BEOSIN
Blockchain Security

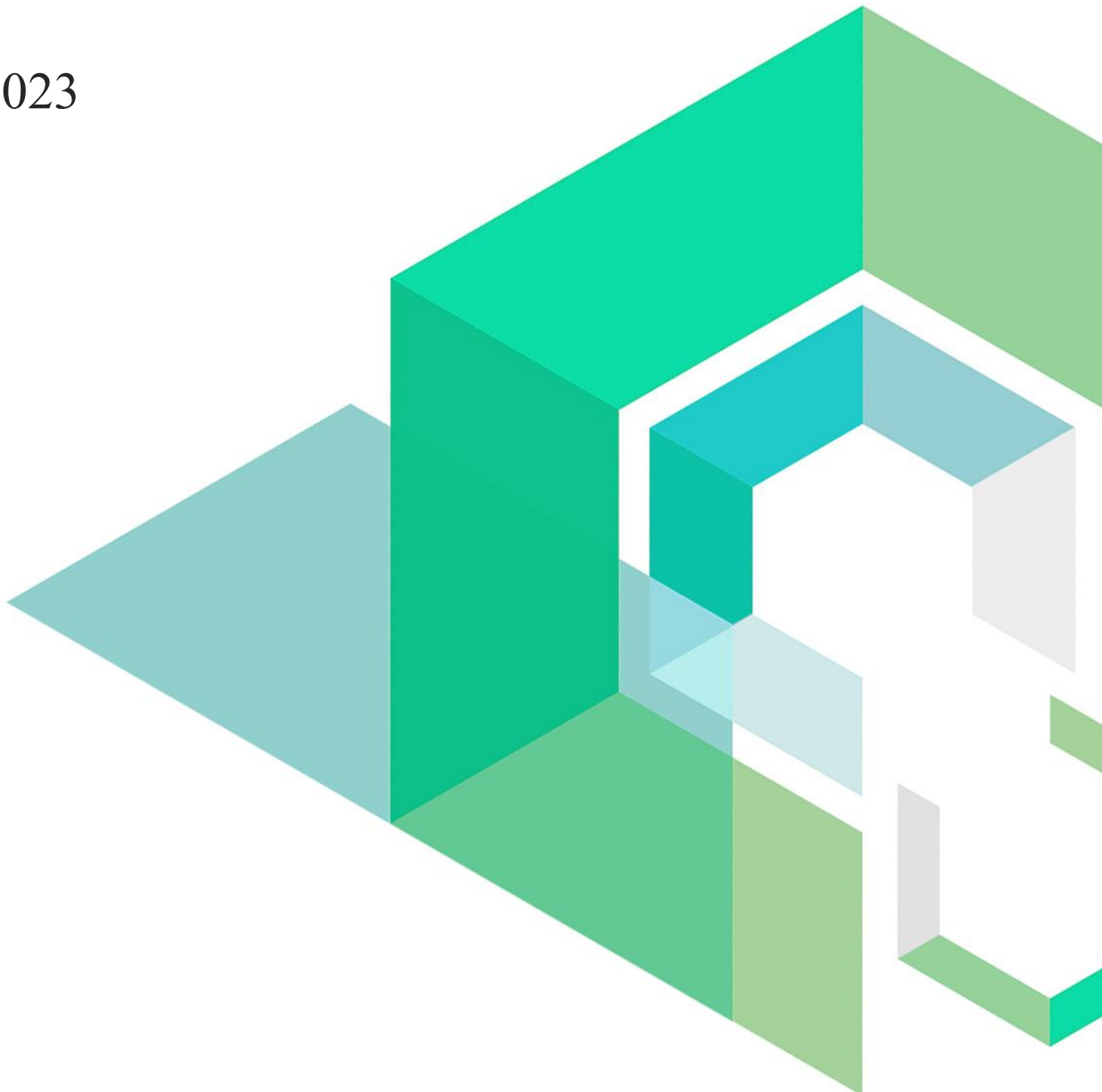
Debox

Smart Contract Security Audit

V1.0

No. 202301031842

Jan 3rd, 2023



Contents

Summary of Audit Results

After auditing, 3 Low and 3 Info-risks were identified in the Debox 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. In some special cases, club owner may not withdraw membership fee and get back the staked NFT. For details, see Debox-1, Debox-2 and Debox-3 in the Findings.

