BEOSIN
Blockchain Security

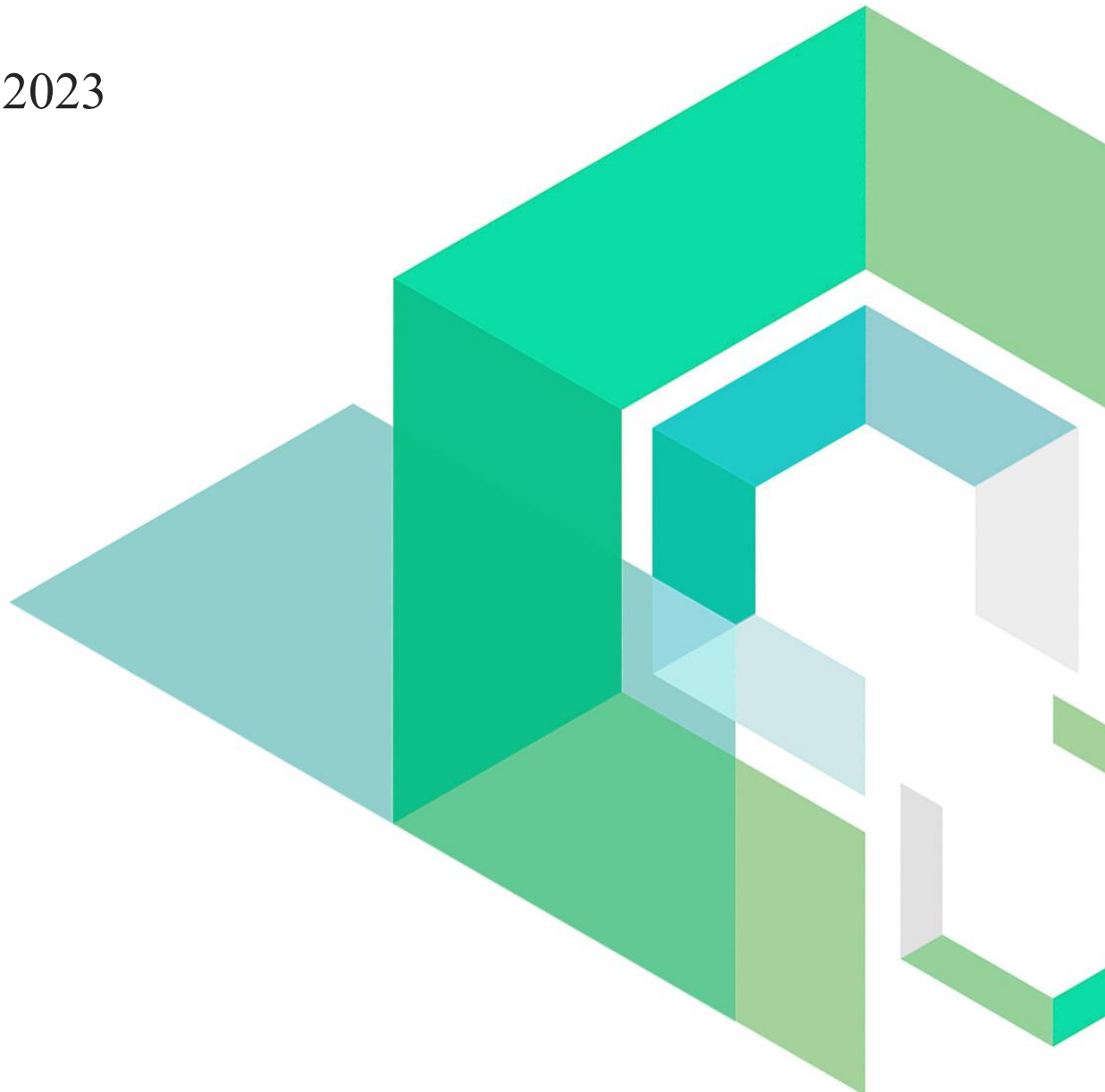
# Lighthouse

Smart Contract Security Audit

V1.0

No. 202301160945

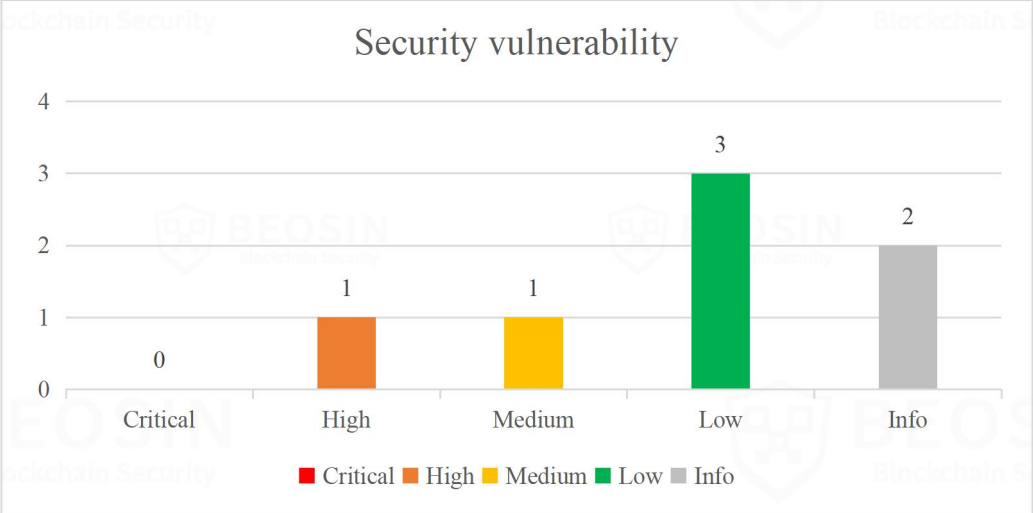
Jan 16th, 2023



# Contents

# Summary of Audit Results

**After auditing, 1 High, 1 Medium, 3 Low and 2 Info-risk items were identified in the Lighthouse project.** Specific audit details will be presented in the **Findings** section. Users should pay attention to the following aspects when interacting with this project:



- **Project Description:**

1. **Business overview**

This audit involves billing, bridge, core, dao, exchange, and nft items. The project can use Stargate to transfer native assets across blockchains, enabling a 1:1 swap of native assets across different chains. For example, users can swap USDC on Ethereum for USDT on BNB.

In the Billing contract, the owner can record system subscription information and can also deactivate system subscriptions for a given id. Owner can add the stable tokens allowed by this contract (including the exchange rate and activation status of the tokens). The contract has a contingency function where the owner can add block numbers to match the growth of the block chain (the subscription time limit is in block numbers). The owner can authorise the balance of the specified token contract to a specified address. In this contract, the user adds himself to an active subscription plan (for which a subscription fee is payable) and can check whether there are active subscriptions in the user's account, with the option to renew or cancel the subscription, depending on the user's needs.

In the Bridger contract, the main function is to enable cross-chain exchange. In this contract, the owner can change the StargateRouter address, add a new asset or change a current asset and the balance of a given token contract can be withdrawn to prevent tokens from being locked in the contract. The *swap* function can be called to exchange tokens across chains (subject to a cross-chain exchange fee). In this contract, only the sargateRouter can call the *sgReceive* function to transfer tokens.

In the core item, there are two contracts, the DepositManager contract, and the Lighthouse contract. In DepositManager, the main purpose is to implement the management of storage space and storage fees. Owners can add or remove tokens (involving token exchange rates), set the initial storage size for new users, set the priceFeed, and whitelist specified accounts. The administrator can adjust storage fees (involving storage fees for platform tokens and stablecoins) and update the user's available storage space. In this contract, the owner can also authorise the amount of tokens that can be spent on a specified account, and can transfer the balance of a specified token contract to a specified wallet address, and can also transfer platform tokens to a specified address. In the Lighthouse contract, only the owner has permission to publish the storage status.

In the exchange item, there are two contracts, of which in the SushiSwap contract, there are three exchange functions that enable the exchange of WETH for tokens. In the UniswapV3 contract, the function of exchanging platform tokens for stable tokens is possible.

