1inch Limit Order Protocol Audit

DECEMBER 16, 2021 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

Introduction

The 1inch team asked us to review and audit their Limit Order Protocol v2 smart contracts. We looked at the code and now publish our results.

Scope

We audited commit `4d94eea25e4dac6271bfd703096a5c4a4d899b4a` of the `1inch/limit-order-protocol` repository. In scope were the following contracts:

- `OrderMixin.sol`
- `OrderRFQMixin.sol`
- `PredicateHelper.sol`
- `RevertReasonParser.sol`
- `Permitable.sol`
- `ChainlinkCalculator.sol`
- `ArgumentsDecoder.sol`
- `AmountCalculator.sol`
- `NonceManager.sol`
- `LimitOrderProtocol.sol`
- `ImmutableOwner.sol`
- `InteractiveNotificationReceiver.sol`
- `AggregatorInterface.sol`
- `IDaiLikePermit.sol`

All other project files and directories (including tests), along with external dependencies and projects, game theory, and

incentive design, were also excluded from the scope of this audit. External code and contract dependencies were assumed to work as documented and back-end services provided by 1inch were assumed to act in the best interest of the protocol.

Overall Health

In general, we found the project's codebase to be readable and well organized, although it could benefit from more extensive documentation, especially around the blocks of assembly code, edge cases of the protocol, assets/predicates/external-resources that will be used, responsibilities/limitations of the back-end service provided, and interactions between actors. The project goes to great lengths to make actions gas-efficient, occasionally even at the risk of making the code more difficult to reason about; we raise issues related to that below. Throughout the audit, the 1inch team was highly available, responsive, and very easy to work with.

System Overview

The Limit Order Protocol enables order `makers` to sign orders off-chain for token swaps. The protocol then facilitates the filling of previously signed orders by order `takers`. Orders are highly extensible, and can static-call external contracts at several points throughout the order filling process. This extensibility imbues the protocol with utility, but it adds both complexity and a greater attack surface for the orders themselves.

It is important to remark that there is no on-chain storage of order details. The fill-state or cancellation status of orders is only tracked via order hashes. This necessitates that orders be shared peer-to-peer or via a centralized party. In this case, the 1inch team intends to act as that centralized party, aggregating signed orders and using those orders as a source of liquidity for their other protocols. Orders will be published via their own API so users are able to interact with them.

This centralization gives the 1inch team extreme control over which orders are published and ultimately executed. This also gives them the ability to censor orders, which could be useful in the case of malicious or deceptive orders, but could also be abused and allow them to frontrun any other user in case of a favorable order by not showing it through the API.

Privileged Roles

Though the contracts the role is used in were out of scope, one privileged role was identified. An `immutableOwner` is set to the creator of a proxy contract at the time of construction, and is used to limit access to the proxy's `external` functions.

External dependencies and trust assumptions

The design of this protocol necessitates off-chain and on-chain components, and this hybrid model can be used to mitigate some attack vectors we identify in our report, but the cost of that ability is increased reliance on the 1inch team and infrastructure.

Additionally, the Limit Order Protocol provides functions that are meant to retrieve prices from Chainlink oracles. We assumed those oracles to be honest, accessible, and properly functioning.

Moreover, due to the flexibility of an order, there are several points of contact with external contracts which are not validated. This means that a malicious user could abuse such calls and impersonate predicates, assets, or oracles with malicious contracts in order to execute actions during order fills. Although the project is protected in some areas against reentrancy, such vectors could cause denial of service attacks or undetected spam orders. The 1inch team is aware that certain issues may appear when using unfamiliar contracts for the protocol and has indicated their intent that only major "blue chip" assets will be fully supported by the project. However, it should be noted that even with the most popular assets there are intrinsic behaviors from each asset that may cause problems on protocols that do not address them properly, such as having a fee during transfers with USDT or returning an error code instead of a success boolean with cTokens.

Findings

Here we present our findings.

Critical severity

None.

High severity

[H01] Inconsistent data passed into `_makeCall`

In the `OrderMixin` contract, the `_makeCall` function is used to transfer assets from the taker to the maker and then from the maker to the taker. In the latter transfer, the `_makeCall` function is incorrectly passed the order's `makerAsset` as the last parameter, when it should be the order's `makerAssetData`.

As a result, any proxy functionality that relies on the `makerAssetData` argument will break.

To be consistent with the earlier call to `_makeCall` and to fully support proxy functionality, consider updating the `order.makerAsset` parameter to `order.makerAssetData`.

Update: *Fixed in pull request #57.*

[H02] Partially-filled private orders can be filled by anyone

The protocol allows the creation of private and public orders. On private orders, only the `allowedSender` address, specified by the maker during the order's creation, is able to fill the order.

However, in the `OrderMixin` contract, validation for the `allowedSender` address is incorrectly scoped, meaning that it is only evaluated inside the logic that handles the first fill of an order. If a private order is partially filled, then the check for the `allowedSender` address is no longer reachable and the order becomes fillable by anyone.

