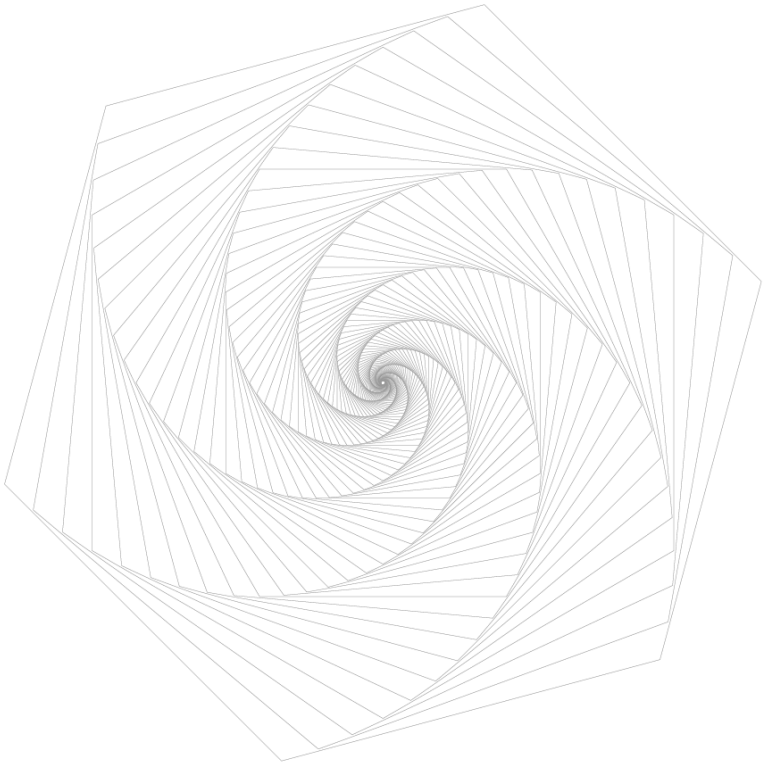
KNOWNSEC
Blockchain Lab

Smart Contract Audit Report



Version description

The revision	Date	Revised	Version
Write documentation	20220516	KNOWNSEC Blockchain Lab	V1.0

Document information

Title	Version	Document Number	Type
OOE Smart Contract Audit Report	V1.0	96c7d024af264ddea4dc547b2a0 520bb	Open to project team

Statement

KNOWNSEC Blockchain Lab only issues this report for facts that have occurred or existed before the issuance of this report, and assumes corresponding responsibilities for this. KNOWNSEC Blockchain Lab is unable to determine the security status of its smart contracts and is not responsible for the facts that will occur or exist in the future. The security audit analysis and other content made in this report are only based on the documents and information provided to us by the information provider as of the time this report is issued. KNOWNSEC Blockchain Lab 's assumption: There is no missing, tampered, deleted or concealed information. If the information provided is missing, tampered with, deleted, concealed or reflected in the actual situation, KNOWNSEC Blockchain Lab shall not be liable for any losses and adverse effects caused thereby.

Directory

1. Summarize

The effective test period of this report is from **May 12, 2022 to May 16, 2022**. During this period, the security and standardization of **OOE smart contracts** will be audited and used as the statistical basis for the report.

The scope of this smart contract security audit does not include external contract calls, new attack methods that may appear in the future, and code after contract upgrades or tampering. (With the development of the project, the smart contract may add a new pool , New functional modules, new external contract calls, etc.), does not include front-end security and server security.

In this audit report, engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (Chapter 6). **The smart contract code of the OOE** is comprehensively assessed as **PASS**.

Since the testing is under non-production environment, all codes are the latest version. In addition, the testing process is communicated with the relevant engineer, and testing operations are carried out under the controllable operational risk to avoid production during the testing process, such as: Operational risk, code security risk.

