AstraSec

1inch Fusion Settlement

Security Audit Report

April 13, 2024

Contents

1 Introduction

1.1 1inch Fusion Settlement

1inch Limit Order protocol allows users to create limit orders off-chain that can be filled on-chain. The new changes in this audit enable the protocol to collect fee from taking amount and cut the logic in Limit Order Settlement into separate files to help modular extension building.

1.2 Source Code

The following source code were reviewed during the audit:

- https://github.com/1inch/limit-order-protocol.git (562fa69)
  - contracts/FeeTaker.sol
- https://github.com/1inch/limit-order-settlement.git (d6f86ed)
  - contracts/Settlement.sol
  - contracts/SimpleSettlement.sol
  - contracts/extensions/BaseExtension.sol
  - contracts/extensions/ExtensionLib.sol
  - contracts/extensions/IntegratorFeeExtension.sol
  - contracts/extensions/ResolverFeeExtension.sol
  - contracts/extensions/WhitelistExtension.sol

And these are the final versions representing all fixes implemented for the issues identified in the audit:

- https://github.com/1inch/limit-order-protocol.git (e0cc4cb)
- https://github.com/1inch/limit-order-settlement.git (c5de861)

2 Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the `1inch Fusion Settlement` project Throughout this audit, we identified a total of 3 issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	-	-	-	-
High	-	-	-	-
Medium	1	-	-	1
Low	2	1	-	1
Informational	-	-	-	-
Undetermined	-	-	-	-

3 Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

~~M-1~~ Lack of Support for Fee in ETH

~~L-1~~ Enhanced Access Control for FeeTaker::postInteraction()

L-2 Revisited _ORDER_FEE_BASE_POINTS in ResolverFeeExtension

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

Severity	Description
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and remediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

3.3 Vulnerability Details

[M-1] Lack of Support for Fee in ETH

Target	Category	IMPACT	LIKELIHOOD	STATUS
IntegratorFeeExtension.sol FeeTaker.sol	Business Logic	Medium	Medium	Addressed

In the 1inch Limit Order protocol, the FeeTaker contract includes a `postInteraction()` routine which is used to collect fees in taking tokens and then transfers the remaining amount to the maker. Since the taking tokens can be either `ETH` or ERC20 tokens, the fee should also accommodate both types.

However, while reviewing the fee collection logic within the `postInteraction()` routine, it appears that it only supports fees in ERC20 tokens (line 45). As a result, when the taking token is `ETH`, it tries to transfer the `ETH` fee by calling `IERC20(order.takerAsset.get()).safeTransfer()` routine, which will result in reverted transaction.

Based on this, it's recommended to introduce support for fees in `ETH`.

FeeTaker::postInteraction()

```
26  function postInteraction(
27      IOrderMixin.Order calldata order,
28      bytes calldata /* extension */,
29      bytes32 /* orderHash */,
30      address /* taker */,
31      uint256 /* makingAmount */,
32      uint256 takingAmount,
33      uint256 /* remainingMakingAmount */,
34      bytes calldata extraData
35  ) external {
36      uint256 fee = takingAmount * uint256(uint24(bytes3(extraData))) / _FEE_BASE;
37      address feeRecipient = address(bytes20(extraData[3:23]));

39      address receiver = order.maker.get();
40      if (extraData.length > 23) {
41          receiver = address(bytes20(extraData[23:43]));
42      }

44      if (fee > 0) {
45          IERC20(order.takerAsset.get()).safeTransfer(feeRecipient, fee);
46      }

48      unchecked {
49          IERC20(order.takerAsset.get()).safeTransfer(receiver, takingAmount - fee
                );
```

```
    }
}
```

Note the same issue exists in the IntegratorFeeExtension contract of Limit Order Settlement protocol.

IntegratorFeeExtension::_postInteraction()

```
function _postInteraction(
    IOrderMixin.Order calldata order,
    bytes calldata extension,
    bytes32 orderHash,
    address taker,
    uint256 makingAmount,
    uint256 takingAmount,
    uint256 remainingMakingAmount,
    bytes calldata extraData
) internal virtual override {
    if (extraData.integratorFeeEnabled()) {
        address integrator = address(bytes20(extraData[:20]));
        uint256 fee = takingAmount * uint256(uint32(bytes4(extraData[20:24]))) /
             _TAKING_FEE_BASE;
        if (fee > 0) {
            IERC20(order.takerAsset.get()).safeTransferFrom(taker, integrator,
                fee);
        }
        extraData = extraData[24:];
    }
    super._postInteraction(order, extension, orderHash, taker, makingAmount,
        takingAmount, remainingMakingAmount, extraData);
}
```

Remediation Revisit the implementation of `FeeTaker::postInteraction()/IntegratorFeeExtension::_postInteraction()` to support the fees in `ETH`.

[L-1] Enhanced Access Control for FeeTaker::postInteraction()

Target	Category	IMPACT	LIKELIHOOD	STATUS
FeeTaker.sol	Access Control	Low	Low	Addressed

The `FeeTaker::postInteraction()` routine, as described in the comment, is a "Callback method that gets called after all fund transfers". It is designed to be invoked exclusively by the LimitOrderProtocol after an order has been filled. However, there is currently no access control implemented for this routine, which poses a potential security risk. As a result, anyone could call this routine with crafted calldata, attempting to drain assets from FeeTaker (line 49). Although the FeeTaker contract is not meant to hold assets, it's advisable to restrict the access to the Limit Order Protocol contract only (add `onlyLimitOrderProtocol` modifier).