To clarify intent around whether any user should be able to fill partially-filled private orders or not, consider either documenting the reason for the current behavior or validating the `allowedSender` address outside of the scope of the first fill to ensure that it will be validated every time a fill is attempted.

Update: *Fixed in pull request #58.*

[H03] Malicious maker could take advantage of partial fills to steal taker's assets

Orders from the `OrderMixin` contract have the ability to be partially filled. To support partial fills, the protocol requires a way to calculate both sides of swaps. Both `getMakerAmount` and `getTakerAmount` fields are defined by the maker of the order for this exact purpose.

When filling an order, takers must provide either the `makingAmount` or the `takingAmount` values as well as a `thresholdAmount` value. There are two different code-paths that can be taken, based on if the `makingAmount` or the `takingAmount` was provided.

The first one is when the `makingAmount` parameter is defined. It could truncate the `makingAmount` value and also calculate the `takingAmount` value for it. In this situation, the `thresholdAmount` ensures that the `takingAmount` value taken is not unexpectedly large.

The second one is when the `takingAmount` parameter is defined. In such case, it will calculate the `makingAmount` value, with the possibility of truncating it and recalculating the `takingAmount` value if that happens. In this situation, the `thresholdAmount` value ensures that the `makingAmount` value returned is not unexpectedly small.

There exist two exploitation methods, each unique to one of the previous mentioned code-paths. These exploitation methods require malicious `getMakerAmount` and `getTakerAmount` functions. A simple implementation of these functions would have an identical behavior to `AmountCalculator`'s `getMakerAmount` and `getTakerAmount` functions, but with a hard-coded switch that will force them to return an attacker controlled value when needed.

The first, less severe exploit pattern involves the first code-path where the `makingAmount` value is specified in a fill order. A malicious maker would wait for a fill order which specifies `makingAmount` to show up in the mempool in order to frontrun it. They would drain all of the value except 1 from the maker's side and then force `_callGetTakerAmount` to return the amount specified in the user's `thresholdAmount` value (or their allowance if it is less). When the user's transaction finally goes through, they will swap their full `thresholdAmount` worth of `takerAsset` for a single unit of `makerAsset`. This exploit is limited by the amount given by the `thresholdAmount` value or the amount of the `takerAsset` the user allowed on the `LimitOrderProtocol` contract.

The second, more severe exploit pattern involves the second code-path where the `takingAmount` value is specified. The malicious maker would similarly wait for a fill order that specified a `takingAmount` value to show up in the mempool. They would frontrun the transaction and force the `makingAmount` value returned by `_callGetMakerAmount` function to be higher than both the `remainingMakerAmount` and the `thresholdAmount`. They would also set the `takingAmount` returned value by `_callGetTakerAmount` to be the amount of `takerAsset` asset allowed on the `LimitOrderProtocol` by the taker. When the taker's transaction goes through, it will truncate the `makingAmount` value and then recalculate the `takingAmount` value. This recalculation is not guaranteed to be lower however, and in this case will drain the taker of all the `takerAsset` that they allowed on the contract. In this code-path, the `thresholdAmount` value is ensuring that the `makingAmount` is not too low, so taking all the taker's `takerAsset` asset is unchecked. The funds lost are bounded by the amount of the `takerAsset` asset the user allowed on the `LimitOrderProtocol` contract.

These exploits are not possible without partial orders and, more specifically, partial orders with malicious `getMakerAmount` and `getTakerAmount` implementations.

The main issue of the `thresholdAmount` value check is that it only covers one side of the swap, but the other side can be manipulated via frontrunning. There are no assurances that the value the taker originally proposed remains unchanged. Consider removing `makingAmount` truncation from both code-paths and reverting if the order cannot support a fill as large as requested. By doing this, the `thresholdAmount` can be used to sufficiently restrict the other side of the swap and avoid unexpected behavior, even in malicious orders.

Update: *Fixed in pull request #83.*

Medium severity

[M01] Static arguments passed after dynamic arguments

In the `OrderMixin` contract, the `getTakerAmount` and `getMakerAmount` bytes fields are used as arguments for the `_callGetTakerAmount` and `_callGetMakerAmount` functions. These calls provide a way to calculate one side of the swap based on the other side, and they allow users to partially fill orders.

The `getTakerAmount`/`getMakerAmount` fields are dynamic variables and are packed in front of the `takerAmount` and `makerAmount` values in the `_callGetTakerAmount` and `_callGetMakerAmount` functions. It is possible for a malicious maker to provide more data than expected in the `getTakerAmount` and `getMakerAmount` fields to push the `takerAmount` and `makerAmount` bytes past where they are assumed to be when being decoded in the next function. This allows the maker to shift the passed in taker or maker amount by a full bytes to the

right and even replace them completely if an extra 32 bytes of data is provided.

