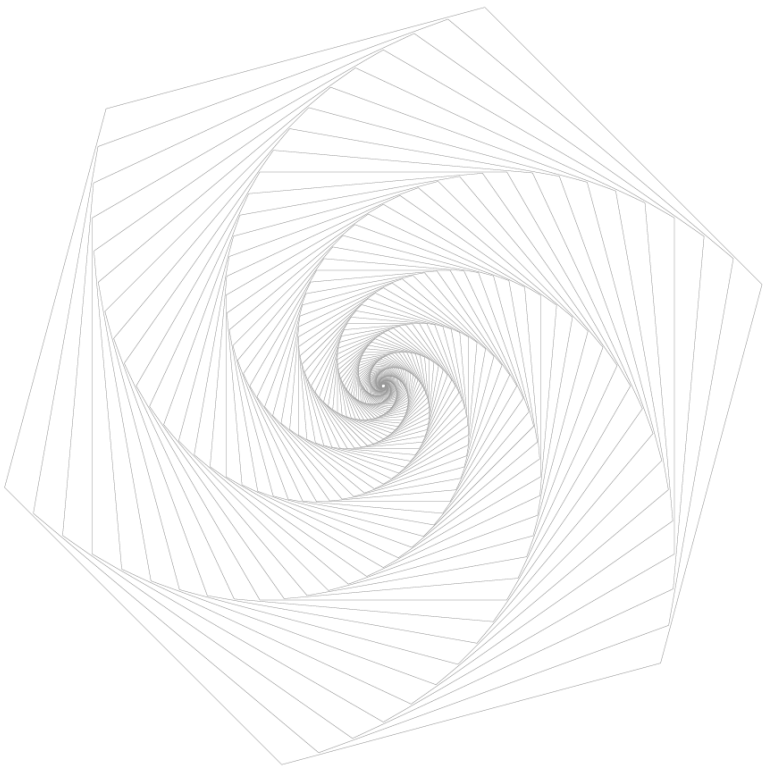
KNOWNSEC

Blockchain Lab

Smart Contract Audit Report



Version description

The revision	Date	Revised	Version
Write documentation	20221219	KNOWNSEC Blockchain Lab	V1.0

Document information

Title	Version	Document Number	Type
OpenOcean Cross_Chain Smart Contract Audit Report	V1.0	9bde5778e1c948bbb4eef805da46deff	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 **December 09, 2022 to December 19, 2022**. During this period, the security and standardization of **OpenOcean Cross_Chain smart contract** 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 OpenOcean Cross_Chain** 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	9bde5778e1c948bbb4eef805da46deff
report query link	https://attest.knownseclab.com/attestation/searchResult?query=9bde5778e1c948bbb4eef805da46deff

2. Item information

2.1. Item description

None.

2.2. The project's website

https://openocean.finance/

2.3. White Paper

https://docs.openocean.finance/

2.4. Review version code

https://bscscan.com/address/0x43c47b76d24ad1f73f5ab12442a016397a5ae9f6#code/

2.5. Contract file and Hash/contract deployment address

The contract documents	MD5
Address.sol	ff9034fc102e96ebb1194d21fc380c31
Context.sol	7093b039d79750162610c51cd75dfce8
errors.sol	cd001a150c54e7468329ab7f040f59f4
IERC20.sol	e6c2fff3f3c6cba4a3b4f06a3f6fec00
ImplBase.sol	5feda8b5bffa0a036f6d1a76beaf6d93
MiddlewareImplBase.sol	6dfbb6bb647ef9d429e73f6e49aae651
Ownable.sol	86d00d592d33469229ed5e5c8668a4bf

Registry.sol	01b5bbbe764e9ced78cf3ae2b429a2a3
SafeERC20.sol	52546a054058393497a7ca9992ed3991

3. External visibility analysis

3.1. Registry contract

Registry					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
outboundTransferTo	external	True	---	payable	
addRoutes	external	True	onlyOwner	---	
disableRoute	external	True	onlyOwner onlyExistingRoute(_routeId)	---	
rescueFunds	external	True	onlyOwner	---	