KNOWNSEC Attest information:

classification	information
report number	96c7d024af264ddea4dc547b2a0520bb
report query link	https://attest.im/attestation/searchResult?qurey=96c7d024af264ddea4dc547b2a0520bb

2. Item information

2.1. Item description

OOE is the world's first complete aggregation protocol for crypto trading, which captures liquidity from DeFi and CeFi markets and enables cross-chain exchange. Our smart routing algorithm finds the best prices from DEXs and CEXes and splits routes to give traders the best prices with low slippage and fast settlement. The product is free to use; OpenOcean users only pay the regular blockchain gas and exchange transaction fees, which are charged by the exchange, not OpenOcean.

OpenOcean aggregates major exchanges (DEXes and CEXes) and the entire Ethereum, Ethereum Layer 2 like Loopping and Polygon, Binance Smart Chain, Solana, HECO, Ontology, TRON, and is Binance Smart Chain, TRON, Ethereum Layer 2 and the first full aggregator for the Binance exchange. We continue to aggregate public chains and exchanges according to the needs of the community.

In addition to pooled swaps, OpenOcean will continue to integrate derivatives, income, lending and insurance products, and launch its own portfolio margin products and smart wealth management services. OpenOcean provides users with APIs and arbitrage tools to operate automated arbitrage strategies.

The vision is to build a complete aggregator for crypto trading to improve capital efficiency and connect siloed islands in the currently fragmented DeFi and CeFi markets. Whether it is a small individual investor or a large institution, everyone should have the opportunity to trade at the best prices and apply their own investment strategies to various crypto asset classes.

OpenOcean has its own token, OOE, as a utility and governance token.

The project's website

2.2 https://openocean.finance

2.3. White Paper

https://docs.openocean.finance/

2.4. Review version code

https://bscscan.com/address/0xA8A0213bb2ce671E457Ec14D08EB9d40E6DA8e2d#code

https://bscscan.com/address/0x63c85eb44932d3b99E3975F4601c330FBD26fcD8#code

2.5. Contract file and Hash/contract deployment address

The contract documents	MD5
AggregatorMock.sol	48b1abac4c151d4c27ebf49e4d44948c
ArgumentsDecoderTest.sol	d32d85c1d92dd0b9623b922d105c7325
LimitOrderProtocol.sol	d32d2baf592a6bbe25b303ecfa603b59
Permitable.sol	0eb1adec1d05b9a93fd834741ae1be6b
ArgumentsDecoder.sol	f188c1809bda586011b3371017aa790d
RevertReasonParser.sol	cdfbe9a2f196d386386b5629a3dc3420
UniversalERC20.sol	032f16f3ed09abbb30cca6f1e941212a
OrderMixin.sol	5bf70d24eb02375a33011e4aba6acb07
OrderRFQMixin.sol	e4bc33da204905be7e7fbbaf4ff7d044

WethUnwrapper.sol	42d04a552d5d386f240b0d738db23fe2
PredicateHelper.sol	33e889f3f82f9d973356a32e4c5ac7c2
ChainlinkCalculator.sol	715da0e0767a8b0c37c43327f2284b62
AmountCalculator.sol	ef981c903622e6c4679cc5289c53932a
ERC721Proxy.sol	9fd6c5b7e9eb6c9a10e51e047fc92246
ImmutableOwner.sol	eb9118aa5d9bd7c99c21e7c9993e4b94
ERC721ProxySafe.sol	cc17bdd50cd30eef3c23c136f3680003
NonceManager.sol	f4c1a5ba0f05f759602bd6b1e3b61359
ERC1155Proxy.sol	e3b4e7ba3916797e43a34dc7b2c893ca
IWithdrawable.sol	61de4986ad406f6ef573a7d89ad50bbd
IDaiLikePermit.sol	0517e16f4028b7275763ce9e9488006e
InteractiveNotificationReceiver.sol	03e46d862b8faa03d8cd3b3c0a975177

