ReWind Security Analysis

by Pessimistic

This report is public

March 24, 2023

Abstract

In this report, we consider the security of smart contracts of ReWind project. Our task is to find and describe security issues in the smart contracts of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. A single audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, a security audit is not investment advice.

Summary

In this report, we considered the security of ReWind smart contracts. We described the audit process in the section below.

The initial audit showed one critical issue: Front-runnable withdrawal. The audit also revealed three issues of medium severity: Front-runnable resource purchase, Better rates via reentrancy, and Overpowered role. Moreover, several low-severity issues were found.

After the initial audit, the developers updated the codebase. In this update, they fixed or addressed all critical and medium-severity issues, and some of the low-severity issues.

General recommendations

We recommend fixing the rest of the issues and limiting powers for the privileged role.

Project overview

Project description

For the audit, we were provided with ReWind project on a private GitHub repository, commit 0fa5353b2e3bc5b3f9078440a415d42a1e034093.

The scope of the audit includes the whole repository.

The documentation for the project includes a document with a brief description of the system and the contracts ABI.

All 51 tests pass successfully. The code coverage is 100%.

The total LOC of audited sources is 404.

Codebase update

After the initial audit, the developers updated the codebase. For the recheck, we were provided with commit bfc1c49f60f6734fac3f66eee10a9913ae8c85c3.

On this commit, all 45 tests pass successfully. The code coverage is 90.22%.

Audit process

We started the audit on February 27, 2023, and finished on March 3, 2023.

We inspected the materials provided for the audit. Then, we contacted the developers for an introduction to the project. After a discussion, we performed preliminary research and specified those parts of the code and logic that require additional attention during an audit:

- The permit functionality is utilized correctly.
- Functions have proper access control.
- USDT is integrated correctly, and its features (i.e., decimals 6) don't break the project logic.
- One cannot manipulate a pool rate to their advantage.
- Upgradability is implemented correctly.

We manually analyzed all the contracts within the scope of the audit and checked their logic. Among other, we verified the following properties of the contracts:

- Gas usage is optimized.
- Arithmetic operations and type casts cannot lead to over/underflow.
- OpenZeppelin dependencies are initialized correctly.

We scanned the project with the static analyzer Slither and our private plugin with an extended set of rules and then manually verified their output.

We ran tests and calculated the code coverage.

We combined in a private report all the verified issues we found during the manual audit or discovered by automated tools.

After the initial audit, we discussed the results with the developers. On March 15, the developers provided us with an updated version of the code. In this update, they fixed some of the issues from our report and provided comments regarding a few other issues.

We reviewed the updated codebase and confirmed the fixes.

Finally, we updated the report.

Manual analysis

The contracts were completely manually analyzed, their logic was checked. Besides, the results of the automated analysis were manually verified. All the confirmed issues are described below.

Critical issues

Critical issues seriously endanger project security. They can lead to loss of funds or other catastrophic consequences. The contracts should not be deployed before these issues are fixed.

C01. Front-runnable withdrawal (fixed)

Using the `withdrawalWithDiscount()` function of the **Operator** contract, a user can convert RWD tokens to a discounted amount of USDT from a specified operator. It relies on a `permit` call, so anyone can front-run a call to `withdrawalWithDiscount()` and re-use the user's permit for other purposes. One can call `withdrawalWithDiscount()` with an altered operator address and claim ReWind tokens with a discount.

Access to this function was restricted. Now, only `SERVICE_ROLE` can call it.

Medium severity issues

Medium severity issues can influence project operation in the current implementation. Bugs, loss of potential income, and other non-critical failures fall into this category, as well as potential problems related to incorrect system management. We highly recommend addressing them.

M01. Front-runnable resource purchase (fixed)

A `spendWithPermit()` function of the **Operator** contract can be used to purchase a specified resource for RWD tokens. However, a call to `spendWithPermit()` can be front-run to force a signer to buy a different resource, e.g., one from another operator.

The same permit can be reused in `withdrawalWithDiscount()`.

Access to this function was restricted. Now, only `SERVICE_ROLE` can call it.

M02. Better rates via reentrancy (fixed)

The specified receivers of an airdrop get a gas refund in the `airdrop()` function of the **Token** contract. If one of the receivers is a smart contract, it can call `withdrawal()` immediately (via re-entrancy).

They will receive more USDT from a better pool ratio, especially if their address is early in the list since the minting of RWD tokens decreases a pool rate. Presumably, the admin has deposited USDT tokens already, but many users have not received their RWD tokens yet.

The issue has been fixed and is not present in the latest version of the code.

M03. Overpowered role (addressed)

In the current implementation, the system depends heavily on the admin role. Thus, some scenarios can lead to undesirable consequences for the project and its users, e.g., if the admin's private keys become compromised. Admin roles have access to the following actions:

- **(fixed)** The `mint` function of the **Token** contract allows minting RWD tokens to any address.
- The `burnFrom` function of the **Token** contract allows burning RWD tokens of any address.
- The `withdrawFunds` function of the **Operator** contract allows withdrawing native currency.

- The `withdrawAnyToken` function of the **Operator** contract allows withdrawing any tokens to an arbitrary address.
- The `withdrawToken` function of the **Token** contract allows withdrawing RWD tokens to an arbitrary address.
- The `withdrawAnyToken` function of the **Token** contract allows withdrawing any tokens (including USDT) to an arbitrary address.

We recommend designing contracts in a trustless manner or implementing proper key management, e.g., setting up a multisig.

> *Comment from the developers: We have taken steps to limit the minter's ability to print tokens above the specified limit. Additionally, we will be using a Safe multisig to cover the remaining roles and further enhance the security of our contract.*

Low severity issues

Low severity issues do not directly affect project operation. However, they might lead to various problems in future versions of the code. We recommend fixing them or explaining why the team has chosen a particular option.

L01. Better rates (commented)

If the Minter calls the `airdrop()` function of the **Token** contract separately from `depositPool()`, users can get a better or worse pool rate than they should, affecting the minted amount of RWD tokens. Consider always executing `airdropWithDepositPool()` instead.

> *<u>Comment from the developers:</u> We have decided to make the airdrop method private in subsequent iterations, as it will likely not be used separately.*

L02. Possible DoS (fixed)

The `airdrop()` function of the **Token** contract iterates through an array of addresses and mints RWD tokens for them. It also refunds native currency for each address, but this address could revert the whole transaction if it is a smart contract, making a current airdrop impossible.

<u>The issue has been fixed and is not present in the latest version of the code.</u>

L03. Extra storage interactions (commented)

The following variables are read from storage inside one function more than once. Consider declaring local variables to optimize gas usage:

- **(fixed)** `refundAmount` in the `_refund()` function of the **Token** contract.
- `token` in the `withdrawalWithDiscount()` function of the **Operator** contract.
- `token` in the `_spend()` function of the **Operator** contract.

> *<u>Comment from the developers:</u> We acknowledge that there is room for optimization in our current implementation. However, given that we are launching on cheap networks, the overhead is currently insignificant for us. Nevertheless, we are committed to optimizing this aspect of the contract in future iterations to improve its efficiency and performance.*

L04. Magic number (fixed)

A `10**30` multiplier is used for precise arithmetic operations in multiple contracts. Consider declaring it as a constant for code clarity in:

- The **Token** contract at lines 140 and 229.
- The **Operator** contract at line 187.

The issues have been fixed and are not present in the latest version of the code.

This analysis was performed by Pessimistic:

Evgeny Marchenko, Senior Security Engineer
Ivan Gladkikh, Security Engineer
Irina Vikhareva, Project Manager
Alexander Seleznev, Founder

March 24, 2023