BEOSIN
Blockchain Security

# Superlibracoin

Smart Contract Security Audit

No. 202406241559

Jun 24th, 2024

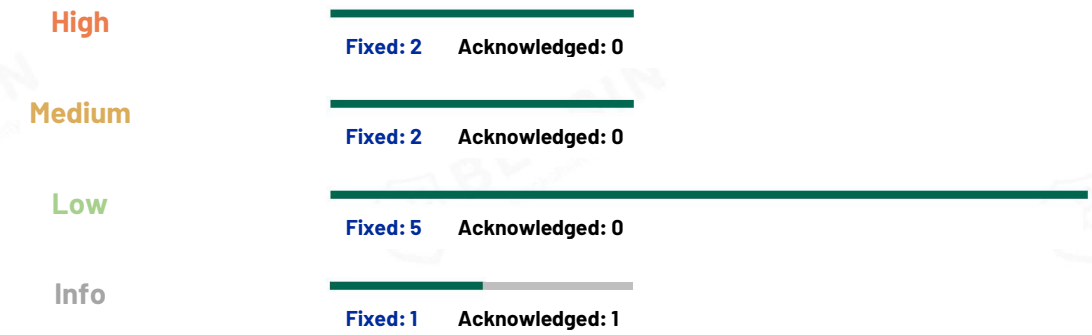
SECURING BLOCKCHAIN ECOSYSTEM

WWW.BEOSIN.COM

# Contents

# Summary of Audit Results

After auditing, 2 High-risks ,2 Medium-risks, 5 Low-risks and 2 Info-item were identified in the Superlibracoin 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:**

  Currently, the SLC and mainCollateralToken within the contract are equivalent to USD. However, if there are any future price adjustments, the `getPrice` and related functions will need to be modified accordingly to continue functioning properly.

- **Project Description:**

The audited project involves SLC token staking, lending, and selling, primarily divided into three modules: Collateral Lending Module, Proxy Module, and Oracle Module.

In the Collateral Lending Module, the functionalities include staking, lending, and selling. The admin can add corresponding collateral assets, with the current limit set to 100 assets. Users can stake collateral assets into the contract for subsequent SLC token lending. Given good collateral credit, users can borrow a certain percentage of SLC tokens based on the value of their collateral assets. If price fluctuations cause the collateral value to drop below the borrowed SLC value, other users and the project team can call the `tokenLiquidate` function to liquidate the user's bad collateral. Users can also buy SLC tokens through regular purchase and sale channels by paying a certain amount of tokens, and sell SLC to obtain other tokens.

In the Proxy Module, users can use the proxy contract to assist with staking, lending, and buying/selling SLC. Additionally, the proxy contract enables the exchange of CFX for SLC.

In the Oracle Module, the contract retrieves relevant token prices from the PYTH Oracle and performs a weighted calculation with the prices from the LP. This helps determine whether the user's collateral in the Collateral Lending Module is in a healthy state.

# 1 Overview

## 1.1 Project Overview

| | |
|---|---|
| **Project Name** | Superlibracoin |
| **Project Language** | Solidity |
| **Platform** | Conflux |
| **File Hash (SHA-256)** | superlibracoinV0.101.zip: 273a24e1e8ddfb6e6d5ca6aee965a46673afdec67ef7772ae764a8770b4e0231 superlibracoinV0.107.zip: 0d4a6dcc15b3c83f608a0cfd640f0104d14122a42ad0db5578bcf77f5f213820 |

## 1.2 Audit Overview

Audit work duration: Jun 19, 2024 – Jun 24, 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 |
|---|---|---|---|
| Superlibracoin-01 | Transfer Logic Error | High | Fixed |
| Superlibracoin-02 | Liquidation Logic Error | High | Fixed |
| Superlibracoin-03 | Exchange Logic Error | Medium | Fixed |
| Superlibracoin-04 | Key Parameter Not Assigned | Medium | Fixed |
| Superlibracoin-05 | Incorrect Issuance Calculation | Low | Fixed |
| Superlibracoin-06 | Inaccurate Total Staked Calculation | Low | Fixed |
| Superlibracoin-07 | No Limit on Adding Assets | Low | Fixed |
| Superlibracoin-08 | Missing Liquidation Incentives | Low | Fixed |
| Superlibracoin-09 | Centralization Risk | Low | Fixed |
| Superlibracoin-10 | Key Functions Missing Events | Info | Partially Fixed |
| Superlibracoin-11 | Oracle Timestamp Not Validated | Info | Acknowledged |