3. External visibility analysis

3.1. LimitOrderProtocol contracts

LimitOrderProtocol					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
setFeeBuyTokenKeep	Public	True	onlyOwner	---	---
setIsNoProtect	external	True	onlyOwner	---	---
balanceOfPair	Public	False	---	---	---
addNoBurnAddress	Public	True	onlyOwner	---	---
setDuration	external	True	onlyOwner	---	---
removeNoBurnAddress	Public	True	onlyOwner	---	---
isContract	Public	False	---	---	---
isContractSender	Public	False	---	---	---
referParent	Public	True	---	---	---
getParent	Public	False	---	---	---
isInNoBurnAddress	Public	False	---	---	---
setStart	external	True	onlyOwner	---	---
setAddLpAddr	external	True	onlyOwner	---	---
checkBuyAmount	internal	True	---	---	---
_transfer	internal	True	---	---	---
recover	external	False	onlyOwner	---	---

3.2. OrderMixin contracts

OrderMixin					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
updateOperator	public	True	onlyOwner	---	---
getOOswap	public	False	---	---	---
setOOswap	public	True	onlyOwner	---	---
remaining	external	False	---	---	---
remainingRaw	external	False	---	---	---
remainingsRaw	external	False	---	---	---
simulateCalls	external	True	---	---	---
cancelOrder	external	True	---	---	---
fillOrder	external	True	---	---	---
hashUnmuteMsg	public	False	---	---	---
fillOrderTo	public	True	---	---	---
checkPredicate	public	False	---	---	---
hashOrder	public	False	---	---	---
_makeCall	private	True	---	---	---
_callGetter	private	False	---	---	---

3.3. OrderRFQMixin contracts

OrderRFQMixin					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**

invalidatorForOrderRFQ	external	False	---	---	---
cancelOrderRFQ	external	True	---	---	---
fillOrderRFQ	external	True	---	---	---
fillOrderRFQTo	public	True	---	---	---
_invalidateOrder	private	True	---	---	---

3.4. WethUnwrapper contracts

WethUnwrapper					
	visibility	**state changes**		**payable reception**	
notifyFillOrder	external	True	---	---	---

4. Code vulnerability analysis

4.1. Summary description of the audit results

Audit results			
audit project	**audit content**	**condition**	**description**
Business security detection			
Code basic vulnerability detection			
	authentication	Pass	After testing, there is no security issue.
	Owner permission control	Pass	After testing, there is no security issue.
	Gas consumption detection	Pass	After testing, there is no security issue.
	call injection attack	Pass	After testing, there is no security issue.

	Low-level function safety	Pass	After testing, there is no security issue.
	Vulnerability of additional token issuance	Pass	After testing, there is no security issue.
	Access control defect detection	Pass	After testing, there is no security issue.
	Fake recharge vulnerabilit y detection	Pass	After testing, there is no security issue.
	Reentry attack detection	Pass	After testing, there is no security issue.

	Replay attack detection	Pass	After testing, there is no security issue.
	Rearrangement attack detection	Pass	After testing, there is no security issue.

