1Inch
Fee Protocol

FINAL REPORT

April ‘2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	1inch – Fee Protocol
Website	1inch.io
Language	Solidity
Methods	Manual Analysis
Github repository	Fusion Protocol - https://github.com/1inch/fusion-protocol/tree/c0197a5b4110e89a7a832fff8e95927e41564e17 Limit Order Protocol - https://github.com/1inch/limit-order-protocol/tree/cf8e50eb24c01c7b276a30a5877840df35e66e67
Resolution 1	Fusion Protocol - https://github.com/1inch/fusion-protocol/blob/c45a48191e73b5876bd8b5889d364bfb9397d5e1 Limit Order Protocol - https://github.com/1inch/limit-order-protocol/tree/67c56aee3b6a9f4982bf487084bd8da1f6638da0

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution
High					
Medium	3			3	
Low	2			2	
Informational	3	2		1	
Governance					
Total	8	2		6	

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

AmountGetterbase

The AmountGetterBase contract is a utility contract used within the 1inch Limit Order Protocol to dynamically calculate either the makingAmount or takingAmount of a trade. It supports external logic via a callback, or defaults to a simple linear price formula if no custom logic is provided.

Appendix: Linear Formula

- When no extraData is provided, the getMakingAmount function falls back to a simple linear pricing formula:

$$makingAmount = \frac{order.makingAmount \ * \ takingAmount}{order.takingAmount}$$

  - ➔ order.makingAmount - The amount of the maker token that the user (maker) is offering to swap.
  - ➔ takingAmount - The actual amount of the taker token the resolver (or taker) is supplying during the fill.
  - ➔ order.takingAmount - The amount of the taker token the user wants to receive in exchange for the order.makingAmount.

- The getTakingAmount function operates similarly to getMakingAmount, but with a slight adjustment: the result is rounded up to ensure the maker receives at least the expected value:

$$takingAmount = \frac{order.takingAmount \ * \ makingAmount}{order.makingAmount}$$

  - ➔ makingAmount - The amount of the maker token the resolver wants to receive in return for supplying.

Privileged Functions

- None

No issues found.

AmountGetterWithFee

The AmountGetterWithFee contract extends the functionality of AmountGetterBase by integrating dynamic fee logic directly into the getMakingAmount and getTakingAmount functions, enabling precise trade amount calculations that account for both integrator and resolver fees.

Appendix: Fees

In the getTakingAmount function, fees are applied to the initially calculated linear taking amount. The process starts with a call to the internal _parseFeeData function, which extracts the integratorFee, integratorShare, resolverFee, whitelistDiscountNumerator, and the whitelist information from the extraData. If the taker is identified as whitelisted, a discount is subsequently applied to the resolver fee.

- In _getTakingAmount, the fees are applied using the following formula(rounding up):

$$totalTakingAmount = \frac{takingAmountBeforeFees * (1e5 + integratorFee + resolverFee)}{1e5}$$

- In _getMakingAmount, the initial making amount is calculated (rounding down):

$$makingAmount = \frac{totalTakingAmountWithFees * 1e5}{(1e5 + integratorFee + resolverFee)}$$

Privileged Functions

- None

Issue_01	Fee rounding may benefit the maker instead of the protocol
Severity	Informational

Description	In _getTakingAmount(), the final taking amount is calculated rounding up the fees:
	
	`return Math.mulDiv(`
	`    super._getTakingAmount(order, extension, orderHash, taker, makingAmount, remainingMakingAmount, tail),`
	`    _BASE_1E5 + integratorFee + resolverFee,`
	`    _BASE_1E5,`
	`    Math.Rounding.Ceil`
	`);`
	
	However, during the fee calculations in _getFeeAmounts(), fees are actually rounded down:
	
	`uint256 integratorFeeTotal = takingAmount.mulDiv(integratorFee, denominator);`
	`integratorFeeAmount = integratorFeeTotal.mulDiv(integratorShare, _BASE_1E2);`
	`protocolFeeAmount = takingAmount.mulDiv(resolverFee, denominator) + integratorFeeTotal - integratorFeeAmount;`
	
	The maker will receive the following amount of tokens in the FeeTaker:
	
	`IERC20(order.takerAsset.get()).safeTransfer(receiver, takingAmount - integratorFeeAmount - protocolFeeAmount);`
	
	However, due to rounding up the fees in _getTakingAmount() and then rounding them down in _getFeeAmounts(), fees that were a product of the positive rounding will actually go to the maker, instead of the fee recipients.