# 1 Overview

## 1.1 Project Overview

| | |
|---|---|
| **Project Name** | Lighthouse |
| **Platform** | BNB Chain、Ethereum、Polygon、Filecoin |
| **Audit scope** | https://github.com/Lighthouse-web3/contracts |
| **Commit Hash** | ca0b022e7a13298f4ccc46367b24a6b1fa961ca8<br>195138f70b4e6d73bbd23b3bb8a956b24b3d83f6<br>6a33bb84642f6e4a7477123adcb76f839c08727d<br>50e98e2ec85a02080f86608d1f71514d50ae52da<br>840b59ab651cb089bd96ea3ef1fb8302248fd745<br>87364b57cc0ab81f4078750116a4ea8d2a20392a<br>06888014cfb133debc8165f932ea1ca7b7188780 |

## 1.2 Audit Overview

Audit work duration: Jan 3, 2023 – Jan 16, 2023

Audit methods: Formal Verification, Static Analysis, Typical Case Testing and Manual Review.

Audit team: Beosin Security Team.

# 2 Findings

| Index | Risk description | Severity level | Status |
|---|---|---|---|
| Lighthouse-1 | No restrictions on access to the *swapTokenToToken* function | High | Fixed |
| Lighthouse -2 | Misjudgement of subscription time | Medium | Fixed |
| Lighthouse -3 | Storage price calculation error in StorageRequest event trigger | Low | Fixed |
| Lighthouse -4 | The *swapExactEthToToken* function return value error | Low | Fixed |
| Lighthouse -5 | Missing judgement on cid parameter | Low | Acknowledged |
| Lighthouse -6 | Redundant code | Info | Fixed |
| Lighthouse -7 | Lack of event triggers | Info | Fixed |

**Status Notes:**

1. Lighthouse -5 is not fixed and may not cause any issue.

# Finding Details:

## [Lighthouse-1] No restrictions on access to the *swapTokenToToken* function

| | |
|---|---|
| **Severity Level** | **High** |
| **Type** | Business Security |
| **Lines** | SushiSwap.sol #L76-91 |
| **Description** | For the *swapTokenToToken* function, we think there is a problem here. The original function is designed to authorize the contract and then call the exchange function for normal exchange, but the actual authorization is for the user, and when the exchange is done, the token spent is not from the contract, but from the user's account. |

```
76    function swapTokenToToken(
77        address tokenAddress,
78        address to,
79        uint256 amountIn,
80        uint256 amountOut,
81        uint256 deadline
82    ) public returns (uint256) {
83        assert(tokenSupported[tokenAddress]);
84        address[] memory path = new address[](2);
85        path[0] = router.WETH();
86        path[1] = tokenAddress;
87        IWETH(router.WETH()).approve(address(router), amountIn);
88
89        uint256[] memory amounts = router.swapExactTokensForTokens(amountIn, amountOut, path, to, deadline);
90        return amounts[0];
91    }
```

Figure 1 Source code of *swapTokenToToken* function (unfixed)

| | |
|---|---|
| **Recommendations** | It is recommended that permission controls be added to calls to this function. |
| **Status** | Fixed. |

```
76    function swapTokenToToken(
77        address tokenAddress,
78        address to,
79        uint256 amountIn,
80        uint256 amountOut,
81        uint256 deadline
82    ) public  onlyOwner returns  (uint256[] memory) {
83        assert(tokenSupported[tokenAddress]);
84        address[] memory path = new address[](2);
85        path[0] = router.WETH();
86        path[1] = tokenAddress;
87        IWETH(router.WETH()).approve(address(router), amountIn);
88
89        return router.swapExactTokensForTokens(amountIn, amountOut, path, to, deadline);
90    }
```

Figure 2 Source code of *swapTokenToToken* function (fixed)