5. Business security detection

5.1. OrderMixin.sol contract fills in order related functions

【Pass】

Audit analysis: The remaining function of the contract returns an order with an unfilled price, and the remainingRaw and remainingsRaw functions are used to return the order when the order exists. The function permissions are correct, and no obvious security issues were found.

```
function remaining(bytes32 orderHash) external view returns(uint256) {  uint256
    amount = _remaining[orderHash];//knownsec Order hash  require(amount !=
    _ORDER_DOES_NOT_EXIST, "LOP: Unknown order"); unchecked { amount -=
    1; }

    return amount;
}

function remainingRaw(bytes32 orderHash) external view returns(uint256)
    { return _remaining[orderHash];
}

function remainingsRaw(bytes32[] memory orderHashes) external view returns(uint256[] memory)
{
    uint256[] memory results = new uint256[](orderHashes.length); for
    (uint256 i = 0; i < orderHashes.length; i++) {

        results[i] = _remaining[orderHashes[i]];//knownsec Get hashes in batches
}
    return results;
}

function cancelOrder(Order memory order) external { //knownsec// Cancel order  require(order.maker
            == msg.sender, "LOP: Access denied"); //knownsec// Whether the
```

```
order creator is the caller


            bytes32 orderHash = hashOrder(order); //knownsec// Get order hash
            uint256 orderRemaining = _remaining[orderHash]; //knownsec// Unfilled price order
found
            require(orderRemaining != _ORDER_FILLED, "LOP: already filled");
            emit OrderCanceled(msg.sender, orderHash, orderRemaining);

            _remaining[orderHash] = _ORDER_FILLED; //knownsec// Set parameters to cancel an
order
        }


        function fillOrderTo(
                    Order memory order, //knownsec// Order quote to fill
                    bytes calldata signature, //knownsec// Signature ensures quote ownership
                    uint256 makingAmount, //knownsec// Make order amount

                    uint256 takingAmount, //knownsec// Take away the order amount
                    uint256 thresholdAmount, //knownsec// Specify the maximum allowed takeAmount
when takingAmount is zero, otherwise specify the minimum allowed makingAmount

                    address target //knownsec// The address that will receive swap
            ) public returns(uint256 /* actualMakingAmount */, uint256 /* actualTakingAmount */)
{ //knownsec// Fill Order
                    require(target != address(0), "LOP: zero target is forbidden"); //knownsec// Swap
address is not 0

                    bytes32 orderHash = hashOrder(order); //knownsec// Order hash


                    {   // Stack too deep
                        uint256 remainingMakerAmount = _remaining[orderHash];
                        require(remainingMakerAmount != _ORDER_FILLED, "LOP: remaining
amount is 0");
                        require(order.allowedSender == address(0) || order.allowedSender ==
msg.sender, "LOP: private order");
                        if (remainingMakerAmount == _ORDER_DOES_NOT_EXIST) { //knownsec// If
the untraded order does not exist
```

```
                // First fill: validate order and permit maker asset
                (bytes memory sig, bytes memory userUnmuteSig) = abi.decode(signature,(bytes,bytes));
                if (userUnmuteSig.length > 2){

require(SignatureCheckerUpgradeable.isValidSignatureNow(operator, orderHash, sig), "LOP: bad signature:operator");

require(SignatureCheckerUpgradeable.isValidSignatureNow(order.maker, hashUnmuteMsg().toEthSignedMessageHash(), userUnmuteSig), "LOP: bad signature:user");

                } else {

require(SignatureCheckerUpgradeable.isValidSignatureNow(order.maker, orderHash, sig), "LOP: bad signature");

                }
                remainingMakerAmount = order.makingAmount; if (order.permit.length >= 20) {
                    // proceed only if permit length is enough to store address
                    (address token, bytes memory permit) = order.permit.decodeTargetAndCalldata();
                    _permitMemory(token, permit);
                    require(_remaining[orderHash] == _ORDER_DOES_NOT_EXIST, "LOP: reentrancy detected");
                }
            } else {
                unchecked { remainingMakerAmount -= 1; }
            }

            // Check if order is valid
            if (order.predicate.length > 0) {
                require(checkPredicate(order), "LOP: predicate returned false");
            }
```

```
// Compute maker and taker assets amount
if ((takingAmount == 0) == (makingAmount == 0))
    { revert("LOP: only one amount should be 0");
} else if (takingAmount == 0) { //knownsec// If takingAmount is 0, then
makingAmount can only be at most remainingMakerAmount
    uint256 requestedMakingAmount = makingAmount; if
    (makingAmount > remainingMakerAmount) {
        makingAmount = remainingMakerAmount;
    }
    takingAmount = _callGetter(order.getTakerAmount, order.makingAmount,
makingAmount, order.takingAmount);

    // check that actual rate is not worse than what was expected
    // takingAmount / makingAmount <= thresholdAmount /
requestedMakingAmount
    require(takingAmount * requestedMakingAmount <= thresholdAmount *
makingAmount, "LOP: taking amount too high");
} else {
    uint256 requestedTakingAmount = takingAmount;
    makingAmount = _callGetter(order.getMakerAmount,
order.takingAmount, takingAmount, order.makingAmount);

    if (makingAmount > remainingMakerAmount)
        { makingAmount = remainingMakerAmount;
        takingAmount = _callGetter(order.getTakerAmount,
order.makingAmount, makingAmount, order.takingAmount);
    }
    // check that actual rate is not worse than what was expected
    // makingAmount / takingAmount >= thresholdAmount /
requestedTakingAmount
    require(makingAmount * requestedTakingAmount >= thresholdAmount *
takingAmount, "LOP: making amount too low");
}

require(makingAmount > 0 && takingAmount > 0, "LOP: can't swap 0
```

```
amount");

                // Update remaining amount in storage
                unchecked {
                    remainingMakerAmount = remainingMakerAmount - makingAmount;
                    _remaining[orderHash] = remainingMakerAmount + 1;
                }
                emit OrderFilled(msg.sender, orderHash, remainingMakerAmount);
            }


            // Taker => Maker
            _makeCall(
                order.takerAsset,
                abi.encodePacked(
                    IERC20Upgradeable.transferFrom.selector,
                    uint256(uint160(msg.sender)),
                    uint256(uint160(order.receiver == address(0)
                                                        ? order.maker :
order.receiver)),
                    takingAmount,
                    order.takerAssetData
                )
            );

            // Maker can handle funds interactively if
            (order.interaction.length >= 20) {
                // proceed only if interaction length is enough to store address
                (address interactionTarget, bytes memory interactionData) =
order.interaction.decodeTargetAndCalldata();
                InteractiveNotificationReceiver(interactionTarget).notifyFillOrder(

                    msg.sender, order.makerAsset, order.takerAsset, makingAmount,
 takingAmount, interactionData
                );
            }
```

```
        // Maker => Taker
        _makeCall(
            order.makerAsset,
            abi.encodePacked(
                IERC20Upgradeable.transferFrom.selector,
                uint256(uint160(order.maker)),
                uint256(uint160(target)),
                makingAmount,
                order.makerAssetData //knownsec// Create Order
            )
        );

        return (makingAmount, takingAmount);
    }
```

