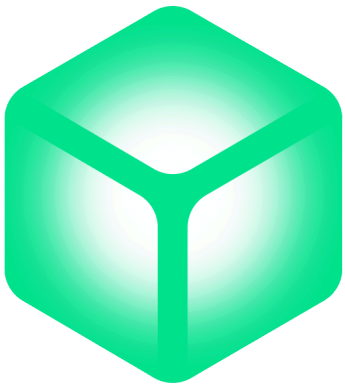

Rubic

Rubic Security Scan Results

by Pessimistic

This is not a security audit

This report is public

May 27, 2024

Abstract

This report considers the security of smart contracts of the Rubic protocol. Our task is to find and describe security issues using the static-analysis tools Slither and Slitherin and help resolve them.

The work is financially covered by the Arbitrum Foundation grant.

Disclaimer

Current work does not give any warranties on the security of the code. It is not an audit or its replacement. Performing this scan, we focused on finding as many crucial issues as possible rather than making sure that the protocol was entirely secure. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

Summary

In this report, we described issues found in smart contracts of the Rubic protocol.

We scanned the codebase and manually rejected or verified all automated findings, revealing five relevant issues.

The developers have commented on three issues.

The entire process is described in the section below.

Scan process

Under the Arbitrum Foundation grant, we researched and developed Arbitrum-specific detectors. They became publicly available with Slitherin v0.6.0 release.

Workflow

This work consisted of five stages:

1. For the scan, we were provided with the Rubic project on the following public GitHub repositories:
    - multi-proxy-rubic, commit a18545d1ae8773a39fc7401e3e3c9fb3cfd67fae;
    - merkle-distributor, commit 6a97c0b3729cbb9287f30aa8db57534db22b9684;
    - staking, commit e8d70aa9249dd51d6d0ecd22e152fa13adc223d6.
2. For the analysis of the protocol, we launched Slither v0.10.1 and Slitherin v0.6.0 on the provided codebase.
3. One auditor manually checked (rejected or accepted) all findings reported by the tools. The second auditor verified this work. We shared all relevant issues with the protocol developers and answered their questions.
4. The developers reviewed the findings and gave comments on issues they do not intend to fix.
5. We prepared this final report summarizing all the issues and comments from the developers.

Issue categories

Within the confines of this work, we were looking for:

- Arbitrum-specific problems;
- Standard vulnerabilities like re-entrancy, overflow, arbitrary calls, etc;
- Non-compliance with popular standards like ERC20 and ERC721;
- Some access control problems;
- Integration issues with some popular DeFi protocols;
- A wide range of code quality and gas efficiency improvement opportunities.

This scan does not guarantee that these issues are not present in the codebase.

Scan results

Issue category	Number of detectors	Status
Compilation	1	1 issue found
Arbitrum Integration	3	Passed
AAVE Integration	1	Passed
Uniswap V2 Integration	7	Passed
OpenZeppelin	2	Passed
ERC-20	7	Passed
ERC-721	2	Passed
Known Bugs	15	Passed
Access Control	3	Passed
Arbitrary Call	5	Passed
Re-entrancy	6	Passed
Weak PRNG	2	Passed
Upgradability	2	Passed
Ether Handling	3	Passed
Low-level Calls	2	1 issue found
Assembly	2	Passed
Inheritance	3	Passed
Arithmetic	2	Passed
Old Solidity Versions Bugs	10	Passed
Code Quality	15	1 issue found
Best Practices	4	Passed
Gas	7	2 issues found

Discovered Issues

Naming

Within the codebase, there are multiple contracts sharing identical names:

- `IFactory` appears in both **RouterHelper.sol** and **IStargateRouter.sol**;
- `IERC20` is present in **RouterHelper.sol** as well as in the @axelar-network and @openzeppelin libraries.

When a codebase includes two contracts with similar names, one of the contracts with the duplicate name may not be included in the compilation artifacts. It is advisable to avoid such occurrences.

> *Comment from the developers: There is no logic in the `IFactory` and `IERC20` interfaces, and if the described issue arises, it will immediately become visible in the tests.*

Unsafe ERC20 interaction

The `sweepTokens` function of the **RubicStaking** and `contsructor` function of the **MerkleDistributorToStaking** contracts ignore the return value from the ERC20 token contract.

According to the ERC20 token standard:

> *Callers MUST handle `false` from returns `(bool success)`. Callers MUST NOT assume that `false` is never returned!*

We recommend using the safe functions from the OpenZeppelin **SafeERC20** library to interact with ERC20 tokens.

Unchecked return values of low level call function

In the **Receiver.sol** contract, the external call's value is not checked within the `pullToken` and `_swapAndCompleteBridgeTokens` functions. Failure to verify the returned value could result in the loss of funds. This is because, in the event of a call revert, the transaction will not revert. To mitigate this risk, it is recommended to thoroughly check returned values.

> *Comment from the developers: The **Receiver.sol** contract is no longer in use and will be removed from the Diamond Proxy.*

Code quality

Within the **Executor.sol** contract, the `diamond` variable could be declared as `immutable`. This action of declaring variables as `immutable` helps decrease gas consumption.

> *Comment from the developers: Agree, it can be declared as `immutable`, but **Executor.sol** contract is also no longer in use and will be removed from the Diamond Proxy.*

External vs public

The `claim` function of the **MerkleDistributorToStaking** contract can be declared as `external`. Consider declaring functions as `external` instead of `public` when possible to improve code readability and optimize gas consumption.

This analysis was performed by Pessimistic:

Egor Dergunov, Junior Security Engineer
Evgeny Bokarev, Junior Security Engineer
Rasul Yunusov, Security Engineer
Pavel Kondratenkov, Senior Security Engineer
Nikita Kirillov, Product Owner
Evgeny Marchenko, CTO
Konstantin Zherebtsov, Business Development Lead

May 27, 2024