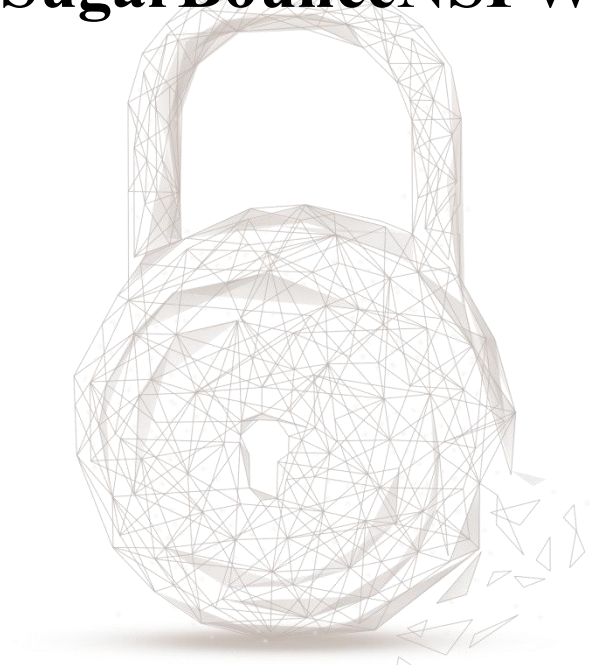
Smart Contract Audit Report

for

SugarBounceNSFW



Audit Number: 202202231750

Contract Name: SugarBounceNSFW

Deployment Platform: BNB Chain

Github Link: https://github.com/SugarBounceNSFW/sugarbounce-lp-yieldfarming

Commit Hash:

9feb77d75953b64628a473610e77906481c85a5b (Initial)

624f739ce68dcaa1723d31835162c60a06841d8c (Final)

Audit Start Date: 2022.02.14

Audit Completion Date: 2022.02.23

Audit Team: Beosin Technology Co. Ltd.

Audit Results Overview

Beosin Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of SugarBounceNSFW smart contracts, including Coding Conventions, General Vulnerability and Business Security. **After auditing, the SugarBounceNSFW smart contracts were found to have 2 Medium-risks, 4 Low-risks, 1 Info items.** The following is the detailed audit information for this project.

Index	Risk items	Risk level	Status
SBMasterChef-1	The *add*, *setSBPerBlock* and *set* function implementation exceptions	Medium	Fixed
SBMasterChef-2	Potential risks of the *pendingRewards* and *_updatePool* function	Low	Fixed
SBMasterChef-3	Centralization risk	Low	Acknowledged
SBMasterChef-4	Parameter type error	Low	Acknowledged
SBMasterChef-5	Reward abnormal risk	Low	Acknowledged
SBMasterChef-6	Lack of judgement on input data	Info	Fixed
Rewarder-1	Owner has a high authority	Medium	Acknowledged

Table 1 – Key Audit Findings

Risk description:

- SBMasterChef-3 is not fixed and may cause potential centralization risk.
- SBMasterChef-4 is not fixed and may cause an exception in the proportion of all staking pools.
- SBMasterChef-5 is not fixed and may cause the user cannot receive the reward normally.
- Rewarder-1 is not fixed and may cause the reward tokens in the rewarder contract to be taken out entirely.

Findings

[SBMasterChef-1 Medium] The *add*, *setSBPerBlock* and *set* function implementation exceptions

Description: When the *add, setSBPerBlock* and *set* functions add or update allocPoint, the accSBPerShare of the pool in use is not updated, which will cause problems in the calculation of the reward of the pool in use.

```
80      function add(uint256 allocPoint, address _lpToken) external onlyOwner nonReentrant {
81          require(!poolExistence[_lpToken], "SBMasterChef: Pool already exists");
82          require(_lpToken != address(0), "SBMasterChef: ZERO address");
83
84          totalAllocPoint = totalAllocPoint + allocPoint;
85          poolExistence[_lpToken] = true;
86
87          poolInfo.push(
88              PoolInfo({
89                  accSBPerShare: 0,
90                  allocPoint: uint128(allocPoint),
91                  lastRewardBlock: block.number,
92                  lpToken: _lpToken
93              })
94          );
95          emit LogPoolAddition(poolInfo.length - 1, allocPoint, _lpToken);
96      }
```

