BEOSIN
Blockchain Security

BunnyfiBridge

Smart Contract Security Audit

No. 202402041153

Feb 4th, 2024

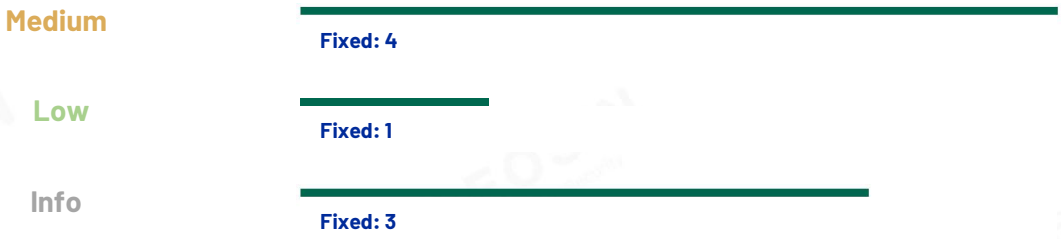
SECURING BLOCKCHAIN ECOSYSTEM

WWW.BEOSIN.COM

Contents

Summary of Audit Results

After auditing, 4 Medium-risk ,1 Low-risk and 3 Info items were identified in the BunnyfiBridge project. Specific audit details will be presented in the Findings section. Users should pay attention to the following aspects when interacting with this project:



- **Risk Description:**

The scope of this audit only includes the BunnyfiBridgeView contract. Other contracts in this project are not within the scope of this audit, and users should be cautious about the risks associated with interactions with other contracts.

- **Project Description:**

Business overview

This audit only covers the BunnyfiBridgeView contract file, and the following descriptions are based on this contract.

BunnyfiBridgeView is a cross-chain project where users can transfer ERC20 and native platform token assets to the BunnyfiBridgeView contract. The project's owner has the authority to modify the corresponding BridgeIdInfo, managing the router, asset, and bridgestatus states within it. The owner can also batch-add users to the blacklist by authorizing the ProxyAdmins address for signature verification. Afterward, the blacklisting status can be restored through the `setUserBlackStatus` function. In the `setUserBlackStatus` function, signature verification is required to make corresponding modifications to the blacklist status.

Additionally, the owner has the capability to transfer ERC20 and platform token assets within the contract to facilitate cross-chain operations.

1 Overview

1.1 Project Overview

Project Name	BunnyfiBridge
Project Language	Solidity
Platform	Ethereum, Arbitrum, Optimism
File Hash(SHA-1)	BunnyfiBridgeView: 35c303c3a0988ad3bc55621b33109d00815018bc(Initial) f244a80975f802d2c4d9d5f445510aa3d41185fe 87e0a070f873928f5f3af60f81a9d255673c37d9 e16b07173503038b64e6288e2dd6c0d6a4ceb787 b4146874f72f8e067d4bb46b848e066fda5a9b4c 64b30048c65a5259602dd314b5371d0a893756f9(Final)

1.2 Audit Overview

Audit work duration: Feb 1, 2024 – Feb 4, 2024

Audit team: Beosin Security Team

1.3 Audit Method

The audit methods are as follows:

1. Formal Verification

Formal verification is a technique that uses property-based approaches for testing and verification. Property specifications define a set of rules using Beosin's library of security expert rules. These rules call into the contracts under analysis and make various assertions about their behavior. The rules of the specification play a crucial role in the analysis. If the rule is violated, a concrete test case is provided to demonstrate the violation.

2. Manual Review

Using manual auditing methods, the code is read line by line to identify potential security issues. This ensures that the contract's execution logic aligns with the client's specifications and intentions, thereby safeguarding the accuracy of the contract's business logic.

The manual audit is divided into three groups to cover the entire auditing process:

The Basic Testing Group is primarily responsible for interpreting the project's code and conducting comprehensive functional testing.

The Simulated Attack Group is responsible for analyzing the audited project based on the collected historical audit vulnerability database and security incident attack models. They identify potential attack vectors and collaborate with the Basic Testing Group to conduct simulated attack tests.