# Finding Details:

## [Superlibracoin-01] Transfer Logic Error

| | |
|---|---|
| **Severity Level** | High |
| **Type** | Business Security |
| **Lines** | slcInterface.sol #L110-152 |
| **Description** | In the slcInterface contract, there are logical errors in functions such as `buySlcByCFX`, `sellSlcToCFX`, `obtainSLC`, `returnSLC`, `CFXPledge`, and `redeemCFX`. For example, in the `CFXPledge` function, the contract converts the platform currency transferred to the contract into wxCFX. However, in the subsequent `licensedAssetsPledge` function logic, this wxCFX is not handled; instead, it reuses wxCFX transferred from msg.sender. If the user has approved the contract, this can result in the transferred platform currency being locked in the contract, causing losses to the user. Similar issues exist in other functions. |
| **Recommendation** | It is recommended to adjust the logic. |
| **Status** | **Fixed.** The project party modified the corresponding logic and avoid transfer logic error. |

```
    function CFXPledge() public payable {
        address payable receiver = payable(this);
        (bool success, ) = receiver.call{value:msg.value}("");
        require(success,"X SLC Interface: CFX Transfer Failed");
        iwxCFX(wxCFX).deposit{value:msg.value}();
        licensedAssetsPledge(wxCFX, msg.value);
    }
```

```
    function CFXPledge() public payable {
        address payable receiver = payable(this);
        (bool success, ) = receiver.call{value:msg.value}("");
        require(success,"X SLC Interface: CFX Transfer Failed");
        iwxCFX(wxCFX).deposit{value:msg.value}();
        IERC20(wxCFX).approve(slcVaults, msg.value);
        iSlcVaults(slcVaults).licensedAssetsPledge( wxCFX, msg.value,
msg.sender);
```

## [Superlibracoin-02] Liquidation Logic Error

| | |
|---|---|
| **Severity Level** | High |
| **Type** | Business Security |
| **Lines** | slcvaults.sol #L321-344 |
| **Description** | In the slcvaults contract, the `tokenLiquidate` function directly uses outputAmount without assignment, leading to two issues:<br>1. If tokenAddr is mainCollateralToken, the user's borrow amount does not decrease, but the collateral is deducted.<br>2. If tokenAddr is not mainCollateralToken, subsequent `xchange` calculations will fail the slippage limit, preventing liquidation. |

```
    function tokenLiquidate(address user,address token, uint amount)
public returns(uint outputAmount) {
        require(amount > 0,"SLC Vaults: Cant Pledge 0 amount");
        require(amount <= userAssetsMortgageAmount[user][token],"SLC
Vaults: amount need <= balance Of user");
        uint factor;
        (factor, ,,) = viewUsersHealthFactor(user);
        require(factor < 1 ether,"Liquidate user Assets, his Health
Factor must < 1");
        address[] memory tokens = new address[](3);
        tokens[0] = token;
        tokens[1] = superLibraCoin;
        tokens[2] = mainCollateralToken;
        IERC20(token).approve(xInterface, amount);

        if(tokens[0] == mainCollateralToken){
            outputAmount = outputAmount * 1 ether *
(10000-licensedAssets[token].liquidationPenalty) / (10000 * slcVaule);
        }else{
            outputAmount = ixInterface(xInterface).xexchange(tokens,
amount, outputAmount, outputAmount / 100, block.timestamp + 100);
            outputAmount = outputAmount * 1 ether *
(10000-licensedAssets[token].liquidationPenalty) / (10000 * slcVaule);
        }
        require(userObtainedSLCAmount[user] >= outputAmount,"");
        slcUnsecuredIssuancesAmount += outputAmount;
        userAssetsMortgageAmount[user][token] -= amount;
```

```
        userObtainedSLCAmount[user] -= outputAmount;}
```

**Recommendation** It is recommended to modify this part of the code according to the logic in the `slcTokenBuy` function.