Figure 1 Source code of *add* function (Unfixed)

```
71      function setSBPerBlock(uint256 _sbPerBlock) external onlyOwner {
72          sbPerBlock = _sbPerBlock;
73          emit LogSetSBPerBlock(msg.sender, _sbPerBlock);
74      }
```

Figure 2 Source code of *setSBPerBlock* function (Unfixed)

```
101     function set(uint256 _pid, uint256 _allocPoint) external onlyOwner {
102         require(_pid < poolInfo.length, "SBMasterChef: Pool does not exist");
103         totalAllocPoint = totalAllocPoint - poolInfo[_pid].allocPoint + _allocPoint;
104         poolInfo[_pid].allocPoint = uint128(_allocPoint);
105         emit LogSetPool(_pid, _allocPoint);
106     }
```

Figure 3 Source code of *set* function (Unfixed)

Fix recommendations: It is recommended that the *add*, *setSBPerBlock* and *set* functions update the accSBPerShare of all pools in use when adding or updating allocPoint.

Status: Fixed.

```
73      function setSBPerBlock(uint256 _sbPerBlock) external onlyOwner {
74          require(_sbPerBlock != sbPerBlock, "It is old value");
75          massUpdatePools();
76
77          sbPerBlock = _sbPerBlock;
78          emit LogSetSBPerBlock(msg.sender, _sbPerBlock);
79      }
```

Figure 4 Source code of *setSBPerBlock* function (Fixed)

```
85      function add(
86          uint256 allocPoint,
87          address _lpToken,
88          bool _withUpdate
89      ) external onlyOwner nonReentrant {
90          require(!poolExistence[_lpToken], "SBMasterChef: Pool already exists");
91          require(_lpToken != address(0), "SBMasterChef: ZERO address");
92
93          if (_withUpdate) {
94              massUpdatePools();
95          }
96
97          totalAllocPoint = totalAllocPoint + allocPoint;
98          poolExistence[_lpToken] = true;
99
100         poolInfo.push(
101             PoolInfo({
102                 accSBPerShare: 0,
103                 allocPoint: uint128(allocPoint),
104                 lastRewardBlock: block.number,
105                 depositedAmount: 0,
106                 lpToken: _lpToken
107             })
108         );
109         emit LogPoolAddition(poolInfo.length - 1, allocPoint, _lpToken);
110     }
```

Figure 5 Source code of *add* function (Fixed)

```
115     function set(
116         uint256 _pid,
117         uint256 _allocPoint,
118         bool _withUpdate
119     ) external onlyOwner {
120         require(_pid < poolInfo.length, "SBMasterChef: Pool does not exist");
121         require(poolInfo[_pid].allocPoint != _allocPoint, "It is old alloc point");
122         if (_withUpdate) {
123             massUpdatePools();
124         }
125 
126         totalAllocPoint = totalAllocPoint - poolInfo[_pid].allocPoint + _allocPoint;
127         poolInfo[_pid].allocPoint = uint128(_allocPoint);
128         emit LogSetPool(_pid, _allocPoint);
129     }
```

Figure 6 Source code of *set* function (Fixed)

```
151     function massUpdatePools() public {
152         uint256 length = poolInfo.length;
153         for (uint256 pid = 0; pid < length; ++pid) {
154             _updatePool(pid);
155         }
156     }
```

Figure 7 Source code of *massUpdatePools* function

[SBMasterChef-2 Low] Potential risks of the *pendingRewards* and *_updatePool* functions

Description: The *pendingRewards* and *_updatePool* functions use the *balanceOf* function to get the amount of staking in the contract. If someone maliciously sends staking tokens to the contract, this may cause exceptions to reward queries and claims for other users.

```
112     function pendingRewards(uint256 _pid, address _user) external view returns (uint256 pending) {
113         PoolInfo storage pool = poolInfo[_pid];
114         UserInfo storage user = userInfo[_pid][_user];
115 
116         uint256 accSBPerShare = pool.accSBPerShare;
117         uint256 lpSupply = IERC20(pool.lpToken).balanceOf(address(this));
118 
119         if (block.number > pool.lastRewardBlock && lpSupply != 0) {
120             uint256 blocks = block.number - pool.lastRewardBlock;
121             uint256 sbReward = (blocks * sbPerBlock * pool.allocPoint) / totalAllocPoint;
122             accSBPerShare = accSBPerShare + ((sbReward * ACC_SB_PRECISION) / lpSupply);
123         }
124         pending = user.pendingRewards + (user.amount * accSBPerShare) / ACC_SB_PRECISION - uint256(user.rewardDebt);
125     }
```