Recommendations	Consider changing the rounding direction in _getFeeAmounts() if this behavior is not expected.

Comments / Resolution	Acknowledged. Reply from 1inch: The rounding behavior is intentional to ensure the maker receives at least the expected amount. The difference is negligible and does not materially impact protocol or integrator revenue.

FeeTaker

The FeeTaker contract is an extension of AmountGetterWithFee that integrates seamlessly with the limit order protocol to collect and distribute fees directly in the taker asset during trade execution. Its primary addition is the implementation of the postInteraction mechanism, allowing the protocol to apply complex fee logic and custom behaviors after a trade has been matched.

This contract enhances the base amount calculation logic from AmountGetterWithFee by enabling fee collection and distribution in ETH or ERC-20 tokens, depending on the trade context. It supports a **whitelisting mechanism** that provides reduced fees for approved users. In cases where a taker is not whitelisted, the contract requires ownership of a designated access token to proceed.

Appendix: Limit Order Protocol

The Limit Order Protocol enables users, referred to as makers, to create signed orders off-chain that define the terms of a token swap. These orders include parameters such as the asset being offered (makingAmount), the asset expected in return (takingAmount), and optional conditions like expiration time or access control. Once signed, these orders are distributed off-chain and can be filled on-chain by other parties—known as resolvers or takers—who agree to the specified terms.

When an order is filled, the following steps are executed:

1) **Order Validation**
   The order is validated to ensure the extension parameters passed by the resolver match the salt provided by the maker.
2) **Amount Computation**
   Either the makingAmount or takingAmount is computed. This step is critical for this audit, as it directly relies on the getter functions of the FeeTaker and SimpleSettlement contracts.
3) **Order Invalidation**
   The order is invalidated based on the maker's specified traits to prevent replay attacks or double execution.
4) **Pre-Interaction Trigger**
   A pre-interaction hook is executed, allowing the maker to prepare or manage funds non-interactively if needed.
5) **Transfer of Making Amount**
   The makingAmount is transferred from the maker to the taker.
6) **Return of Taking Amount**

The taker transfers the takingAmount to the designated recipient.
In the context of FeeTaker, the recipient is the FeeTaker contract itself, which will then manage the distribution of the received funds internally.

7) **Post-Interaction Trigger**
A post-interaction hook is triggered, during which the FeeTaker distributes the collected assets to the appropriate recipients.

Appendix: Post Interaction

The postInteraction() function is invoked by the limit order protocol after an order has been filled. It is responsible for distributing the taker's payment by sending the appropriate fee amounts to the protocolFeeRecipient and integratorFeeRecipient, while transferring the remaining portion— the takingAmount minus all applicable fees—to the final recipient, typically the order maker or the specified recipient.

Core Invariants:

INV 1: A resolver must be whitelisted or have _ACCESS_TOKEN to fill an order

Privileged Functions

- rescueFunds

Issue_02	FeeTaker is not compatible with fee-on-transfer tokens
Severity	**Medium**
Description	When a resolver is filling an order, they are first transferring the takingAmount to the FeeTaker contract(in OrderMixin.sol): *IERC20(order.takerAsset.get()).safeTransferFromPermit2(msg.sender , receiver, takingAmount);* In the FeeTaker, fees are sent to the corresponding fee recipients and the following transfer to the maker is made: *IERC20(order.takerAsset.get()).safeTransfer(receiver, takingAmount - integratorFeeAmount - protocolFeeAmount);* Trying to transfer *takingAmount - integratorFeeAmount - protocolFeeAmount* will revert due to insufficient balance, when fee-on-transfer tokens are used.
Recommendations	Consider adding a balance check, prior to executing the transfer.
Comments / Resolution	Acknowledged. Reply from 1inch: Fee-on-transfer tokens are filtered out during backend validation before it is accepted and made available for submission.