Users already have to manually review the `getTakerAmount` and `getMakerAmount` fields in the order, but this technique is rather hard to spot. Also worth noting, this attack even applies to the internally trusted `getMakerAmount` and `getTakerAmount` functions. For most attacks, providing a reasonable threshold amount will prevent loss of funds.

To prevent this, consider encoding the static arguments before the dynamic arguments to avoid giving the dynamic arguments a method to control the static arguments.

Update: *Not fixed. The 1inch team stated:*

> We'll take extra care with getters validation. We'll try to implement sanity validation of getters in our sdk that will help with filtering potentially malicious orders.

[M02] ERC721 orders can be manipulated

It is possible to exchange more than just ERC20s via the `OrderMixin` by deploying a contract that shares the same function selector as IERC20's `transferFrom`, and providing that contract as the `makerAsset` or the `takerAsset` in an order.

The out-of-scope proxies, namely, `ERC721Proxy`, `ERC721ProxySafe`, and `ERC1155Proxy` contracts follow this pattern to provide support for `ERC721` and `ERC1155` tokens. Since the proxies must be called with the same pattern as an IERC20 `transferFrom` call, the signature must start with `address from`, `address to` and `uint256 amount`. Anything else that the proxies require can be passed in after, and is defined in the order as `makerAssetData` and `takerAssetData`.

ERC1155s can naturally transfer multiple of the same id tokens at once, which means the `ERC1155Proxy` contract makes use of the `amount` field. On the other hand, `ERC721`s do not have an obvious use for the `amount` field. Since they represent non-fungible tokens, a specific tokenId will only have one in existence, rendering the `amount` field useless. Because of this, the implementation for both `ERC721Proxy` and `ERC721ProxySafe` contracts use the required `amount` field as the `tokenId` instead.

This overloading of the `amount` parameter creates the possibility of partially filling `ERC721` orders in order to purchase separately listed tokens at discounted prices. For instance, there could be a case where a single user has multiple `ERC721`s of the same contract permitted to be transferred by the `ERC721Proxy` contract and lists them in separate limit orders. If the limit orders also provide the `getMakerAmount` and `getTakerAmount` fields, it will be possible to partially fill these `ERC721` orders. Since the order's `amount` field actually corresponds to the `tokenId`, a malicious user can place a partial fill on the `ERC721` with the higher tokenId, resulting in a `makingAmount`/`takingAmount` of an `ERC721` that could correspond to a lower `tokenId`. The result is the `ERC721` with the lower `tokenId` would be transferred at the price of `(higher tokenId price) * (lower tokenId's id) / (higher tokenId's id)`.

This exploit has a few requirements:

- Multiple `ERC721`s from the same contract to be allowed on either `ERC721` proxy by a single owner.
- Open order for one of the `ERC721`s that is not the lowest `tokenId` of the ones allowed.
- Partial fills allowed on the order.

To completely remove the possibility of partial `ERC721` fills, consider separating the `amount` and `tokenId` arguments. Whether the arguments are separated or not, consider also documenting this to alert users of this behavior and to avoid this pattern in the future.

Update: *Fixed in pull request #59.*

[M03] Undocumented decimal assumptions

The `LimitOrderProtocol` contract inherits the `ChainlinkCalculator` contract through the `OrderMixin` contract. This contract exposes two functions to enable the usage of Chainlink oracles during the predicates check and the lookup of the maker amount/taker amount.

However, the contract makes undocumented assumptions about the number of decimals that the Chainlink oracles should report in, as well as the number of decimals that the function parameters should contain. In certain scenarios, this could lead to unexpected behaviors, including the mis-pricing of assets and the unintentional loss of funds.

More specifically, throughout the contract the implicit assumption is that the Chainlink oracles will report with 18 decimals of precision. However, not all Chainlink oracles report with this number of decimals. In fact, if the oracle reports a token pair that is in terms of a currency (USD, for instance), it will only have 8 decimals of precision. Since there are no restrictions on *which* oracles can be used, implicit assumptions should not be made about the number of decimals they will report with.

Relatedly, there is an implicit assumption that the `amount` parameter for the `ChainlinkCalculator` functions will use 18 decimals, together with the misleading explicit declaration that the `singlePrice` function `Calculates price of token relative to ETH scaled by 1e18`. In reality, even with an oracle that *does* report with 18 decimals, the return value of the `singlePrice` function would be scaled by the number of decimals of the `amount` parameter, which may not necessarily be 18 decimals.

Similarly, the `doublePrice` function assumes that two Chainlink oracles will report with the same number of decimals, causing the result of the function to deviate from expectations.

Consider explicitly documenting assumptions regarding the number of decimals that parameters and return values should be in terms of. Furthermore, consider either limiting calculations that depend on oracles that break those assumptions, or having the relevant calculations take the actual number of decimals into account.