**Status** **Fixed.** The project party modified the corresponding logic and avoid liquidation logic error.

```
    function tokenLiquidate(address user,address token, uint amount)
public returns(uint outputAmount) {
        require(amount > 0,"SLC Vaults: Cant Pledge 0 amount");
        require(amount <= userAssetsMortgageAmount[user][token],"SLC
Vaults: amount need <= balance Of user");
        uint factor;
        (factor, ,,) = viewUsersHealthFactor(user);
        require(factor < 1 ether,"SLC Vaults: Liquidate user Assets, his
Health Factor must < 1");
        address[] memory tokens = new address[](3);
        tokens[0] = token;
        tokens[1] = superLibraCoin;
        tokens[2] = mainCollateralToken;
        IERC20(token).approve(xInterface, amount);
        if(tokens[0] == mainCollateralToken){
            outputAmount = amount * 1 ether *
(10000-licensedAssets[token].liquidationPenalty) / (10000 * slcVaule);
        }else{
            (outputAmount,) =
ixInterface(xInterface).xExchangeEstimateInput(tokens, amount);
            outputAmount = ixInterface(xInterface).xexchange(tokens,
amount, outputAmount, outputAmount / 100, block.timestamp + 100);
            outputAmount = outputAmount * 1 ether *
(10000-licensedAssets[token].liquidationPenalty) / (10000 * slcVaule);
        }
        require(userObtainedSLCAmount[user] >= outputAmount,"");
        slcUnsecuredIssuancesAmount += outputAmount;
        userObtainedSLCAmount[user] -= outputAmount;
        userAssetsMortgageAmountSum[token] -= amount;
        userAssetsMortgageAmount[user][token] -= amount;
        IERC20(token).transfer(msg.sender,amount/10000);
        emit TokenLiquidate(msg.sender, token, amount, outputAmount);
    }
```

## [Superlibracoin-03] Exchange Logic Error

| | |
|---|---|
| **Severity Level** | Medium |
| **Type** | Business Security |
| **Lines** | slcvaults.sol #L222-245 |
| **Description** | In the slcvaults contract, the xExchangeEstimateOutput function in slcTokenSell is passed the outAmount calculated by xExchangeEstimateInput, which is the amount of TokenAddr obtained from selling SLC tokens. Therefore, the logic in xExchangeEstimateOutput should determine how much mainCollateralToken is needed for the outAmount of TokenAddr according to the tokensT exchange path. However, the function does not reverse the tokensT exchange path. |

```
    function slcTokenSell(address TokenAddr, uint amount)
public  returns(uint outputAmount) {
        iSlc(superLibraCoin).burnSLC(msg.sender,amount);
        if(slcUnsecuredIssuancesAmount > outputAmount){
            slcUnsecuredIssuancesAmount -= outputAmount;
        }else{
            slcUnsecuredIssuancesAmount = 0;
        }
        address[] memory tokens = new address[](2);
        tokens[0] = superLibraCoin;
        tokens[1] = TokenAddr;
        (outputAmount,) =
ixInterface(xInterface).xExchangeEstimateInput(tokens, amount);
        outputAmount = outputAmount * 95 / 100 ;
        address[] memory tokensT = new address[](3);
        tokensT[0] = mainCollateralToken;
        tokensT[1] = superLibraCoin;
        tokensT[2] = TokenAddr;
        (amount,) =
ixInterface(xInterface).xExchangeEstimateOutput(tokensT,
outputAmount);
        IERC20(mainCollateralToken).approve(xInterface, amount);
        outputAmount =
ixInterface(xInterface).xexchange(tokensT,amount,outputAmount,output
Amount / 50, block.timestamp + 100);
```

```
        IERC20(TokenAddr).transfer(msg.sender,outputAmount);
    }
```

**Recommendation** It is recommended to reverse tokensT before calling xExchangeEstimateOutput.

**Status** **Fixed.** The project party modified the code of the xExchangeEstimateOutput function and performed an inverted calculation on tokens.

```
    function xExchangeEstimateOutput(address[] memory tokens,uint
amountOut) external view returns(uint input, uint[3] memory
priceImpactAndFees, uint b) {
        uint[4] memory inputAmount;
        uint[4] memory outputAmount;
        address[] memory _lp = new address[](tokens.length);
        uint i;
        uint[2] memory priceCumulative;
        require(tokens.length>1&&tokens.length<=5,"X SWAP Interface:
exceed MAX path lengh:2~5");
        outputAmount[tokens.length-1] = amountOut;
        require( amountOut > 0,"X SWAP Interface: Input need > 0");
        priceImpactAndFees[1] = 10000;
        priceImpactAndFees[2] = 10000;
        for(i=tokens.length-1;i>0;i--){
            _lp[i]=iXunionFactory(xfactory).getPair(tokens[i],
tokens[i-1]);
            (input,) = getLpSettings(_lp[i]);
            priceImpactAndFees[0] += input;
            (inputAmount[i],,priceCumulative,b) =
            ixCore(xCore).swapCalculation3(_lp[i],tokens[i-1],outputA
mount[i]);
            outputAmount[i-1] = inputAmount[i];
            priceImpactAndFees[1] = priceImpactAndFees[1] *
priceCumulative[0] / priceCumulative[1];
            priceImpactAndFees[2] = priceImpactAndFees[2] *
getLpPrice(_lp[i]) / 1 ether;}
        input = inputAmount[1];
    }
```