Issue_03	Order maker can perform a gas-griefing attack on resolver
Severity	**Low**

Description	In _postInteraction(), the FeeTaker contract will try to transfer native ETH to the order's receiver, which is specified by the maker:
	
	`if (order.takerAsset.get() == address(_WETH) && order.makerTraits.unwrapWeth()) {`
	`        ...`
	`        _sendEth(receiver, takingAmount - integratorFeeAmount - protocolFeeAmount);`
	
	_sendEth() will perform the following external call:
	
	`  function _sendEth(address target, uint256 amount) private {`
	`      (bool success, ) = target.call{value: amount}("");`
	
	As the user who invokes the _postInteraction() function is actually the resolver, they will be paying for all the gas fees of the call.
	The maker of the order can use the external call to trick the resolver into paying higher gas fees by adding gas-expensive logic in their receive() function.
	The maker can also perform the same attack in the pre-interaction and post-interaction calls, which are specified in the order creation.

Recommendations	Consider using a low level call with a gas limit.
Comments / Resolution	Acknowledged. Reply from 1inch: We acknowledge the potential griefing vector. In practice, resolvers simulate transactions and won't execute orders that are unprofitable, regardless of underlying logic.

Issue_04	Same check differs across the codebase
Severity	**Informational**

Description	In FeeTaker.sol, the length check is as follows
	
	`if (tail.length >= 20) {`
	`IPostInteraction(address(bytes20(tail))).postInteraction(order, extension, orderHash, taker, makingAmount, takingAmount, remainingMakingAmount, tail[20:]);`
	`}`
	
	However, the same check in the OrderMixin.sol contract is the following.
	
	`if (data.length > 19) {`
	`listener = address(bytes20(data));`
	`data = data[20:];`
	`}`
	`IPostInteraction(listener).postInteraction(`
	`order, extension, orderHash, msg.sender, makingAmount, takingAmount, remainingMakingAmount, data`
	`);`
	
	The length checks differ, even though they function the exact same way.

Recommendations	Consider changing one of the checks to match the other.
Comments / Resolution	Resolved by following recommended mitigation.

SimpleSettlement

The SimpleSettlement contract extends FeeTaker to implement advanced Dutch auction mechanics, surplus fee extraction, and dynamic gas cost compensation, making it the core settlement layer for Fusion-mode limit orders.

It enhances the base fee-taking and amount-calculation logic by introducing time-dependent pricing via rate bumps that gradually decrease over the duration of an auction. These adjustments are computed using a linear model, allowing for flexible and gas-efficient price discovery.

Appendix: Dutch Auction

A Dutch auction mechanism is employed to dynamically determine the fill price of a Fusion order, with pricing change bounded between auctionStartTime and auctionFinishTime. Unlike traditional Dutch auctions that decrease price linearly over time, the 1inch Fusion implementation divides the auction duration into multiple segments(rateBumps)—each with its own custom rate of price reduction. This segmented approach enables more precise control over auction dynamics and can optimize outcomes based on evolving market conditions.

The rate bump between two consecutive rateBump points is calculated using linear interpolation as follows:

$$auctionBump = \frac{(block.timestamp - currentPointTime) * nextRateBump + (nextPointTime - block.timestamp) * currentRateBump}{(nextPointTime - currentPointTime)}$$

- ➔ currentRateBump - the starting rate bump for the segment
- ➔ currentPointTime - the starting timestamp of the currentRateBump
- ➔ nextRateBump - the ending rate bump of the segment
- ➔ nextPointTime - the ending timestamp for the nextRateBump

Appendix: Gas Compensation

To incentivize early order fulfillment, especially during periods of elevated network congestion, the protocol introduces a gas compensation mechanism. This feature ensures that resolvers are not discouraged from interacting with orders when gas prices are high, helping maintain consistent liquidity and order execution.

