Security Audit Report for StaFi rBNB Contracts

Date: June 13, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.06.14 10:51:37 +08'00'

Contents

Report Manifest

Item	Description
Client	StaFi
Target	StaFi rBNB Contracts

Version History

Version	Date	Description
1.0	June 13, 2023	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The objective of this audit is to review the code repository of StaFi rBNB Contracts [1] for the StaFi project. rBNB by StaFi is a decentralized DeFi product that provides a liquid staking solution for staking BNB on the BNB Smart Chain (BSC). rBNB was initially issued on the StaFi chain, and the upgrade being audited involves an overall technical architecture enhancement, with the migration of relevant logic and data to BSC. Users can perform operations such as stake, unstake, and withdraw through the stake manager contract on BSC. The scope of this audit is limited to contracts located within the `contracts/rbnb` folder of the original repository. Other files are not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
StaFi rBNB Contracts	`Version 1`	`8071fdd1fa886a9b9a009362467a50da2f659e3b`
	`Version 2`	`73dbc2e3546cc6a4785f966c87a19f5adf2c497e`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

[1] https://github.com/stafiprotocol/rtoken-contracts

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **one** potential issue. We also have **two** recommendations and **two** notes.

- Low Risk: 1
- Recommendation: 2
- Note: 2

ID	Severity	Description	Category	Status
1	Low	Potential insufficient verification of payable addresses	Software Security	Acknowledged
2	-	Check the threshold with the number of voters	Recommendation	Fixed
3	-	Verify the number of voters	Recommendation	Fixed
4	-	Centralization risk	Note	-
5	-	Arbitrage opportunity on new era	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential insufficient verification of payable addresses

Severity Low

Status Acknowledged

Introduced by `Version 1`

Description For the `stakeWithPool` and `unstakeWithPool` functions in the `StakeManager` contract, a check is performed to determine if `msg.sender` is payable (i.e., able to receive the native token). This check for a payable address is implemented in lines 283-284 of the following code snippet.

```
280    function stakeWithPool(address _poolAddress, uint256 _stakeAmount) public payable {
281        require(msg.value >= _stakeAmount.add(getStakeRelayerFee()), "fee not enough");
282        require(_stakeAmount >= minStakeAmount, "amount not enough");
283        require(bondedPools.contains(_poolAddress), "pool not exist");
284        (bool success, ) = msg.sender.call{gas: transferGas}("");
285        require(success, "staker not payable");
```

Listing 2.1: StakeManager.sol

Unfortunately, this check for payable addresses is not comprehensive. While the check can determine if the destination address has implemented the `fallback` or `receive` function, it cannot ascertain if the `fallback` function is marked as `payable`. In other words, a contract with a non-payable `fallback` can pass this check, but it would still be unable to receive native tokens.

Impact Some contracts that are unable to receive native tokens may still pass this check.

Suggestion Refactor the corresponding check.

Feedback from the Project We think these checks are useful in some scenarios. For example, if the caller is a contract, the contract must implement the receive function or the fallback function modified by payable. So maybe we should keep these checks.

2.2 Additional Recommendation

2.2.1 Check the threshold with the number of voters

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `Multisig` contract serves as a base contract that implements multi-signature functionality for the privileged `newEra` function within the `StakeManager` contract. In typical implementations of threshold voting, a limit exists between the threshold and the number of voters. Generally, the threshold should be greater than $\frac{2}{3}$ of the total voters. It is recommended to implement this check in the `changeThreshold` function.

```
68    function changeThreshold(uint256 _newThreshold) external onlyAdmin {
69        require(voters.length() >= _newThreshold && _newThreshold > 0, "invalid threshold");
70        threshold = _newThreshold.toUint8();
71    }
```

Listing 2.2: Multisig.sol

Impact This may result in an excessively low threshold.

Suggestion Implement a check to maintain the lower limit of the threshold.

2.2.2 Verify the number of voters

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Multisig` contract, a 16-bit bitmap is present within the `Proposal` structure to record the vote history. As a result, there can be a maximum of 16 voters. However, this limit is not enforced in the `initMultisig` and `addVoter` functions.

```
20    struct Proposal {
21        ProposalStatus _status;
22        uint16 _yesVotes; // bitmap, 16 maximum votes
23        uint8 _yesVotesTotal;
24    }
```

Listing 2.3: Multisig.sol

The `_yesVotes` bitmap holds the vote state bitmap. The index for each voter is calculated using the following functions.

```
20    function getVoterIndex(address _voter) public view returns (uint256) {
21        return voters._inner._indexes[bytes32(uint256(_voter))];
22    }
23
24    function voterBit(address _voter) internal view returns (uint256) {
25        return uint256(1) << getVoterIndex(_voter).sub(1);
26    }
```

