Security Audit Report for Reward Distributor Contract

Date: Dec 10, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.12.13 21:16:01 +08'00'

Contents

Report Manifest

Item	Description
Client	Glori Finance
Target	Reward Distributor Contract

Version History

Version	Date	Description
1.0	Dec 10, 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 focus of this audit is the `contracts/RewardDistributor.sol` file within the Reward Distributor Contract of Glori Finance [1]. Please note that this file is the only one within the scope of our audit. While all other smart contracts in the repository, which is a fork of the reputable Compound V2 Protocol [2], are considered reliable in terms of both functionality and security, these files are not included in the scope of the audit.

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
Reward Distributor Contract	`Version 1`	`edbc4dc12a5aebe6d8504ec67eca0b5bf961e888`

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/GloriFinance/glorifinance

[2] https://github.com/compound-finance/compound-protocol

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 [3] and Common Weakness Enumeration [4]. 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.

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

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

Chapter 2 Findings

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

- Low Risk: 1
- Recommendation: 3
- Note: 1

ID	Severity	Description	Category	Status
1	Low	Potential problems when setting reward token addresses	Software Security	Confirmed
2	-	Remove unused payable modifier	Recommendation	Confirmed
3	-	Remove the unused fallback function	Recommendation	Confirmed
4	-	Use the `SafeERC20` library	Recommendation	Confirmed
5	-	Centralization risk	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential problems when setting reward token addresses

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The `RewardDistributor` contract distribute token rewards to the users according to their usage of the protocol (similar to the Compound Protocol). Instead of distributing rewards of a fixed token (`COMP` in the Compound Protocol), the `RewardDistributor` can distribute different tokens to the users, and the tokens are recorded in the `rewardAddresses` state variable. However, we found several issues in the code logic related to the reward token addresses:

1. The `getRewardAddress` function is redundant, as the `rewardAddresses` state variable is public.
2. The parameter `rewardType` of the `getRewardAddress` function is `uint256`, while the `rewardType` for other functions is `uint8`.
3. The `addRewardAddress` function for adding a reward token does not check if there are more than $256 = 2^8$ reward addresses, because the `rewardType` for most functions is `uint8`.

```
function getRewardAddress(uint256 rewardType) public view returns (address) {
    return rewardAddresses[rewardType];
}
```

Listing 2.1: RewardDistributor.sol

```
function addRewardAddress(address newRewardAddress) public {
    require(msg.sender == admin, "only admin can add new reward address");
    rewardAddresses.push(newRewardAddress);
    uint8 rewardType = uint8(rewardAddresses.length - 1);
    emit RewardAdded(rewardType, newRewardAddress);
}
```

Listing 2.2: RewardDistributor.sol

Impact The logic for reward addresses handling has several problems.

Suggestion Refactor the code to fix the issues pointed above.

2.2 Additional Recommendation

2.2.1 Remove unused payable modifier

Status Confirmed

Introduced by `Version 1`

Description In the `claimReward` functions, there are unused `payable` modifiers which are recommended to be removed for more clean code logic.

```
/**
 * @notice Claim all the COMP/ETH accrued by holder in all markets
 * @param holder The address to claim COMP/ETH for
 */
function claimReward(uint8 rewardType, address payable holder) public {
    return claimReward(rewardType, holder, comptroller.getAllMarkets());
}

/**
 * @notice Claim all the COMP/ETH accrued by holder in the specified markets
 * @param rewardType 0 = COMP, 1 = ETH
 * @param holder The address to claim COMP/ETH for
 * @param cTokens The list of markets to claim COMP/ETH in
 */
function claimReward(uint8 rewardType, address payable holder, CToken[] memory cTokens) public
    {
    address payable[] memory holders = new address payable[](1);
    holders[0] = holder;
    claimReward(rewardType, holders, cTokens, true, true);
}

/**
 * @notice Claim all COMP/ETH accrued by the holders
 * @param rewardType 0 = COMP, 1 = ETH
 * @param holders The addresses to claim COMP/ETH for
 * @param cTokens The list of markets to claim COMP/ETH in
 * @param borrowers Whether or not to claim COMP/ETH earned by borrowing
 * @param suppliers Whether or not to claim COMP/ETH earned by supplying
 */
function claimReward(
    uint8 rewardType,
    address payable[] memory holders,
    CToken[] memory cTokens,
    bool borrowers,
    bool suppliers
```

```
) public payable {
```

Listing 2.3: RewardDistributor.sol

Impact The `payable` modifiers are not used.

Suggestion Remove the unused `payable` modifiers.

2.2.2 Remove the unused fallback function

Status Confirmed

Introduced by `Version 1` The `RewardDistributor` contract implements a fallback function which enables the contract to receive native tokens. However, there is no way to withdraw native tokens from the contract. It is recommended that contracts without logic related to native tokens should not implement the fallback function to prevent users from mistakenly transfer native tokens to the contract.

Description

```
function() external payable {}
```

Listing 2.4: RewardDistributor.sol

Impact The contract can receive native token transfer without any method to withdraw.

Suggestion Remove the unused fallback function.

2.2.3 Use the `SafeERC20` library

Status Confirmed

Introduced by `Version 1`

Description When sending reward tokens to the users, the `RewardDistributor` contract directly invokes plain ERC-20 `transfer` interface. Due to the potential diversity of the tokens set in the `rewardAddresses` state variable, it is recommended to use the `SafeERC20` library from OpenZeppelin to handle the potential corner cases of different implementations of the ERC-20 tokens.

```
function grantRewardInternal(uint8 rewardType, address payable user, uint256 amount) internal
     returns (uint256) {
    address rewardAddress = rewardAddresses[rewardType];
    EIP20Interface reward = EIP20Interface(rewardAddress);
    uint256 rewardRemaining = reward.balanceOf(address(this));
    if (amount > 0 && amount <= rewardRemaining) {
        reward.transfer(user, amount);
        return 0;
    }

    return amount;
}
```

Listing 2.5: RewardDistributor.sol

Impact Reward token transfer can silently fail due to inconsistent ERC-20 token implementations.

Suggestion Replace plain ERC-20 transfer with `SafeERC20` library from OpenZeppelin.

2.3 Note

2.3.1 Centralization risk

Description In the `RewardDistributor` contract, there are potential centralization risks:

- Existing reward token addresses can be modified through `setRewardAddress` privileged function. If a reward token is changed to another token with lower price (or less value), users that has not claimed can suffer from losses.
- The project admin is able to withdraw any token inside the contract through the `_grantReward` privileged function.

```
function setRewardAddress(uint8 rewardType, address newRewardAddress) public {
    require(msg.sender == admin, "only admin can set reward address");
    address oldRewardAddress = rewardAddresses[rewardType];
    rewardAddresses[rewardType] = newRewardAddress;
    emit RewardAddressChanged(rewardType, oldRewardAddress, newRewardAddress);
}
```

Listing 2.6: RewardDistributor.sol

```
function _grantReward(uint8 rewardType, address payable recipient, uint256 amount) public {
    require(adminOrInitializing(), "only admin can grant reward");
    uint256 amountLeft = grantRewardInternal(rewardType, recipient, amount);
    require(amountLeft == 0, "insufficient reward for grant");
    emit RewardGranted(rewardType, recipient, amount);
}
```

Listing 2.7: RewardDistributor.sol