Figure 8 Source code of *pendingRewards* function (Unfixed)

```
135     function _updatePool(uint256 pid) private {
136         PoolInfo storage pool = poolInfo[pid];
137         if (block.number > pool.lastRewardBlock) {
138             uint256 lpSupply = IERC20(pool.lpToken).balanceOf(address(this));
139             if (lpSupply > 0) {
140                 uint256 blocks = block.number - pool.lastRewardBlock;
141                 uint256 sbReward = (blocks * sbPerBlock * pool.allocPoint) / totalAllocPoint;
142                 pool.accSBPerShare = pool.accSBPerShare + uint128((sbReward * ACC_SB_PRECISION) / lpSupply);
143             }
144             pool.lastRewardBlock = block.number;
145             emit LogUpdatePool(pid, pool.lastRewardBlock, lpSupply, pool.accSBPerShare);
146         }
147     }
```

Figure 9 Source code of *_updatePool* function (Unfixed)

Fix recommendations: It is recommended that a separate variable is used to store the total staking amount.

Status: Fixed.

```
135     function pendingRewards(uint256 _pid, address _user) external view returns (uint256 pending) {
136         PoolInfo storage pool = poolInfo[_pid];
137         UserInfo storage user = userInfo[_pid][_user];
138 
139         uint256 accSBPerShare = pool.accSBPerShare;
140         uint256 lpSupply = pool.depositedAmount;
141 
142         if (block.number > pool.lastRewardBlock && lpSupply != 0) {
143             uint256 blocks = block.number - pool.lastRewardBlock;
144             uint256 sbReward = (blocks * sbPerBlock * pool.allocPoint) / totalAllocPoint;
145             accSBPerShare = accSBPerShare + ((sbReward * ACC_SB_PRECISION) / lpSupply);
146         }
147         pending = user.pendingRewards + (user.amount * accSBPerShare) / ACC_SB_PRECISION - uint256(user.rewardDebt);
148     }
```

Figure 10 Source code of *pendingRewards* function (Fixed)

```
166     function _updatePool(uint256 pid) private {
167         PoolInfo storage pool = poolInfo[pid];
168         if (block.number > pool.lastRewardBlock) {
169             uint256 lpSupply = pool.depositedAmount;
170             if (lpSupply > 0) {
171                 uint256 blocks = block.number - pool.lastRewardBlock;
172                 uint256 sbReward = (blocks * sbPerBlock * pool.allocPoint) / totalAllocPoint;
173                 pool.accSBPerShare = pool.accSBPerShare + uint128((sbReward * ACC_SB_PRECISION) / lpSupply);
174             }
175             pool.lastRewardBlock = block.number;
176             emit LogUpdatePool(pid, pool.lastRewardBlock, lpSupply, pool.accSBPerShare);
177         }
178     }
```

Figure 11 Source code of *_updatePool* function (Fixed)

[SBMasterChef-3 Low] Centralization risk

Description: The contract owner can call the *setRewarder*, *setSBPerBlock*, *add* and *set* functions to modify the relevant parameters in the contract, which may carry some risk of centralization.

```
65      function setRewarder(IRewarder _rewarder) external onlyOwner {
66          require(address(_rewarder) != address(rewarder), "It is old rewader");
67          require(address(_rewarder) != address(0), "SBMasterChef: ZERO address");
68          rewarder = _rewarder;
69
70          emit LogSetRewarder(msg.sender, address(_rewarder));
71      }
```

Figure 12 Source code of *setRewarder* function

Fix recommendations: It is recommended to use multi-signature wallet accounts, DAO or Timelock as the owner of the contract.

Status: Acknowledged. The project team replied that they will keep owner's address in secret place.