## [Lighthouse-2] Misjudgement of subscription time

| | |
|---|---|
| **Severity Level** | Medium |
| **Type** | Business Security |
| **Lines** | Billing.sol #L136-158 |
| **Description** | In the *purchaseSubscription* function, it needs to check if the next subscription has reached its expiry date, but the wrong frequencyOfDeduction variable is used, which should be compared with nextDeductionInNumOfBlocks. |

```
136    function purchaseSubscription(address account) internal returns (bool) {
137        UserSubscription storage subscription = userToSubscription[account];
138        require(subscription.occuranceLeft > 0, "subscription expired or doesn't exist");
139        require(
140            block.number - subscription.lastDebit >
141                contractSubscriptions[subscription.systemDefinedSubscriptionID].frequencyOfDeduction
142        );
143        if (
144            IERC20MetadataUpgradeable(subscription.tokenAddress).balanceOf(_msgSender()) <
145            getAmountToBeDeducted(subscription.tokenAddress, subscription.systemDefinedSubscriptionID)
146        ) {
147            return false;
148        }
149        IERC20Upgradeable(subscription.tokenAddress).transferFrom(
150            account,
151            address(this),
152            getAmountToBeDeducted(subscription.tokenAddress, subscription.systemDefinedSubscriptionID)
153        );
154        subscription.occuranceLeft = subscription.occuranceLeft - 1;
155        subscription.lastDebit = uint96(block.number);
156        emit Purchase(account, subscription.systemDefinedSubscriptionID, subscription.tokenAddress);
157        return true;
158    }
```

Figure 3 Source code of *purchaseSubscription* function (unfixed)

| | |
|---|---|
| **Recommendations** | It is recommended to replace 'frequencyOfDeduction' with 'nextDeductionInNumOfBlock'. |
| **Status** | Fixed. |

```
136    function purchaseSubscription(address account) internal returns (bool) {
137        UserSubscription storage subscription = userToSubscription[account];
138        require(subscription.occuranceLeft > 0, "subscription expired or doesn't exist");
139        require(
140            block.number - subscription.lastDebit >
141                contractSubscriptions[subscription.systemDefinedSubscriptionID].nextDeductionInNumOfBlocks
142        );
143        if (
144            IERC20MetadataUpgradeable(subscription.tokenAddress).balanceOf(_msgSender()) <
145            getAmountToBeDeducted(subscription.tokenAddress, subscription.systemDefinedSubscriptionID)
146        ) {
147            return false;
148        }
149        IERC20Upgradeable(subscription.tokenAddress).transferFrom(
150            account,
151            address(this),
152            getAmountToBeDeducted(subscription.tokenAddress, subscription.systemDefinedSubscriptionID)
153        );
154        subscription.occuranceLeft = subscription.occuranceLeft - 1;
155        subscription.lastDebit = uint96(block.number);
156        emit Purchase(account, subscription.systemDefinedSubscriptionID, subscription.tokenAddress);
157        return true;
158    }
```

Figure 4 Source code of *purchaseSubscription* function (fixed)

## [Lighthouse-3] Storage price calculation error in StorageRequest event trigger

| | |
|---|---|
| **Severity Level** | Low |
| **Type** | Business Security |
| **Lines** | Lighthouse.sol #L63-80<br>DepositManager.sol #L74-78<br>Lighthouse.sol #L28-36 |
| **Description** | In the lighthouse contract, the fourth parameter printed to the StorageRequest event triggered by the *store* function has a miscalculation of the file storage price. The _costOfStorage variable is assigned to the storage space size and should use the storage price multiplied by the file size. |

```
63      function store(
64          string calldata cid,
65          string calldata config,
66          string calldata fileName,
67          uint256 fileSize
68      ) external {
69          uint256 currentTime = block.timestamp;
70          Deposit.updateStorage(msg.sender, fileSize, cid);
71          emit StorageRequest(
72              msg.sender,
73              cid,
74              config,
75              Deposit.costOfStorage() * fileSize,
76              fileName,
77              fileSize,
78              currentTime
79          );
80      }
```

Figure 5 Source code of *store* function in the Lighthouse contract (unfixed)

```
74  function initialize() public initializer {
75      __Ownable_init();
76      __UUPSUpgradeable_init();
77      _costOfStorage = 214748365; // Byte per Dollar in these case 1gb/5$  which is eqivalent too ((1024**3) / 5)
78  }
```

Figure 6 Source code of *initialize* function in the DepositManager contract

```
28      event StorageRequest(
29          address indexed uploader,
30          string cid,
31          string config,
32          uint256 fileCost,
33          string fileName,
34          uint256 timestamp
35      );
36
```

