# 1inch Security Review

## Pashov Audit Group

Conducted by: Alex Murphy, HickupHH3, SpicyMeatball, ubermensch

March 25th 2024 - March 29th 2024

# Contents

# 1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

# 2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

# 3. Introduction

A time-boxed security review of the **limit-order-protocol** and **limit-order-settlement** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

# 4. About 1inch

1inch limit orders protocol allows traders to create and execute classic limit orders, as well as a wide variety of custom orders, including the exchange of non-ERC20 tokens, dynamic exchange rates, verification of conditions for order filling, execution of arbitrary code.

# 5. Risk Classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| Likelihood: High | Critical | High | Medium |
| Likelihood: Medium | High | Medium | Low |
| Likelihood: Low | Medium | Low | Low |

## 5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

## 5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

## 5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

# 6. Security Assessment Summary

*review commit hashes*:

- 198a6d719aef8b6cd0b29e18a128f4abcf978046
- d6f86ed124b38e2af114cd821a6bf3beed94193b

*fixes review commit hashes*:

- 8c19fb95792c8fec76ed8f11554300dc0056416d
- 9cd284fb6d9856cf297c5662805efbd85227a6a8

## Scope

The following smart contracts were in scope of the audit:

For `limit-order-protocol`:

- `FeeTaker`

For `limit-order-settlement`:

- `Settlemen`
- `SimpleSettlement`
- `extensions/BaseExtension`
- `extensions/ExtensionLib`
- `extensions/ResolverFeeExtension`
- `extensions/WhitelistExtension`
- `extensions/IntegratorFeeExtension`

# 7. Executive Summary

Over the course of the security review, Alex Murphy, HickupHH3, SpicyMeatball, ubermensch engaged with 1inch to review 1inch. In this period of time a total of **6** issues were uncovered.

## Protocol Summary

| | |
|---|---|
| **Protocol Name** | 1inch |
| **Date** | March 25th 2024 - March 29th 2024 |
| **Protocol Type** | Limit order protocol |

## Findings Count

| **Severity** | **Amount** |
|---|---|
| Medium | 2 |
| Low | 4 |
| **Total Findings** | **6** |

# Summary of Findings

| ID | Title | Severity | Status |
|---|---|---|---|
| [M-01] | FeeTaker is not accounting for unwrapWeth Feature | Medium | Resolved |
| [M-02] | FeeTaker is incompatible with fee-on-transfer tokens | Medium | Acknowledged |
| [L-01] | Unverified integratorFee Within Reasonable Range | Low | Resolved |
| [L-02] | Limitation of timeDelta to Around 18 Hours Due to 16-bit Encoding | Low | Acknowledged |
| [L-03] | Unverified timeDelta Leading to Transaction Panic | Low | Acknowledged |
| [L-04] | Whitelist Extension cannot handle the case when there is no whitelist | Low | Acknowledged |

# 8. Findings

## 8.1. Medium Findings

### [M-01] FeeTaker is not accounting for `unwrapWeth` Feature

#### Severity

**Impact:** Medium

**Likelihood:** Medium

#### Description

The `FeeTaker` currently does not account for scenarios where the `unwrapWeth` feature is enabled. This oversight becomes apparent when `takerAsset` is `WETH`, and `unwrapWeth` is activated, causing the limit order protocol to unwrap the `WETH` and send the taking amount as `ETH` directly to the maker listener (`FeeTaker`). The `postInteraction` function, however, is designed to only handle ERC20 tokens and not ETH. Consequently, this discrepancy leads to transaction failures as the contract attempts to transfer `WETH` to the `receiver` and the `feeRecipient` without having adequate `WETH` to fulfill these transfers. In addition to that `FeeTaker` does not have a receive method to be able to accept `ETH`. The same issue applies to the `IntegratorFeeExtension` contract.

```
// Taker => Maker
if (order.takerAsset.get() == address(_WETH) && msg.value > 0) {
    if (msg.value < takingAmount) revert Errors.InvalidMsgValue();
    if (msg.value > takingAmount) {
        unchecked {
            // solhint-disable-next-line avoid-low-level-calls
            (bool success, ) = msg.sender.call{value: msg.value - takingAmount}
              ("");
            if (!success) revert Errors.ETHTransferFailed();
        }
    }

    if (order.makerTraits.unwrapWeth()) {
        // solhint-disable-next-line avoid-low-level-calls
        (bool success, ) = order.getReceiver().call{value: takingAmount}("");
        if (!success) revert Errors.ETHTransferFailed();
    } else {
        _WETH.safeDeposit(takingAmount);
        _WETH.safeTransfer(order.getReceiver(), takingAmount);
    }
```

```
function postInteraction(
    IOrderMixin.Order calldata order,
    bytes calldata /* extension */,
    bytes32 /* orderHash */,
    address /* taker */,
    uint256 /* makingAmount */,
    uint256 takingAmount,
    uint256 /* remainingMakingAmount */,
    bytes calldata extraData
) external {
    uint256 fee = takingAmount * uint256(uint24(bytes3(extraData))) / _FEE_BASE;
    address feeRecipient = address(bytes20(extraData[3:23]));

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

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

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