[SBMasterChef-4 Low] Parameter type error

Description: The type of allocPoint input in *add* and *set* functions of SBMakerChef contract is uint256, but the type of actual allocPoint is uint128. If the input allocPoint exceeds 2**128, it may cause an exception in the proportion of all staking pools.

```
80      function add(uint256 allocPoint, address _lpToken) external onlyOwner nonReentrant {
81          require(!poolExistence[_lpToken], "SBMasterChef: Pool already exists");
82          require(_lpToken != address(0), "SBMasterChef: ZERO address");
83
84          totalAllocPoint = totalAllocPoint + allocPoint;
85          poolExistence[_lpToken] = true;
86
87          poolInfo.push(
88              PoolInfo({
89                  accSBPerShare: 0,
90                  allocPoint: uint128(allocPoint),
91                  lastRewardBlock: block.number,
92                  lpToken: _lpToken
93              })
94          );
95          emit LogPoolAddition(poolInfo.length - 1, allocPoint, _lpToken);
96      }
```

Figure 13 Source code of *add* function (Unfixed)

```
101     function set(uint256 _pid, uint256 _allocPoint) external onlyOwner {
102         require(_pid < poolInfo.length, "SBMasterChef: Pool does not exist");
103         totalAllocPoint = totalAllocPoint - poolInfo[_pid].allocPoint + _allocPoint;
104         poolInfo[_pid].allocPoint = uint128(_allocPoint);
105         emit LogSetPool(_pid, _allocPoint);
106     }
```

Figure 14 Source code of *set* function (Unfixed)

Fix recommendations: It is recommended to modify to only input allocPoint of type uint128.

Status: Acknowledged. The project team replied that actually allocPoint is not so big as 2**128. They guess it can be at maximum 100 or 1000, so 100 * 100 or 1000 * 100 in smart contract.

[SBMasterChef-5 Low] Reward abnormal risk

Description: The *setRewarder* function of SBMakerChef contract can modify the rewarder address, there may be a potential risk that if the rewarder address is modified as a malicious address may cause the user cannot receive the reward normally.

```
64    function setRewarder(IRewarder _rewarder) external onlyOwner {
65        require(address(_rewarder) != address(0), "SBMasterChef: ZERO address");
66        rewarder = _rewarder;
67
68        emit LogSetRewarder(msg.sender, address(_rewarder));
69    }
```

Figure 15 Source code of *setRewarder* function (Unfixed)

Fix recommendations: It is recommended to modify it to set once.

Status: Acknowledged. The project team replied that it is very rare case to change Rewarder. They will be careful whenever setting Rewarder.

[SBMasterChef-6 Info] Lack of judgement on input data

Description: The *setRewarder*, *setSBPerBlock* and *set* functions do not determine whether the value entered is the same as the previous value.

```
64    function setRewarder(IRewarder _rewarder) external onlyOwner {
65        require(address(_rewarder) != address(0), "SBMasterChef: ZERO address");
66        rewarder = _rewarder;
67
68        emit LogSetRewarder(msg.sender, address(_rewarder));
69    }
```

Figure 16 Source code of *setRewarder* function (Unfixed)

```
71    function setSBPerBlock(uint256 _sbPerBlock) external onlyOwner {
72        sbPerBlock = _sbPerBlock;
73        emit LogSetSBPerBlock(msg.sender, _sbPerBlock);
74    }
```

Figure 17 Source code of *setSBPerBlock* function (Unfixed)

```
101   function set(uint256 _pid, uint256 _allocPoint) external onlyOwner {
102       require(_pid < poolInfo.length, "SBMasterChef: Pool does not exist");
103       totalAllocPoint = totalAllocPoint - poolInfo[_pid].allocPoint + _allocPoint;
104       poolInfo[_pid].allocPoint = uint128(_allocPoint);
105       emit LogSetPool(_pid, _allocPoint);
106   }
```

Figure 18 Source code of *set* function (Unfixed)

Fix recommendations: It is recommended to determine whether the new value is the same as the original value.

Status: Fixed.

```
65    function setRewarder(IRewarder _rewarder) external onlyOwner {
66        require(address(_rewarder) != address(rewarder), "It is old rewader");
67        require(address(_rewarder) != address(0), "SBMasterChef: ZERO address");
68        rewarder = _rewarder;
69
70        emit LogSetRewarder(msg.sender, address(_rewarder));
71    }
```

Figure 19 Source code of *setRewarder* function (Fixed)

```
73     function setSBPerBlock(uint256 _sbPerBlock) external onlyOwner {
74         require(_sbPerBlock != sbPerBlock, "It is old value");
75         massUpdatePools();
76
77         sbPerBlock = _sbPerBlock;
78         emit LogSetSBPerBlock(msg.sender, _sbPerBlock);
79     }
```