Figure 7 Source code of StorageRequest event in the Lighthouse contract

| | |
|---|---|
| **Recommendations** | It is recommended to use the *getStorageCost* function for storage price calculation. |
| **Status** | Fixed. |

```
68      function store(
69          string calldata cid,
70          string calldata config,
71          string calldata fileName,
72          uint256 fileSize
73      ) external {
74          uint256 currentTime = block.timestamp;
75          Deposit.updateStorage(msg.sender, fileSize, cid);
76          emit StorageRequest(msg.sender, cid, config, Deposit.getStorageCost(fileSize), fileName, fileSize, currentTime);
77      }
```

Figure 8 Source code of *store* function in the Lighthouse contract (fixed)

## [Lighthouse-4] The *swapExactEthToToken* function return value error

| | |
|---|---|
| **Severity Level** | Low |
| **Type** | Business Security |
| **Lines** | SushiSwap.sol #L93-106 |
| **Description** | In the SushiSwap contract, there are three exchange functions, among which the *swapExactEthToToken* function is WETH to tokens, for example, 100WETH to 200 tokens, what we want to know here is the number of tokens exchanged, not the number of WETH, so here amounts[1] should be returned instead of amounts[0]. |

```
93    function swapExactEthToToken(
94        address tokenAddress,
95        address to,
96        uint256 amountOut,
97        uint256 deadline
98    ) public payable returns (uint256) {
99        assert(tokenSupported[tokenAddress]);
100       address[] memory path = new address[](2);
101       path[0] = router.WETH();
102       path[1] = tokenAddress;
103
104       uint256[] memory amounts = router.swapExactETHForTokens{ value: msg.value }(amountOut, path, to, deadline);
105       return amounts[0];
106   }
```

Figure 9 Source code of *swapExactEthToToken* function (unfixed)

| | |
|---|---|
| **Recommendations** | It is recommended to return amounts[1]. |
| **Status** | Fixed. The project has modified this to return the full array. |

```
92    function swapExactEthToToken(
93        address tokenAddress,
94        address to,
95        uint256 amountOut,
96        uint256 deadline
97    ) public onlyOwner payable returns (uint256[] memory) {
98        assert(tokenSupported[tokenAddress]);
99        address[] memory path = new address[](2);
100       path[0] = router.WETH();
101       path[1] = tokenAddress;
102
103       return router.swapExactETHForTokens{ value: msg.value }(amountOut, path, to, deadline);
104   }
```

Figure 10 Source code of *swapExactEthToToken* function (fixed)

## [Lighthouse-5] Missing judgement on cid parameter

| | |
|---|---|
| **Severity Level** | Low |
| **Type** | Business Security |
| **Lines** | Lighthouse.sol #L114-121<br>Lighthouse.sol #L63-80 |
| **Description** | In the Lighthouse contract, for the *publishStorageStatus* function, only the owner can call it to set the corresponding active value. We have a question, is this function just to record the status of a cid? Or does it prevent the cid from being stored arbitrarily by the user, since there is a store function in the contract that needs to be passed in as an argument. If it is the second case, is it necessary to add a status judgment on cid? |

```
114     function publishStorageStatus(
115         string calldata cid,
116         string calldata dealIds,
117         bool active
118     ) external onlyOwner {
119         // restrict it to only to the owner address
120         statuses[cid] = Status(dealIds, active);
121     }
```

Figure 11 Source code of *publishStorageStatus* function (unfixed)

```
63     function store(
64         string calldata cid,
65         string calldata config,
66         string calldata fileName,
67         uint256 fileSize
68     ) external {
69         uint256 currentTime = block.timestamp;
70         Deposit.updateStorage(msg.sender, fileSize, cid);
71         emit StorageRequest(
72             msg.sender,
73             cid,
74             config,
75             Deposit.costOfStorage() * fileSize,
76             fileName,
77             fileSize,
78             currentTime
79         );
80     }
```

Figure 12 Source code of *store* function (unfixed)

| | |
|---|---|
| **Recommendations** | It is recommended to add a status determination for the cid parameter to the *store* function. |
| **Status** | Acknowledged. The project owner responded that when a deal is made on filecoin the idea was to publish that on the blockchain. |