4. Code vulnerability analysis

4.1. Summary description of the audit results

Audit results			
audit project	**audit content**	**condition**	**description**
Business security detection	Outbound transfer function	Pass	After testing, there is no security issue.
	New routing function	Pass	After testing, there is no security issue.
	Rescue money function	Pass	After testing, there is no security issue.
	High authority issue	Pass	After testing, there is no security issue.
Code basic vulnerability detection	Compiler version security	Pass	After testing, there is no security issue.
	Redundant code	Pass	After testing, there is no security issue.
	Use of safe arithmetic library	Pass	After testing, there is no security issue.
	Not recommended encoding	Pass	After testing, there is no security issue.
	Reasonable use of require/assert	Pass	After testing, there is no security issue.
	fallback function safety	Pass	After testing, there is no security issue.
	tx.origin 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.
	Numerical overflow detection	Pass	After testing, there is no security issue.
	Arithmetic accuracy error	Pass	After testing, there is no security issue.
	Wrong use of random number detection	Pass	After testing, there is no security issue.
	Unsafe interface use	Pass	After testing, there is no security issue.
	Variable coverage	Pass	After testing, there is no security issue.
	Uninitialized storage pointer	Pass	After testing, there is no security issue.
	Return value call verification	Pass	After testing, there is no security issue.
	Transaction order dependency detection	Pass	After testing, there is no security issue.
	Timestamp dependent attack	Pass	After testing, there is no security issue.
	Denial of service attack detection	Pass	After testing, there is no security issue.
	Fake recharge vulnerability 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. Outbound transfer function【Pass】

Audit analysis: Conduct a security audit on the logic of the outbound transfer function in the contract Registry.sol, which is used to determine the content related to the outbound transfer of tokens across the chain bridge. Check the validity of the incoming parameters and whether there are logical errors in the outbound process.

```
function outboundTransferTo(UserRequest calldata _userRequest)
    external
    payable
    {
        require(_userRequest.amount != 0, MovrErrors.INVALID_AMT);

        // make sure bridge ID is not 0
        require( //knownsec// Cross-link bridge id is not 0
            _userRequest.bridgeRequest.id != 0,
            MovrErrors.INVALID_BRIDGE_ID
        );

        // make sure bridge input is provided
        require( //knownsec// Cross-link bridge input is not 0
            _userRequest.bridgeRequest.inputToken != address(0),
            MovrErrors.ADDRESS_0_PROVIDED
        );

        // load middleware info and validate
        RouteData memory middlewareInfo = routes[
        _userRequest.middlewareRequest.id
        ];
        require(
```

```
            middlewareInfo.route != address(0) &&
            middlewareInfo.isEnabled &&
            middlewareInfo.isMiddleware,
            MovrErrors.ROUTE_NOT_ALLOWED
        );

        // load bridge info and validate
        RouteData memory bridgeInfo = routes[_userRequest.bridgeRequest.id];
        require(
            bridgeInfo.route != address(0) &&
            bridgeInfo.isEnabled &&
            !bridgeInfo.isMiddleware,
            MovrErrors.ROUTE_NOT_ALLOWED
        );

        emit ExecutionCompleted(
            _userRequest.middlewareRequest.id,
            _userRequest.bridgeRequest.id,
            _userRequest.amount
        );

        // if middlewareID is 0 it means we dont want to perform a action before bridging
        // and directly want to move for bridging
        if (_userRequest.middlewareRequest.id == 0) { //knownsec// Determine if the user needs middleware
            // perform the bridging
            ImplBase(bridgeInfo.route).outboundTransferTo{value: msg.value}(
                _userRequest.amount,
                msg.sender,
                _userRequest.receiverAddress,
                _userRequest.bridgeRequest.inputToken,
                _userRequest.toChainId,
                _userRequest.bridgeRequest.data
```

```
        );
        return;
    }

    // we first perform an action using the middleware
    // we determine if the input asset is a native asset, if yes we pass
    // the amount as value, else we pass the optionalNativeAmount
    uint256 _amountOut = MiddlewareImplBase(middlewareInfo.route)
    .performAction{
    value: _userRequest.middlewareRequest.inputToken ==
    NATIVE_TOKEN_ADDRESS
    ? _userRequest.amount +
    _userRequest.middlewareRequest.optionalNativeAmount
    : _userRequest.middlewareRequest.optionalNativeAmount
    }(
        msg.sender,
        _userRequest.middlewareRequest.inputToken,
        _userRequest.amount,
        address(this),
        _userRequest.middlewareRequest.data
    );

    // we mutate this variable if the input asset to bridge Impl is NATIVE
    uint256 nativeInput = _userRequest.bridgeRequest.optionalNativeAmount;

    // if the input asset is ERC20, we need to grant the bridge implementation approval
    if (_userRequest.bridgeRequest.inputToken != NATIVE_TOKEN_ADDRESS) {
        IERC20(_userRequest.bridgeRequest.inputToken).safeIncreaseAllowance(
            bridgeInfo.route,
            _amountOut
        );
    } else {
        // if the input asset is native we need to set it as value
```

```
            nativeInput =
            _amountOut +
            _userRequest.bridgeRequest.optionalNativeAmount;
        }

        // send off to bridge
        ImplBase(bridgeInfo.route).outboundTransferTo{value: nativeInput}(
            _amountOut,
            address(this),
            _userRequest.receiverAddress,
            _userRequest.bridgeRequest.inputToken,
            _userRequest.toChainId,
            _userRequest.bridgeRequest.data
        );
        }
```

