortgage token balance of this contract
        require(balance >= amount, "Insufficient balance");
        // update state
        userStaked[msg.sender] -= amount; //knownsec// Update the user's mortgage balance
        totalStaked -= amount; //knownsec// Update the total number of mortgage tokens
        // transfer asset
        stakeToken.safeTransfer(msg.sender, amount); //knownsec// Send tokens to the caller
        emit Withdraw(msg.sender, amount, block.number); //knownsec// Triggers an extraction event
    }
```

Recommendation：nothing.

3.3. MultiReward contract harvest reward token feature 【PASS】

Audit Analysis: The claimBatch, claim, and claimReward functions of

multiReward contracts enable bulk withdrawal of all user reward tokens, withdrawal of all reward tokens for specified users, and extraction of award tokens specified by specified users.

```
    function claimBatch(address []memory users) public {//knownsec// Bulk harvest address rewards
        for (uint i = 0; i < users.length; i++) {
            claim(users[i]);
        }
    }

    function claim(address user) public {//knownsec// Harvest all reward tokens for the specified address
        for (uint i = 1; i <= rewardsId; i++) {
            updateRewardIndex(i);
            distributeReward(i, user);
        }
    }

    function claimReward(address user, uint rewardId) public {//knownsec// Harvest the specified reward tokens for the specified user
        updateRewardIndex(rewardId);
```

```
        distributeReward(rewardId, user);
    }
```

Recommendation：nothing.

3.4. OpenOcean Exchange Contract Redemption Arbitrage Feature【PASS】

Audit analysis: The swap function of OpenOcean Exchange contract realizes the function of redeeming arbitrage. Users can exchange arbitrage the best token price difference between Defi and Cefi by selecting a specified token transaction.

```
    function swap(
        IOpenOceanCaller caller, //knownsec//  Interface contract implementation address
        SwapDescription calldata desc, //knownsec// Swap description information
        IOpenOceanCaller.CallDescription[] calldata calls //knownsec//  The function call information
    ) external payable whenNotPaused returns (uint256 returnAmount) {
        require(desc.minReturnAmount > 0, "Min return should not be 0"); //knownsec// A minimum redemption return is required that the quantity returned is greater than 0
        require(calls.length > 0, "Call data should exist");  //knownsec// The call data is greater than 0

        uint256 flags = desc.flags; //knownsec// Get the flag bit
```

```
        IERC20 srcToken = desc.srcToken; //knownsec//  Gets the source token address of the
user's payment
        IERC20 dstToken = desc.dstToken; //knownsec//  Gets the target token address
obtained by the user

        if (flags & _REQUIRES_EXTRA_ETH != 0) {
            require(msg.value > (srcToken.isETH() ? desc.amount : 0), "Invalid
msg.value");//knownsec// Msg.value is required to be greater than the number of user sources
redeeming tokens
        } else {
            require(msg.value == (srcToken.isETH() ? desc.amount : 0), "Invalid
msg.value");//knownsec// Ask msg.value equal to the number of tokens redeemed by the user
source
        }

        if (flags & _SHOULD_CLAIM != 0) { //knownsec//  Cannot be ETH, the source
address can only be tokens
            require(!srcToken.isETH(), "Claim token is ETH");
            _claim(srcToken, desc.srcReceiver, desc.amount, desc.permit); //knownsec//
// Call the internal function, pass in the source token, the source token recipient, redeem the
quantity, redeem the signature
```

```
        }

        address dstReceiver = (desc.dstReceiver == address(0)) ? msg.sender : desc.dstReceiver; //knownsec//        // Determine the recipient address of the target token

        uint256 initialSrcBalance = (flags & _PARTIAL_FILL != 0) ? srcToken.universalBalanceOf(msg.sender) : 0; //knownsec//    // Gets the number of caller source tokens

        uint256 initialDstBalance = dstToken.universalBalanceOf(dstReceiver); //knownsec// // Get the initial number of recipients' target tokens

        caller.makeCalls{value: msg.value}(calls); //knownsec// Call the target contract to specify the function

        uint256 spentAmount = desc.amount; //knownsec// Gets the number of transfer tokens

        returnAmount = dstToken.universalBalanceOf(dstReceiver).sub(initialDstBalance); //knownsec// Calculate the quantity returned after redemption, and now the token balance minus the previous initial token balance

        if (flags & _PARTIAL_FILL != 0) {

            spentAmount = initialSrcBalance.add(desc.amount).sub(srcToken.universalBalanceOf(msg.sender));
```

```
//knownsec// The amount spent
                require(returnAmount.mul(desc.amount) >=
desc.minReturnAmount.mul(spentAmount), "Return amount is not enough");//knownsec// The
number of returns required is greater than the amount spent
            } else {
                require(returnAmount >= desc.minReturnAmount, "Return amount is not
enough");
            }

            _emitSwapped(desc, srcToken, dstToken, dstReceiver, spentAmount, returnAmount);
//knownsec// Triggers a redemption arbitrage event
        }

        function _claim(
            IERC20 token,
            address dst,
            uint256 amount,
            bytes calldata permit
        ) private {
            // TODO: Is it safe to call permit on tokens without implemented permit? Fallback
will be called. Is it bad for proxies?
```

```
            if (permit.length == 32 * 7) {
                // solhint-disable-next-line avoid-low-level-calls
                (bool success, bytes memory result) = address(token).call(abi.encodeWithSelector(IERC20Permit.permit.selector, permit));
//knownsec// Call the function selector and parameters specified in the source token address
                if (!success) {
                    revert(RevertReasonParser.parse(result, "Permit call failed: "));
                }
            }

            token.safeTransferFrom(msg.sender, dst, amount); //knownsec// The caller sends the token to the destination address
        }
```

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

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 tokens 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 tokens 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 tokens. When the **call.value()** function to send tokens 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 testing, the security problem 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 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 tokens 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 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

Remix is a browser-based compiler and IDE that allows users to build Ethereum contracts and debug transactions in the Solidity language.

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.

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