1inch Audit

LimitOrder Protocol

April 2021

By CoinFabrik

Introduction

CoinFabrik was asked to audit the Limit Order Protocol contracts for 1inch. Limit orders are contracts between a maker, who generates the order, and a taker, who accepts the order. A maker defines an order (offchain) including a pair of tokens, an exchange rate and a total amount which he wants to exchange--among other parameters. A taker who accepts this order can fill the order and have it executed, if certain conditions hold.

We have audited the contracts. a summary of our discoveries, what is their current status, and then we will show the details of our findings.

Summary

We analyzed the contracts received from the private github repository :

[1inch/limit-order-protocol (github.com)]

This audit is based on the following commit:

`fc528b390bad66927b9316470e4c86d06df58563`

Contracts

The audited contracts are:

- contracts/LimitOrderProtocol.sol: main functions
- contracts/helpers/AmountCalculator.sol: helper functions calculating maker and taker amounts
- contracts/helpers/PredicateHelper.sol: helper functions for building predicates
- contracts/helpers/ImmutableOwner.sol: modifier
- contracts/helpers/ERC20Proxy.sol: proxy helper
- contracts/helpers/ERC1155Proxy.sol: proxy helper
- contracts/helpers/ERC721Proxy.sol: proxy helper
- contracts/helpers/NonceManager.sol: nonces

- contracts/libraries/ArgumentsDecoder.sol: help decoding arguments
- contracts/libraries/UncheckedAddress.sol: includes calls to (external) token functions
- contracts/interfaces/IEIP1271.sol: interface
- contracts/interfaces/InteractiveMaker.sol: interface

Analyses

The following analyses were performed:

- Misuse of the different call methods
- Integer overflow errors
- Division by zero errors
- Outdated version of Solidity compiler
- Front running attacks
- Reentrancy attacks
- Misuse of block timestamps
- Softlock denial of service attacks
- Functions with excessive gas cost
- Missing or misused function qualifiers
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling
- Failure to use a withdrawal pattern
- Insufficient validation of the input parameters
- Incorrect handling of cryptographic signatures

Specific Analyses for this Project

We further tested design flaws allowing makers/takers to take advantage of each other. This includes the following analyses:

1. We tested whether *takers* could fill orders with malicious parameters, allowing them to take advantage of *makers*, e.g., with an improved `takingAmount` / `makingAmount` ratio.
2. We tested border conditions by adapting the supplied truffle tests to include security-relevant scenarios.
3. Finally, since the contract makes calls to external contracts from parameters, we looked for vulnerabilities arising from these calls.

Findings and Fixes

ID	Title	Severity	Status
CR-1	Front-running attack in malicious order leads to arbitrary rate set by maker	Critical	Fixed
EN-1	Use of deprecated functions	Enhancement	Not fixed

Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. They must be fixed **immediately**.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of but can be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.
- **Enhancement:** These kinds of findings do not represent a security risk. They are best practices that we suggest to implement.

This classification is summarized in the following table:

SEVERITY	EXPLOITABLE	ROADBLOCK	TO BE FIXED
Critical	Yes	Yes	Immediately
Medium	In the near future	Yes	As soon as possible
Minor	Unlikely	No	Eventually
Enhancement	No	No	Eventually

Issues Found by Severity

Critical Severity

CR-01 Front-running attack in malicious order leads to arbitrary rate setting by maker (fixed)

The LimitOrder contract allows a maker to define orders where a `makingAmount`, a `takingAmount`, `getMakerAmount` and `getTakerAmount` parameters are set (among others). When `fillOrder` is called by a taker, the `makingAmount` and `takingAmount` values of the exchange are computed on the fly, and may use these `getMakerAmount` and `getTakerAmount` functions defined by the maker arbitrarily.

A malicious maker could create an order setting custom `getMakerAmount` and `getTakerAmount` functions on an arbitrary maker-controlled contract together with a different `makingAmount/takingAmount` price. The maker's contract could be implemented so that if the taker evaluates these `getMakerAmount` and `getTakerAmount`, he obtains the original `makingAmount` and `takingAmount` values of the order. However, when the taker calls `fillOrder()` with `makingAmount` set to 0, the maker may execute a front-running attack calling a setter in the maker's malicious contract which changes the value returned by `getMakerAmount`. Hence, when fillOrder is executed, it will use the new value which may be lower than the value expected by the taker (`makingAmount`). (Respectively with `takingAmount=0` and `getTakerAmount`.)

Recommendation

The expected behavior of fillOrder should be documented and in particular, what is the intended use of `thresholdAmount`, `getMakerAmount` and `getTakerAmount`, so that parties are fully aware of the possible order execution outcomes.

Follow up

1inch implemented a solution which eliminates the threat by adding a parameter, `thresholdAmount,` to fillOrder with which the taker stops any attack.

Before placing the transfers, the code in fillOrder requires that

```
takingAmount <= thresholdAmount
```

when takingAmount is determined by `getMakerAmount`, and it requires that

```
makingAmount >= thresholdAmount
```

when makingAmount is determined by `getMakerAmount`.

The solution thus eliminates the threat in the case where the taker enters thresholdAmount according to his needs: for example, a taker which sets a nonzero `makingAmount` value in `fillOrder` may specify the `thresholdAmount` to be equal to the `takingAmount` he expects to receive (so that the maker cannot make the order's `takingAmount` any bigger); respectively, the taker entering a `takingAmount` value in the `fillOrder` may specify the `thresholdAmount` to be equal to the expected `makingAmount` (so that the maker cannot make the order's `makingAmount` any smaller). When the `thresholdAmount` values differ from what the taker expects to receive, this may prevent the transaction from happening or may allow the maker to take an advantage.

An alternative solution, proposed and discarded by 1inch, includes a different additional parameter, `minPrice`, to be added to `fillOrder` and the contract requires that

```
makingAmount * 1e18 / takingAmount >= minPrice.
```

This solution is analogous to the `thresholdAmount` solution. Generally speaking, a taker may perceive problematic paying a price below the one entered when calling `fillOrder()`; he may even dismiss `thresholdAmount` (or `minPrice`) as he expects the order to be executed according to what he inputs without going into the additional steps.

Medium Severity

No issues found.

Minor Severity

No issues found.

Enhancements

EN-01 Use of deprecated functions

The functions `remaining()` and `fillOrder()` make use of `value1.sub(value2, error_message)` which is deprecated (see link).

Recommendation

Use `trySub()` instead.

Conclusion

We found the contracts to be concise with scarce documentation. The code reduces to a single contract. Tests and testing scripts included in the repository were helpful to understand the limit order execution flow, and also in evaluating certain scenarios. We included in our analysis checks to design flaws specific to this model.

A critical issue was discovered by 1inch that allows a malicious maker to deceive takers in receiving a lower-than-expected exchange rate. The issue was fixed by providing the taker with a security parameter which ensures that attacks are not possible when the parameter is chosen respecting his expectations.

Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the 1Inch project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.