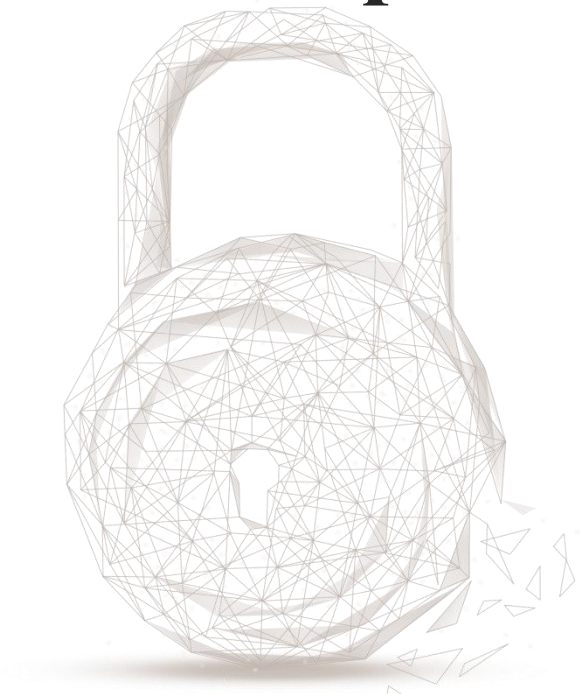
BEOSIN
Blockchain Security

Smart contract security audit report



Audit Number: 202111051700

Project Contract Name: NFTBombMarketUpgradeable

Project Deployment Platform: Binance Smart Chain

Project Contract Hash:

Contract Name	Hash(SHA256)
NFTBombMarketUpgradeable	ca5fa9feefaca1f5694c23be4678191fc19b0a015a041076933f2ffb69dc14b0 (Initial) 7f901a27047f17c800b5c2868a772010fdd06e3f9b5d35e74bd70a42186640c2 (Final)

Audit Start Date: 2021.10.25

Audit Completion Date: 2021.11.05

Audit Result: Pass

Audit Team: Beosin Technology Co. Ltd.

Audit Results Explained

Beosin Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of NFTBombMarketUpgradeable smart contracts, including Coding Standards, Security, and Business Logic. **After auditing, the NFTBombMarketUpgradeable project was found to have 9 risk items: 3 High-risks, 1 Medium-risk,1 Low-risk, 4 Info-risks. As of the completion of the audit, part of the risk items have been fixed or properly handled. The overall result of the NFTBombMarketUpgradeable smart contract is Pass.** The following is the detailed audit information for this project.

Index	Risk description	Risk level	Fix results
NFTB-1	*setFeeRate* function has too much authority	High	Ignored
NFTB-2	*cancelOrder* function has too much authority	High	Fixed
NFTB-3	The *buyOne* and *cancelOrder* functions are at risk of reentry	High	Fixed
NFTB-4	*buyOne* function does not determine the time	Medium	Fixed
NFTB-5	*_payThePrice* function implementation defects	Low	Fixed
NFTB-6	Redundant codes	Info	Fixed
NFTB-7	Compiler alerts	Info	Fixed
NFTB-8	Suggested optimizations for the *_transferAsset* function	Info	Fixed
NFTB-9	Suggested optimizations for the *buyOne* function	Info	Fixed

Table 1. Risk Statistics

Risk explained :

Item NFTB-1 is not fixed and may cause the seller not to get any revenue if the variable denominator is set to a value no greater than 10,000.

Risk descriptions and fix results explained

[NFTB-1 High] *setFeeRate* function has too much authority

Description: The contract owner can call the *setFeeRate* function to set the fee percentage. If the denominator is not greater than 10,000, the order creator may have no revenue.

```
38      function setFeeRate(uint256 feeRate_) external virtual onlyManager{
39          require(feeRate_ <= 10000, "Market: Invalid feeRate");
40          feeRate = feeRate_;
41          emit SetFeeRate(address(this), feeRate_, msg.sender);
42      }
```

Figure 1 source code of *setFeeRate* function

Fix recommendations: It is recommended that the fee rate be limited to a reasonable range.

Fix results: Ignored.

