xlending

Smart Contract Security Audit

No. 202407261622

Jul 26th, 2024

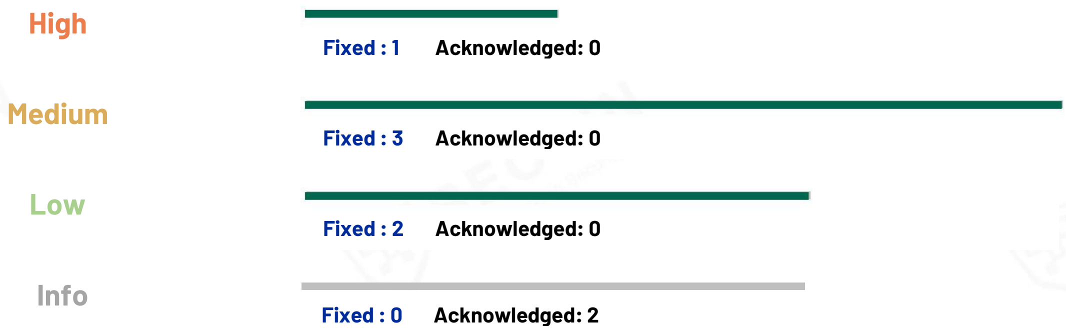
SECURING BLOCKCHAIN ECOSYSTEM

WWW.BEOSIN.COM

Contents

Summary of Audit Results

After auditing, 1 High, 3 Medium, 2 Low and 2 Info-risk item were identified in the xlending 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

xlending is primarily composed of the following contracts:

coinFactory Contract: This contract is used to create and manage DepositCoin and LoanCoin contracts. The depositOrLoanCoin contracts created are non-transferable ERC20 tokens, with minting and burning controlled by the manager contract.

lendingCoreAlgorithm Contract: This contract is used to calculate and query the interest rates for collateral and loans of specified tokens.

lendingVaults Contract: This contract is used to store and manage users' collateral assets.

lendingManager Contract: This contract handles all business logic. Users can perform operations such as deposit, lend, withdraw, repay, and liquidating bad assets. When deposit, users receive DepositCoin, which is destroyed upon withdrawal and includes a reward. After deposit, users can borrow, receiving LoanCoin, which is destroyed upon repayment along with interest. During liquidation, the liquidator must pay off the lend user's loan tokens, then the liquidator will destroy the lend user's DepositCoin and LoanCoin and transfer the lend user's collateral tokens to the liquidator. Relevant parameters are set by the setter.

lendingInterface Contract: This is an external interface contract for the lendingManager. All functions for deposit, lend, withdraw, repay, and querying, except for liquidation, can be accessed through this contract. For user convenience, new functions are implemented for handling platform tokens and regular tokens, as well as for full repayment and full withdrawal operations. These new functions preprocess the operations before calling the corresponding functions in the lendingManager contract.

1 Overview

1.1 Project Overview

Project Name	xlending
Project Language	Solidity
Platform	Conflux Network
Code base	https://github.com/artixv/xlending
Commit Id	aa28bb97b92ad8c57ee36092d1634f9589fa69d6 d8916d750fc5261fd92ac6c7b3fad7b4073d17d6 addcdfb8715df96cbd64302056fa4d678914e0b6 53485e707691c6b84860b39ec0475643ac668fca 750ddc15d7711767d82f12267455088efa83ed5e 353101a4c5e7a9b6bb875867ced597dbb69915dd accc904dfb7765c52a57cd1a6a7b973934b65fa0 6a902209349ce3afae20f330b360ceafa7ad23de

1.2 Audit Overview

Audit work duration: Jul 17, 2024 – Jul 26, 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
xlending-01	Liquidation formula error	High	Fixed
xlending-02	Excessive lendingInterface privileges	Medium	Fixed
xlending-03	Centralized Risks	Medium	Fixed
xlending-04	Conflict of judgment condition	Medium	Fixed
xlending-05	CreateDeAndLoCoin lacks permission checks	Low	Fixed
xlending-06	Unit error in tokenLiquidateEstimate function	Low	Fixed
xlending-07	Missing trigger event	Info	Acknowledged
xlending-08	Redundant code	Info	Acknowledged

Finding Details:

