BEOSIN
Blockchain Security

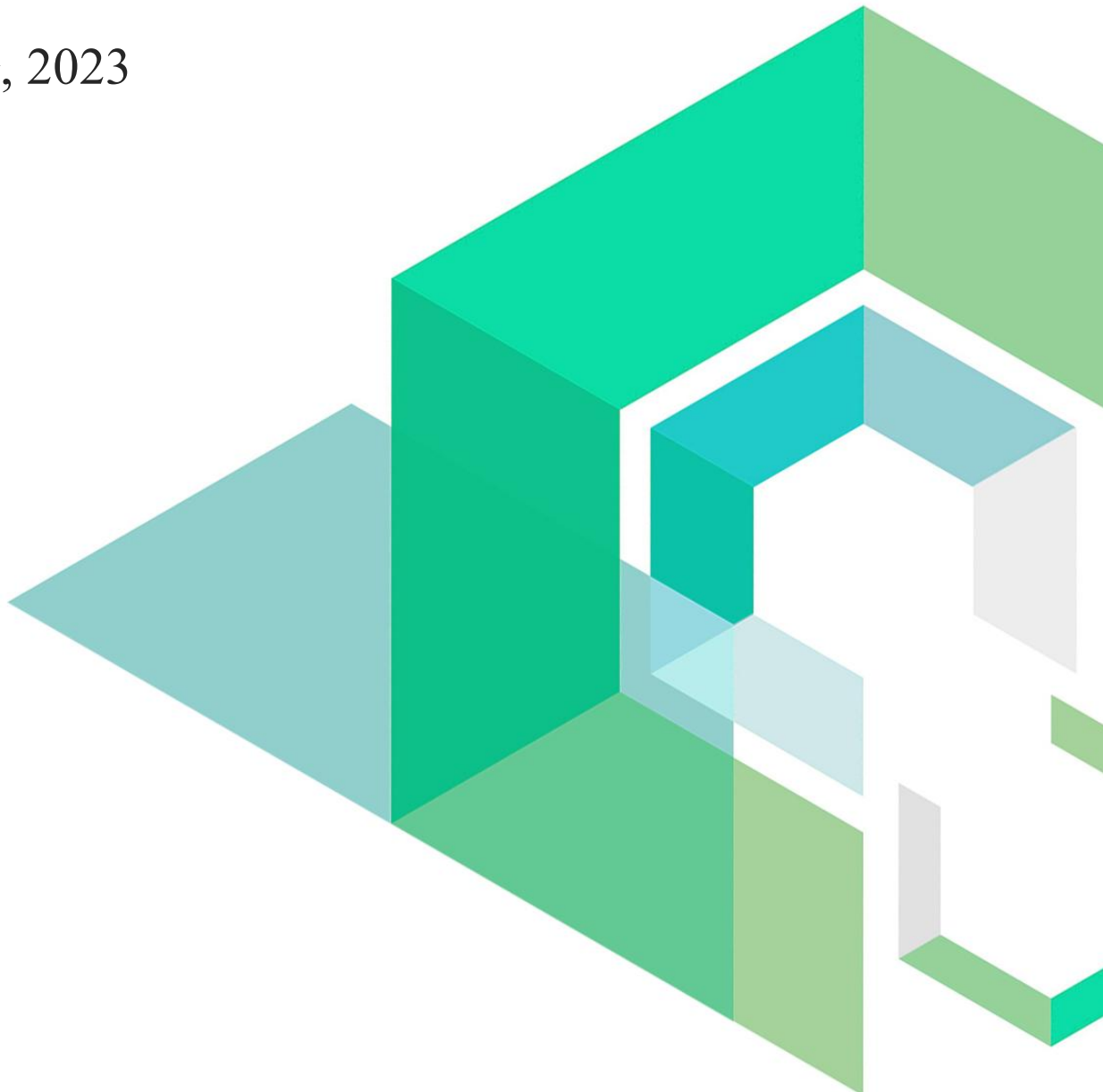
BetDex

Smart Contract Security Audit

V1.0

No. 202305111200

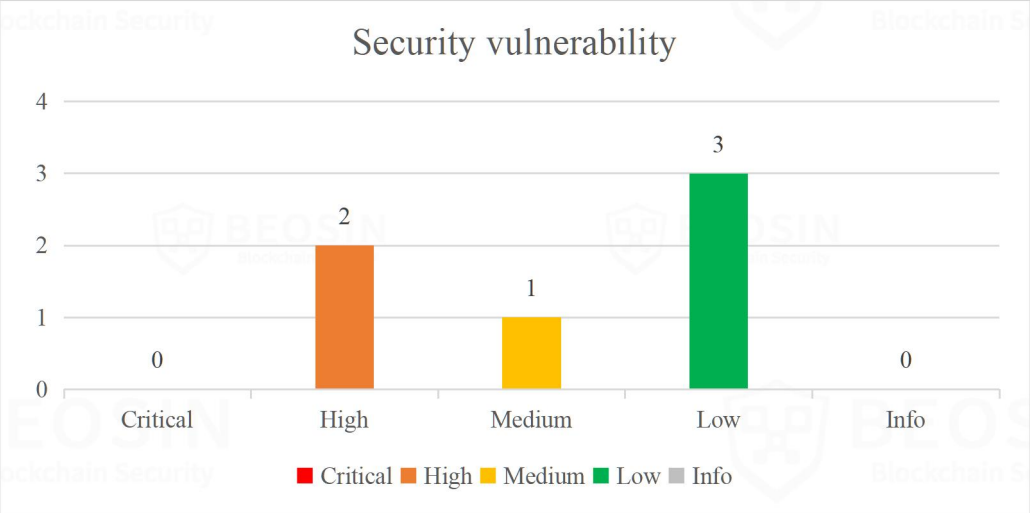
May 11th, 2023



Contents

Summary of Audit Results

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



*Notes:

- **Risk Description:**

1. ROLE_MANAGER can be passed in any lottery result and Maxodds has no restricted range. If the private key to this address is lost, an attacker can take control of the lottery results and Maxodds to steal large amounts of funding. If the ROLE_MANAGER role does not resolve the bet, the user's funds will be locked in the contract and cannot be withdrawn.
2. There is no limit to the feeToRoom variable, which may result in the loss of all funds if the user is rejected withdrawal.
3. Room owner is unable to withdraw his principal when a user bets or the contract is suspended.
4. The project does not support deflationary tokens.

- **Project Description:**

1. Business overview

The BetDex project is a betting lottery project. Participated by the project party, room owner and betting users. The project party can set up some privileged accounts through the contract to manage and run the contract. For example, roles such as ROLE_RANDOMIZER, ROLE_MANAGER, and ROLE_AUDITOR have the authority to call specific functions.

The logic of the project is that the room owner creates a room, pledges tokens to the room and pays a certain amount of handling fees. After that, the user can place a bet, which will pledge the user's token and lock it, and lock the token in the room according to the odds set by the user. The odds set by the user cannot be higher than the upper limit of odds set by the project party. After the bet is placed, the