Figure 20 Source code of *setSBPerBlock* function (Fixed)

```
115   function set(
116       uint256 _pid,
117       uint256 _allocPoint,
118       bool _withUpdate
119   ) external onlyOwner {
120       require(_pid < poolInfo.length, "SBMasterChef: Pool does not exist");
121       require(poolInfo[_pid].allocPoint != _allocPoint, "It is old alloc point");
122       if (_withUpdate) {
123           massUpdatePools();
124       }
125
126       totalAllocPoint = totalAllocPoint - poolInfo[_pid].allocPoint + _allocPoint;
127       poolInfo[_pid].allocPoint = uint128(_allocPoint);
128       emit LogSetPool(_pid, _allocPoint);
129   }
```

Figure 21 Source code of *set* function (Fixed)

[Rewarder-1 Medium] Owner has a high authority

Description: The *withdrawAsset* function of Rewarder contract can withdraw any tokens from the contract. This may cause the user to not receive sufficient rewards.

```
34      function withdrawAsset(
35          address _token,
36          address _to,
37          uint256 _amount
38      ) external onlyOwner {
39          TransferHelper.safeTransfer(_token, _to, _amount);
40      }
41  }
```

Figure 22 Source code of *withdrawAsset* function (Unfixed)

Fix recommendations: It is recommended to change it to not being able to withdraw the reward tokens.

Status: Acknowledged. The project team replied that there can be several cases such as admin deposit wrong asset or excessive reward tokens. In that case admin should withdraw assets from rewarder contract.

Other Audit Items Descriptions

1. Centralization risk

The contract owner can call the *setRewarder*, *setSBPerBlock*, *add* and *set* functions to modify the relevant parameters in the contract, which may carry some risks of centralization.

- If the contract owner calls the *add* and *set* functions to add or modify the allocPoint of the staking pool and input more than 2**128 data, it may cause the proportion of all staking pools to be small, resulting in abnormal reward calculation.

  Project team replied: But actually allocPoint is not so big as 2**128. They guess it can be at maximum 100 or 1000, so 100 * 100 or 1000 * 100 in smart contract.

- The *setRewarder* function of SBMakerChef contract can modify the rewarder address, there may be a potential risk that if the rewarder address is modified as a malicious address may cause the user cannot receive the reward normally.

  Project team replied: It is very rare case to change Rewarder. They will be careful whenever setting Rewarder.

- The *withdrawAsset* function of Rewarder contract can withdraw any tokens from the contract.

  Project team replied: They will always tries to deposit enough reward tokens. All assets in Rewarder will be deposited by their admin. There can be several cases such as admin deposit wrong asset or excessive reward tokens. In that case admin should withdraw assets from rewarder contract.

2. Precautions for staking tokens

It is not recommended to use non-standard BEP-20 tokens as staking tokens, such as SafeMoon tokens, tokens that charge a fee for transferring funds, and inflation-deflation tokens.

3. Staking Bonus Related

If there are not enough reward tokens in the Rewarder contract, the user may receive less reward tokens than expected, and the unclaimed rewards are stored in the pendingRewards.

Appendix 1 Vulnerability Severity Level and Status Description

- Vulnerability Severity 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.

- Fix Results Status

Status	Description
Fixed	The project team fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project team confirms and chooses to ignore the issue.

Appendix 2 Description of 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
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.

3. Business Security

3.1 Business Logic

Whether the business logic is designed clearly and flawlessly.

3.2 Business Implementations

Whether the code implementation conforms to comments, project whitepaper, etc.

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.

Appendix 4 About Beosin

BEOSIN is a leading global blockchain security company dedicated to the construction of blockchain security ecology, with team members coming from professors, post-docs, PhDs from renowned universities and elites from head Internet enterprises who have been engaged in information security industry for many years. BEOSIN has established in-depth cooperation with more than 100 global blockchain head enterprises; and has provided security audit and defense deployment services for more than 1,000 smart contracts, more than 50 blockchain platforms and landing application systems, and nearly 100 digital financial enterprises worldwide. Relying on technical advantages, BEOSIN has applied for nearly 50 software invention patents and copyrights.

Official Website

https://beosin.com

E-mail

contact@beosin.com

Twitter

https://twitter.com/Beosin_com