Security advice: None.

5.2. LimitOrderProtocol.sol contract swap related functions

【Pass】

Audit analysis: The swap function of the contract is used to transfer tokens to the mining pool. The function permissions are correct, and no obvious security issues were found.

```
function swap(address from, address[] calldata path, uint[] calldata amounts, address fee,
bytes calldata swapExtraData) public payable onlyOperator {
    require(path.length == 2 && amounts.length == 2, "invalid args");
    address ooSwap = getOOswap();//knownsec Get swap pool address
    require(ooSwap != address(0), "ooswap is zero");

    Param memory vars;
```

```
    vars.isETH = IERC20Upgradeable(path[0]).isETH(); if
    (!vars.isETH) {
        IERC20Upgradeable(path[0]).safeTransferFrom(from, address(this), amounts[0]);
        IERC20Upgradeable(path[0]).safeIncreaseAllowance(ooSwap, amounts[0]);

    }
    (vars.success, vars.result) = ooSwap.call{value : msg.value}(swapExtraData);
    require(vars.success, "swap failed");
    if (!vars.isETH)
        {  IERC20Upgradeable(path[0]).safeApprove(ooSwap, 0);

    }
    uint256 returnAmount = abi.decode(vars.result, (uint256));//knownsec Returns the number of
tokens

    require(returnAmount >= amounts[1], "returnAmount is too low");
    IERC20Upgradeable(path[1]).universalTransfer(from, amounts[1]);
    uint delta = returnAmount - amounts[1];//
    if (delta > 0) {//knownsec If enough tokens are returned  address
        to = fee == address(0) ? owner() : fee;
        IERC20Upgradeable(path[1]).universalTransfer(to, delta);

    }
}
```

Security advice: None.