Security advice: None.

5.2. New routing function【Pass】

Audit analysis: Security audit of the logic of the new routing function in the contract Registry.sol, which is used to add routes for the contract owner. Checking the validity of incoming parameters, adding process for logical errors, permission verification issues.

```
function addRoutes(RouteData[] calldata _routes)
    external
    onlyOwner
    returns (uint256[] memory)
    { //knownsec// Only the Owner can add Routers
        require(_routes.length != 0, MovrErrors.EMPTY_INPUT);
```

```
        uint256[] memory _routeIds = new uint256[](_routes.length);
        for (uint256 i = 0; i < _routes.length; i++) {
            require(
                _routes[i].route != address(0),
                MovrErrors.ADDRESS_0_PROVIDED
            );
            routes.push(_routes[i]);
            _routeIds[i] = routes.length - 1;
            emit NewRouteAdded(
                i,
                _routes[i].route,
                _routes[i].isEnabled,
                _routes[i].isMiddleware
            );
        }

        return _routeIds;
        }
```

Security advice: None.

5.3. Rescue money function【Pass】

Audit analysis: Conduct a security audit on the logic of the rescue fund function in the contract Registry.sol, which is used to refund the tokens transferred in by mistake. Check the validity of incoming parameters, add process for logical errors, permission verification issues.

```
function rescueFunds(
        address _token,
        address _receiverAddress,
        uint256 _amount
```

```
) external onlyOwner { //knownsec// Rescue money
        IERC20(_token).safeTransfer(_receiverAddress, _amount);
}
```

Security advice: None.

5.4. High authority issue【Pass】

Audit analysis: Security audit of each function logic in the contract Registry.sol, found that rescueFunds, disableRoute, addRoutes all exist only owner permissions operability, there is a high risk of permissions, the official added multi-signature so the risk is lifted.

```
function addRoutes(RouteData[] calldata _routes)
    external
    onlyOwner
    returns (uint256[] memory)
    { //knownsec// Only the Owner can add Routers
        require(_routes.length != 0, MovrErrors.EMPTY_INPUT);
        uint256[] memory _routeIds = new uint256[](_routes.length);
        for (uint256 i = 0; i < _routes.length; i++) {
            require(
                _routes[i].route != address(0),
                MovrErrors.ADDRESS_0_PROVIDED
            );
            routes.push(_routes[i]);
            _routeIds[i] = routes.length - 1;
            emit NewRouteAdded(
                i,
                _routes[i].route,
                _routes[i].isEnabled,
```

```
                _routes[i].isMiddleware
            );
        }

        return _routeIds;
    }

    ///@notice disables the route    if required.
    function disableRoute(uint256 _routeId)
    external
    onlyOwner
    onlyExistingRoute(_routeId)
    { //knownsec// Close an existing Router
        routes[_routeId].isEnabled = false;
        emit RouteDisabled(_routeId);
    }

    function rescueFunds(
        address _token,
        address _receiverAddress,
        uint256 _amount
    ) external onlyOwner { //knownsec// Rescue money
        IERC20(_token).safeTransfer(_receiverAddress, _amount);
    }
```

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 0.8.3 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: Selective removal of redundant code.

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**
User pledged token assets	buyMTT、sellMTT	**PASS**

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