[xlending-01] Liquidation formula error

Severity Level	High
Type	Business Security
Lines	lendingManager.sol#L525-545
Description	1. The `tokenLiquidate` function calculates the `usedAmount` using an uninitialized temporary variable, which is not aligned with the estimation in the `tokenLiquidateEstimate` function. The current calculation method results in a `usedAmount` of 0, indicating that the user does not need to pay collateral when liquidating. 2. Before calculating the user's health factor, the user's staking rewards are not updated, and some users who do not meet the liquidation conditions after the reward settlement will be liquidated. 3. After the liquidation is completed and the user's tokens are burned, the global pledge rate and interest rate are not updated. 4. When `liquidateAmount=amountlending` executes full liquidation, `usedAmount` may be less than `amountDeposit` due to calculation, and the `depositToken` of the user being liquidated will not be completely destroyed. There will be a situation where the user's mortgage has been fully liquidated, but still holds the debt token `depositToken`.

```
    function tokenLiquidate(address user,
        address liquidateToken,
        uint    liquidateAmount,
        address depositToken) public returns(uint usedAmount) {
        require(liquidateAmount > 0,"Lending Manager: Cant Pledge 0
amount");
        require(viewUsersHealthFactor(user) < 1 ether,"Lending Manager:
Users Health Factor Need < 1 ether");
...
        require( amountLending >= liquidateAmount,"Lending Manager:
amountLending >= liquidateAmount");
        usedAmount  = usedAmount *
iSlcOracle(oracleAddr).getPrice(liquidateToken) / 1 ether;
        usedAmount = usedAmount * (UPPER_SYSTEM_LIMIT -
```

```
licensedAssets[liquidateToken].liquidationPenalty) * 1 ether /
                (UPPER_SYSTEM_LIMIT *
iSlcOracle(oracleAddr).getPrice(depositToken));
        require( amountDeposit >= usedAmount,"Lending Manager:
amountLending >= liquidateAmount");
        iLendingVaults(lendingVault).vaultsERC20Approve(liquidateToke
n, liquidateAmount);
        IERC20(depositToken).transferFrom(msg.sender, lendingVault,
usedAmount);
        IERC20(liquidateToken).transferFrom(lendingVault, msg.sender,
liquidateAmount);
        iDepositOrLoanCoin(assetsDepositAndLend[liquidateToken][0]).b
urnCoin(user,liquidateAmount);
        iDepositOrLoanCoin(assetsDepositAndLend[depositToken][1]).bur
nCoin(user,usedAmount);
    }
```

Recommendation

It is recommended that:

1. Modify the algorithm in `tokenLiquidate` according to the design requirements.

2. Execute the `_beforeUpdate` function to update the reward before `viewUsersHealthFactor` judges.

3. After destroying the tokens, execute the `_assetsValueUpdate` function to update the interest rate.

4. Add a judgment before destroying `depositToken`. If `liquidateAmount==amountlending`, the destroyed amount is `amountDeposit`. If it is less than amount, the destroyed amount is `usedAmount`.

Status

Fixed. In Commit 53485e707691c6b84860b39ec0475643ac668fca, the liquidation function was redesigned. Now only assets that meet the liquidation conditions but have not entered `badDebt` can be liquidated. A new `badDebtDeduction` function is added to process assets that have entered `badDebt`. The `badDebtCollectionAddress` is set by the setter. The project party declares that the `badDebt` is borne by the project party itself.

```
    function badDebtDeduction(address user) public {
        require(_userTotalDepositValue(user) <=
_userTotalLendingValue(user)*102/100,"Lending Manager: should be bad
debt.");
```

```
        for(uint i=0;i<assetsSerialNumber.length;i++){
            iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumbe
r[i]][0]).mintCoin(badDebtCollectionAddress,IERC20(assetsDepositAndL
end[assetsSerialNumber[i]][0]).balanceOf(user));
            iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumbe
r[i]][1]).mintCoin(badDebtCollectionAddress,IERC20(assetsDepositAndL
end[assetsSerialNumber[i]][1]).balanceOf(user));
            iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumbe
r[i]][0]).burnCoin(user,IERC20(assetsDepositAndLend[assetsSerialNumb
er[i]][0]).balanceOf(user));
            iDepositOrLoanCoin(assetsDepositAndLend[assetsSerialNumbe
r[i]][1]).burnCoin(user,IERC20(assetsDepositAndLend[assetsSerialNumb
er[i]][1]).balanceOf(user));
        }
    }
    function tokenLiquidate(address user,
                            address liquidateToken,
                            uint    liquidateAmount,
                            address depositToken) public returns(uint
usedAmount) {
        _beforeUpdate(liquidateToken);
        _beforeUpdate(depositToken);
        require(_userTotalDepositValue(user) >
_userTotalLendingValue(user)*102/100,"Lending Manager: Require users
not bad debt.");
        require(liquidateAmount > 0,"Lending Manager: Cant Pledge 0
amount");
        require(viewUsersHealthFactor(user) < 1 ether,"Lending Manager:
Users Health Factor Need < 1 ether");
...
        usedAmount = liquidateAmount *
iSlcOracle(oracleAddr).getPrice(liquidateToken) / 1 ether;
        usedAmount = usedAmount * (UPPER_SYSTEM_LIMIT -
licensedAssets[liquidateToken].liquidationPenalty) * 1 ether /
                                (UPPER_SYSTEM_LIMIT *
iSlcOracle(oracleAddr).getPrice(depositToken));
        require( amountDeposit >= usedAmount,"Lending Manager:
amountDeposit >= usedAmount");
...
```

```
        iDepositOrLoanCoin(assetsDepositAndLend[liquidateToken][0]).b
urnCoin(user,liquidateAmount);
        iDepositOrLoanCoin(assetsDepositAndLend[depositToken][1]).bur
nCoin(user,usedAmount);
        _assetsValueUpdate(liquidateToken);
        _assetsValueUpdate(depositToken);
        emit AssetsDeposit(liquidateToken, liquidateAmount, user);
        emit RepayLoan(depositToken, usedAmount, user);
    }
```