FeeTaker::postInteraction()

```
26 function postInteraction(
27     IOrderMixin.Order calldata order,
28     bytes calldata /* extension */,
29     bytes32 /* orderHash */,
30     address /* taker */,
31     uint256 /* makingAmount */,
32     uint256 takingAmount,
33     uint256 /* remainingMakingAmount */,
34     bytes calldata extraData
35 ) external {
36     uint256 fee = takingAmount * uint256(uint24(bytes3(extraData))) / _FEE_BASE;
37     address feeRecipient = address(bytes20(extraData[3:23]));

39     address receiver = order.maker.get();
40     if (extraData.length > 23) {
41         receiver = address(bytes20(extraData[23:43]));
42     }

44     if (fee > 0) {
45         IERC20(order.takerAsset.get()).safeTransfer(feeRecipient, fee);
46     }

48     unchecked {
49         IERC20(order.takerAsset.get()).safeTransfer(receiver, takingAmount - fee
               );
50     }
51 }
```

Remediation Improve the `FeeTaker::postInteraction()` routine and allow only the LimitOrderProtocol contract to access it.

[L-2] Revisited _ORDER_FEE_BASE_POINTS in ResolverFeeExtension

Target	Category	IMPACT	LIKELIHOOD	STATUS
ResolverFeeExtension.sol	Business Logic	Medium	Low	Mitigated

The ResolverFeeExtension contract works as an extension in the settlement contract to impose a resolver fee through the `postInteraction()` routine. The resolver fee is calculated via the `_getResolverFee()` routine, which utilizes a fixed fee base point `_ORDER_FEE_BASE_POINTS = 1e15`, with which the calculated resolver fee will not fall below $1e15$.

What's more, the resolver fee is implemented as a credit allowance, which is secured by depositing an equivalent amount of `_TOKEN` via the `FeeBank::_depositFor()` routine (line 108). As a result, if the `_TOKEN` has fewer decimals than 15, the fee imposed on the order taker becomes prohibitively expensive. For instance, if `_TOKEN` is `USDC`, which has 6 decimals, the order taker has to deposit a minimum of $1e9$ `USDC` to cover the resolver fee.

Given this, it's recommended to select `_TOKEN` judiciously to maintain the fee amount at a reasonable level.

EscrowFactory::createDstEscrow()

```
18  uint256 private constant _ORDER_FEE_BASE_POINTS = 1e15;

20  constructor(IERC20 token) FeeBankCharger(token) {}

22  function _getResolverFee(
23      uint256 fee,
24      uint256 orderMakingAmount,
25      uint256 actualMakingAmount
26  ) internal pure virtual returns(uint256) {
27      return fee * _ORDER_FEE_BASE_POINTS * actualMakingAmount / orderMakingAmount;
28  }
```

FeeBank::_depositFor()

```
106 function _depositFor(address account, uint256 amount) internal returns (uint256
        totalAvailableCredit) {
107   if (account == address(0)) revert ZeroAddress();
108   _TOKEN.safeTransferFrom(msg.sender, address(this), amount);
109   unchecked {
110       _accountDeposits[account] += amount;  // overflow is impossible due to
              limited _TOKEN supply
111   }
112   totalAvailableCredit = _CHARGER.increaseAvailableCredit(account, amount);
113 }
```

Remediation Select the `_TOKEN` judiciously and adjust the `_ORDER_FEE_BASE_POINTS` to a suitable value based on the chosen `_TOKEN`.

Status This issue has been mitigated. The team has confirmed that they will use `1INCH` on the networks where it is available, other tokens with 18 decimals will be used on other networks.

4 Appendix

4.1 About AstraSec

`AstraSec` is a blockchain security company that serves to provide high-quality auditing services for blockchain-based protocols. With a team of blockchain specialists, `AstraSec` maintains a strong commitment to excellence and client satisfaction. The audit team members have extensive audit experience for various famous DeFi projects. `AstraSec`'s comprehensive approach and deep blockchain understanding make it a trusted partner for the clients.

4.2 Disclaimer

The information provided in this audit report is for reference only and does not constitute any legal, financial, or investment advice. Any views, suggestions, or conclusions in the audit report are based on the limited information and conditions obtained during the audit process and may be subject to unknown risks and uncertainties. While we make every effort to ensure the accuracy and completeness of the audit report, we are not responsible for any errors or omissions in the report.

We recommend users to carefully consider the information in the audit report based on their own independent judgment and professional advice before making any decisions. We are not responsible for the consequences of the use of the audit report, including but not limited to any losses or damages resulting from reliance on the audit report.

This audit report is for reference only and should not be considered a substitute for legal documents or contracts.

4.3 Contact

Name	AstraSec Team
Phone	+86 176 2267 4194
Email	contact@astrasec.ai
Twitter	https://twitter.com/AstraSecAI