6. Code basic vulnerability detection

6.1. Compiler version security【Pass】

Check to see if a secure compiler version is used in the contract code implementation.

Detection results: After testing, the compiler version is greater than or equal to 0.6.0 in the smart contract code, and there is no such security problem.

Security advice: None.

6.2. Redundant code【Pass】

Check that the contract code implementation contains redundant code.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.3. Use of safe arithmetic library【Pass】

Check to see if the SafeMath security abacus library is used in the contract code implementation.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.4. Not recommended encoding【Pass】

Check the contract code implementation for officially uns recommended or deprecated coding methods.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.5. Reasonable use of require/assert【Pass】

Check the reasonableness of the use of require and assert statements in contract code implementations.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.6. Fallback function safety【Pass】

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

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.7. tx.origin authentication【Pass】

tx.origin is a global variable of Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts makes contracts vulnerable to phishing-like attacks.z

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.8. Owner permission control【Pass】

Check that theowner in the contract code implementation has excessive permissions. For example, modify other account balances at will, and so on.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.9. Gas consumption detection【Pass】

Check that the consumption of gas exceeds the maximum block limit.
Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.10. call injection attack【Pass】

When a call function is called, strict permission control should be exercised, or the function called by call calls should be written directly to call calls.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.11. Low-level function safety【Pass】

Check the contract code implementation for security vulnerabilities in the use of call/delegatecall

The execution context of the call function is in the contract being called, while the execution context of the delegatecall function is in the contract in which the function is currently called.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.12. Vulnerability of additional token issuance【Pass】

Check to see if there are functions in the token contract that might increase the total token volume after the token total is initialized.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.13. Access control defect detection【Pass】

Different functions in the contract should set reasonable permissions, check whether the functions in the contract correctly use pubic, private and other keywords for visibility modification, check whether the contract is properly defined and use modifier access restrictions on key functions, to avoid problems caused by overstepping the authority.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.14. Numerical overflow detection【Pass】

The arithmetic problem in smart contracts is the integer overflow and integer overflow, with Solidity able to handle up to 256 digits (2^256-1), and a maximum number increase of 1 will overflow to get 0. Similarly, when the number is an unsigned type, 0 minus 1 overflows to get the maximum numeric value.

Integer overflows and underflows are not a new type of vulnerability, but they are particularly dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if the likelihood is not anticipated, which can affect the reliability and safety of the program.

Detection results: The security issue is not present in the smart contract code after

detection.

Security advice: None.

6.15. Arithmetic accuracy error【Pass】

Solidity has a data structure design similar to that of a normal programming language, such as variables, constants, arrays, functions, structures, and so on, and there is a big difference between Solidity and a normal programming language - Solidity does not have floating-point patterns, and all of Solidity's numerical operations result in integers, without the occurrence of decimals, and without allowing the definition of decimal type data. Numerical operations in contracts are essential, and numerical operations are designed to cause relative errors, such as sibling operations: 5/2 x 10 x 20, and 5 x 10/2 x 25, resulting in errors, which can be greater and more obvious when the data is larger.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.16. Incorrect use of random numbers【Pass】

Random numbers may be required in smart contracts, and while the functions and variables provided by Solidity can access significantly unpredictable values, such as block.number and block.timestamp, they are usually either more public than they seem, or are influenced by miners, i.e. these random numbers are somewhat predictable, so

malicious users can often copy it and rely on its unpredictability to attack the feature.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.17. Unsafe interface usage【Pass】

Check the contract code implementation for unsafe external interfaces, which can be controlled, which can cause the execution environment to be switched and control contract execution arbitrary code.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.18. Variable coverage【Pass】

Check the contract code implementation for security issues caused by variable overrides.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.19. Uninitialized storage pointer【Pass】

A special data structure is allowed in solidity as a strut structure, while local

variables within the function are stored by default using stage or memory.