Update: *Fixed in pull request #75.*

Low severity

[L01] Constants not declared explicitly

There are a few occurrences of literal values being used with unexplained meaning in the codebase. For example:

- In the `OrderMixin` contract, the `_remaining` mapping is semantically overloaded (as explained in the issue ***Semantic overloading of mapping***) to track the amount of asset remaining for a partially filled order *as well as* if an order has been completely filled. Specifically, `0` means that no fills associated with an order have been made, `1` means an order can no longer be filled, and anything larger than `1` means that there is a remaining amount associated with the order that can potentially be filled.
- In the `ChainlinkCalculator` contract, the literal value `1e18` is used in the `singlePrice` function.

To improve the code's readability and facilitate refactoring, consider defining a constant for every magic number, giving it a clear and self-explanatory name. For complex values, consider adding an inline comment explaining how they were calculated or why they were chosen.

Update: *Fixed in pull request #75 and pull request #76.*

[L02] Malicious parties could prevent the execution of permitable orders

The `OrderMixin` contract allows maker users to submit permitable orders so those can be executed in one transaction, rather than having to have a separate transaction for approvals. Also, order takers can submit their own permit during the

filling of the order for the same purpose.

However, because the maker's permit is contained inside the order, both the maker's and the taker's permits would be accessible while the order-fill transaction is in the mempool. This would make it possible for any malicious user to take those permits and execute them on the respective asset contracts while frontrunning the fill transaction. Because these permits have a `nonce` to prevent a double spending attack, the order's fill transaction would fail as a result of trying to use the same permit that was just used during the frontrun.

Although there is no security risk, and the maker could create a new order and pre-approve the transaction, this attack could certainly impact the usability of permitable orders. Indeed, a motivated attacker could block *all* permitable orders with this attack. Consider validating if the permit was already submitted, or if the allowance is enough, during the order fills. Also consider letting users know about this possible attack during order composition.

Update: *Not fixed. The 1inch team states:*

> We had approval checks before but decided to simplify permit flow to just revert on unsuccessful approvals. We'll think about the ways to notify makers about the issue.

[L03] Duplicated code

There are instances of duplicated code within the codebase. Duplicating code can lead to issues later in the development lifecycle and leaves the project more prone to the introduction of errors. Such errors can inadvertently be introduced when functionality changes are not replicated across all instances of code that should be identical. Examples of duplicated code include:

- In the `Permitable` contract, the `_permit` and `_permitMemory` functions are duplicates.
- The calculations in the `getMakerAmount` and `getTakerAmount` functions are duplicated in the `fillOrderRFQTo` function when calculating making and taking amounts, respectively.

Rather than duplicating code, consider having just one contract or library containing the duplicated code and using it whenever the duplicated functionality is required.

Update: *Partially fixed in pull request #60.*

[L04] Erroneous or misleading test suite

There are instances in the test suite where the tests deviate from their expected behavior. For instance:

- The `ChainlinkCalculator` contract is inherited by the `OrderMixin` contract. However, during the tests the `AmountCalculator.arbitraryStaticCall` function is used to call the `ChainlinkCalculator` contract as an external, independent contract. Even though the result is the one expected, the test should reflect the behavior with the current design of the system and anticipated use case by calling `ChainlinkCalculator` functions directly without using the arbitrary static call.
- Though the proxy contracts were out of scope, we noticed that when testing the protocol with ERC721 assets, the `ERC721Proxy` contract is not used to swap the assets in its test suite.

As the test suite itself is outside the scope of this audit, please consider thoroughly reviewing the test suite to make sure all tests run successfully according to the specifications of the protocol.

Update: *Fixed in pull request #57, pull request #59, and pull request #61.*

[L05] Errors and omissions in events

Throughout the codebase, events are generally emitted when sensitive changes are made to the contracts. However, many events lack indexed parameters and/or are missing important parameters. For example:

- The `OrderRFQMixin.OrderFilledRFQ`, `OrderMixin.OrderFilled`, and `OrderMixin.OrderCanceled` events should index the `orderHash` parameter.
- The `OrderRFQMixin.OrderFilledRFQ` and `OrderMixin.OrderFilled` events should be more complete, including the `maker`, `taker`, `target`, `amountMaking`, and `amountTaking` where possible.

There are also sensitive actions that are lacking events, such as:

- In the `OrderRFQMixin` contract, the `cancelOrderRFQ` function does not emit an event when an order is canceled.

Consider more completely indexing existing events and adding new parameters where they are lacking. Also, consider emitting all events in such a complete manner that they could be used to rebuild the state of the contract by off-chain services.

Update: *Not fixed. However, the 1inch team did add an `orderRemaining` parameter to the `OrderCanceled` event in pull request #62.*

The 1inch team states:

> We found that only a limited subset of data is required to satisfy frontend needs. In the case of extensive analysis, all the suggested fields are available via tracing. For `OrderRFQMixin` we expect market makers to build their own sophisticated way of tracking what orders have been canceled.