[xlending-02] Excessive lendingInterface privileges

Severity Level	**Medium**
Type	Business Security
Lines	lendingManager.sol#L429-450
Description	The `lendingInterface` permission in the contract is set by the `setter`. The original intention is to set it to the lendingInterface contract, but it can be assigned to multiple addresses in the code implementation, and the assigned addresses are not checked. Taking the `withdrawDeposit` function as an example, once a malicious address holds the `lendingInterface` permission, it can withdraw any user's mortgage to malicious address. The same problem exists in the `lendAsset` function. The malicious address can use other people's mortgages to borrow for malicious address.

```
    function withdrawDeposit(address tokenAddr, uint amount, address
user) public  {
        if(lendingInterface[msg.sender]==false){
            require(user == msg.sender,"Lending Manager: Not registered
as slcInterface or user need be msg.sender!");
        }
        require(amount > 0,"Lending Manager: Cant Pledge 0 amount");
        if(userMode[user] == 0){
            require(licensedAssets[tokenAddr].maxLendingAmountInRIM
== 0,"Lending Manager: Wrong Token in Risk Isolation Mode");
        }else if(userMode[user] == 1){
            require((tokenAddr == userRIMAssetsAddress[user]),"Lending
Manager: Wrong Token in Risk Isolation Mode");
        }else {
            require((licensedAssets[tokenAddr].lendingModeNum ==
userMode[user]),"Lending Manager: Wrong Mode, Need in same homogeneous
Mode");
        }
        //need + vualt add accept amount function (only manager)
        iLendingVaults(lendingVault).vaultsERC20Approve(tokenAddr,
amount);
        _beforeUpdate(tokenAddr);
        IERC20(tokenAddr).transferFrom(lendingVault,msg.sender,amount
);
```

	`` iDepositOrLoanCoin(assetsDepositAndLend[tokenAddr][0]).burnCoin(user,amount); _assetsValueUpdate(tokenAddr); uint factor; (factor) = viewUsersHealthFactor(user); if(userMode[user] > 1){ require(factor >= homogeneousFloorOfHealthFactor,"Your Health Factor <= homogeneous Floor Of Health Factor, Cant redeem assets"); }else{ require(factor >= nomalFloorOfHealthFactor,"Your Health Factor <= nomal Floor Of Health Factor, Cant redeem assets"); } emit WithdrawDeposit(tokenAddr, amount, user); } ```
Recommendation	It is recommended to add permissions to grant `lendingInterface`. If there is only one `lendingInterface` contract in business needs, modify it to only be set once. If there are multiple, add a whitelist and only set the addresses in the whitelist.
Status	**Partially Fixed.** In Commit 353101a4c5e7a9b6bb875867ced597dbb69915dd, the lendingInterface permission was changed to be granted only to one contracts . ``` function updatelendingInterface(address _interface) external onlySetter{ require(isContract(_interface),"Lending Manager: Interface MUST be a contract."); lendingInterface = _interface; emit LendingInterfaceSetup(_interface); } ```

[xlending-03] Centralized Risks

Severity Level	**Medium**
Type	Business Security
Lines	lendingManager.sol#L150-159
Description	The `setter` permissions in the project, taking the lendingmanager contract as an example, can modify key parameters and interacting external contracts. Arbitrary modification of the health factor may make users more vulnerable to liquidation, and modifying the external contract address to a malicious address may cause security risks.

```
    function setlendingInterface(address _interface, bool _ToF)
external onlySetter{
        lendingInterface[_interface] = _ToF;
        emit LendingInterfaceSetup(_interface, _ToF);
    }
    function setFloorOfHealthFactor(uint nomal, uint homogeneous)
external onlySetter{
        nomalFloorOfHealthFactor = nomal;
        homogeneousFloorOfHealthFactor = homogeneous;
        emit FloorOfHealthFactorSetup( nomal, homogeneous);
    }
```

Recommendation	It is recommended that `setter` permissions in a project be managed using a multi-signature wallet.
Status	**Fixed.** The project owner stated that it would use a multi-signature wallet to manage setter permissions.