contract will record the betting status in the variable. After that, ROLE_MANAGER will use the *resolveBet* function to draw a prize. The lottery result is passed in as a parameter, and the betting status will be automatically deleted after the lottery. If the user has a reward, it will be recorded in the contract variable.

After a prize has been drawn, the user can withdraw their balance or use it to continue to place bets. There is a daily limit on the amount that can be withdrawn by a user, above which the ROLE_AUDITOR has to approve the withdrawal and a fee is charged to the room owner if the withdrawal is refused. The ROLE_AUDITOR role on the project side can authorise the Admin to withdraw the commission paid by the room owner.

1 Overview

1.1 Project Overview

Project Name	BetDex
Platform	EVM Compatible Chains
Audit scope	https://github.com/betdexproject/contract
Commit Hash	cf7c5f8d28a85b25ec38f3992efe4b1f5b53876a 68e73de8931d2c80dd325b58060d23841c98bd54

1.2 Audit Overview

Audit work duration: Apr 26, 2023 – May 11, 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
BetDex-1	Bet After Result	High	Fixed
BetDex-2	The owner can withdraw the user's funds	High	Fixed
BetDex-3	Centralization Risk	Medium	Acknowledged
BetDex-4	User withdrawal fees scope is not limited	Low	Acknowledged
BetDex-5	Deflationary tokens not considered	Low	Acknowledged
BetDex-6	WithdrawReserves function is not properly designed	Low	Acknowledged

Status Notes:

BetDex-3 is unfixed, if the private key to "ROLE_MANAGER" address is lost, an attacker can take control of the lottery results and Maxodds to steal large amounts of funding. And if the "ROLE_MANAGER" role does not resolve the bet, the user's funds will be locked in the contract and cannot be withdrawn.