[NFTB-2 High] *cancelOrder* function has too much authority

Description: The contract owner can call the *cancelOrder* function to cancel the order created by the user and get the tokens that were sent to the contract when the user created the order.

```
161     function cancelOrder(uint256 orderId_) external onlyManagerOrSellor(orderId_) {
162         // 1. return the asset
163         Order storage order = orderList[orderId_];
164         require(order.status == Status.onlist, "Market: invalid orderId");
165
166         _transferAsset(
167             address(this),
168             msg.sender,
169             order.nftAddr,
170             order.tokenId,
171             order.tokenAmount,
172             order.assetType
173         );
174         // 2. set Order status to 2
175         order.status = Status.sold;
176         emit CancelOrder(address(this), orderId_, msg.sender);
177     }
```

Figure 2 source code of *cancelOrder* function (Unfixed)

Fix recommendations: It is suggested to change it to cancel the order and return the coins to the order creator.

Fix results: Fixed.

```
150     function cancelOrder(uint256 orderId_) external virtual onlyManagerOrSeller(orderId_) nonReentrant {
151         // 1. return the asset
152         Order storage order = orderList[orderId_];
153         require(order.status == Status.onlist, "Market: invalid orderId");
154 
155         _transferAsset(
156             address(this),
157             order.seller,
158             order.nftAddr,
159             order.tokenId,
160             order.tokenAmount,
161             order.assetType
162         );
163         // 2. set Order status to 2
164         order.status = Status.sold;
165         emit CancelOrder(address(this), orderId_, msg.sender);
166     }
```

Figure 3 source code of *cancelOrder* function (Fixed)

[NFTB-3 High] The *buyOne* and *cancelOrder* functions are at risk of reentry

Description: The buyOne and cancelOrder functions are at risk of reentry attacks.

```
208     function buyOne(uint256 orderId) external payable {
209         require(orderId <= currentOrderId,"Market: invalid orderId");
210         Order storage order = orderList[orderId];
211         require(order.status == Status.onlist, "Market: not onlist");
212 
213         _payThePrice(order);
214         _transferAsset(address(this), msg.sender, order.nftAddr, order.tokenId, order.tokenAmount, order.assetType);
215         order.status = Status.sold;
216         emit BuyOne(address(this), msg.sender, order.nftAddr, order.tokenId, order.tokenAmount, order.assetType);
217         //emit log
218     }
```

Figure 4 source code of *buyOne* function (Unfixed)

```
161     function cancelOrder(uint256 orderId_) external onlyManagerOrSellor(orderId_) {
162         // 1. return the asset
163         Order storage order = orderList[orderId_];
164         require(order.status == Status.onlist, "Market: invalid orderId");
165 
166         _transferAsset(
167             address(this),
168             msg.sender,
169             order.nftAddr,
170             order.tokenId,
171             order.tokenAmount,
172             order.assetType
173         );
174         // 2. set Order status to 2
175         order.status = Status.sold;
176         emit CancelOrder(address(this), orderId_, msg.sender);
177     }
```

Figure 5 source code of cancelOrder function (Unfixed)

Fix recommendations: It is recommended that token transfers be executed at the end.

Fix results: Fixed.

```
function buyOne(uint256 orderId) external virtual payable nonReentrant{
    require(orderId > 0,"Market: invalid orderId");
    require(orderId <= currentOrderId,"Market: invalid orderId");
    require(orderList[orderId].exTime > block.timestamp, "Market: time expired");
    Order storage order = orderList[orderId];
    require(order.status == Status.onlist, "Market: not onlist");

    _payThePrice(order);
    order.status = Status.sold;
    _transferAsset(address(this), msg.sender, order.nftAddr, order.tokenId, order.tokenAmount, order.assetType);
    emit BuyOne(address(this), msg.sender, order.nftAddr, order.tokenId, order.tokenAmount, order.assetType);
}
```

Figure 6 source code of *buyOne* function (Fixed)

```
function cancelOrder(uint256 orderId_) external virtual onlyManagerOrSeller(orderId_) nonReentrant {
    // 1. return the asset
    Order storage order = orderList[orderId_];
    require(order.status == Status.onlist, "Market: invalid orderId");

    _transferAsset(
        address(this),
        order.seller,
        order.nftAddr,
        order.tokenId,
        order.tokenAmount,
        order.assetType
    );
    // 2. set Order status to 2
    order.status = Status.sold;
    emit CancelOrder(address(this), orderId_, msg.sender);
}
```