2. If _dAddr setting as the contract address will cause the club owner unable to withdraw membership fee and can't get back the staked NFT.

- **Project Description:**

1. Business overview

In the Debox project, users can create club by staking NFT (NFT must be in the whitelist). When creating a club, club owner can set the membership validity period and monthly fee for membership. Membership validity period should 6 or 12 months only unchangeable and re-join should wait to the next membership period. The monthly fee which is between 0.01 ethers and 10 ethers can be modified by club owner and each modification must be higher than the previous setting fee. Users can join the club by paying the membership fee (the monthly membership fee multiplied by the validity period of the membership). Joined Members can quit club and withdraw unused membership fees. There are two ways to quit: one is the member quit by himself (non refundable two-month membership fee is deducted), and the other is that the member quit by club owner. In addition, the club owner can withdraw the membership fee used by the member (the platform will charge 5% fee). After the club owner withdraws all the membership fee of the club, the club owner can release the club and get back the staked NFT.

1 Overview

1.1 Project Overview

Project Name	Debox
Platform	Ethereum
GitHub	https://github.com/debox-pro/debox-contract/blob/main/knowledge_payment/club_payment.sol
Commit Hash	0003cdd84c693c4d724814855b19afa0ad0415d7 1ae5fbf700a1788b3d30a233e82beb6644789761 9475366752002284 4852f82e26f182cd40cc8c6d

1.2 Audit Overview

Audit work duration: Dec 26, 2022 – Jan 03, 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
Debox-1	Out of gas	Low	Fixed
Debox-2	The calculateReceivableAmount function is poorly designed	Low	Acknowledged
Debox-3	Business Logics	Low	Acknowledged
Debox-4	No trigger event	Info	Acknowledged
Debox-5	Compiler version not fixed	Info	Acknowledged
Debox-6	Redundant code	Info	Acknowledged

Status Notes:

1. Debox-1 is fixed.
2. Debox-2 is not fixed. The *withdraw* function require receiveableAmount to be bigger than withdrawAmount. The attacker can control the value of receiveableAmount to be smaller than withdrawAmount, so that the withdraw function cannot be called successfully.
3. Debox-3 is not fixed. Because club.withdrawAmount is not an accumulative value in the *withdraw* function, club.withdrawAmount will not be equal to club.balance. *The releaseClub* call will fail, which will cause the club owner to fail to get back the NFT.
4. Debox-4 is not fixed and may not cause any issue.
5. Debox-5 is not fixed and may not cause any issue.
6. Debox-6 is not fixed and may not cause any issue.

Finding Details:

[Debox-1] Out of gas

Severity Level	**Low**
Type	Coding Conventions
Lines	club_payment.sol#L181-198 (0003cdd84c693c4d724814855b19afa0ad0415d7) club_payment.sol#L200-231 (0003cdd84c693c4d724814855b19afa0ad0415d7)
Description	As shown in the figure below, if the member array (idx_array) is too long, the function may fail to execute due to too many cycles and gas exhaustion.

```
181    function refundAll(address nft_ca, uint256 token_id) external callerIsUser {
182        Club memory club = getStakeInfo(nft_ca, token_id);
183        require(club.owner == msg.sender, "invalid owner address for current club");
184        uint256[] memory idx_array = _clubPayRecordIndexs[club.id];
185        for (uint i = 0; i < idx_array.length; ++i) {
186            PayRecord storage record = _allPayRecords[idx_array[i]];
187            if (record.refundType == RefundType.NONE && !isExpire(record.payTime, club.payMonths)) {
188                uint256 months = block.timestamp.sub(record.payTime).div(SECONDS_OF_MONTH);
189                uint256 remain_months = club.payMonths - months;
190                uint256 amount = record.payAmount.mul(remain_months).div(club.payMonths);
191                _staking[nft_ca][token_id].balance = _staking[nft_ca][token_id].balance.sub(amount);
192                record.refundType = RefundType.OWNER_REFUND;
193                record.refundAmount = amount;
194                payable(record.payAddr).transfer(amount);
195                emit Refund(msg.sender, club.id, record.payAddr, RefundType.OWNER_REFUND, amount);
196            }
197        }
198    }
```