## [Superlibracoin-04] Key Parameter Not Assigned

| | |
|---|---|
| **Severity Level** | **Medium** |
| **Type** | Business Security |
| **Lines** | slcvaults.sol #L250-252, L266-268, L286-288, L303-305 |
| **Description** | In the slcvaults contract, no function assigns a value to slcInterface, causing slcInterface to always be false. If the slcinterface contract calls functions like `licensedAssetsPledge`, `redeemPledgedAssets`, `obtainSLC`, and `returnSLC`, it will throw an exception, halting subsequent logic. |
| **Recommendation** | It is recommended to add a function to set the slcInterface parameter. |
| **Status** | **Fixed.** The project party added the `setSlcInterface` function. |

```
    function licensedAssetsPledge(address TokenAddr, uint amount,
address user) public  {

        if(slcInterface[msg.sender]==false){

            require(user == msg.sender,"SLC Vaults: Not registered as
slcInterface or user need be msg.sender!");

        }
```

```
    function setSlcInterface(address _ifSlcInterface, bool _ToF)
external onlySetter{
        slcInterface[_ifSlcInterface] = _ToF;
    }
```

## [Superlibracoin-05] Incorrect Issuance Calculation

| | |
|---|---|
| **Severity Level** | Low |
| **Type** | Business Security |
| **Lines** | slcvaults.sol #L222-228 |
| **Description** | In the slcvaults contract, the `slcTokenSell` function deducts from the total issuance before assigning outputAmount, resulting in the total issuance always incrementing since outputAmount is initially 0. This leads to inaccurate subsequent queries.<br>`function slcTokenSell(address TokenAddr, uint amount) public returns(uint outputAmount) { iSlc(superLibraCoin).burnSLC(msg.sender,amount); if(slcUnsecuredIssuancesAmount > outputAmount){ slcUnsecuredIssuancesAmount -= outputAmount; }else{ slcUnsecuredIssuancesAmount = 0; }` |
| **Recommendation** | It is recommended to deduct the total SLC issuance after calculating outputAmount. |
| **Status** | **Fixed.** The project party modified the code logic of `slcTokenSell` and changed the deduction of outputAmount to the deduction of amount.<br>`function slcTokenSell(address TokenAddr, uint amount) public returns(uint outputAmount) { iSlc(superLibraCoin).burnSLC(msg.sender,amount); if(slcUnsecuredIssuancesAmount > amount){ slcUnsecuredIssuancesAmount -= amount; }else{ slcUnsecuredIssuancesAmount = 0; }` |

## [Superlibracoin-06] Inaccurate Total Staked Calculation

| | |
|---|---|
| **Severity Level** | Low |
| **Type** | Business Security |
| **Lines** | slcvaults.sol #L340-343 |
| **Description** | In the slcvaults contract, the `tokenLiquidate` function only reduces the user's staked token amount without decreasing the total staked token amount. As a result, subsequent queries for the total staked tokens are inaccurate.<br>`require(userObtainedSLCAmount[user] >= outputAmount,"");`<br>`slcUnsecuredIssuancesAmount += outputAmount;`<br>`userAssetsMortgageAmount[user][token] -= amount;`<br>`userObtainedSLCAmount[user] -= outputAmount;` |
| **Recommendation** | It is recommended to also reduce the total staked token amount during liquidation. |
| **Status** | **Fixed.** The project party modified the corresponding logic and deducted the value of userObtainedSLCAmount simultaneously.<br>`require(userObtainedSLCAmount[user] >= outputAmount,"");`<br>`slcUnsecuredIssuancesAmount += outputAmount;`<br>`userObtainedSLCAmount[user] -= outputAmount;`<br>`userAssetsMortgageAmountSum[token] -= amount;`<br>`userAssetsMortgageAmount[user][token] -= amount;` |