BetDex-4 is unfixed, which may result in the loss of all funds if the user is rejected withdrawal.

BetDex-5 is unfixed, so the project does not support deflationary tokens.

BetDex-6 is unfixed, so when a user bets or the contract is suspended, the room owner cannot withdraw his capital.

Finding Details:

[BetDex-1] Bet After Result

Severity Level	High
Type	Business Security
Lines	BetDex.sol #L333
Description	The betting time limit is imperfect, and an attacker could wait until the project owner calls the *resolveBet* function to open the prize (the emphasis here is on the need to send the prize out twice; the hacker could create his own room and submit multiple bets to his room, which could result in not enough gas and a rollback of the transaction if the administrator sends out the prize all at once, then the administrator would need to send it twice), get the results and then place the bets, thus potentially emptying the prize pool. As an example, first the attacker obtains multiple signatures of the results, which contain the signatures of the correct answers. When the results are announced, the attacker can take the signatures of the correct results and place a bet, as the *_validateSignatures* function is incorrectly determined, resulting in "(input.timestamp - SIGNATURE_TIMESTAMP_THRESHOLD) < block.timestamp" is constant.

```
328    function _validateSignatures(Input calldata input, bytes32 hash)
329        internal
330        view
331        returns (bool)
332    {
333        require((input.timestamp - SIGNATURE_TIMESTAMP_THRESHOLD) < block.timestamp, "INVALID_TIMESTAMP");
334        return _validateAuthorization(keccak256(abi.encodePacked(hash, input.timestamp)), input.v, input.r, input.s);
335    //     return _validateAuthorization(hash, input.v, input.r, input.s, input.signatureVersion, input.extraSignature);
336    }
337
```

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

Recommendations	It is recommended that users be restricted to betting times that can only be done before the results are announced. It should be noted that the allowed betting time and the lottery opening time should be separated by a certain range to prevent hackers from monitoring the transactions and making a robbery.
Status	Fixed.

```
324    function _validateSignatures(Input calldata input, bytes32 hash)
325        internal
326        view
327        returns (bool)
328    {
329        require((input.timestamp + SIGNATURE_TIMESTAMP_THRESHOLD) >= block.timestamp, "INVALID_TIMESTAMP");
330        return _validateAuthorization(keccak256(abi.encodePacked(hash, input.timestamp)), input.v, input.r, input.s);
331    }
332
```

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

```
355     function _preparePlaceBets(Input calldata input, bytes32[] memory betHashes) internal returns (uint256, uint256)
356         uint256 totalPotentialWinningAmount;
357         uint256 totalBetAmount;
358         address bettorAddress;
359         uint betLength = input.bets.length;
360         for (uint i = 0; i < betLength; i = _uncheckedInc(i)) {
361             if (i == 0) {
362                 bettorAddress = input.bets[0].bettor;
363             }
364             require(input.bets[i].amount > minBetLimit[rooms[input.roomId].contractAddress], "Invalid amount");
365             require(bettorAddress == input.bets[i].bettor, "Not allow multiple bettor");
366             require(input.bets[i].odds <= maxOdds, "Exceed Max Odds");
367             require(matchResolved[input.bets[i].matchId] == false, "Match resolved");
368             uint256 potentialWinningAmount = (input.bets[i].amount * input.bets[i].odds) / (10**ODDS_DECIMAL);
369             totalPotentialWinningAmount += potentialWinningAmount;
370             totalBetAmount += input.bets[i].amount;
371             // Start 44000 gas fee
372             bytes32 betHash = betHashes[i];
373             Bet storage myBet = bets[betHash];
374             require(myBet.bettor == address(0), "BET_EXISTS");
375             myBet.bettor = input.bets[i].bettor;
376             myBet.target = input.bets[i].target;
377             myBet.potentialWinningAmount = potentialWinningAmount;
```

Figure 3 Source code of *_preparePlaceBets* function (fixed)