Figure 1 Source code of *refundAll* function

As shown in the figure below, if the member array (idx_array) is too long, the transaction gas exceeds the gas limit, and the *getWithdrawBalance* call fails, causing the *withdraw* function call to fail, resulting in the club owner being unable to call the *releaseClub* function to release the club, and the club owner's NFT is locked in the contract.

```
200     function getWithdrawBalance(address nft_ca, uint256 token_id) public view returns (uint256) {
201         Club memory club = getStakeInfo(nft_ca, token_id);
202         uint256 amount = 0;
203         uint256[] memory idx_array = _clubPayRecordIndexs[club.id];
204         for (uint i = 0; i < idx_array.length; ++i) {
205             PayRecord memory record = _allPayRecords[idx_array[i]];
206             if (record.refundType == RefundType.NONE) {
207                 if (isExpire(record.payTime, club.payMonths)) {
208                     amount = amount.add(record.payAmount);
209                 }
210                 else {
211                     uint256 stay_months = block.timestamp.sub(record.payTime).div(SECONDS_OF_MONTH);
212                     amount = amount.add(record.payAmount.mul(stay_months).div(club.payMonths));
213                 }
214             }
215             else {
216                 amount = amount.add(record.payAmount.sub(record.refundAmount));
217             }
218         }
219         return amount.sub(club.withdrawAmount);
220     }
221
222     function withdraw(address nft_ca, uint256 token_id) external callerIsUser {
223         require(_staking[nft_ca][token_id].owner == msg.sender, "invalid owner address for current club");
224         uint256 balance = getWithdrawBalance(nft_ca, token_id);
225         require(balance > 0, "have no balance to withdraw");
226         uint256 fees = balance.div(20);
227         payable(msg.sender).transfer(balance.sub(fees));
228         _dAddr.transfer(fees);
229         _staking[nft_ca][token_id].withdrawAmount = _staking[nft_ca][token_id].withdrawAmount.add(balance);
230         emit Withdraw(msg.sender, _staking[nft_ca][token_id].id, balance);
231     }
```

Figure 2 Source code of related functions

Recommendations	It is recommended to execute in batches, or design new logic that without loop iteration.
Status	Fixed. The problem shown in Figure 1 has been fixed, and the code after the fix is shown in Figure 3.

```
200     function batchRefund(address nft_ca, uint256 token_id, address[] calldata addrs) external callerIsUser {
201         Club memory club = getStakeInfo(nft_ca, token_id);
202         require(club.owner == msg.sender, "invalid owner address for current club");
203         for (uint idx = 0; idx < addrs.length; ++idx) {
204             uint256 member_idx = _clubMemberPayIndex[club.id][addrs[idx]];
205             if (member_idx <= 0) {
206                 continue;
207             }
208             PayRecord storage record = _clubPayRecords[club.id][member_idx];
209             if (record.refundType == RefundType.NONE && record.expireTime > block.timestamp) {
210                 uint256 remain_months = calculateRemainMonths(record.payTime, club.payMonths, RefundType.OWNER_REFUND);
211                 uint256 amount = record.payAmount.mul(remain_months).div(club.payMonths);
212                 _staking[nft_ca][token_id].balance = _staking[nft_ca][token_id].balance.sub(amount);
213                 record.refundType = RefundType.OWNER_REFUND;
214                 record.refundAmount = amount;
215                 payable(record.payAddr).transfer(amount);
216                 emit Refund(msg.sender, club.id, record.payAddr, RefundType.OWNER_REFUND, amount);
217             }
218         }
219     }
```

Figure 3 Source code of *batchRefund* function (Fixed)

