SHERLOCK

Security Review For 1inch

Collaborative Audit Prepared For: **1inch**
Lead Security Expert(s): **0x73696d616f**
oot2k
Date Audited: **April 14 - April 17, 2025**

Introduction

This security review focuses on the fee extension for limit order protocol

Scope

Repository: 1inch/fusion-protocol

Audited Commit: c0197a5b4110e89a7a832fff8e95927e41564e17

Final Commit: c0197a5b4110e89a7a832fff8e95927e41564e17

Files:

- contracts/SimpleSettlement.sol

Repository: 1inch/limit-order-protocol

Audited Commit: cf8e50eb24c01c7b276a30a5877840df35e66e67

Final Commit: 889c3976f32a4bfae3e567242491f4f954f1f616

Files:

- contracts/extensions/AmountGetterBase.sol
- contracts/extensions/AmountGetterWithFee.sol
- contracts/extensions/FeeTaker.sol

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	0	3

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue L-1: Taker fee is underestimated due to incorrect fee calculation

Source: https://github.com/sherlock-audit/2025-04-1inch/issues/5

Summary

The taker fee is calculated in a wrong way leading to an underestimation of the fee.

Vulnerability Detail

The taker fee is calculated by picking up the maker amount and multiplying by (1 + fee). The maker amount is the amount the user receives. Consider the following example: Assume that the ratio between the maker and taker assets is the same. The taker wants a making amount of 1000, paying a fee of `1000 * 10% = 100`, so the taking amount is 1100. This means that effectively, the taker pays a fee of `100 / 1100` = 9.1%‘, as they pay 1100 and receive 1000.

Same goes for the inverse calculation.

Impact

Fee is smaller than intended.

Code Snippet

https://github.com/sherlock-audit/2025-04-1inch/blob/main/limit-order-protocol/contracts/extensions/AmountGetterWithFee.sol#L33-L37

https://github.com/sherlock-audit/2025-04-1inch/blob/main/limit-order-protocol/contracts/extensions/AmountGetterWithFee.sol#L55-L60

Tool Used

Manual Review

Recommendation

The correct implementations are:

```
takingAmount = makerAmount / (1 - feeRate)
makerAmount = takingAmount * (1 - feeRate)
```

Discussion

SevenSwen

I believe my previous explanation may have misled you – I apologize for the confusion. Hopefully, things will become clearer now.

When calculating `getTakingAmount` / `getMakingAmount`, both `integrationFee` and `protocolFee` are taken into account (link). Therefore, when calculating the actual fees in `_postInteraction`, we need to reverse that logic in order to correctly determine the total fees and the takingAmount that will be passed to the maker.

The call chain for getTakingAmount will ultimately look like this: `SimpleSettlement.getTakingAmount` → `SimpleSettlement._getTakingAmount` → `FeeTaker._getTakingAmount` (undefined) → `AmountGetterWithFee._getTakingAmount` → `AmountGetterBase._getTakingAmount`

0xsimao

Yeah, that makes sense, thank you, but the issue still stands? You'd have to change here also, as you mentioned, to just `takingAmount.mul(integratorFee)` (same logic to other fee types). So, 1000 taking amount, would become 1000/0.9 = 1111.111 actual taking amount in the `getTakingAmount()` flow above. Then, in `_postInteraction`, the fee is 0.1, being 1111.111*0.1 = 111.1111, which means an effective fee of `10\%` -> user pays 1111.111 and gets a maker amount equivalent to `90\%` of that, or 1000.

zZoMROT

Following up on the previous explanation, we'd like to clarify the intended behavior behind the fee calculation and explain why the observed outcome aligns with our protocol's design.

In our implementation, the `takingAmount` specified in the order is the maximum amount that the taker is willing to pay. The fee is deducted from this amount, meaning the fee is included within `takingAmount` in finally order, not added on top of it. The maker receives the remaining portion after fee deduction.

This design ensures that the taker never pays more than the amount which signed off in the order - this one improves predictability and avoids unexpected costs on filling.

When we fill with `takingAmount`, we calculate `makingAmount` using:

```
makingAmount = takingAmount * BASE / (BASE + fee)
```

This reflects the model where the commission is taken in `takingAsset` terms and included in the total amount the taker sends. While this does result in a slightly lower effective fee (as a percentage of `takingAmount` from order), that is an intentional outcome aligned with the design goals.