Figure 7 source code of *cancelOrder* function (Fixed)

[NFTB-4 Medium] *buyOne* function does not determine the time

Description: The buyOne function does not internally determine the exTime set when the order is created.

```
function buyOne(uint256 orderId) external payable {
    require(orderId <= currentOrderId,"Market: invalid orderId");
    Order storage order = orderList[orderId];
    require(order.status == Status.onlist, "Market: not onlist");

    _payThePrice(order);
    _transferAsset(address(this), msg.sender, order.nftAddr, order.tokenId, order.tokenAmount, order.assetType);
    order.status = Status.sold;
    emit BuyOne(address(this), msg.sender, order.nftAddr, order.tokenId, order.tokenAmount, order.assetType);
    //emit log
}
```

Figure 8 source code of *buyOne* function (Unfixed)

Fix recommendations: It is recommended to add the judgment of exTime.

Fix results: Fixed.

```
197     function buyOne(uint256 orderId) external virtual payable nonReentrant{
198         require(orderId > 0,"Market: invalid orderId");
199         require(orderId <= currentOrderId,"Market: invalid orderId");
200         require(orderList[orderId].exTime > block.timestamp, "Market: time expired");
201         Order storage order = orderList[orderId];
202         require(order.status == Status.onlist, "Market: not onlist");
203 
204         _payThePrice(order);
205         order.status = Status.sold;
206         _transferAsset(address(this), msg.sender, order.nftAddr, order.tokenId, order.tokenAmount, order.assetType);
207         emit BuyOne(address(this), msg.sender, order.nftAddr, order.tokenId, order.tokenAmount, order.assetType);
208     }
```

Figure 9 source code of *buyOne* function (Fixed)

[NFTB-5 low] *_payThePrice* function implementation defects

Description: The *_payThePrice* function satisfies the condition that msg.value is not less than price if the payment token is a BNB. If msg.value is greater than price, it may cause the extra BNB to lock up in the contract.

```
222     function _payThePrice(Order storage order) internal{
223         uint256 price = order.price;
224         bytes32 assetType = order.assetType;
225         address paymentToken = order.paymentToken;
226         address seller = order.seller;
227 
228         address feeTo = addrc.getAddr("FEETO");
229         uint256 feeAmount = price.mul(feeRate).div(denominator);
230         uint256 remaining = price.sub(feeAmount);
231 
232         if(paymentToken == address(1)){
233             require(msg.value >= price, 'Market: Not enough ether to buy");
234             payable(feeTo).transfer(feeAmount);
235             payable(seller).transfer(remaining);
236             return;
237         }
238 
239         IERC20(paymentToken).transferFrom(msg.sender, feeTo, feeAmount);
240         IERC20(paymentToken).transferFrom(msg.sender, seller, remaining);
241     }
```

Figure 10 source code of *_payThePrice* function (Unfixed)

Fix recommendations: It is recommended to change it to 'msg.value==price'.

Fix results: Fixed.

```
212     function _payThePrice(Order storage order) virtual internal{
213         uint256 price = order.price;
214         address paymentToken = order.paymentToken;
215         address seller = order.seller;
216 
217         address feeTo = addrc.getAddr("FEETO");
218         uint256 feeAmount = price.mul(feeRate).div(denominator);
219         uint256 remaining = price.sub(feeAmount);
220 
221         if(paymentToken == address(1)){
222             require(msg.value == price, "Market: invalid amount ether to buy");
223             payable(feeTo).transfer(feeAmount);
224             payable(seller).transfer(remaining);
225             return;
226         }
```

Figure 11 source code of *_payThePrice* function (Fixed)

[NFTB-6 Info] Redundant codes

Description: There are some redundant codes in the contract.

```
222     function _payThePrice(Order storage order) internal{
223         uint256 price = order.price;
224         bytes32 assetType = order.assetType;
225         address paymentToken = order.paymentToken;
226         address seller = order.seller;
227 
228         address feeTo = addrc.getAddr("FEETO");
229         uint256 feeAmount = price.mul(feeRate).div(denominator);
230         uint256 remaining = price.sub(feeAmount);
231 
232         if(paymentToken == address(1)){
233             require(msg.value >= price, "Market: Not enough ether to buy");
234             payable(feeTo).transfer(feeAmount);
235             payable(seller).transfer(remaining);
236             return;
237         }
238 
239         IERC20(paymentToken).transferFrom(msg.sender, feeTo, feeAmount);
240         IERC20(paymentToken).transferFrom(msg.sender, seller, remaining);
241     }
```

Figure 12 source code of _payThePrice function

```
32      // ==== fee ====
33      address public feeTo;
34      uint256 public feeRate;      // 300
35      uint256 public denominator; // 10000
```

Figure 13 partial source code of the contract

Fix recommendations: It is recommended to remove the redundant codes.