The compensation is implemented through a dynamic gasBump, which estimates the gas cost and adjusts the effective price accordingly. The calculation factors in the current block.basefee, and the result is subtracted from the auction's rateBump to keep the net incentive aligned with prevailing transaction fees:

$$gasBump = \frac{gasBumpEstimate \, * \, block.basefee}{gasPriceEstimate \, * \, GasPriceBase}$$

- ➔ gasBumpEstimate - a gas bump that is computed off-chain
- ➔ gasPriceEstimate - the gas price used for computing the gasBumpEstimate

Gas is compensated only if auctionBump > gasBump, ensuring that orders provide sufficient incentive for timely fulfillment. This results in the following rateBump:

$$rateBump = auctionBump - gasBump$$

Appendix: Surplus Fee

Surplus fee is charged on the positive difference between the filling price and the reference price.
First, the scaledExpectedTakingAmount is calculated:

$$scaledEstimatedTakingAmount = \frac{estimatedTakingAmount * makingAmount}{order.makingAmount}$$

- ➔ estimatedTakingAmount - an estimate made by the backend of how many taking tokens the maker is expected to receive based on the current market price

After that the surplus fee is calculated based on the actualTakingAmount and the scaledEstimatedTakingAmount:

$$surplusFee = \frac{(actualTakingAmount - scaledEstimatedTakingAmount) * protocolSurplusFee}{100}$$

➔ actualTakingAmount - takingAmount without the integrator and the protocol fees

Appendix: Rate Bump

The rateBump is calculated based on the current auctionBump and gasBump, as previously mentioned. It is then applied to either the makingAmount or the takingAmount always rounding in favor of the maker:

$$takingAmount = \frac{initialTakingAmount * (BasePoints + rateBump)}{BasePoints}$$

The rateBump is applied on top of the initial taking amount and is rounded up.

$$makingAmount = \frac{initialTakingAmount * BasePoints}{(BasePoints + rateBump)}$$

The making amount is calculated based on the rateBump and is rounded down in favor of the maker.

Appendix: Whitelist

The SimpleSettlement contract implements a custom whitelist mechanism based on time-gated access for resolvers.

Resolvers in the whitelist are sorted in chronological order by their allowedTime, which defines the earliest timestamp at which each resolver is permitted to fill the order.

- Each resolver entry occupies 12 bytes:
    - 10 bytes: lower 80 bits (half) of the resolver's address (for compact storage)
    - 2 bytes: a time offset, indicating how much later this resolver is allowed to fill the order compared to the previous one

The first resolver in the list has an allowedTime equal to the base allowed time. Each subsequent resolver's allowedTime is calculated by adding the stored offset to the previous resolver's time. This ensures that resolvers later in the list can only fill the order at the same or a later time than those before them.

Core Invariants:

INV 1: Whitelisted addresses cannot fill orders prior to their allowedTime

INV 2: Surplus fee is charged whenever an order is filled with a higher amount than the estimated.

Privileged Functions

- None.

Issue_05	Makers can use a discounted gasBump
Severity	**Medium**
Description	In _getRateBump(), the gasBump is subtracted from the auctionBump: *return (auctionBump > gasBump ? auctionBump - gasBump : 0, tail);* This serves as a compensation for the taker, being able to use a discounted rate. However, it also affects the maker, as they will receive less tokens than the current auction state. The gasBump is computed based on the gasGasBumpEstimate and the price change since the gasPriceEstimate was taken. *uint256 gasBump = gasBumpEstimate == 0 || gasPriceEstimate == 0* *? 0* *: gasBumpEstimate * block.basefee / gasPriceEstimate / _GAS_PRICE_BASE;* However, gasBumpEstimate is calculated by the backend based on the gas cost of a simulated order fulfillment. This estimate may not accurately reflect the actual cost of the current transaction, especially if there have been implementation changes or upgrades to the pre-interaction or post-interaction logic. Consider the following scenario: 1) Maker creates an order off-chain, providing the order details and a pre-interaction listener contract. 2) The transaction is processed by the backend and the gasBumpEstimate is set. 3) After the order is created, the maker changes the implementation of the listener contract with a more gas expensive one. As a result, the maker will be able to use a decreased gasBump, increasing the takingAmount and also increasing the transaction

	fee for the taker. While it may slow down the order fulfillment, by doing this the maker will be able to limit taker's profit and secure a bit more optimal fill price.
Recommendations	Consider implementing gas tracking so that such attacks are not feasible.
Comments / Resolution	Acknowledged. Reply from 1inch: We acknowledge the theoretical possibility but consider it economically impractical. Proxy upgrades are on-chain and costly, while gasBump is a minor adjustment. Resolvers are professional traders who simulate execution before submission and skip unprofitable orders, so the risk of abuse is minimal in practice.