## [Superlibracoin-07] No Limit on Adding Assets

| | |
|---|---|
| **Severity Level** | Low |
| **Type** | Business Security |
| **Lines** | slcvaults.sol #L94-101 |
| **Description** | In the slcvaults contract, the `viewUsersHealthFactor` function checks the health factor, limiting asset types to fewer than 100. However, the `licensedAssetsRegister` function does not enforce this limit and does not allow removal of assets. If the admin mistakenly adds more than 100 asset types, the project's collateral lending functionality will fail. |
| **Recommendation** | It is recommended to limit the asset types during registration. |
| **Status** | **Fixed.** The project party limit the asset types during registration. |

```
    function licensedAssetsRegister(address _asset, uint MaxLTV, uint
LiqPenalty,uint MaxDepositAmount) public onlySetter {
        require(licensedAssets[_asset].assetAddr == address(0),"SLC
Vaults: asset already registered!");
        assetsSerialNumber.push(_asset);
        licensedAssets[_asset].assetAddr = _asset;
        licensedAssets[_asset].maximumLTV = MaxLTV;
        licensedAssets[_asset].liquidationPenalty = LiqPenalty;
        licensedAssets[_asset].maxDepositAmount = MaxDepositAmount;
    }
```

```
    function licensedAssetsRegister(address _asset, uint MaxLTV, uint
LiqPenalty,uint MaxDepositAmount) public onlySetter {
        require(licensedAssets[_asset].assetAddr == address(0),"SLC
Vaults: asset already registered!");
        require(assetsSerialNumber.length < 100,"SLC Vaults: Too Many
Assets");
        assetsSerialNumber.push(_asset);
        licensedAssets[_asset].assetAddr = _asset;
        licensedAssets[_asset].maximumLTV = MaxLTV;
        licensedAssets[_asset].liquidationPenalty = LiqPenalty;
        licensedAssets[_asset].maxDepositAmount = MaxDepositAmount;
    }
```

## [Superlibracoin-08] Missing Liquidation Incentives

| | |
|---|---|
| **Severity Level** | Low |
| **Type** | Business Security |
| **Lines** | staking.sol #L321-344 |
| **Description** | In the slcvaults contract, the `tokenLiquidate` function only clears the user's bad loans but does not reward the liquidator, negatively impacting the liquidator's motivation. |

```
    function tokenLiquidate(address user,address token, uint amount)
public returns(uint outputAmount) {
        require(amount > 0,"SLC Vaults: Cant Pledge 0 amount");
        require(amount <= userAssetsMortgageAmount[user][token],"SLC
Vaults: amount need <= balance Of user");
        uint factor;
        (factor, ,,) = viewUsersHealthFactor(user);
        require(factor < 1 ether,"Liquidate user Assets, his Health
Factor must < 1");
        address[] memory tokens = new address[](3);
        tokens[0] = token;
        tokens[1] = superLibraCoin;
        tokens[2] = mainCollateralToken;
        IERC20(token).approve(xInterface, amount);
        if(tokens[0] == mainCollateralToken){
            outputAmount = outputAmount * 1 ether *
(10000-licensedAssets[token].liquidationPenalty) / (10000 * slcVaule);
        }else{
            outputAmount = ixInterface(xInterface).xexchange(tokens,
amount, outputAmount, outputAmount / 100, block.timestamp + 100);
            outputAmount = outputAmount * 1 ether *
(10000-licensedAssets[token].liquidationPenalty) / (10000 * slcVaule);
        }
        require(userObtainedSLCAmount[user] >= outputAmount,"");
        slcUnsecuredIssuancesAmount += outputAmount;
        userAssetsMortgageAmount[user][token] -= amount;
        userObtainedSLCAmount[user] -= outputAmount;
    }
```

| | |
|---|---|
| **Recommendation** | It is recommended to allocate a portion of the liquidation as a reward for the liquidator. |

| Status | **Fixed.** The project party allocate a portion of the liquidation as a reward for the liquidator.<br>`IERC20(token).transfer(msg.sender,amount/10000);` |
| --- | --- |

