1inch
Smart Contract Audit

coinspect

Smart Contract Audit
Limit Order Protocol

Prepared for 1inch • April 2021

v210605

1. Executive Summary

In April, 1inch engaged Coinspect to perform a source code review of its new limit-order-protocol smart contract to be added to their exchange platform. The objective of the audit was to evaluate the security of the smart contracts being developed.

The assessment was conducted on contracts from the Git repository at https://github.com/1inch/limit-order-protocol obtained on **April 5th**.

Overall, Coinspect did not find any high risk security vulnerabilities that would result in stolen or lost user funds. The smart contracts reviewed rely on many user defined external function calls, and many validations of market orders are expected to be performed off-chain. As **off-chain components were not in scope for this engagement**, it is not clear how the system will interact with end-users and what validations will be performed or what information will be provided to the users of the system.

The current implementation does guarantee that:

- each order is only taken once or until no more funds remain,
- transfers are successfully executed for both sides of the trade: maker to taker, and taker to maker.

However, if unsuspecting users are lured into taking orders from untrusted sources, they could end up receiving less funds than expected and it is advised this risk is clearly documented. In order to enforce a minimum expected amount to be received for an exchange at the smart contract layer, it is recommended to add a parameter for the taker to specify the minimum amount that is expected.

In June 2021 Coinspect reviewed several modifications that were introduced by 1inch in order to improve the limit-order-protocol smart contracts in response to the suggestions provided in this report. As a result the most important recommendations, including the issue mentioned in the previous paragraph, were correctly addressed as detailed in 5. Remediations.

The following issues were identified during the assessment:

High Risk	Medium Risk	Low Risk
0	0	1

The only low risk finding reported details how a function callback set by order makers could be abused to mine gas tokens (See INC-001).

The following report details the tasks performed and the vulnerabilities found during this audit as well as several suggestions aimed at improving the overall code quality and warnings regarding potential issues.

2. Introduction

1inch is a DEX aggregator that offers its users the best rates by discovering the most efficient swapping routes across all leading DEXes.

The focus of this audit was their new limit orders protocol which is currently being developed .

The audit started on April 5th and was conducted on the Git repository at https://github.com/1inch/limit-order-protocol. The last commit reviewed during this engagement was `fc528b390bad66927b9316470e4c86d06df58563` from **January 6th**:

```
commit fc528b390bad66927b9316470e4c86d06df58563
Author: Anton Bukov <k06aaa@gmail.com>
Date:   Sat Apr 3 00:44:31 2021 +0200

   Remove *NoPartialFill methods and adapt tests
```

The scope of the audit was limited to the latest version of the following Solidity source files, shown here with their sha256sum hash:

```
9b90bb7dd6c9e8d3c56fd3fa5c5b6239ab3bf61b7c2f89b56376917bbc55aee3  ./mocks/TokenMock.sol
1c13c4de9632150402e1787b23aae098258fe81f60af1bbd7376c8c3b3066fe0  ./libraries/UncheckedAddress.sol
8029f3b3d948812747a9e16a0d2f6dead4b04ab9cb46225cafbca6fac8f6812a  ./libraries/ArgumentsDecoder.sol
cba07ee07bce9b1d42c52932cf452204734dd81a2312f57f256805ff896183e8  ./interfaces/InteractiveMaker.sol
48567a2da613e85c64d2a40d0d713d93a0c729adef2907cebfad98ecece20696  ./interfaces/IEIP1271.sol
230703523c2cab650cc5fc73d5b7d18e14fdff236e6083dec182284cb3e9b159  ./LimitOrderProtocol.sol
f2452df1093aeb92f059aa87b0ec07cea52914e4b3e6d859b1c80f2f817930d9  ./helpers/NonceManager.sol
ba906a2647c9c1a9d00cb20bc641bc96b5a4976de5c3d07821e82992ec85db4f  ./helpers/ERC20Proxy.sol
db93a616b51f2dd8947a50318bf0a5500e239ffcaa6577298aa3b44193cad29d  ./helpers/AmountCalculator.sol
80571c0371b018280b1102fae0d20c0ad23d993af461be5e3771f8d836a712d0  ./helpers/PredicateHelper.sol
083f9925682589391ea303fa57f15ea814c88e7384973811e08f2f9182418345  ./helpers/ERC1155Proxy.sol
3d9f2b295316b240adc96f09480c0cd0bfefe140f8a4b8f4765d6ad32e375c64  ./helpers/ERC721Proxy.sol
53b01db5ff8625c4084f748a7223c98259e3e032a3b168a6030dfa370301d0ea  ./helpers/ImmutableOwner.sol
```

These smart contracts interact with multiple other contracts which were not part of this review.