[xlending-04] Conflict of judgment condition

Severity Level	**Medium**
Type	Business Security
Lines	lendingVaults.sol#L55-60
Description	In the `excessDisposal` function, the require judgment requires that the contract balance is greater than D-L, but the amount transferred is D-L-contract balance, which will cause an error in the subtraction.

```
    function excessDisposal(address token) public onlyRebalancer(){
        uint amountD =
iDepositOrLoanCoin(iLendingManager(lendingManager).assetsDepositAndLendAddrs(token)[0]).totalSupply();
        uint amountL =
iDepositOrLoanCoin(iLendingManager(lendingManager).assetsDepositAndLendAddrs(token)[1]).totalSupply();
        require(IERC20(token).balanceOf(address(this)) > amountD -
amountL,"Lending Manager: Cant Do Excess Disposal, asset not enough!");
        IERC20(token).transfer(msg.sender,amountD - amountL-
IERC20(token).balanceOf(address(this)));
    }
```

Recommendation	It is recommended that the number of transfers be modified according to design requirements.
Status	**Fixed.** The transfer amount is modified to a value that meets the require judgment.

```
    function excessDisposal(address token) public onlyRebalancer(){
        uint amountD =
iDepositOrLoanCoin(iLendingManager(lendingManager).assetsDepositAndLendAddrs(token)[0]).totalSupply();
        uint amountL =
iDepositOrLoanCoin(iLendingManager(lendingManager).assetsDepositAndLendAddrs(token)[1]).totalSupply();
        require(IERC20(token).balanceOf(address(this)) > amountD -
amountL,"Lending Manager: Cant Do Excess Disposal, asset not enough!");
        IERC20(token).transfer(msg.sender,IERC20(token).balanceOf(address(this)) + amountL - amountD);
    }
```

[xlending-05] CreateDeAndLoCoin lacks permission checks

Severity Level	Low
Type	Business Security
Lines	coinFactory.sol#L29-47
Description	The `createDeAndLoCoin` function does not have any permission check, and anyone can call this function to create a `depositOrLoanCoin`, which may lead to a waste of contract storage resources.

```
    function createDeAndLoCoin(address token) external returns (address[2] memory _pAndLCoin) {
        require(token != address(0), 'Coin Factory: ZERO_ADDRESS');
        require(getDepositCoin[token] == address(0), 'Coin Factory: COIN_EXISTS');// single check is sufficient
        require(lendingManager != address(0), 'Coin Factory: Coin manager NOT Set');
        require(rewardContract != address(0), 'Coin Factory: Reward Contract NOT Set');
        require(depositType != 0, 'Coin Factory: Reward Type NOT Set');
        bytes32 _salt1 = keccak256(abi.encodePacked(token,msg.sender, "Deposit Coin"));
        bytes32 _salt2 = keccak256(abi.encodePacked(token,msg.sender, "Loan Coin"));
        //Only ERC20 Tokens Can creat pairs
        _pAndLCoin[0] = address(new depositOrLoanCoin{salt: _salt1}(0,token,lendingManager, rewardContract, strConcat(string(ERC20(token).symbol()), " Deposit Coin"),strConcat(string(ERC20(token).symbol()), " DCoin")));  //
        _pAndLCoin[1] = address(new depositOrLoanCoin{salt: _salt2}(1,token,lendingManager, rewardContract,strConcat(string(ERC20(token).symbol()), " Loan Coin"),strConcat(string(ERC20(token).symbol()), " LCoin")));
        getDepositCoin[token] = _pAndLCoin[0];
        getLoanCoin[token] = _pAndLCoin[1];
        iRewardMini(rewardContract).factoryUsedRegist(_pAndLCoin[0], depositType);
        iRewardMini(rewardContract).factoryUsedRegist(_pAndLCoin[1], LoanType);
        emit DepositCoinCreated( token, _pAndLCoin[0]);
```