[L06] Storage changes during event emission

In the `NonceManager` contract, when the `NonceIncreased` event is emitted, the nonce of the message sender is also increased.

Executing multiple operations simultaneously can make the codebase harder to reason about, more prone to errors, and can lead to operations being overlooked or misunderstood.

To improve the overall intentionality, readability, and clarity of the code, consider increasing the nonce value before emitting the event.

Update: *Fixed in pull request #63.*

[L07] Inconsistent decoding methodologies could cause outcome discrepancies

To support all of its extensibility and flexibility, the Limit Order Protocol routinely has to deal with dynamic bytes data and arbitrary return values from external contracts. As a result, the protocol includes an `ArgumentsDecoder` library to more efficiently convert dynamic bytes values into basic data types. However, this library is not used exclusively, and in some cases `abi.decode` is used instead. Additionally, some contracts are using `abi coder v1` while others are using `abi coder v2`. The former performs more similarly to the `ArgumentsDecoder` library, whereas the latter performs additional checks when decoding.

The inconsistent usage of these different decoding methodologies can result in subtle discrepancies between the intention and actual behavior of the codebase.

For instance, the `simulateCalls` function only uses the `ArgumentsDecoder.decodeBool` function. If the `simulateCalls` function is used to check calls that would be made in the predicate part of an order, then its results could deviate from what

actually occurs when the predicate conditions are evaluated, because different decoding methodologies are employed.

So, for instance, if a predicate makes an external `staticcall` to some function that returns a `uint256` value greater than one rather than the expected `bool`, then that call will revert, because the return value is decoded with `abi coder v2`'s `abi.decode` which will not accept such values as `bool`. However, if the exact same call is made with `simulateCalls`, then it will simply be marked as `true`, because `decodeBool` treats any value larger than zero as `true`.

To make the `simulateCalls` function fully mirror the behavior of actual predicate calls, consider modifying it to use `abi.decode`.

Update: *Fixed in pull request #82.*

[L08] Lack of input validation

The `fillOrderToWithPermit` and `fillOrderTo` functions of the `OrderMixin` contract, as well as the `fillOrderRFQToWithPermit` and `fillOrderRFQTo` functions of the `OrderRFQMixin` contract, do not validate the `target` address parameter.

This makes it possible for a user to inadvertently pass in the zero address and, as a result, lock up the assets they are meant to receive after filling an order.

To ensure that users do not accidentally lock up their funds, consider validating that the `target` address does not equal the zero address in the cited functions.

Update: *Fixed in pull request #78.*

[L09] Low unit test coverage

The unit test coverage for the entire project is around 75%, with some of the contracts having particularly low coverage.

Considering the importance of unit tests to validate code and prevent regressions when refactoring and developing new features, we encourage significantly increasing unit test coverage to at least 95%, and including edge cases that cover even unlikely situations.

Update: *Not fixed.*

[L10] Misleading or incomplete inline documentation

Throughout the codebase, a few instances of misleading and/or incomplete inline documentation were identified and should be fixed.

The following are instances of misleading inline documentation:

- In the `ChainlinkCalculator` contract, the `singlePrice` function's NatSpec `@notice` tag says that it `Calculates price of token relative to ETH scaled by 1e18`, but in fact, its result is the *value* of `amount` tokens scaled by `1e18`, where the oracle may not report in terms of ETH (for a pair not including ETH, for instance).
- In the `OrderRFQMixin` contract, the `invalidatorForOrderRFQ` function's NatSpec `@return` tag is misleading, because the quote may not have been filled for the respective invalidator bit to have been set. The order can also have been canceled.
- On lines 147, 165, and 188 of `OrderMixin.sol`, the NatSpec `@param` tags are ungrammatical.
- On line 20 of `ERC1155Proxy.sol`, the `@notice` tag states that the computed hash is the result of hashing the `func_733NCGU` function, where it should be the `func_301JL5R` function instead.

The following are instances of incomplete inline documentation:

- Functions in the `AmountCalculator` contract do not describe any of the parameters.
- In the `ChainlinkCalculator` contract, the `singlePrice` and `doublePrice` functions do not describe all of the parameters.
- In the `ImmutableOwner` contract, the public variable and modifier have no NatSpec.
- In the `InteractiveNotificationReceiver` contract, the `notifyFillOrder` function does not describe any of the parameters.
- In the `LimitOrderProtocol` contract, the `DOMAIN_SEPARATOR` function has no NatSpec.
- Events and mappings in the `NonceManager` have no NatSpec.
- In the `OrderRFQMixin` contract, `cancelOrderRFQ*` functions do not describe the return values.
- In the `OrderMixin` contract, several functions lack complete NatSpec.
- On line 168 of `OrderMixin.sol` and on line 71 of `OrderRFQMixin.sol`, it is missing the `@dev` tag.
- Functions in the `PredicateHelper` contract do not describe all of the parameters.