3. Assessment

The limit order protocol is implemented in the `LimitOrderProtocol.sol` smart contract, which is not upgradable.

All contracts are specified to be compiled with an up-to-date Solidity compiler version 0.8.0 at least (`pragma solidity ^0.8.0`). It is recommended to lock the compiler pragma to use a specific compiler version in order to guarantee the contract is deployed with the exact same version which was used to test the contracts during development. There is only one minor compilation warning which is not security relevant.

All the 32 existing tests passed. The coverage at the moment is around 82% and it is recommended that tests are extended in order to increase coverage as much as possible.

No system description, specs, or documentation were provided for the audit. The code includes limited comments and it is recommended for contracts to be fully annotated using NatSpec for all public interfaces. This rich documentation can be provided for functions and their return values and can include developer-focused messages and end-user facing messages that are shown to the end user when they sign a transaction.

`LimitOrderProtocol` implements an off-chain order book exchange which is responsible for executing pre-signed transfer orders after validating certain conditions. Both the order maker and taker need to pre-approve the exchange contract for it to be able to transfer funds from the maker to the taker and vice versa. Alternatively, the maker can provide in the order a signed permit transaction that will be executed by the exchange the first time a fill takes place.

The provided `transferFrom` transaction (or alternative proxy transfer function) is patched by the exchange with the appropriate taker address and amount and executed via an EVM call instruction.

Wrappers are used to call the `transferFrom` transaction provided in the orders, the transaction success is checked, and if a value is returned it is parsed as a boolean and checked to be true. This is performed to make sure the contract works as expected with different implementations of ERC20 tokens. By doing this, the `LimitOrderContract` contract guarantees funds from both sides of the operation were effectively transferred.

The protocol is designed to be as generic as possible, allowing exchanges between any ERC20 compliant token. Orders contain arbitrary external function calls and is the taker's responsibility to fully understand and validate the order being taken.

The exchange contract itself does not hold any funds, it validates the provided orders and executes the exchanges, which are included in the order itself. Its main responsibility is verifying each order is executed only once (or many times when the fill is partial, always stopping when the maximum amount is reached).

The protocol allows orders to be created and signed, and presumably be posted off-chain. Order takers can then pick a signed order from an off-chain source such as a website and initiate a swap.

There will be a service that keeps track of all the orders and updates their status accordingly. This service was not reviewed during this engagement and its exact responsibilities are unknown.

All public functions are accessible by anybody. The only functions with access control modifiers are the `ERC*Proxy` functions (described later on) that are intended to be invoked by the `LimitOrderProtocol` contract itself.

The exchange main entry points are the following 2 functions invoked by order takers:

1. `function fillOrderRFQ(OrderRFQ memory order, bytes memory signature) external {`

2. `function fillOrder(Order memory order, bytes calldata signature, uint256 makingAmount, uint256 takingAmount) external returns(uint256, uint256)`

These functions receive a maker order with its corresponding maker signature.

The off-chain handling of orders: how they are presented to takers, how they are indexed, how potentially unexecutable/fake orders are filtered, etc, were not in scope for this engagement so they were not reviewed.

The system handles two types of orders:

1. *RFQ orders*: simpler processing performed by `fillOrderRFQ()`, they have an expiration timestamp that can be invalidated by the maker or get invalidated once filled, only full fills are possible.
2. *Limit Market Orders*: these orders incorporate more features. For example: they include a permit transaction (to be called the first time `fillorder()` is called with the order), they allow partial fills, and they include an interactive callback to the maker that is called when the order is taken.

Also, private orders are possible, where the taker can only be the one specified in the order by the maker at creation time.

EIP1271 (Standard Signature Validation Method for Contracts) is supported. If the account that signed the maker order, as obtained with ecrecover, does not match the order maker address, the maker's `isValidSignature` function is called in order to validate the transfer. This enables the creation of orders that move funds stored in smart contracts such as wallets, that are not able to sign the orders themselves. The `isValidSignature` is called statically, which prevents gas mining and similar attack vectors.