Fix results: Fixed.

[NFTB-7 Info] Compiler alerts

Description: Compiler alerts exist when the contract is compiled.

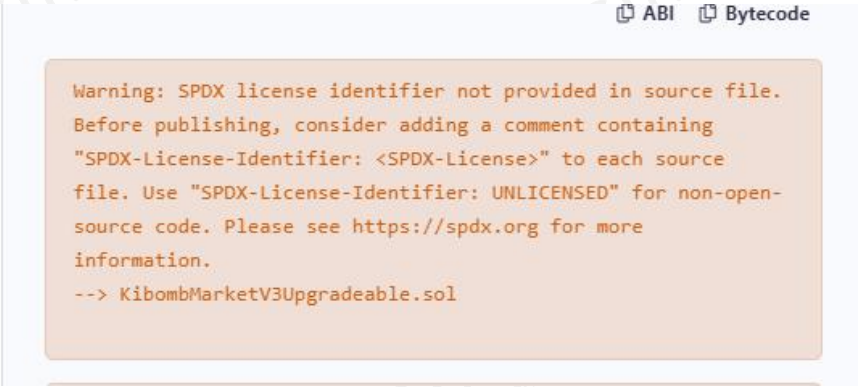


Figure 14 compiler alerts

Fix recommendations: It is recommended to eliminate compiler alerts.

Fix results: Fixed.

[NFTB-8 Info] Suggested optimizations for the *_transferAsset* function

Description: Since the value of assetType_ is of type bytes32, the post-conditions will not be met after the preceding conditions are met.

```
181   function _transferAsset(
182       address from_,
183       address to_,
184       address assetAddress_,
185       uint256 tokenId_,
186       uint256 tokenAmount_,
187       bytes32 assetType_
188   ) internal {
189       if( assetType_ == TYPE_NFT721){
190           IERC721(assetAddress_).safeTransferFrom(from_, to_,
              tokenId_);
191           console.log(msg.sender)
              ;                                   /// test should be
              removed
192       }
193
194       if(assetType_  == TYPE_NFT1155){
195           IERC1155(assetAddress_).safeTransferFrom(from_, to_,
              tokenId_, tokenAmount_, "0x00");
196       }
197
198       if(assetType_  == TYPE_ERC20){
199           if(from_ == address(this)){
200               IERC20(assetAddress_).transfer(to_, tokenAmount_);
201               return;
202           }
203           IERC20(assetAddress_).transferFrom(from_, to_,
              tokenAmount_);
204       }
205   }
```

Figure 15 source code of *_transferAsset* function (Unfixed)

Fix recommendations: It is recommended to return directly at the end of the 'if' internal code after the previous 'if' condition is satisfied to reduce unnecessary gas consumption.

Fix results: Fixed.

```
169     function _transferAsset(
170         address from_,
171         address to_,
172         address assetAddress_,
173         uint256 tokenId_,
174         uint256 tokenAmount_,
175         bytes32 assetType_
176     ) internal virtual {
177         if( assetType_ == TYPE_NFT721){
178             IERC721(assetAddress_).safeTransferFrom(from_, to_, tokenId_);
179             return;
180         }
181 
182         if(assetType_  == TYPE_NFT1155){
183             IERC1155(assetAddress_).safeTransferFrom(from_, to_, tokenId_, tokenAmount_, "0x00");
184             return;
185         }
186 
187         if(assetType_  == TYPE_ERC20){
188             if(from_ == address(this)){
189                 IERC20(assetAddress_).transfer(to_, tokenAmount_);
190                 return;
191             }
192             IERC20(assetAddress_).transferFrom(from_, to_, tokenAmount_);
193         }
194     }
```