The Expert Analysis Group is responsible for analyzing the overall project design, interactions with third parties, and security risks in the on-chain operational environment. They also conduct a review of the entire audit findings.

3. Static Analysis

Static analysis is a method of examining code during compilation or static analysis to detect issues. Beosin-VaaS can detect more than 100 common smart contract vulnerabilities through static analysis, such as reentrancy and block parameter dependency. It allows early and efficient discovery of problems to improve code quality and security.

2 Findings

Index	Risk description	Severity level	Status
BunnyfiBridge-01	The contract lacks a function for withdrawing ERC20 tokens	Medium	Fixed
BunnyfiBridge-02	The contract lacks an initialization function for permissions	Medium	Fixed
BunnyfiBridge-03	Potential risk of contract self-destruction	Medium	Fixed
BunnyfiBridge-04	Checking the result of token transfers	Medium	Fixed
BunnyfiBridge-05	Signature is missing nonce and chainid	Low	Fixed
BunnyfiBridge-06	The contract lacks event triggering	Info	Fixed
BunnyfiBridge-07	Name specification	Info	Fixed
BunnyfiBridge-08	The redundant code	Info	Fixed

Finding Details:

[BunnyfiBridge-01] The contract lacks a function for withdrawing ERC20 tokens

Severity Level	Medium
Type	Business Security
Lines	BunnyfiBridge.sol #L50-61
Description	By examining the `getAssetBalance` function and its code context, it is evident that the contract is designed to store ERC20 token assets. However, the contract only implements a `withdraw` function for the native platform coin, and lacks a `withdrawal` function for ERC20 assets. This results in the ERC20 tokens being locked in the contract with no means of transfer.

```
    function getAssetBalance(uint256 id)
        public
        view
        returns (uint256[] memory assetbalance)
    {
        assetbalance = new uint256[](BridgeId[id].asset.length);
        for (uint256 i = 0; i < BridgeId[id].asset.length; i++) {
            assetbalance[i] = IERC20(BridgeId[id].asset[i]).balanceOf(
                address(this)
            );
        }
    }

    function withdraw() external onlyOwner {
        payable(msg.sender).transfer(payable(address(this)).balance);
    }
```

Recommendation	It is recommended to add a function for withdrawing ERC20 tokens.
Status	**Fixed.**

```
    function withdrawOtherTokens( address wtoken,address to, uint256 amount) external onlyOwner {
        SafeERC20.safeTransfer(IERC20(wtoken), to, amount);
        emit WithdrawOtherTokens(msg.sender, wtoken, to, amount);
    }
```

[BunnyfiBridge-02] The contract lacks an initialization function for permissions

Severity Level	Medium
Type	Business Security
Lines	BunnyfiBridge.sol #L9-19
Description	As an implementation contract, the contract lacks an initialization function for the owner's permission, resulting in the owner address being consistently set to 0 after deployment and unable to be changed. This prevents the implementation of permission-related functionalities.

```
contract BunnyfiBridgeView is Initializable, OwnableUpgradeable {
    using SafeMath for uint256;

    mapping(address => bool) public ProxyAdmins;

    struct BridgeIdInfo {
        uint256 chainid;
        address[] rounter;
        address[] asset;
        bool bridgestatus;
    }
```

Recommendation	If not using a proxy, it is recommended to add an `initialization` function for the owner. If using a proxy, consider calling the `_disableInitializers` function in the constructor and implementing the initialization function for the owner.
Status	**Fixed.** The project team has abandoned the use of the proxy pattern and opted to manage permissions by inheriting the Ownable contract.

```
contract BunnyfiBridgeView is Ownable {
    using SafeMath for uint256;

    mapping(address => bool) public ProxyAdmins;

    struct BridgeIdInfo {
        address[] rounter;
        address[] asset;
        bool bridgestatus;
    }
```

[BunnyfiBridge-03] Potential risk of contract self-destruction

Severity Level	**Medium**
Type	Business Security
Lines	BunnyfiBridge.sol #L155-159

Description	The contract contains a `selfDestruct` function, granting the contract owner the authority to destroy the contract.
	`function selfDestruct() external onlyOwner {`
	`selfdestruct(payable(owner()));`
	`}`

