Impossible Finance

Impossible Finance Security Analysis

by Pessimistic

This report is public

February 4, 2022

Abstract 2
Disclaimer 2
Summary 2
General recommendations 2
Project overview 3
Project description 3
Procedure 4
Manual analysis 5
Critical issues 5
Medium severity issues 5
Low severity issues 6
Code quality 6
Tests issues 6
Notes 7
Out of the scope 7

Abstract

In this report, we consider the security of smart contracts of Impossible Finance 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. One 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, security audit is not an investment advice.

Summary

In this report, we considered the security of Impossible Finance smart contracts. We performed our audit according to the procedure described below.

The audit only showed a few issues of low severity.

The project has a documentation and tests, all tests pass.

General recommendations

We recommend fixing the mentioned issues and implementing CI to run tests, calculate code coverage, and analyze code with linters and security tools.

Project overview

Project description

For the audit, we were provided with Impossible Finance project on a public GitHub repository, commit d7f80b754a9a4e5647a646de659abe43c0a3a796.

The scope of the audit included only two files:

- **ImpossiblePair.sol**
- **ImpossibleERC20.sol**

The documentation for the project includes:

- README.md file.
- Online documentation.
- Post on Medium.

All tests pass. However, the code coverage calculation fails.

The total LOC of audited sources is 488.

Procedure

In our audit, we consider the following crucial features of the code:

1. Whether the code is secure.
2. Whether the code corresponds to the documentation (including whitepaper).
3. Whether the code meets best practices.

We perform our audit according to the following procedure:

- Automated analysis
  - We scan project’s code base with automated tools: Slither and SmartCheck.
  - We manually verify (reject or confirm) all the issues found by tools.
- Manual audit
  - We manually analyze code base for security vulnerabilities.
  - We assess overall project structure and quality.
- Report
  - We reflect all the gathered information in 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.

The audit showed no critical issues.

Medium severity issues

Medium issues can influence project operation in 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.

The audit showed no issues of medium severity.

Low severity issues

Low severity issues do not directly affect project operation. However, they might lead to various problems in the future versions of the code. We recommend taking them into account.

Code quality

- Consider using OpenZeppelin ECDSA library to more reliably verify signatures in **ImpossibleERC20** contract at line 124.
- Consider inheriting **ImpossibleERC20** contract from OpenZeppelin draft-EIP712 to improve code readability and to avoid re-implementing existing solutions.
- Consider declaring constant instead of raw numeric values in **ImpossiblePair** contract:
  - `1000000` value at lines 321 and 328.
  - `10000` value at lines 556–557, 569, and 576.
- Best practice is to use full-sized types (e.g., `uint256`) for all calculations and interfaces. If you tightly pack shorter types to save gas, consider casting types during storage access, i.e., immediately after reading or before writing a variable. This style improves readability and prevents certain types of overflow issues.

  We recommend changing the following functions in **ImpossiblePair** contract: `getBoost`, `linInterpolate`, `makeXybk`, `updateBoost`, `updateTradeFees`.
- Consider using SafeCast from OpenZeppelin to exclude overflow possibility at lines 394-395 in **ImpossiblePair**.

Tests issues

The project has tests. All tests pass successfully. However, the code coverage calculation fails.

Notes

Out of the scope

The architecture of the project supposes that users do not call methods of **ImpossiblePair** contract directly. One should use **ImpossibleRouter** contract for swap and mint operations since it performs them atomically. Violation of atomicity raises the risk of losing supplied underlying tokens since anyone can withdraw any tokens that exceed the current reserve value.

This behavior is standard for Uniswap-like pools, whose security relies on the router contract. However, **ImpossibleRouter** contract is out of the scope of this audit.

This analysis was performed by Pessimistic:

Evgeny Marchenko, Senior Security Engineer
Daria Korepanova, Security Engineer
Boris Nikashin, Analyst
Irina Vikhareva, Project Manager

February 4, 2022