Clear inline documentation is fundamental for outlining the intentions of the code. Mismatches between the inline documentation and the implementation can lead to serious misconceptions about how the system is expected to behave. Consider fixing these errors to avoid confusion for developers, users, and auditors alike.

Update: *Partially fixed. Misleading documentation addressed in pull request #75 and pull request #77.*

The 1inch team states:

> We've fixed misleading docs. Completion of the docs will be done later.

[L11] DoS orders possible when using hooks

The `OrderMixin` contract implements functionality to fill generic off-chain swap orders which could have conditions for their success. During order fills, the order can check the predefined "predicate" conditions before continuing with execution.

However, because these predicate conditions could target the logic of any arbitrary contract, a malicious maker could trick takers into believing that an order behaves correctly and that it is valid when checking it off-chain, but then failing when attempting to fill the same order on-chain. This change in the predicate behavior could either be made by frontrunning some variable state on which the predicates depend, by examining the gas sent or even which addresses are involved in the call, or by some other logic.

Furthermore, if the maker defined a interaction during the swap, the `interactionTarget` contract could revert itself or revoke the allowance to prevent a successful order fill, essentially leading to the same result as malicious predicates.

Although assets will not be at risk, users or bots finding a favorable order will have the increased burden of trying to identify these sorts of spam orders that can seem legitimate on the surface. In the case that they fail to identify these sorts of orders they will incur wasted gas costs. To reduce the amount of spam orders, consider restricting the available targets for these hooks. Also consider warning users about this possibility before they attempt to fill orders.

Update: *Not fixed. The 1inch team states:*

> We handle that on our backend and we'll think about the ways to notify possible takers about the issue.

[L12] Rounding can be unfavorable for `taker`

In the `OrderMixin` and `OrderRFQMixin` contracts, when an order is being filled and the taker provides only a `makingAmount` or `takingAmount` amount, the protocol attempts to calculate the counterpart amount of the swap.

There are two issues with these calculations, the first being that there is no documentation or logic limiting the number of decimals that the amount parameters should use, which we addressed in the ***Undocumented decimal assumptions*** issue. The second issue is that, in the course of these calculations, the protocol rounds in the favor of the maker. The rounding issue can be greatly exacerbated when the implicit decimal assumptions are broken, but even when everything is in the expected terms, rounding will occur with small, odd amounts.

Consider allowing the taker to specify a minimum amount of `makerAsset` asset that they are willing to receive together with a maximum amount of `takerAsset` asset they are willing to swap, so that the acceptance of any rounding is more explicit.

Update: *Not fixed. The 1inch team states:*

> Threshold amount should be enough for taker's protection.

[L13] Contradictory order handling when lacking parameters

In the `OrderMixin` contract, the `fillOrderTo` function makes internal calls to the `_callGetMakerAmount` and `_callGetTakerAmount` functions whenever a fill is attempted and either the `makingAmount` or the `takingAmount` parameters are zero, respectively, or if the `makingAmount` value is larger than the `remainingMakerAmount` value.

The `_callGetMakerAmount` and `_callGetTakerAmount` calls will lead to reversions if the order was not created with the `getMakerAmount` or `getTakerAmount` parameters, respectively, and a partial fill is being executed.

An inline comment alongside `_callGetMakerAmount` and an inline comment alongside `_callGetTakerAmount` claim that "only whole fills are allowed" if the order was not created with `getMakerAmount` or `getTakerAmount` parameters.

However, there are code paths for which this does not apply, because those paths do not check the `length`s of both `getMakerAmount` and `getTakerAmount` parameters.

Specifically, when a `taker` specifies a `takerAmount` value for an order which only has a `getMakerAmount`, unless that call to `getMakerAmount` returns an amount larger than `remainingMakerAmount`, a partial fill can be executed in contradiction to the inline documentation.

This leaves the intentionality of those code paths unclear. If this is the expected behavior, consider modifying the inline documentation so that it is more explicit. If this is unintentional behavior, consider always checking the lengths of both the `getMakerAmount` and the `getTakerAmount` parameters simultaneously so that the implementation reinforces the behavior described by the inline documentation.

Update: *Fixed in pull request #79.*

[L14] Using deprecated Chainlink calls

The `ChainlinkCalculator` contract is intended to be used to query Chainlink oracles. It does so via making calls to their `latestTimestamp` and `latestAnswer` methods, both of which have been deprecated. In fact, the methods are no longer present in the API of Chainlink aggregators as of version three.

To avoid potential future incompatibilities with Chainlink oracles, consider using the `latestRoundData` method instead.

Update: *Fixed in pull request #67.*

Notes & Additional Information

[N01] Not importing interfaces

The `AggregatorInterface` interface appears to be a subset of code copied from `ChainLink`'s public code repository. The full interface is included in `ChainLink`'s contract npm package.

When possible, to lessen the potential for interface mismatches and resultant issues, rather than re-defining and/or rewriting another project's interfaces, consider using interfaces installed via their official npm packages instead.