Recommendation	It is recommended to remove this function.
Status	**Fixed.**

[BunnyfiBridge-04] Checking the result of token transfers

Severity Level	Medium
Type	Business Security
Lines	BunnyfiBridge.sol #L145-156
Description	On some chains, especially for ERC20 tokens that do not provide a return value (such as USDT on the ETH chain), using the `transfer` function for transfers does not provide a `true/false` return value. This will result in an exception during the transfer, causing the transfer to fail and leading to the corresponding assets being locked in the contract.

```
    function withdrawOtherTokens(
        address wtoken,
        address to,
        uint256 amount
    ) external onlyOwner returns (bool sent) {
        require(
            to != address(this) && to != address(0),
            "Error target address"
        );
        uint256 _contractBalance =
IERC20(wtoken).balanceOf(address(this));
        if (amount > 0 && amount <= _contractBalance) {
            sent = IERC20(wtoken).transfer(to, amount);
        } else {
            return false;
        }
    }
```

Recommendation	It is recommended to use the `safeTransfer` function instead of `transfer` for token transfers to avoid transfer exceptions caused by the lack of a return value (for example, USDT on ETH).
Status	**Fixed.**

```
    function withdrawOtherTokens( address wtoken,address to,
uint256 amount) external onlyOwner {

        SafeERC20.safeTransfer(IERC20(wtoken), to, amount);

        emit WithdrawOtherTokens(msg.sender, wtoken, to,
```

```
amount);
    }
```

[BunnyfiBridge-05] Signature is missing nonce and chainid

Severity Level	Low
Type	Business Security
Lines	BunnyfiBridge.sol #L78-117
Description	If the BunnyfiBridge contract is deployed on multiple chains and configured with the same ProxyAdmins, there is a potential issue of signature reuse across these chains. With a single signature, a user can be blacklisted repeatedly, and this action can be executed across multiple chains.

```
    function setUserBlack(address user, bytes memory signature)
external {
        // check
        bytes32 hash = keccak256(abi.encodePacked(msg.sender, user));
        bytes32 message = ECDSA.toEthSignedMessageHash(hash);
        address receivedAddress = ECDSA.recover(message, signature);
        require(
            receivedAddress != address(0) &&
                ProxyAdmins[receivedAddress] == true,
            "signer error"
        );
        UserBlackList[user] = true;
        emit SetUserBlack(msg.sender);
    }

    function setUserBlackList(address[] memory users, bytes memory
signature)
        external
    {
        // check
        bytes32 hash = keccak256(abi.encodePacked(msg.sender,
users));
        bytes32 message = ECDSA.toEthSignedMessageHash(hash);
        address receivedAddress = ECDSA.recover(message, signature);
        require(
            receivedAddress != address(0) &&
                ProxyAdmins[receivedAddress] == true,
            "signer error"
        );
        for (uint256 i = 0; i < users.length; i++) {
```

```
            UserBlackList[users[i]] = true;
        }
        emit SetUserBlack(msg.sender);
    }
```

Recommendation	It is recommended to include nonce and chainid for verification in the signature.
Status	**Fixed.**

```
    function setUserBlackStatus(
        address user,
        bool status,
        bytes memory signature
    ) external {
        // check
        bytes32 hash = keccak256(
            abi.encodePacked(
                msg.sender,
                user,
                status,
                _useNonce(msg.sender),
                block.chainid
            )
        );
        address receivedAddress = ECDSA.recover(hash, signature);
        require(
            receivedAddress != address(0) &&
                ProxyAdmins[receivedAddress] == true,
            "signer error"
        );
        require(UserBlackList[user] != status , "Error Set User
Status");
        UserBlackList[user] = status;
        emit SetUserBlackStatus(
            msg.sender,
            _useNonce(msg.sender),
            block.chainid,
            user,
            status
        );
```

```
    }

    function setUserBlackList(address[] memory users, bytes memory
signature)
        external
    {
        // check
        bytes32 hash = keccak256(
            abi.encodePacked(
                msg.sender,
                users,
                _useNonce(msg.sender),
                block.chainid
            )
        );
        address receivedAddress = ECDSA.recover(hash, signature);
        require(
            receivedAddress != address(0) &&
                ProxyAdmins[receivedAddress] == true,
            "signer error"
        );
        for (uint256 i = 0; i < users.length; i++) {
            UserBlackList[users[i]] = true;
        }
        emit SetUserBlack(msg.sender, _useNonce(msg.sender),
block.chainid);
    }
```