Listing 2.4: Multisig.sol

In the code snippet below, the `_checkProposal` function may fail on Line 99 due to the cast to `uint16` for the `_yesVotes` bitmap. For instance, if there are 17 voters and the 17th voter casts their vote, the `_checkProposal` would revert because of the cast.

```
90    function _checkProposal(bytes32 _proposalId) internal view returns (Proposal memory proposal)
          {
91        proposal = proposals[_proposalId];
92
93        require(uint256(proposal._status) <= 1, "proposal already executed");
94        require(!_hasVoted(proposal, msg.sender), "already voted");
95
96        if (proposal._status == ProposalStatus.Inactive) {
97            proposal = Proposal({_status: ProposalStatus.Active, _yesVotes: 0, _yesVotesTotal: 0});
98        }
99        proposal._yesVotes = (proposal._yesVotes | voterBit(msg.sender)).toUint16();
100       proposal._yesVotesTotal++;
101   }
```

Listing 2.5: Multisig.sol

Impact If there are more than 16 voters, the function `_checkProposal` would revert.

Suggestion Verify the number of voters before adding a new voter.

2.3 Note

2.3.1 Centralization risk

Description The reward distribution mechanism for rBNB relies on a custom time unit called "era", which represents a period or timeframe. However, the duration of each era is not fixed, as the end of an era is determined by invoking the privileged `newEra` function. Moreover, the reward distribution information is not derived from the native BNB staking contract but is passed as parameters to the `newEra` function instead.

This introduces a risk of centralization, as privileged accounts have control over the timing and amount of reward distribution.

Feedback from the Project Not all rewards will be synchronized from BC to the BSC contract, so in order to ensure the accuracy of the reward, we need to query more accurate data through the BC API and synchronize it to the contract. Of course, if all rewards can be queried in the BSC contract in the future, we will upgrade the contract and get the reward directly from the BSC staking contract to avoid this risk.

2.3.2 Arbitrage opportunity on new era

Description The reward distribution mechanism for rBNB relies on a custom time unit called "era", and the `newEra` privileged function is employed to initiate a new era and distribute rewards to stakers. The reward distribution rate is established by the `rate` state variable, which represents the conversion rate between rBNB and staked BNB. When the `newEra` function is invoked, it claims staking rewards and increases the `rate`, allowing rBNB tokens to be converted into a greater amount of staked BNB.

However, the stake-unstake process lacks a locking period, which creates an arbitrage opportunity where the `newEra` function call can be sandwiched by stake-unstake transactions. By staking just prior

to calling the `newEra` function and unstaking afterward, anyone can secure risk-free profits due to the era switch. Although the BNB acquired from the unstake process must be locked, this arbitrage opportunity is cost-free.

The following code snippet demonstrates that the quantity of BNB to be unstaked is determined by the current `rate` value in the `unstakeWithPool` function.

```
305    function unstakeWithPool(address _poolAddress, uint256 _rTokenAmount) public payable {
306        require(_rTokenAmount > 0, "rtoken amount zero");
307        require(msg.value >= getUnstakeRelayerFee(), "fee not enough");
308        require(bondedPools.contains(_poolAddress), "pool not exist");
309        (bool success, ) = msg.sender.call{gas: transferGas}("");
310        require(success, "unstaker not payable");
311        require(unstakeOfUser[msg.sender].length() <= 100, "unstake number limit"); //todo test max
               limit number
312
313        uint256 unstakeFee = _rTokenAmount.mul(unstakeFeeCommission).div(1e18);
314        uint256 leftRTokenAmount = _rTokenAmount.sub(unstakeFee);
315        uint256 tokenAmount = leftRTokenAmount.mul(rate).div(1e18);
```

Listing 2.6: StakeManager.sol

The following code snippet shows the `rate` update in the `newEra` function.

```
550    // update rate
551    uint256 newRate = totalNewActive.mul(1e18).div(totalRTokenSupply);
552    uint256 rateChange = newRate > rate ? newRate.sub(rate) : rate.sub(newRate);
553    require(rateChange.mul(1e18).div(rate) < rateChangeLimit, "rate change over limit");
554
555    rate = newRate;
556    eraRate[_era] = newRate;
```

Listing 2.7: StakeManager.sol

Feedback from the Project We have considered this issue, and it is indeed possible to arbitrage. However, the BNB is not arrived immediately after unstaking, and it needs to wait for 16 days before withdrawing these funds, there will be some locking costs. So we think it's generally acceptable.