Update: *Fixed in pull request #66.*

[N02] Deprecated project dependencies

During the installation of the project's dependencies, NPM warns that one of the packages installed, `Highlight`, "will no longer be supported or receive security updates in the future".

Even though it is unlikely that this package could cause a security risk, consider upgrading the dependency that uses this package to a maintained version.

Update: *Fixed in pull request #64.*

[N03] External calls to view methods are not staticcalls

Throughout most of the codebase, the protocol explicitly makes external calls via OpenZeppelin's `functionStaticCall` method to restrict the possibility for state changes when they are either not expected or not desirable. However, in the `ChainlinkCalculator` contract, despite the intention of making external calls only to `view` methods on Chainlink oracles, the external calls in the `singlePrice` and `doublePrice` functions are not made via explicit `staticcall`s.

While we did not identify any immediate security concerns stemming from this, to reduce the attack surface, improve consistency, and clarify intent, consider using explicit `staticcall`s, for all external calls to `view` functions in the `ChainlinkCalculator` contract.

Update: *Not fixed. The 1inch team states:*

> We think that syntax complication nullifies improvements in consistency.

[N04] Not failing early with invalid orders

In the `OrderMixin` contract, the `fillOrderTo` function handles the special condition when an order has not been previously submitted (`remainingMakerAmount == 0`), but it does not explicitly handle the condition when the order is no longer valid (`remainingMakerAmount == 1`).

In the latter scenario, the function will eventually revert, but only after burning non-trivial amounts of gas. To clarify intent, increase readability, and reduce gas usage, consider explicitly handling the invalid-order scenario towards the beginning of the function.

Update: *Fixed in pull request #68.*

[N05] Helper contracts not marked as abstract

In Solidity, the keyword `abstract` is used for contracts that are either not functional contracts in their own right, or are not meant to be used as such. Instead, `abstract` contracts are inherited by other contracts in the system to create stand-alone

functional contracts.

Throughout the codebase, there are examples of helper contracts that are not marked as abstract, despite the fact that they are not meant to be deployed on their own. For instance, the `AmountCalculator`, `ChainlinkCalculator`, `ImmutableOwner`, `NonceManager`, and `PredicateHelper` contracts are all comprised of a base set of functions which are intended to be used by inheriting contracts.

Consider marking helper contracts as `abstract` to clearly signify that they are designed solely to add functionality to contracts that inherit them.

Update: *Not fixed. The 1inch team states:*

> Those helpers can be deployed separately. They are inherited only for gas savings.

[N06] Inconsistent function ordering

Throughout the codebase, function ordering generally follows the recommended order in the Solidity Style Guide, which is: `constructor`, `fallback`, `external`, `public`, `internal`, `private`.

However, in the the `OrderMixin` contract, the `public` `checkPredicate` function deviates from the style guide, bisecting the `external` functions.

To improve the project's overall legibility, consider standardizing function ordering throughout the codebase, as recommended by the Solidity Style Guide.

Update: *Fixed in pull request #69.*

[N07] Inconsistent order fill flow

The `OrderMixin` and `RFQOrderMixin` contracts both handle the filling of signed orders, but the general order flow between the two contracts is inconsistent.

`OrderMixin`'s `fillOrderTo` function follows this general flow (pseudo-code):

```
if ((takingAmount == 0) == (makingAmount == 0))
else if (takingAmount == 0)
else
  (handle makingAmount == 0)

THEN

swapTokens
```

Whereas `RFQOrderMixin`'s analogous `fillOrderRFQTo` function follows this flow (pseudo-code):

```
if (takingAmount == 0 && makingAmount == 0)
else if (takingAmount == 0)
else if (makingAmount == 0)
else
  revert

THEN

swapTokens
```

There are no insights from the documentation as to why the first conditional in each of these two functions differs, or why `takingAmount` and `makingAmount` cannot both be zero in the latter function. Also, the case where both a `makingAmount` and a `takingAmount` are provided is much easier to reason about in the `fillOrderRFQTo` function, since it is handled clearly in the final `else` block.

To clarify intent and increase the overall readability of the code, consider either standardizing the general order flow across these two contracts, or explicitly documenting why the differences exist.

Update: *Not fixed. The 1inch team states:*

> This is due to custom pricing functions in limit orders. Since `getMakerAmount` can potentially substantially differ from `getTakerAmount`, we thought that it is better not to make default option for the taker as it'll probably confuse them in cases when those getters will be different.

[N08] Error messages are inconsistently formatted or misleading

Throughout the codebase, the `require` and `revert` error messages, which are meant to notify users of the particular conditions causing a transaction to fail, were found to be inconsistently formatted.

For instance, each one of the error messages on lines 85 of `OrderMixin.sol`, 16 of `ERC721ProxySafe.sol`, and 26 of `Permitable.sol` employ a different style.