[BunnyfiBridge-06] The contract lacks event triggering

Severity Level	Info
Type	Coding Conventions
Lines	BunnyfiBridge.sol #L78-117
Description	In the `setIdInfoByOwner` function, the corresponding event is not triggered after the owner modifies the BridgeId information. Similarly, in the `setUserBlack` and `setUserBlackList` functions, no events are triggered after setting `UserBlackList` to true.

```
    function setIdInfoByOwner(uint256 id, BridgeIdInfo memory info)
        external
        onlyOwner
    {
        BridgeId[id] = info;
    }

    function setUserBlack(address user, bytes memory signature)
external {
        // check
        bytes32 hash = keccak256(abi.encodePacked(msg.sender, user));
        bytes32 message = ECDSA.toEthSignedMessageHash(hash);
        address receivedAddress = ECDSA.recover(message, signature);
        require(
            receivedAddress != address(0) &&
                ProxyAdmins[receivedAddress] == true,
            "signer error"
        );
        UserBlackList[user] = true;
        emit SetUserBlack(msg.sender);
    }

    function setUserBlackList(address[] memory users, bytes memory
signature)
        external
    {
        // check
        bytes32 hash = keccak256(abi.encodePacked(msg.sender,
users));
```

```
        bytes32 message = ECDSA.toEthSignedMessageHash(hash);
        address receivedAddress = ECDSA.recover(message, signature);
        require(
            receivedAddress != address(0) &&
                ProxyAdmins[receivedAddress] == true,
            "signer error"
        );
        for (uint256 i = 0; i < users.length; i++) {
            UserBlackList[users[i]] = true;
        }
        emit SetUserBlack(msg.sender);
    }
```

Recommendation It is recommended to add relevant events and trigger them in the corresponding functions.

Status **Fixed.**

```
    function setUserBlackStatus(
        address user,
        bool status,
        bytes memory signature
    ) external {
        // check
        bytes32 hash = keccak256(
            abi.encodePacked(
                msg.sender,
                user,
                status,
                _useNonce(msg.sender),
                block.chainid
            )
        );
        address receivedAddress = ECDSA.recover(hash, signature);
        require(
            receivedAddress != address(0) &&
                ProxyAdmins[receivedAddress] == true,
            "signer error"
        );
        require(UserBlackList[user] != status, "Error Set User
Status");
        UserBlackList[user] = status;
```

```
        emit SetUserBlackStatus(
            msg.sender,
            _useNonce(msg.sender),
            block.chainid,
            user,
            status
        );
    }

    function setUserBlackList(address[] memory users, bytes memory
signature)
        external
    {
        // check
        bytes32 hash = keccak256(
            abi.encodePacked(
                msg.sender,
                users,
                _useNonce(msg.sender),
                block.chainid
            )
        );
        address receivedAddress = ECDSA.recover(hash, signature);
        require(
            receivedAddress != address(0) &&
                ProxyAdmins[receivedAddress] == true,
            "signer error"
        );
        for (uint256 i = 0; i < users.length; i++) {
            UserBlackList[users[i]] = true;
            emit SetUserBlackStatus(
                msg.sender,
                _useNonce(msg.sender),
                block.chainid,
                users[i],
                true
            );
        }
        emit SetUserBlack(msg.sender, _useNonce(msg.sender),
```

```
block.chainid);
    }
```