[BetDex-2] The owner can withdraw the user's funds

Severity Level	High
Type	Business Security
Lines	BetDex.sol #L113
Description	The *withdrawReserves* function is intended to allow the owner of the room to withdraw the funds staked to the contract. But in the owner permission piece, not only the current room owner can withdraw, but also the ROLE_MANAGER role can withdraw the user's funds. If the private key of the ROLE_MANAGER role is lost, the user's funds will be lost.

```
155     function withdrawReserves(
156         bytes32 roomId,
157         address _recipient,
158         uint256 _amount
159     ) external whenNotPaused nonReentrant onlyOwners(roomId) {
160         require(rooms[roomId].lockedReverse == 0, "LOCKED");
161         uint256 roomBalance = roomAvailableReserve(roomId);
162         require(roomBalance >= _amount, "INSUFFICIENT_BALANCE");
163         rooms[roomId].reverse -= _amount;
164         Token.withdrawal(rooms[roomId].contractAddress, _recipient, _amount);
165         emit ReservesWithdrawn(
166             roomId,
167             rooms[roomId].contractAddress,
168             msg.sender,
169             _recipient,
170             _amount
171         );
172     }
```

Figure 4 Source code of *withdrawReserves* function (unfixed)

```
105
106     modifier onlyAdmin() {
107         require(hasRole(DEFAULT_ADMIN_ROLE, msg.sender), "PERMISSION_DENIED");
108         _;
109     }
110
111     modifier onlyOwners(bytes32 roomId) {
112         require(
113             rooms[roomId].owner == msg.sender || hasRole(ROLE_MANAGER, msg.sender),
114             "PERMISSION_DENIED"
115         );
116         _;
117     }
118
```

Figure 5 Source code of *onlyOwners* modifier (unfixed)

Recommendations	It is recommended to delete this part of the code logic.
Status	Fixed.

```
107     modifier onlyOwners(bytes32 roomId) {
108         require(
109             rooms[roomId].owner == msg.sender,
110             "PERMISSION_DENIED"
111         );
112         _;
113     }
```

Figure 6 Source code of *onlyOwners* modifier (fixed)

[BetDex-3] Centralization Risk

Severity Level	Medium
Type	Business Security
Lines	BetDex.sol #L479
Description	If the private key of ROLE_MANAGER is compromised, the user's funds will be lost. Since ROLE_MANAGER can be passed in with arbitrary prize results and there is no limit to the Maxodds, an attacker can control the amount of profit by modifying the Maxodds by a large amount. Other setup parameters are the same and will not be pointed out here.

```
473
474     function resolveBet(
475         uint256 matchId,
476         uint16[] calldata _targets,
477 //      uint256 maxRoomLength,
478         uint256 maxBetLength
479     ) external whenNotPaused onlyRole(ROLE_MANAGER) {
480         uint256 processedRoomLength = gameBetRoomsArrProcessed[matchId];
481         uint256 roomLength = gameBetRoomsArr[matchId].length - processedRoomLength; // .min(maxRoo
482         uint256 targetLength = _targets.length;
483         uint256 processedRoomCount = 0;
484         uint256 totalProcessed = 0;
485         require(roomLength > 0, "NO_BETS");
486         for (uint i = 0; i < targetLength; i = _uncheckedInc(i)) {
487             targets[_targets[i]] = true;
488         }
```

Figure 7 Source code of *resolveBet* function (unfixed)

Recommendations	1. Maxodds should be limited to a reasonable range. 2. It is recommended that the relevant permission addresses are managed by a multi-signature wallet.
Status	Acknowledged.

[BetDex-4] User withdrawal fees scope is not limited

Severity Level	Low
Type	Business Security
Lines	BetDex.sol #L260-268
Description	There is no limit on the number of Feetoroom in the *rejectWithdrawal* function, which may cause users to lose all funds.

```
257         }
258     }
259
260     function rejectWithdrawal(uint256 id, uint256 feeToRoom) external whenNotPaused onlyRole(ROLE_AUDITOR) {
261         PendingWithdrawal storage pendingWithdrawal = pendingWithdrawals[id];
262         require(pendingWithdrawal.id == id, "PENDING_WITHDRAWAL_NOT_EXISTS");
263         require(pendingWithdrawal.executed == false, "WITHDRAWAL_PROCESSED");
264         pendingWithdrawal.executed = true;
265         rooms[pendingWithdrawal.roomId].reverse += feeToRoom;
266         uint256 feeToUser = pendingWithdrawal.amount - feeToRoom;
267         userBalances[pendingWithdrawal.roomId][pendingWithdrawal.userAddress] += feeToUser;
268         emit RejectWithdrawal(id, pendingWithdrawal.roomId, rooms[pendingWithdrawal.roomId].contractAddress, msg.sender,
269     }
270
271     function userRequestWithdrawalTimeout(uint256 id) external whenNotPaused nonReentrant {
272         PendingWithdrawal storage pendingWithdrawal = pendingWithdrawals[id];
273         require(pendingWithdrawal.id == id, "PENDING_WITHDRAWAL_NOT_EXISTS");
```