Figure 16 source code of *_transferAsset* function (Fixed)

[NFTB-9 Info] Suggested optimizations for the *buyOne* function

Description: When creating an order, currentOrderId starts from 1. The function internally only makes a judgment that is less than currentOrderId.

```
208     function buyOne(uint256 orderId) external payable {
209         require(orderId <= currentOrderId,"Market: invalid orderId");
210         Order storage order = orderList[orderId];
211         require(order.status == Status.onlist, "Market: not onlist");
212 
213         _payThePrice(order);
214         _transferAsset(address(this), msg.sender, order.nftAddr, order.tokenId,
            order.tokenAmount, order.assetType);
215         order.status = Status.sold;
216         emit BuyOne(address(this), msg.sender, order.nftAddr, order.tokenId, order.
            tokenAmount, order.assetType);
217         //emit log
218     }
```

Figure 17 source code of *buyOne* function (Unfixed)

Fix recommendations: It is recommended to determine whether the input orderId is 0 or not.

Fix results: Fixed.

```
197     function buyOne(uint256 orderId) external virtual payable nonReentrant{
198         require(orderId > 0,"Market: invalid orderId");
199         require(orderId <= currentOrderId,"Market: invalid orderId");
200         require(orderList[orderId].exTime > block.timestamp, "Market: time expired");
201         Order storage order = orderList[orderId];
202         require(order.status == Status.onlist, "Market: not onlist");
203 
204         _payThePrice(order);
205         order.status = Status.sold;
206         _transferAsset(address(this), msg.sender, order.nftAddr, order.tokenId, order.tokenAmount, order.assetType);
207         emit BuyOne(address(this), msg.sender, order.nftAddr, order.tokenId, order.tokenAmount, order.assetType);
208     }
```

Figure 18 source code of *buyOne* function (Fixed)

Other audit items explained

1. Trading Fee Information

After an order created by a seller is purchased by a user, a handling fee is deducted. The handling fee rate is set by the contract owner. If the denominator is not greater than 10,000, the seller may not get any revenue.

Appendix 1 Description of Vulnerability Level

Vulnerability Level	Description	Example
Critical	Vulnerabilities that lead to the complete destruction of the project and cannot be recovered. It is strongly recommended to fix.	Malicious tampering of core contract privileges and theft of contract assets.
High	Vulnerabilities that lead to major abnormalities in the operation of the contract due to contract operation errors. It is strongly recommended to fix.	Unstandardized docking of the USDT interface, causing the user's assets to be unable to withdraw.
Medium	Vulnerabilities that cause the contract operation result to be inconsistent with the design but will not harm the core business. It is recommended to fix.	The rewards that users received do not match expectations.
Low	Vulnerabilities that have no impact on the operation of the contract, but there are potential security risks, which may affect other functions. The project party needs to confirm and determine whether the fix is needed according to the business scenario as appropriate.	Inaccurate annual interest rate data queries.
Info	There is no impact on the normal operation of the contract, but improvements are still recommended to comply with widely accepted common project specifications.	It is needed to trigger corresponding events after modifying the core configuration.

Appendix 2 Audit Categories and Details

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
3	Business Security	Business Logics
		Business Implementations

1. Coding Conventions

1.1. Compiler Version Security

The old version of the compiler may cause various known security issues. Developers are advised to specify the contract code to use the latest compiler version and eliminate the compiler alerts.

1.2. Deprecated Items

The Solidity smart contract development language is in rapid iteration. Some keywords have been deprecated by newer versions of the compiler, such as throw, years, etc. To eliminate the potential pitfalls they

may cause, contract developers should not use the keywords that have been deprecated by the current compiler version.

1.3. Redundant Code

Redundant code in smart contracts can reduce code readability and may require more gas consumption for contract deployment. It is recommended to eliminate redundant code.

1.4. SafeMath Features

Check whether the functions within the SafeMath library are correctly used in the contract to perform mathematical operations, or perform other overflow prevention checks.

1.5. require/assert Usage

Solidity uses state recovery exceptions to handle errors. This mechanism will undo all changes made to the state in the current call (and all its subcalls) and flag the errors to the caller. The functions assert and require can be used to check conditions and throw exceptions when the conditions are not met. The assert function can only be used to test for internal errors and check non-variables. The require function is used to confirm the validity of conditions, such as whether the input variables or contract state variables meet the conditions, or to verify the return value of external contract calls.