The problem shown in Figure 2 has been fixed, but it will cause new problems such as Debox-2 and Debox-3.

```
221     function getWithdrawAmont(PayRecord memory record, uint8 pay_months) internal view returns (uint256) {
222         if (record.refundType == RefundType.NONE) {
223             if (record.expireTime > block.timestamp) {
224                 uint256 stay_months = block.timestamp.sub(record.payTime).div(SECONDS_OF_MONTH);
225                 return record.payAmount.mul(stay_months).div(pay_months);
226             }
227             else {
228                 return record.payAmount;
229             }
230         }
231         return record.payAmount.sub(record.refundAmount);
232     }
233
234     function calculateReceivableAmount(address nft_ca, uint256 token_id, uint256 start, uint256 end) external callerIsUser {
235         Club memory club = getStakeInfo(nft_ca, token_id);
236         uint256 amount = 0;
237         PayRecord[] memory records = _clubPayRecords[club.id];
238         for (uint idx = start; idx < end && idx < records.length; ++idx) {
239             amount = amount.add(getWithdrawAmont(records[idx], club.payMonths));
240         }
241         _staking[nft_ca][token_id].receivableAmount = amount;
242         emit CalculateReceivableAmount(msg.sender, club.id, amount);
243     }
244
245     function withdraw(address nft_ca, uint256 token_id) external callerIsUser {
246         Club storage club = _staking[nft_ca][token_id];
247         require(club.owner == msg.sender, "invalid owner address for current club");
248         require(club.receivableAmount > club.withdrawAmount, "have no balance to withdraw");
249         uint256 balance = club.receivableAmount.sub(club.withdrawAmount);
250         uint256 fees = balance.div(20);
251         club.withdrawAmount = club.receivableAmount;
252         _dAddr.transfer(fees);
253         payable(msg.sender).transfer(balance.sub(fees));
254         emit Withdraw(msg.sender, _staking[nft_ca][token_id].id, balance);
255     }
256 }
```

Figure 4 Source code of related functions (Fixed)

[Debox-2] The calculateReceivableAmount function is poorly designed

Severity Level	**Low**
Type	Business Security
Lines	club_payment.sol#L234-256 (9475366752002284485 2f82e26f182cd40cc8c6d)
Description	The *calculateReceivableAmount* function can be called by anyone, causing the parameter receiveableAmount to be controlled. The attacker can control the value of receiveableAmount to be smaller than withdrawAmount, so that the *withdraw* function cannot be called successfully.

```
234     function calculateReceivableAmount(address nft_ca, uint256 token_id, uint256 start, uint256 end) external callerIsUser {
235         Club memory club = getStakeInfo(nft_ca, token_id);
236         uint256 amount = 0;
237         PayRecord[] memory records = _clubPayRecords[club.id];
238         for (uint idx = start; idx < end && idx < records.length; ++idx) {
239             amount = amount.add(getWithdrawAmont(records[idx], club.payMonths));
240         }
241         _staking[nft_ca][token_id].receivableAmount = amount;
242         emit CalculateReceivableAmount(msg.sender, club.id, amount);
243     }
244
245     function withdraw(address nft_ca, uint256 token_id) external callerIsUser {
246         Club storage club = _staking[nft_ca][token_id];
247         require(club.owner == msg.sender, "invalid owner address for current club");
248         require(club.receivableAmount > club.withdrawAmount, "have no balance to withdraw");
249         uint256 balance = club.receivableAmount.sub(club.withdrawAmount);
250         uint256 fees = balance.div(20);
251         club.withdrawAmount = club.receivableAmount;
252         _dAddr.transfer(fees);
253         payable(msg.sender).transfer(balance.sub(fees));
254         emit Withdraw(msg.sender, _staking[nft_ca][token_id].id, balance);
255     }
256 }
```

Figure 5 Source code of *calculateReceivableAmount* functions

Recommendations	It is recommended to add call permission.
Status	Acknowledged.