Figure 8 Source code of *rejectWithdrawal* function (unfixed)

Recommendations	It is recommended that feeToRoom takes a percentage, such as three percent.
Status	Acknowledged.

[BetDex-5] Deflationary tokens not considered

Severity Level	Low
Type	Business Security
Lines	BetDex.sol#139-171
Description	Deflationary tokens(Such as Safemoon) are not considered when creating a room in a contract, and the amount of the initiated transfer may not match the received one, which can lead to insufficient balance for subsequent transfers.

```
139     function depositReserves(
140         bytes32 roomId,
141         uint256 _amount,
142         uint256 _feeAmount
143     ) external whenNotPaused nonReentrant onlyOwners(roomId) {
144         Room storage room = rooms[roomId];
145         Token.deposit(room.contractAddress, msg.sender, _amount + _feeAmount);
146         room.reverse += _amount;
147         room.availableFee += _feeAmount;
148         emit ReservesDeposited(roomId, msg.sender, room.contractAddress, _amount);
149         emit FeeDeposited(roomId, msg.sender, room.contractAddress, _feeAmount);
150     }
151 
152     /// @notice Withdraw from the room. Only available to room owners.
153     /// @param _recipient Address of the withdraw recipient.
154     /// @param _amount Amount to withdraw.
155     function withdrawReserves(
156         bytes32 roomId,
157         address _recipient,
158         uint256 _amount
159     ) external whenNotPaused nonReentrant onlyOwners(roomId) {
160         require(rooms[roomId].lockedReverse == 0, "LOCKED");
161         uint256 roomBalance = roomAvailableReserve(roomId);
162         require(roomBalance >= _amount, "INSUFFICIENT_BALANCE");
163         rooms[roomId].reverse -= _amount;
164         Token.withdrawal(rooms[roomId].contractAddress, _recipient, _amount);
165         emit ReservesWithdrawn(
166             roomId,
167             rooms[roomId].contractAddress,
168             msg.sender,
169             _recipient,
170             _amount
171         );
172     }
```

Figure 9 Source code of related function (unfixed)

Recommendations	Determine if the number of tokens transferred is the same by checking the amount before and after the transfer, or restrict the deflationary token collateral.
Status	Acknowledged.

[BetDex-6] *WithdrawReserves* function is not properly designed

Severity Level	Low
Type	Business Security
Lines	BetDex.sol #L159-160
Description	Theoretically, the contract should not restrict the user to withdraw the principal, and when suspended, the user can still take out the principal of the funds. In line 160, it is judged that the lockedReverse must be equal to zero, which we think should not be necessary, because in the following *roomAvailableReserve* function, the user can only take out the withdrawable part.

```
154     /// @param _amount Amount to withdraw.
155     function withdrawReserves(
156         bytes32 roomId,
157         address _recipient,
158         uint256 _amount
159     ) external whenNotPaused nonReentrant onlyOwners(roomId) {
160         require(rooms[roomId].lockedReverse == 0, "LOCKED");
161         uint256 roomBalance = roomAvailableReserve(roomId);
162         require(roomBalance >= _amount, "INSUFFICIENT_BALANCE");
163         rooms[roomId].reverse -= _amount;
164         Token.withdrawal(rooms[roomId].contractAddress, _recipient, _amo
165         emit ReservesWithdrawn(
166             roomId,
167             rooms[roomId].contractAddress,
168             msg.sender,
169             _recipient,
170             _amount
171         );
172     }
173
```

Figure 10 Source code of *withdrawReserves* function (unfixed)

Recommendations	Remove whenNotPaused from *withdrawReserves* function and remove 160 lines of judgment.
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	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