[BunnyfiBridge-07] Name specification

Severity Level	Info
Type	Coding Conventions
Lines	BunnyfiBridge.sol #L37-43
Description	Using the chainid name is more in line with the current business logic. In the `getIdInfoRouter` function, the term rounter does not have a corresponding English meaning. The literal meaning of `receivedAddress` is the address for receiving token assets, which is inconsistent with the actual address being verified in the function logic.

```
    struct BridgeIdInfo {
        address[] rounter;
        address[] asset;
        bool bridgestatus;
    }

    function getChainIdInfoRounter(uint256 chainid)
        public
        view
        returns (address[] memory router)
    {
        router = BridgeId[chainid].router;
    }
    function getIdInfoAsset(uint256 id)
        public
        view
        returns (address[] memory asset)
    {
        asset = BridgeId[id].asset;
    }

    function setUserBlackStatus(
        address user,
        bool status,
        bytes memory signature
    ) external {
        // check
        bytes32 hash = keccak256(
```

```
            abi.encodePacked(
                msg.sender,
                user,
                status,
                _useNonce(msg.sender),
                block.chainid
            )
        );
        address receivedAddress = ECDSA.recover(hash, signature);
        require(
            receivedAddress != address(0) &&
                ProxyAdmins[receivedAddress] == true,
            "signer error"
        );
```

Recommendation It is recommended to change the id to chainid. Replace the corresponding word.

Status **Fixed.**

```
    struct BridgeIdInfo {
        address[] router;
        address[] asset;
        bool bridgestatus;
    }
    function getChainIdInfoRouter(uint256 chainid)
        public
        view
        returns (address[] memory router)
    {
        router = BridgeId[chainid].router;
    }

    function setUserBlackStatus(
        address user,
        bool status,
        bytes memory signature
    ) external {
        // check
        bytes32 hash = keccak256(
            abi.encodePacked(
                msg.sender,
```

```
                user,
                status,
                _useNonce(msg.sender),
                block.chainid
            )
        );
        address adminAddress = ECDSA.recover(hash, signature);
        require(
            adminAddress != address(0) &&
                ProxyAdmins[adminAddress] == true,
            "signer error"
        );
```

[BunnyfiBridge-08] The redundant code

Severity Level	Info
Type	Coding Conventions
Lines	BunnyfiBridge.sol #L138
Description	The triggering of the event here is redundant, as the parameters are duplicated from the loop above.

```
    function setUserBlackList(address[] memory users, bytes memory
signature)
        external
    {
        // check
        bytes32 hash = keccak256(
            abi.encodePacked(
                msg.sender,
                users,
                _useNonce(msg.sender),
                block.chainid
            )
        );
        address receivedAddress = ECDSA.recover(hash, signature);
        require(
            receivedAddress != address(0) &&
                ProxyAdmins[receivedAddress] == true,
            "signer error"
        );
        for (uint256 i = 0; i < users.length; i++) {
            UserBlackList[users[i]] = true;
            emit SetUserBlackStatus(
                msg.sender,
                _useNonce(msg.sender),
                block.chainid,
                users[i],
                true
            );
        }
        emit SetUserBlack(msg.sender, _useNonce(msg.sender),
block.chainid);
```

```
    }
```

Recommendation It is recommended to remove the event here to avoid code redundancy.

Status **Fixed.**

```
    function setUserBlackList(address[] memory users, bytes memory
signature)
        external
    {
        // check
        bytes32 hash = keccak256(
            abi.encodePacked(
                msg.sender,
                users,
                _useNonce(msg.sender),
                block.chainid
            )
        );
        address adminAddress = ECDSA.recover(hash, signature);
        require(
            adminAddress != address(0) &&
                ProxyAdmins[adminAddress] == true,
            "signer error"
        );
        for (uint256 i = 0; i < users.length; i++) {
            UserBlackList[users[i]] = true;
            emit SetUserBlackStatus(
                msg.sender,
                _useNonce(msg.sender),
                block.chainid,
                users[i],
                true
            );
        }
    }
```

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1(Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Likelihood \ Impact	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	Medium	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

3.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

3.1.4 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

3.1.5 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

3.2 Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
		Third-party Protocol Interface Consistency
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Asset Tradability
		Arbitrage Attack

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

3.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in blockchain.

3.4 About Beosin

Beosin is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions. Beosin has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, Beosin has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Blockchain Security

Official Website
https://www.beosin.com

Telegram
https://t.me/beosin

Twitter
https://twitter.com/Beosin_com

Email
service@beosin.com