## [Superlibracoin-09] Centralization Risk

| | |
|---|---|
| **Severity Level** | Low |
| **Type** | Business Security |
| **Lines** | slcvaults.sol #L349-359 |
| **Description** | The `rebalance` function can swap tokens within the slcvaults contract and transfer them out through the `excessDisposal` function. If the private keys of the rebalancer or setter are compromised, it could result in malicious extraction of users' staked tokens. |
| **Recommendation** | It is recommended to manage the rebalancer and setter using a multi-signature wallet or similar mechanism. |
| **Status** | **Fixed.** The project party will use a multi-signature wallet to manage it later. |

```
    function rebalance(address[] memory tokens, uint amount) public
onlyRebalancer() returns(uint outputAmount){
        IERC20(tokens[0]).approve(xInterface, amount);
        (outputAmount,) =
ixInterface(xInterface).xExchangeEstimateInput(tokens, amount);
        outputAmount =
ixInterface(xInterface).xexchange(tokens,amount,outputAmount,outputA
mount / 100, block.timestamp + 100);
    }
    function excessDisposal(address token, uint amount) public
onlyRebalancer(){
        require(IERC20(token).balanceOf(address(this)) > amount +
userAssetsMortgageAmountSum[token],"SLC Vaults: Cant Do Excess
Disposal, asset not enough!");
        IERC20(token).transfer(msg.sender,amount);
    }
```

## [Superlibracoin-10] Key Functions Missing Events

| | |
|---|---|
| **Severity Level** | Info |
| **Type** | Coding Conventions |
| **Description** | Multiple functions setting key parameters lack events, such as `slcValueRevaluate` and `setup`. They may not be able to obtain the corresponding information in a timely manner. |

```
    function setup( address _superLibraCoin,
                    address _mainCollateralToken,
                    address _xInterface,
                    address _oracleAddr ) external onlySetter{
        superLibraCoin = _superLibraCoin;
        mainCollateralToken = _mainCollateralToken;
        xInterface = _xInterface;
        oracleAddr = _oracleAddr;
    }
    function slcValueRevaluate(uint newVaule) public  onlySetter {
        slcVaule = newVaule;
    }
    function mainCollateralTokenSetting(address token)
public  onlySetter {
        mainCollateralToken = token;
    }
```

| | |
|---|---|
| **Recommendation** | It is recommended to trigger events for all functions that set key parameters to facilitate future project maintenance. |
| **Status** | **Partially Fixed.** The project party added partially event triggers. |

```
    function userModeSetting(uint8 _mode,address
_userModeAssetsAddress) public {
        require(userObtainedSLCAmount[msg.sender] == 0,"SLC Vaults:
Cant Change Mode before return all SLC.");
        userMode[msg.sender] = _mode;
        userModeAssetsAddress[msg.sender] = _userModeAssetsAddress;
        emit UserModeSetting(msg.sender, _mode,
_userModeAssetsAddress);
    }
```

## [Superlibracoin-11] Oracle Timestamp Not Validated

| | |
|---|---|
| **Severity Level** | Info |
| **Type** | Business Security |
| **Lines** | slcOracle.sol #L57-76 |
| **Description** | In the slcOracle contract, the `getPythBasicPrice` function does not validate the publishTime of the retrieved price, which may result in using outdated prices. This leads to a loss of price timeliness. |

```
    function getPythBasicPrice(bytes32 id) internal view returns
(PythStructs.Price memory price){
        price = IPyth(pythAddr).getPriceUnsafe(id);
    }
    function pythPriceUpdate(bytes[] calldata updateData) public
payable {
        uint fee = IPyth(pythAddr).getUpdateFee( updateData);
        IPyth(pythAddr).updatePriceFeeds{ value: fee }(updateData);
    }
    function getPythPrice(address token) public view returns (uint
price){
        PythStructs.Price memory priceBasic;
        uint tempPriceExpo ;
        if(TokenToPythId[token] != bytes32(0)){
            priceBasic = getPythBasicPrice(TokenToPythId[token]);
            tempPriceExpo = uint(int256(18+priceBasic.expo));
            price = uint(int256(priceBasic.price)) *
(10**tempPriceExpo);
        }else{
            price = 0;
        }
    }
```

| | |
|---|---|
| **Recommendation** | It is recommended to validate the timestamp of the oracle price. |
| **Status** | **Acknowledged.** |

# 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.

### 3.1.3 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.

### 3.1.4 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