	`emit LoanCoinCreatedX( token, _pAndLCoin[1]);` `}`
Recommendation	It is recommended to add permission checks for the caller, such as requiring the caller to be lendingManager contract.
Status	**Fixed.** Added restriction requiring the caller to be lendingManager.

```
 function createDeAndLoCoin(address token) external returns (address[2] memory _pAndLCoin) {
        require(msg.sender == lendingManager, 'Coin Factory: msg.sender MUST be lendingManager.');
...
    }
```

[xlending-06] Unit error in tokenLiquidateEstimate function

Severity Level	Low
Type	Business Security
Lines	lendingManager.sol#L553-580
Description	The final unit of maxAmounts[0] in the last else condition of the tokenLiquidateEstimate function is quantity, which is different from the value in other conditions.

```
    function tokenLiquidateEstimate(address user,
            address liquidateToken,
            address depositToken) public view returns(uint[2] memory
maxAmounts){
        if(viewUsersHealthFactor(user) >= 1 ether){
            uint[2] memory zero;
            return zero;
        }
        uint amountliquidate =
iDepositOrLoanCoin(assetsDepositAndLend[liquidateToken][0]).balanceO
f(user);
        uint amountDeposit =
iDepositOrLoanCoin(assetsDepositAndLend[depositToken][1]).balanceOf(
user);
        amountliquidate = amountliquidate *
iSlcOracle(oracleAddr).getPrice(liquidateToken) / 1 ether;
        amountDeposit = amountDeposit *
iSlcOracle(oracleAddr).getPrice(depositToken) / 1 ether
    * UPPER_SYSTEM_LIMIT / (UPPER_SYSTEM_LIMIT -
licensedAssets[liquidateToken].liquidationPenalty);
        if(amountliquidate < amountDeposit){
            maxAmounts[0] = amountliquidate;
            maxAmounts[1] = amountliquidate * (UPPER_SYSTEM_LIMIT -
licensedAssets[liquidateToken].liquidationPenalty) * 1 ether
        / (UPPER_SYSTEM_LIMIT *
iSlcOracle(oracleAddr).getPrice(depositToken));
        }else if(amountliquidate == amountDeposit){
            maxAmounts[0] = amountliquidate;
            maxAmounts[1] = amountDeposit;
        }else{
```

	`` maxAmounts[1] = amountDeposit; maxAmounts[0] = amountDeposit * 1 ether / iSlcOracle(oracleAddr).getPrice(depositToken); } } ```
Recommendation	It is recommended to unify the units of the return values according to actual design requirements.
Status	**Fixed.** The unit error part of this function has been redesigned and modified.

```
    function tokenLiquidateEstimate(address user,
                            address liquidateToken,
                            address depositToken) public view
returns(uint[2] memory maxAmounts){
...
            maxAmounts[0] = depositMaxValue * 1 ether /
liquidateTokenPrice;//At this point, this Token deposit of the liquidated
user will be fully liquidated
            maxAmounts[1] = amountDeposit;

        }
    }
```

[xlending-07] Missing trigger event

Severity Level	Info
Type	Coding Conventions
Lines	TOKEN.sol #L134-148
Description	Some administrator functions (such as `setup` in the lendingManager contract) did not trigger events when modifying key contract parameters.

```
    function setup( address _superLibraCoin,
                    address _xInterface,
                    address _coinFactory,
                    address _lendingVault,
                    address _riskIsolationModeAcceptAssets,
                    address _coreAlgorithm,
                    address _oracleAddr ) external onlySetter{
        superLibraCoin = _superLibraCoin;
        coinFactory = _coinFactory;
        xInterface = _xInterface;
        oracleAddr = _oracleAddr;
        lendingVault = _lendingVault;
        coreAlgorithm = _coreAlgorithm;
        riskIsolationModeAcceptAssets =
_riskIsolationModeAcceptAssets;
    }
```

Recommendation	It is recommended to emit events when modifying critical variables is a recommended practice as it provides a standardized way to capture and communicate important changes within the contract. Events enable transparency and allow external systems and users to easily track and react to these modifications.
Status	**Acknowledged.**

[xlending-08] Redundant code

Severity Level	Info
Type	Coding Conventions
Lines	lendingManager.sol#L113-117， L139
Description	In the lendingManager contract, `xInterface` and `slcValue` do not have specific application scenarios and are redundant codes.

```
    function slcValueRevaluate(uint newValue) public  onlySetter {
        slcValue = newValue;
        emit SlcValue(superLibraCoin,newValue);
    }
......
        xInterface = _xInterface;
```

Recommendation	It is recommended for the project team to remove the redundant code.
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