```
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
    );
}
```

## Recommendations

To resolve this issue, it's recommended to implement an additional check within the `postInteraction` function to ascertain whether the transaction involves `WETH` that has been unwrapped to `ETH`. If the transaction involves unwrapping `WETH` to `ETH`, the contract should be equipped to handle the direct transfer of `ETH` to the `receiver` and `feeRecipient`, rather than attempting to transfer `WETH` as an ERC20 token. Also, the contract should have a receive method to accept `ETH`.

# [M-02] FeeTaker is incompatible with fee-on-transfer tokens

## Severity

**Impact:** Medium

**Likelihood:** Medium

# Description

When a fee-on-transfer token is used as a taker asset, the `FeeTaker.sol` contract receives `takingAmount - tokenFee`. This can cause the contract to revert because it assumes that the entire `takingAmount` is available on its balance.

```
function postInteraction(
        IOrderMixin.Order calldata order,
        bytes calldata /* extension */,
        bytes32 /* orderHash */,
        address /* taker */,
        uint256 /* makingAmount */,
        uint256 takingAmount,
        uint256 /* remainingMakingAmount */,
        bytes calldata extraData
    ) external {
        ---SNIP---
        unchecked {
>>          IERC20(order.takerAsset.get()).safeTransfer
  (receiver, takingAmount - fee);
        }
    }
```

# Recommendations

Validate the balance before transferring to the recipient

```
function postInteraction(
        IOrderMixin.Order calldata order,
        bytes calldata /* extension */,
        bytes32 /* orderHash */,
        address /* taker */,
        uint256 /* makingAmount */,
        uint256 takingAmount,
        uint256 /* remainingMakingAmount */,
        bytes calldata extraData
    ) external {
+       uint256 balance = IERC20(order.takerAsset.get()).balanceOf(address
+ (this));
+       if (balance < takingAmount) takingAmount = balance;

          ---SNIP---
    }
```

# 8.2. Low Findings

## [L-01] Unverified integratorFee Within Reasonable Range

The `IntegratorFeeExtension` does not enforce a verification process to ensure that the `integratorFee` is below a predefined threshold (`_TAKING_FEE_BASE`) or within a generally acceptable range. This lack of validation could potentially lead to scenarios where the `integratorFee` exceeds expected values. Ensuring that fees are kept within reasonable limits is crucial for maintaining user trust and contract integrity.

## [L-02] Limitation of `timeDelta` to Around 18 Hours Due to 16-bit Encoding

The `timeDelta` parameter, currently limited by a 2-byte (16-bit) encoding, restricts the maximum duration to 18 hours, 12 minutes, and 15 seconds. This limitation, while typically adequate for the swift nature of swap operations and auction adjustments, introduces a constraint on user flexibility. It's observed that the duration, chosen for its alignment with frequent, short-term swaps, could benefit from an extended range. By increasing the encoding to 3 bytes, the contract could offer a greater variety of timing options to users.

```
function _getAuctionBump(
  uint256auctionStartTime,
  uint256auctionFinishTime,
  uint256initialRateBump,
  bytescalldatapointsAndTimeDeltas
) private view returns (uint256
      ...
      while (pointsAndTimeDeltas.length > 0) {
          uint256 nextRateBump = uint24(bytes3(pointsAndTimeDeltas[:3]));
          uint256 nextPointTime = currentPointTime + uint16(bytes2
            (pointsAndTimeDeltas[3:5]));
      ...
}
```

# [L-03] Unverified timeDelta Leading to Transaction Panic

The `timeDelta` parameter is not properly verified to ensure it is different from zero before proceeding with the division. This oversight can lead to scenarios where the transaction unexpectedly panics. It is recommended that a validation check is implemented to verify `timeDelta` to be different from zero.

```
function _getAuctionBump(
  uint256auctionStartTime,
  uint256auctionFinishTime,
  uint256initialRateBump,
  bytescalldatapointsAndTimeDeltas
) private view returns (uint256
      ...
      while (pointsAndTimeDeltas.length > 0) {
          uint256 nextRateBump = uint24(bytes3(pointsAndTimeDeltas[:3]));
          uint256 nextPointTime = currentPointTime + uint16(bytes2
            (pointsAndTimeDeltas[3:5]));
          if (block.timestamp <= nextPointTime) {
              return ((block.timestamp - currentPointTime) * nextRateBump +
              //(nextPointTime - block.timestamp) * currentRateBump) / (nextPointTime
          }
      ...
}
```

# [L-04] Whitelist Extension cannot handle the case when there is no whitelist

The `WhitelistExtension` contract is designed to allow only certain resolvers to fill a maker order. However, it is possible to have no whitelist, meaning that any resolver can fill the maker order. If the resolver count is set to zero, the `_postInteraction` function in the `WhitelistExtension` contract will always revert.

Recommendation:

- Modify the `_postInteraction` function in the `WhitelistExtension` contract to skip the `_isWhitelisted` check if the resolver count is zero.
- Decouple the check for whether the auction has started from the whitelist check logic.