Maker orders can include "predicates" that are evaluated before the order is executed, these predicates are executed statically so the state is not affected, preventing gas mining and similar attack vectors.

Potential griefing and/or denial of service threat scenarios exist where malicious makers could introduce unfillable orders into the order book. Unfillable orders can be created in many ways: failing signature verification when `isValidSignature` is called, consuming all available gas during the maker interactive notification, lack of balance or allowance and orders being cancelled). As a consequence, griefed takers that try to fill the order will waste gas and will have problems finding real orders. These issues are, by design, handled off-chain by the 1inch order aggregator service. Loopholes in the validations performed by this service would result in this scenario being exploitable and takers being griefed.

The 1inch team explained users can also verify fillability of the order via the two helper functions provided in the exchange contract: `simulateTransferFroms` and `checkPredicate`. The former allows checking whether the maker has enough funds and approvals while the latter allows checking the predicate value.

This is a list of external calls that are performed during the order filling process flow, this calls are performed to addresses and with call data provided by the creator of the limit order:

1. `permit()`
2. `notifyFillOrder()` interactive maker call
3. `transferFrom()` twice, once for each token involved in the swap
4. `_validate()` static call to `isValidSignature()`
5. `_callGetMakerAmount()` / `_callGetTakerAmount()` static calls

Because the `notifyFillOrder` is not static in order to allow the maker to handle the funds interactively, it could be abused in order to mine gas tokens and similar attack vectors. It is recommended to consider making this call static or limiting the amount of gas passed. This is detailed in the INC-001 nofityFillOrder gas mining finding.

Orders provide a maker defined function to calculate the amount taken from the maker or taker, depending if the `fillOrder` function invocation specifies the amount to be taken or provided (only one of the parameters makingAmount or takingAmount can be different to 0). If the full value in the order is requested, there is no calculation: the amount specified in the order is used for the other side. But, if the fill is not complete, the provided callback is invoked. The callback transaction includes the maker and taker amount for each token involved, nevertheless there is no check to verify the amounts encoded in the callback are the same as the values in `makerAssetData` and `takerAssetData`. The default implementations for these calculations round the result in favor of the maker. However, the maker can create an order with custom `order.getTakerAmount` and/or `order.getMakerAmount` fields. These callbacks could be set up in a way where the amount sent to the taker is the minimum (it can not be 0) instead of the expected value.

For example, this is how an order is created in the test provided:

```
    function buildOrderWithSalt (exchange, salt, makerAsset, takerAsset, makerAmount, takerAmount,
taker = zeroAddress, predicate = '0x', permit = '0x', interaction = '0x') {
       return {
           salt: salt,
           makerAsset: makerAsset.address,
           takerAsset: takerAsset.address,
           makerAssetData:
                makerAsset.contract.methods.transferFrom(wallet, taker, makerAmount).encodeABI(),
           takerAssetData:
                takerAsset.contract.methods.transferFrom(taker, wallet, takerAmount).encodeABI(),
           getMakerAmount:
                exchange.contract.methods.getMakerAmount(makerAmount,
                                takerAmount, 0).encodeABI().substr(0, 2 + 68 * 2),
           getTakerAmount:
                exchange.contract.methods.getTakerAmount(makerAmount,
                                takerAmount, 0).encodeABI().substr(0, 2 + 68 * 2),
           predicate: predicate,
           permit: permit,
           interaction: interaction,
       };
   }
```

Again, the taker is responsible for verifying how the order is constructed, and checking if the default implementation is used or not, and if the correct parameters are passed. An evil order could be created that for example, pays 1 token in exchange for any mount sent by the taker.

Regarding this issue, 1inch responded that orders will be presented to users though their own service, with all their fields. Also, 1inch plans to provide an SDK to parse and check the predicate values as well as `getMakerAmount` and `getTakerAmount` functions.

There is still potential for takers being lured into taking orders from alternative sources. This risk should be clearly documented, warning users that non-validated orders from untrusted sources could result in lost funds.

One way to remove the need to trust and/or fully understand the maker order would be allowing the taker to specify the minimum amount expected in exchange for the swap.

The `AmountCalculator.sol` smart contract (from which the `LimitOrderProtocol` smart contract inherits) allows an arbitrary static call via the `arbitraryStaticCall` function:

```
function arbitraryStaticCall(address target, bytes memory data) external view returns(uint256) {
        (bytes memory result) =
                target.uncheckedFunctionStaticCall(data, "AC: arbitraryStaticCall");
       return abi.decode(result, (uint256));
   }
```

and could be utilized to implement a custom `getMakerAmount`. This function is not used anywhere in the project and should be removed unless needed.