1.6. Gas Consumption

The smart contract virtual machine needs gas to execute the contract code. When the gas is insufficient, the code execution will throw an out of gas exception and cancel all state changes. Contract developers are required to control the gas consumption of the code to avoid function execution failures due to insufficient gas.

1.7. Visibility Specifiers

Check whether the visibility conforms to design requirement.

1.8. Fallback Usage

Check whether the Fallback function has been used correctly in the current contract.

2. General Vulnerability

2.1. Integer overflow

Integer overflow is a security problem in many languages, and they are especially dangerous in smart contracts. Solidity can handle up to 256-bit numbers (2**256-1). If the maximum number is increased by 1, it will overflow to 0. Similarly, when the number is a uint type, 0 minus 1 will underflow to get the maximum number value. Overflow conditions can lead to incorrect results, especially if its possible results are not expected, which may affect the reliability and safety of the program. For the compiler version after Solidity 0.8.0, smart contracts will perform overflow checking on mathematical operations by default. In the previous compiler versions, developers need to add their own overflow checking code, and SafeMath library is recommended to use.

2.2. Reentrancy

The reentrancy vulnerability is the most typical Ethereum smart contract vulnerability, which has caused the DAO to be attacked. The risk of reentry attack exists when there is an error in the logical order of calling the call.value() function to send assets.

2.3 Pseudo-random Number Generator (PRNG)

Random numbers may be used in smart contracts. In solidity, it is common to use block information as a random factor to generate, but such use is insecure. Block information can be controlled by miners or obtained by attackers during transactions, and such random numbers are to some extent predictable or collidable.

2.4. Transaction-Ordering Dependence

In the process of transaction packing and execution, when faced with transactions of the same difficulty, miners tend to choose the one with higher gas cost to be packed first, so users can specify a higher gas cost to have their transactions packed and executed first.

2.5. DoS(Denial of Service)

DoS, or Denial of Service, can prevent the target from providing normal services. Due to the immutability of smart contracts, this type of attack can make it impossible to ever restore the contract to its normal working state. There are various reasons for the denial of service of a smart contract, including malicious revert when acting as the recipient of a transaction, gas exhaustion caused by code design flaws, etc.

2.6. Function Call Permissions

If smart contracts have high-privilege functions, such as coin minting, self-destruction, change owner, etc., permission restrictions on function calls are required to avoid security problems caused by permission leakage.

2.7. call/delegatecall Security

Solidity provides the call/delegatecall function for function calls, which can cause call injection vulnerability if not used properly. For example, the parameters of the call, if controllable, can control this contract to perform unauthorized operations or call dangerous functions of other contracts.

2.8. Returned Value Security

In Solidity, there are transfer(), send(), call.value() and other methods. The transaction will be rolled back if the transfer fails, while send and call.value will return false if the transfer fails. If the return is not correctly judged, the unanticipated logic may be executed. In addition, in the implementation of the transfer/transferFrom function of the token contract, it is also necessary to avoid the transfer failure and return false, so as not to create fake recharge loopholes.

2.9. tx.origin Usage

The tx.origin represents the address of the initial creator of the transaction. If tx.origin is used for permission judgment, errors may occur; in addition, if the contract needs to determine whether the caller is the contract address, then tx.origin should be used instead of extcodesize.

2.10. Replay Attack

A replay attack means that if two contracts use the same code implementation, and the identity authentication is in the transmission of parameters, the transaction information can be replayed to the other contract to execute the transaction when the user executes a transaction to one contract.

2.11. Overriding Variables

There are complex variable types in Solidity, such as structures, dynamic arrays, etc. When using a lower version of the compiler, improperly assigning values to it may result in overwriting the values of existing state variables, causing logical exceptions during contract execution.

Appendix 3 Disclaimer

This report is made in response to the project code. No description, expression or wording in this report shall be construed as an endorsement, affirmation or confirmation of the project. This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the documents and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin. Due to the technical limitations of any organization, this report conducted by Beosin still has the possibility that the entire risk cannot be completely detected. Beosin disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin Technology.

Official Website

https://lianantech.com

E-mail

market@lianantech.com

Twitter

https://twitter.com/Beosin_com