Additionally, some error messages are misleading:

- `LOP: one of amounts should be 0` should be `at least one of the amounts should be 0`, since either or both amounts can be zero.
- `LOP: only one amount should be 0` should be `LOP: one, and only one, amount should be 0`.

Error messages are intended to notify users about failing conditions, so they should provide enough information so that appropriate corrections can be made to interact with the system. Uninformative error messages greatly damage the overall user experience, thus lowering the system's quality. Moreover, inconsistently formatted error messages can introduce unnecessary confusion. Therefore, consider reviewing the entire codebase to make sure every `require` and `revert` statement has an error message that is consistently formatted, accurate, informative, and user-friendly.

Update: *Partially fixed in pull request #81.*

[N09] Inconsistent use of named return variables

There is an inconsistent use of named return variables in the `OrderMixin` contract. Some functions return named variables, others return explicit values, and others declare a named return variable but override it with an explicit return statement.

Consider adopting a consistent approach to return values throughout the codebase by removing all named return variables, explicitly declaring them as local variables, and adding the necessary return statements where appropriate. This would improve both the explicitness and readability of the code, and it may also help reduce regressions during future code refactors.

Update: *Fixed in pull request #73.*

[N10] Order's hash calculation is not open to the API

The `external` functions `remaining`, `remainingRaw` and `remainingsRaw` all expect an order hash for successful operation.

However, the helper function `_hash`, which returns the hash of an order, has `private` visibility. This means that users will have to pack parts of the orders and domain strings manually in order to obtain the hash of an order.

To avoid the potential for mistakes when calculating order hashes and to provide users with a method for generating an order's respective hash, consider extending the visibility of the `_hash` function to `public` and refactoring the name to `hash` to be consistent with the rest of the code.

Update: *Fixed in pull request #74.*

[N11] Semantic overloading of mapping

The `_remaining` mapping in the `OrderMixin` contract is semantically overloaded to track the status of orders and the remaining amount of assets available for those orders.

The three states that it can take on are:

- `0`: The order hash has not been seen yet.
- `1`: The order has been either canceled or completely filled.
- Any `uint` larger than `1`: The remaining `makerAmount` available to be filled on the order plus `1`.

This semantic overloading requires wrapping and unwrapping of this value during `lookup`, `cancellation`, `initialization`, and `storage`.

Semantic overloading and the necessary logic to enable it can be prone to error and can make the codebase harder to understand and reason about, it may also open the door for regressions in future updates to the code.

To improve the code's readability, consider tracking the completion state of orders in a separate mapping.

Update: *Not fixed. The 1inch team cited that a fix would increase gas costs for the `fillOrder` function.*

[N12] Orders with permit allow calls to arbitrary contracts

The `OrderMixin` contract inherits the `Permitable` contract to allow for single-transaction order filling with assets that accept such `permit` calls to modify allowances.

However, the calls to the `Permitable` contract do not validate whether the target is a permitable asset nor if it is even an asset, which could allow a malicious user to pass the address of an arbitrary contract which could execute another call before the order fill completes.

Although the contract is protected against reentrancy, reducing the attack surface and preventing the calling of external contracts during execution is always recommended. Consider either restricting the asset involved in the permit to the assets

involved in the order or to an assets whitelist for the protocol.

Update: *Not fixed. The 1inch team states:*

> `OrderMixin` does not actually have info about actual tokens as `makerAsset` and `takerAsset` sometimes are proxies or other intermediate contracts and info about actual tokens is stored in some arbitrary bytes. So there is no viable way to restrict which asset `permit` is called on.

[N13] `solhint` is never re-enabled

Throughout the codebase, there are a couple of `solhint-disable` statements, specifically those on line 23 and on line 41 of `RevertReasonParser.sol`, that are not terminated with `solhint-enable`.

In favor of explicitness and to be as restrictive as possible when disabling `solhint`, consider using `solhint-disable-line` or `solhint-disable-next-line` instead, similar to line 16 of the same file.

Update: *Fixed in pull request #72.*

[N14] Typos

The codebase contains the following typos:

- On line 18 of `OrderMixin.sol` and line 11 of `OrderRFQMixin.sol`, `v1` should be `v2`.
- On lines 147, 165, and 188 of `OrderMixin.sol`, `it's` should be `if it's`.

Additionally the project's `README` (out of scope for this audit) contains the following typos:

- `Ket` should be `Key`.
- `stategies` should be `strategies`.
- `cancelation` should be `cancellation`.

Consider correcting these typos to improve code readability.

Update: *Fixed in pull request #71 and pull request #77.*

[N15] Use of `uint` instead of `uint256`

To favor explicitness, all instances of `uint` should be declared as `uint256`. In particular, those in the `for` loops on lines 98 and 119 of `OrderMixin.sol` and lines 16 and 30 of `PredicateHelper.sol`.

Update: *Fixed in pull request #70.*

Conclusions

3 high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.