The existence of store (memory) and memory (memory) is two different concepts, solidity allows pointers to point to an uninitialized reference, while uninitialized local stage causes variables to point to other stored variables, resulting in variable overrides, and even more serious consequences, and should avoid initializing the task variable in the function during development.

Detection results: After detection, the smart contract code does not have the problem.

Security advice: None.

6.20. Return value call verification 【Pass】

This issue occurs mostly in smart contracts related to currency transfers, so it is also known as silent failed sending or unchecked sending.

In Solidity, there are transfer methods such as transfer(), send(), call.value(), which can be used to send tokens to an address, the difference being: transfer send failure will be throw, and state rollback; Call.value returns false when it fails to send, and passing all available gas calls (which can be restricted by incoming gas_value parameters) does not effectively prevent reentration attacks.

If the return values of the send and call.value transfer functions above are not checked in the code, the contract continues to execute the subsequent code, possibly with unexpected results due to token delivery failures.

Detection results: The security issue is not present in the smart contract code after

detection.

Security advice: None.

6.21. Transaction order dependency【Pass】

Because miners always get gas fees through code that represents an externally owned address (EOA), users can specify higher fees to trade faster. Since blockchain is public, everyone can see the contents of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transactions at a higher cost to preempt the original solution.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.22. Timestamp dependency attack【Pass】

Block timestamps typically use miners' local time, which can fluctuate over a range of about 900 seconds, and when other nodes accept a new chunk, they only need to verify that the timestamp is later than the previous chunk and has a local time error of less than 900 seconds. A miner can profit from setting the timestamp of a block to meet as much of his condition as possible.

Check the contract code implementation for key timestamp-dependent features.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.23. Denial of service attack【Pass】

Smart contracts that are subject to this type of attack may never return to normal operation. There can be many reasons for smart contract denial of service, including malicious behavior as a transaction receiver, the exhaustion of gas caused by the artificial addition of the gas required for computing functionality, the misuse of access control to access the private component of smart contracts, the exploitation of confusion and negligence, and so on.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.24. Fake recharge vulnerability【Pass】

The transfer function of the token contract checks the balance of the transfer initiator (msg.sender) in the if way, when the balances < value enters the else logic part and return false, and ultimately does not throw an exception, we think that only if/else is a gentle way of judging in a sensitive function scenario such as transfer is a less rigorous way of coding.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.25. Reentry attack detection【Pass】

The call.value() function in Solidity consumes all the gas it receives when it is used to send tokens, and there is a risk of re-entry attacks when the call to the call tokens occurs before the balance of the sender's account is actually reduced.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.26. Replay attack detection【Pass】

If the requirements of delegate management are involved in the contract, attention should be paid to the non-reusability of validation to avoid replay attacks

In the asset management system, there are often cases of entrustment management, the principal will be the assets to the trustee management, the principal to pay a certain fee to the trustee. This business scenario is also common in smart contracts.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.27. Rearrangement attack detection【Pass】

A reflow attack is an attempt by a miner or other party to "compete" with a smart contract participant by inserting their information into a list or mapping, giving an attacker the opportunity to store their information in a contract.

Detection results: After detection, there are no related vulnerabilities in the smart contract code.

Security advice: None.

7. Appendix A: Security Assessment of Contract Fund Management

Contract fund management		
The type of asset in the contract	**The function is involved**	**Security risks**
		SAFE

Check the security of the management of **digital currency assets** transferred by users in the business logic of the contract. Observe whether there are security risks that may cause the loss of customer funds, such as **incorrect recording, incorrect transfer, and backdoor** withdrawal of the **digital currency assets** transferred into the contract.

Official Website

www.knownseclab.com

E-mail

blockchain@knownsec.com