[Debox-3] Business Logics

Severity Level	**Low**
Type	Business Security
Lines	club_payment.sol#L234-256 (9475366752002284485 2f82e26f182cd40cc8c6d) club_payment.sol#L142-150 (9475366752002284485 2f82e26f182cd40cc8c6d)
Description	Because club.withdrawAmount is not an accumulative value in the *withdraw* function, club.withdrawAmount will not be equal to club.balance. *The releaseClub* call will fail, which will cause the club owner to fail to get back the NFT.

```
234     function calculateReceivableAmount(address nft_ca, uint256 token_id, uint256 start, uint256 end) external callerIsUser {
235         Club memory club = getStakeInfo(nft_ca, token_id);
236         uint256 amount = 0;
237         PayRecord[] memory records = _clubPayRecords[club.id];
238         for (uint idx = start; idx < end && idx < records.length; ++idx) {
239             amount = amount.add(getWithdrawAmont(records[idx], club.payMonths));
240         }
241         _staking[nft_ca][token_id].receivableAmount = amount;
242         emit CalculateReceivableAmount(msg.sender, club.id, amount);
243     }
244
245     function withdraw(address nft_ca, uint256 token_id) external callerIsUser {
246         Club storage club = _staking[nft_ca][token_id];
247         require(club.owner == msg.sender, "invalid owner address for current club");
248         require(club.receivableAmount > club.withdrawAmount, "have no balance to withdraw");
249         uint256 balance = club.receivableAmount.sub(club.withdrawAmount);
250         uint256 fees = balance.div(20);
251         club.withdrawAmount = club.receivableAmount;
252         _dAddr.transfer(fees);
253         payable(msg.sender).transfer(balance.sub(fees));
254         emit Withdraw(msg.sender, _staking[nft_ca][token_id].id, balance);
255     }
256 }
```

Figure 6 Source code of related functions

```
142     function releaseClub(address nft_ca, uint256 token_id) external callerIsUser {
143         Club memory club = getStakeInfo(nft_ca, token_id);
144         require(club.owner == msg.sender, "invalid owner address for current club");
145         require(club.balance == club.withdrawAmount, "please to withdraw/refund before release");
146         IERC721(nft_ca).transferFrom(address(this), msg.sender, token_id);
147         delete _staking[nft_ca][token_id];
148         delete _clubPayRecords[club.id];
149         emit ReleaseClub(msg.sender, club.id);
150     }
```

Figure 7 Source code of *releaseClub* function

Recommendations	It is recommended to design new logic.
Status	Acknowledged.

[Debox-4] No trigger event

Severity Level	Info
Type	Coding Conventions
Lines	club_payment.sol#L63-70 (0003cdd84c693c4d724814855b19afa0ad0415d7)
Description	Owner modified keyword parameters did not trigger the event.

```
63      function setAllowedContract(address nft_ca, bool isAllowed) external onlyOwner {
64          _stakedActive[nft_ca] = isAllowed;
65      }
66
67      function modifyDAddr(address new_addr) external onlyOwner {
68          require(new_addr != address(0), "invalid address");
69          _dAddr = payable(new_addr);
70      }
```

Figure 8 Source code of related functions

Recommendations	It is recommended to add and trigger corresponding events.
Status	Acknowledged.

[Debox-5] Compiler version not fixed

Severity Level	Info
Type	Coding Conventions
Lines	club_payment.sol#L4 (0003cdd84c693c4d724814855b19afa0ad0415d7)
Description	The compiler version is not fixed, which may cause some unexpected risks.

```
4  pragma solidity ^0.8.4;
5
6  import "@openzeppelin/contracts/token/ERC721/IERC721.sol";
7  import "@openzeppelin/contracts/access/Ownable.sol";
8  import "@openzeppelin/contracts/utils/math/SafeMath.sol";
9
10 contract ClubPayment is Ownable {
```

Figure 9 Source code of related problem

Recommendations	It is recommended to fix the compiler version.
Status	Acknowledged.

[Debox-6] Redundant code

Severity Level	Info
Type	Coding Conventions
Lines	club_payment.sol#L73-75 (9475366752002284485 2f82e26f182cd40cc8c6d)
Description	This function is not used.
Recommendations	It is recommended to delete this code.
Status	Acknowledged.

```
73    function isExpire(uint256 pay_time, uint8 pay_months) internal view returns (bool) {
74        return pay_time.add(SECONDS_OF_MONTH.mul(pay_months)) < block.timestamp;
75    }
```

Figure 10 Source code of *isExpire* function

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.

Impact / Likelihood	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