## [Lighthouse-6] Redundant code

| | |
|---|---|
| **Severity Level** | Info |
| **Type** | Coding Conventions |
| **Lines** | Billing.sol #L214-220 |
| **Description** | In the *cancelSubscription* function, there is an approve function, but it is to this contract authorization, not to the user authorization, and has no practical effect. |

```
214     function cancelSubscription() external {
215         require(userToSubscription[_msgSender()].occuranceLeft != 0, "No active subscription");
216         userToSubscription[_msgSender()].occuranceLeft = 0;
217         userToSubscription[_msgSender()].isCancelled = true;
218         IERC20Upgradeable(userToSubscription[_msgSender()].tokenAddress).approve(address(this), 0);
219         emit CancelSubscription(_msgSender(), userToSubscription[_msgSender()].systemDefinedSubscriptionID);
220     }
```

Figure 13 Source code of *cancelSubscription* function (unfixed)

| | |
|---|---|
| **Recommendations** | It is recommended to delete this step. |
| **Status** | Fixed. |

```
213     /// @dev this function cancels Subscription renewal for a user
214     function cancelSubscription() external {
215         require(userToSubscription[_msgSender()].occuranceLeft != 0, "No active subscription");
216         userToSubscription[_msgSender()].occuranceLeft = 0;
217         userToSubscription[_msgSender()].isCancelled = true;
218         emit CancelSubscription(_msgSender(), userToSubscription[_msgSender()].systemDefinedSubscriptionID);
219     }
220 }
```

Figure 14 Source code of *cancelSubscription* function (fixed)

## [Lighthouse-7] Lack of event triggers

| | |
|---|---|
| **Severity Level** | Info |
| **Type** | Coding Conventions |
| **Lines** | DepositManager.sol#L138-159 |
| **Description** | No event triggered when changing key parameter variables. |

```
138     function setDataCapForNewUsers(uint256 _initialDataCapForNewUsers) public onlyOwner {
139         initialDataCapForNewUsers = _initialDataCapForNewUsers;
140     }
141
142     /**
143     * @dev the function initializated the priceFeed Aggregator
144     * see {chainlink for more}
145
146     * @param aggregatorAddressFeed designated chainlink priceFeed address
147     */
148     function setPriceFeed(address aggregatorAddressFeed) public onlyOwner {
149         priceFeed = AggregatorV3Interface(aggregatorAddressFeed);
150     }
151
152     /**
153     * @dev the function allows the owner to modify the cost of storage
154
155     * @param newCost value to update cost to
156     */
157     function setCostOfStorage(uint256 newCost) public accountCheck(accountType.MANAGER) {
158         _costOfStorage = newCost;
159     }
```

Figure 15 Source code of *setDataCapForNewUsers, setPriceFeed, setCostOfStorage* functions (unfixed)

| | |
|---|---|
| **Recommendations** | It is recommended to add relevant events and trigger them in the corresponding functions. |
| **Status** | Fixed. |

```
166     function setDataCapForNewUsers(uint256 _initialDataCapForNewUsers) public onlyOwner {
167         initialDataCapForNewUsers = _initialDataCapForNewUsers;
168         emit AdminChangeEvent(
169             _msgSender(),
170             address(0),
171             _initialDataCapForNewUsers,
172             adminChangeType.SET_DATA_CAP_FOR_NEW_USERS
173         );
174     }
175
176     /**
177     * @dev the function initializated the priceFeed Aggregator
178     * see {chainlink for more}
179
180     * @param aggregatorAddressFeed designated chainlink priceFeed address
181     */
182     function setPriceFeed(address aggregatorAddressFeed) public onlyOwner {
183         priceFeed = AggregatorV3Interface(aggregatorAddressFeed);
184         emit AdminChangeEvent(_msgSender(), aggregatorAddressFeed, 0, adminChangeType.SET_PRICE_FEED);
185     }
186
187     /**
188     * @dev the function allows the owner to modify the cost of storage
189
190     * @param newCost value to update cost to
191     */
192     function setCostOfStorage(uint256 newCost) public accountCheck(accountType.MANAGER) {
193         _costOfStorage = newCost;
194         emit AdminChangeEvent(_msgSender(), address(0), newCost, adminChangeType.SET_COST);
195     }
```

Figure 16 Source code of *setDataCapForNewUsers, setPriceFeed, setCostOfStorage* functions (fixed)

# 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 | High | 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/+dD8Bnqd133RmNWNl

**Twitter**

https://twitter.com/Beosin_com

**Email**

Contact@beosin.com