Issue_06	Gas compensation should not be linearly dependent on the taking amount
Severity	**Medium**
Description	In _getRateBump(), the gasBump will be discounted from the final rate bump: *return (auctionBump > gasBump ? auctionBump - gasBump : 0, tail);* The rateBump is proportionally applied to the filling amount. The problem is, discounting gasBump from the auctionBump affects the total amount of assets to pay as premium based on the progress of the dutch auction. The gas to compensate is proportional to the filled amount, instead of the actual gas spent to execute the Taker's tx, for example: An order for 1m USD - resolver1 fills 500k - resolver2 fills 5k The two resolvers executes the tx in the same block, (so block.basefee is the same as well as the auctionBump) Let's suppose the auctionBump is 4% and gasBump is 2%, so, rateBump would be returned as 2%. - resolver1 would have to pay 20k(4%) for the auctionBump, but in the end only pays 10k (2%). Meaning, **because of the gasBump, it got a discount of 10k USD.** - resolver2 would have to pay 200(4%), but it pays only 100(2%). Meaning, **resolver2 got a discount of 100 USD.** So, **resolver1 got a discount of 10k while resolver2 got a discount of 100 USD. Even though the 2 resolvers would have spent the same amount of gas.** - That discount comes from the Maker's pocket in the sense they won't receive the correct premium determined by the

	dutch auction.
Recommendations	Consider calculating first the amount of assets the auctionBump represents, and then, discount the gas to compensate from that amount of assets. - Do not mix the gas to compensate with the auctionBump itself. Threat the gas to compensate as an amount of assets rather than a bump (%). After discounting gas to compensate from amount of assets that represents the auctionBump: - Add the remaining it to the \`takingAmount\` - Discount the remaining from the \`makingAmount\`
Comments / Resolution	Acknowledged. Reply from 1inch: We acknowledge the observation, but this is an intentional design choice. The gasBump is not a direct gas refund - it adjusts the auction curve dynamically based on block.basefee changes to ensure fillability. It is applied proportionally to the filled amount to discourage splitting orders into small fills, which would otherwise cause significant losses to makers. This incentivizes resolvers to fill larger portions, improving execution efficiency and fairness across all participants.

Issue_07	Time between resolvers in the whitelist is limited due to 16 bit encoding
Severity	**Low**
Description	In _isWhitelistPostInteractionImpl(), each resolver is defined using 12 bytes - 10 bytes for the maskedTakerAddress and 2 bytes for the allowedTime between the current resolver and the next taker: *allowedTime += uint16(bytes2(whitelist[10:]));* Only two bytes are used for the time delta between two resolvers which means that it will be limited to 65,536 seconds, or around 18 hours and 12 minutes which may not be sufficient for some auctions.
Recommendations	Consider encoding to 3 bytes instead of 2.
Comments / Resolution	Acknowledged. Reply from 1inch: We consider 2 bytes sufficient for our expected use cases. In edge cases where longer delays are required, a workaround is available by inserting a zero_address as a placeholder resolver to artificially extend the allowed time window.

Issue_08	scaledEstimatedtakingAmount is rounded down
Severity	**Informational**

Description	Throughout the codebase, taking amounts are rounded up. However, when estimatedTakingAmount is scaled, scaledEstimatedTakingAmount is rounded down.
	
	uint256 scaledEstimatedTakingAmount = estimatedTakingAmount.mulDiv(makingAmount, order.makingAmount);
	if (actualTakingAmount > scaledEstimatedTakingAmount) {
	
	Therefore, when actualTakingAmount is compared against the scaledEstimatedTakingAmount, it is comparing a value which is rounded up while the other is rounded down. While the difference will be at most 1 wei, we think it's important to make sure that taking amounts rounding is uniform throughout the codebase.

Recommendations	Consider rounding scaledEstimatedTakingAmount up.
Comments / Resolution	Resolved.