Regarding the calls used to transfer funds, only calls using selectors between `IERC20.transferFrom.selector` and this value plus 10 are allowed. This is intended to be used with the provided proxy contracts, which implement functions with precalculated function hashes in that range:

1. `ERC20Proxy`
2. `ERC7211Proxy`
3. `ERC1155Proxy`

For example, this is the proxy function in the `ERC20Proxy` contract, which is passed an ERC20 token address as a parameters and executes a `safeTransferFrom`:

```
   // keccak256("func_50BkM4K(address,address,uint256,address)") = 0x23b872de
   function func_50BkM4K(address from, address to, uint256 amount, IERC20 token) external
        onlyImmutableOwner {
       token.safeTransferFrom(from, to, amount);
   }
```

The `LimitOrderProtocol` contract inherits from these 3 contracts and only the contract itself is allowed to invoke these proxy functions. This feature can be used to handle swaps of different types of tokens by pointing either `makerAssetData` or `takerAssetData` in the order.

4. Recommendations

The following list sums up the most important suggestions provided in this report:

1. Add a parameter for the taker to specify the minimum amount expected to be received for the exchange. Remove the need to trust and/or fully understand the maker order by allowing the taker to specify the minimum amount expected in exchange for the swap.
2. Clearly document the risks of taking non-validated orders from untrusted sources.
3. Limit the amount of gas that is passed to `InteractiveMaker.notifyFillOrder` or make the call static if possible.
4. Improve tests/coverage (events not tested, Permit functionality not tested, custom `getTakeAmount` not tested, etc).
5. Remove arbitrary calls in `AmountCalculator` if not needed.
6. Add inline NatSpec documentation to every function.
7. Lock Solidity compiler version pragma to use a specific version of the compiler.

5. Remediations

The 1inch team introduced improvements in response to the most important recommendations provided in this report. These were reviewed in June 2021 and they are detailed below.

Regarding Coinspect's *recommendation 1*, it was addressed by PR#14. The new `thresholdAmount` argument was introduced in the `fillOrder` function and this effectively protects makers and takers from being front-runned or abused by malicious orders. This was later modified in PR#13 which switched to a `minPrice` argument instead of `thresholdAmount` with the same purpose and result as before. This pull request also enables partial fills for RFQ orders by adding the `makingAmount` and `takingAmount` parameters to the `fillOrderRFQ` function.

The `test/ChainlinkExample.js` test introduced in PR#16 provides an example of how the `arbitraryStaticCall` is intended to be used (it was unused before) and renders *recommendation 5* not applicable anymore. This pull request also removed the optimization that bypassed the call to `_callGetTakerAmount` and `_callGetMakerAmount` functions when the full order was being filled; these functions are always called now instead.

6. Summary of Findings

ID	Description	Risk	Fixed
INC-001	notifyFillOrder abuse gas mining	Low	✘

7. Findings

INC-001 notifyFillOrder abuse gas mining

Total Risk	Impact	Location
Low	Low	LimitOrderProtocol.sol
Fixed	Likelihood	
✘	Low	

Description

The `nofiyFillOrder` callback could be abused in order to mine gas tokens with all the gas supplied to the `fillOrder` function.

The `fillOrder` function passes all the available gas to the interactive notification function set by the maker in the order:

```
// Maker can handle funds interactively
if (order.interaction.length > 0) {
    InteractiveMaker(order.makerAssetData.decodeAddress(0))
        .notifyFillOrder(order.makerAsset, order.takerAsset, makingAmount,
                         takingAmount, order.interaction);
```

Even though static calls are used for most external calls, the `notifyFillOrder` callback is not static in order to allow the maker to handle the funds interactively.

Takers could limit the gas supplied to a sensible amount. However, it could be hard for them to know what constitutes a reasonable amount to be consumed by the interactive maker notification callback.

Recommendation

If possible, make the maker notifier callback function static. Alternatively, restrict the amount of gas it gets passed.

8. Disclaimer

The information presented in this document is provided "as is" and without warranty. Source code reviews are a “point in time” analysis and as such it is possible that something in the code could have changed since the tasks reflected in this report were executed. This report should not be considered a perfect representation of the risks threatening the analyzed system.