Security Audit Report for RingCore Contracts

Date: Feb 20, 2024

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by BlockSec Audit Team
Date: 2024.02.21 15:13:34 +08'00'

Contents

1 Introduction **1**

1.1 About Target Contracts . . . 1
1.2 Disclaimer . . . 1
1.3 Procedure of Auditing . . . 1
1.3.1 Software Security . . . 2
1.3.2 DeFi Security . . . 2
1.3.3 NFT Security . . . 2
1.3.4 Additional Recommendation . . . 2
1.4 Security Model . . . 3

2 Findings **4**

2.1 DeFi Security . . . 4
2.1.1 Potential unfairness in the reward distribution . . . 4
2.2 Additional Recommendation . . . 5
2.2.1 Optimize gas usage . . . 5
2.3 Notes . . . 6
2.3.1 Potential centralization risks . . . 6

Report Manifest

Item	Description
Client	Few
Target	RingCore Contracts

Version History

Version	Date	Description
1.0	Feb 20, 2024	First Version

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team 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 released detailed analysis reports of high-impact security incidents. 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 target of this audit is RingCore Contracts, a yield-farming project of Few. Users stake whitelisted tokens on BLAST to the staking contract and receive a fixed APY in terms of token amount.

The auditing process is iterative. Specifically, we will 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. Our audit report is responsible for the only initial version (i.e., `Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project	Version	File MD5
ring-core-contracts.zip	`Version 1`	`8c508a58811b04592d3de2dcb5d9176a`
	`Version 2`	`ec7f74208033f9c7840da7fc8d29bfc1`

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.

1.3 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
* Access control
* 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 [1] and Common Weakness Enumeration [2]. 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.

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

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

Chapter 2 Findings

In total, we find **one** potential issue. Besides, we also have **one** recommendation and **one** note.

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

ID	Severity	Description	Category	Status
1	Low	Potential unfairness in the reward distribution	DeFi Security	Fixed
2	-	Optimize gas usage	Recommendation	Fixed
3	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential unfairness in the reward distribution

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `FixedStakingRewards` contract contains potential unfairness in its reward distribution. Specifically, the `earn` function calculates a user's reward as follows:

```
90    function earned(uint256 index, address account) public override view returns (uint256) {
91        StakingInfo storage info = stakingInfos[index];
92        return
93            info.balances[account].mul(
94                info.rewardPerTokenPerSecond
95            ).mul(
96                lastTimeRewardApplicable(index).sub(info.lastUpdateTime[account])
97            ).div(
98                1e18
99            ).add(
100               info.rewards[account]
101           );
102   }
```

Listing 2.1: FixedStakingRewards.sol

Here, `lastTimeRewardApplicable` is the earlier timestamp of either `block.timestamp` or `periodFinish`. This could result in potential unfairness when `periodFinish` is updated to a future timestamp through the `setPeriodFinish` function. Consequently, for stakers involved in the previous period, rewards will continue to accumulate from the old `periodFinish` to the newly established one. This means that these stakers inadvertently collect additional rewards during this interval.

Furthermore, `rewardPerTokenPerSecond` can be altered using the `setRewardPerTokenPerSecond` function. Should a change in `rewardPerTokenPerSecond` occur, the rewards from the previous period that have not yet been recalculated will be determined using the updated `rewardPerTokenPerSecond`.

```
229    function setRewardPerTokenPerSecond(uint256 index, uint256 _rewardPerTokenPerSecond) external
           override {
230        require(msg.sender == rewardSetter, 'Ring: FORBIDDEN');
231        StakingInfo storage info = stakingInfos[index];
232        require(info.stakingToken != address(0), 'FixedStakingRewards::set: not deployed yet');
233        info.rewardPerTokenPerSecond = _rewardPerTokenPerSecond;
234
235        emit SetRewardPerTokenPerSecond(index, info.stakingToken, msg.sender,
               _rewardPerTokenPerSecond);
236    }
```

Listing 2.2: FixedStakingRewards.sol

Impact Unfair reward distribution may cause unexpected losses.

Suggestion Revise the logic accordingly.

Feedback from the Project We (the project owner) might extend or shorten the staking period, we might increase/decrease the staking reward rate based on the actual price fluctuation. We agree that this might not be fair if we decrease the staking reward rate, thus we add a `updateRewardFor` method that is used for update for stakers before we set a new staking reward rate if we decrease the rate.

2.2 Additional Recommendation

2.2.1 Optimize gas usage

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `FewWrappedToken` constructor stores the original token's name in the `name` storage variable, then loads it, adds a prefix, and stores it to storage again. These actions can be consolidated into a single `SSTORE` operation by directly concatenating the name strings during the initial assignment. A similar issue exists with the `symbol`. Additionally, the `name` variable is not initialized in line 66.

```
58    constructor() public {
59        uint chainId;
60        assembly {
61            chainId := chainid()
62        }
63        DOMAIN_SEPARATOR = keccak256(
64            abi.encode(
65                keccak256('EIP712Domain(string name,string version,uint256 chainId,address
                      verifyingContract)'),
66                keccak256(bytes(name)),
67                keccak256(bytes('1')),
68                chainId,
69                address(this)
70            )
71        );
72
73        factory = msg.sender;
74        token = IFewFactory(msg.sender).parameter();
```

```
75        name = IERC20Detailed(token).name();
76        name = string(abi.encodePacked("Few Wrapped ", name));
77        symbol = IERC20Detailed(token).symbol();
78        symbol = string(abi.encodePacked("fw", symbol));
79
80        BLAST.configureClaimableGas();
81    }
```

Listing 2.3: FewWrappedToken.sol

Impact N/A

Suggestion Revise the redundant operations.

2.3 Notes

2.3.1 Potential centralization risks

Introduced by Version 1

Description The admins for this project have multiple privileged entries to modify critical configurations, which brings centralization risks.

Feedback from the Project All these privileged entries will be controlled by a Timelock and the Timelock will be governed by a DAO Governa contract on mainnet launch. Every configuration will need to go through DAO governance with an approved proposal.