In the reverse case - when filling with `makingAmount` and calculating how much the taker must provide to deliver that amount to the maker - we use the inverse formula:

```
takingAmount = makingAmount * (BASE + fee) / BASE
```

This ensures that the total amount the taker sends is enough to cover both the maker's amount and the fee.

We also want to clarify that this approach assumes that `_getMakingAmount` and `_getTakingAmount` are homogeneous functions of degree 1, meaning they are strictly proportional to their inputs (i.e., of the form `f(x) = a * x`). This allows us to safely apply a scaling factor outside the function. If either function included additive terms (`+ b`), such as thresholds or fixed adjustments, this assumption would no longer hold, and the proportional fee adjustment would be invalid. We are aware of this limitation and enforce proportionality in those functions to preserve correctness.

To summarize:

- `takingAmount` in the order already includes the fee
- the taker never pays more than what is specified in the order
- the maker receives the net amount after fee deduction
- we apply the appropriate scaling depending on the calculation direction
- the model is valid under the assumption of proportional (degree-1 homogeneous) logic in `_getMakingAmount` and `_getTakingAmount`

Please let us know if you would like us to document this explicitly in the code or specification.

Issue L-2: uniTransfer() limits the gas limit to 5k in outdated 1inch solidity-utils lib

Source: https://github.com/sherlock-audit/2025-04-1inch/issues/6

Summary

`UniERC20::uniTransfer()` limits the gas to 5k in the solidity-utils lib version 6.3.0.

Vulnerability Detail

See the version used in the current commit of the protocol here.

The solidity-utils lib has been updated in 6.4.0 and now forwards all gas, which means that the only fix needed is bumping this version in the package.json file.

Impact

Sweeping could fail due to this limitation. Another owner would have to be used, that most likely can not be a multisig so it could be a problem.

Code Snippet

https://github.com/sherlock-audit/2025-04-1inch/blob/main/limit-order-protocol/contracts/extensions/FeeTaker.sol#L97-L99

Tool Used

Manual Review

Recommendation

Bump the solidity utils version to the latest.

Discussion

SevenSwen

Strictly speaking, it's a coincidence that the next version of our library no longer has a gas limit. We didn't originally intend to remove this feature (since it can potentially offer some protection against reentrancy attacks). However, on the zkSync network, gas

calculations turned out to be different, and it’s currently not possible to set a specific constant value (at least for now).

So, as part of discussing this trade-off, we decided to remove the gas limit to make the logic consistent across all networks.

That said, it does seem like we really need to update the library if we want to deploy this protocol on zkSync.

0xsimao

That's interesting, but in the context of a sweep function, there is no reentrancy issue

Issue L-3: Significant rounding error in whitelist discount that could be avoided

Source: https://github.com/sherlock-audit/2025-04-1inch/issues/7

Summary

Whitelisted users pay less resolver fee but the discount is applied too early leading to rounding errors.

Vulnerability Detail

The resolver fee is discounted by doing the following:

```
if (isWhitelisted) {
    resolverFee = resolverFee * whitelistDiscountNumerator / _BASE_1E2;
}
```

As can be seen, it is applied in the `resolverFee` itself, which just has `1e5` precision.

Impact

Code Snippet

https://github.com/sherlock-audit/2025-04-1inch/blob/main/limit-order-protocol/contracts/extensions/AmountGetterWithFee.sol#L88

Tool Used

Manual Review

Recommendation

In `_getMakingAmount()` it may not be possible to apply the fix directly because it is dividing by the `resolverFee`. Probably adding a precision multiplier would be better. For example, scaling it to 1e18 fixes the issue: Suppose the fee is 9999, taking amount is 10_000 USD, share is `50%`. Scaled fee = resolver fee * 1e18 / 1e5 / 2 = 9999 * 1e18 / 1e5 / 2 = 4.9995e16 (notice how the 0.5 is kept). Fixed total resolver fee = takingAmount * resolverFee / 1e18 = 10_000e6 * 4.9995e16 / 1e18 = 499.95e6. The 0.05 USD component would be lost without scaling the fee. It may not be much but with enough volume it adds up.

Discussion

zZoMROT

Thanks, that's a helpful observation. We'll take this into account when considering potential improvements to fee precision.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.