1inch Exchange AggregationRouter V5

Date	August 2022
Auditors	George Kobakhidze, Chingiz Mardanov

1 Executive Summary

This report presents the results of our engagement with **1inch** to review **AggregationRouter V5**.

The review was conducted over two weeks, from **August 15 2022** to **September 16 2022** by **Chingiz Mardanov** and **George Kobakhidze**. A total of 40 person-days were spent.

2 Scope

Our review focused on the following repositories and commit hashes:

- https://github.com/1inch/1inch-contract @ 3461c75d00481d111f0323d7e1cb6e56b0dc7ec0,
- https://github.com/1inch/limit-order-protocol @ 403be0f583b863b3b13373c23e8f2652f5ef720d,
- https://github.com/1inch/solidity-utils @ 32b04c767db00d4ddc2428534fd6e90abbacb12e.

The list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the **1inch** team, we identified the following priorities for our review:

1. Ensure that the system is implemented consistently with the intended functionality, and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

General comments

Security Properties

1inch team excelled at writing fast and efficient code that will minimize the gas cost for the end users of the protocol. A lot of code was written in Yul and readability of the code fell victim as a result. This could eventually lead to code that is hard to audit, jeopardizing the security of the protocol. We would also like to suggest increasing the amount of technical comments and documentation available that would explain reasoning behind certain decisions saving code readers time, which could be both future auditors, other systems integrating 1inch into their protocols, and even new hires for the 1inch team itself.

Limit Order Protocol V3

Actors

The relevant actors are listed below with their respective abilities:

- **Maker** - this is the actor responsible for placing orders. One could otherwise refer to makers as sellers. Makers typically will format an `Order` object, sign it and submit it to the 1inch backend server.
- **Taker** - this is the actor responsible for filling orders. Otherwise referred to as buyer. Takers find the attractive order via 1inch services and then submit the fill transaction on chain, thus paying gas for the entire exchange.
- **1inch Team** - the primary method of finding and communicating order hashes and their signatures between makers and takers happens through the 1inch system. That system would allow to post information to it, and then, similarly, it would let users read information from it, either through a front-end or an API call.

Trust Model

A lot of the infrastructure of 1inch Limit Order Protocol V3 is actually off chain. Both makers and takers will rely on the SDK to place and fill orders to the 1inch system. In addition to that, a lot of the code depends on providing correct arguments to the deployed contracts as well as generating correct ids. With increasing optimization, the complexity of parameters that are being passed is concerning. Such complexity might eventually lead to a mistake in off-chain components that will get propagated into the contracts. Users of 1inch must trust that the off-chain infrastructure is reliable, operates correctly, and provides the correct data up to the specification declared by the 1inch team and their smart contracts. Due to this, the 1inch team is able to partially censor, block, front-run and generally do whatever they want with access to the orders. While it is true that users can share orders and signatures in a P2P manner, and this service done by the 1inch team is needed and acts as a public good since the creation of orders is not on-chain and effectively gasless, this still introduces a centralization and a reliance vector on the 1inch team's systems, both in terms of trust and availability. None of the off-chain infrastructure was part of this engagement.

4 1inch Contract

Each issue has an assigned severity:

- `Minor` issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- `Medium` issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- `Major` 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.
- `Critical` issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 _WETH private constant address in UnoswapRouter makes for tricky deployments to other chains `Minor`

Resolution

The 1inch team acknowledged but is unable to implement changes due to stack-too-deep errors that would require a large refactor of this already well-used and tested library.

To interact with WETH, the `UnoswapRouter` uses a hardcoded `_WETH` private constant in the contract. Therefore, this currently needs changing every time the contract is deployed to a different chain, as noted by the comment within the contract:

1inch-contract/contracts/routers/UnoswapRouter.sol:L24-L26

```
/// @dev WETH address is network-specific and needs to be changed before deployment.
/// It can not be moved to immutable as immutables are not supported in assembly
address private constant _WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
```

As the comment also points out, the choice to not make it an immutable variable is not possible since they are not supported in assembly, and the `UnoswapRouter` contract is highly efficient and almost entirely written in assembly. However, the other contracts within the scope of this audit, do utilize setting a `private immutable` variable for WETH in the constructor, and some of them then initialize a new `address` variable derived from this `private immutable` variable, thereby allowing the `address` variable to be used in the assembly blocks instead:

UnoswapV3Router

1inch-contract/contracts/routers/UnoswapV3Router.sol:L33-L37

```
IWETH private immutable _WETH;  // solhint-disable-line var-name-mixedcase

constructor(IWETH weth) {
    _WETH = weth;
}
```

ClipperRouter

1inch-contract/contracts/routers/ClipperRouter.sol:L18-L24

```
IWETH private immutable _WETH;  // solhint-disable-line var-name-mixedcase
IClipperExchangeInterface private immutable _clipperExchange;

constructor(IWETH weth, IClipperExchangeInterface clipperExchange) {
    _clipperExchange = clipperExchange;
    _WETH = weth;
}
```

1inch-contract/contracts/routers/ClipperRouter.sol:L101

```
address weth = address(_WETH);
```

1inch-contract/contracts/routers/ClipperRouter.sol:L112

```
if iszero(call(gas(), weth, 0, ptr, 0x64, 0, 0)) {
```

OrderMixin

limit-order-protocol/contracts/OrderMixin.sol:L63-L70

```
IWETH private immutable _WETH;  // solhint-disable-line var-name-mixedcase
/// @notice Stores unfilled amounts for each order plus one.
/// Therefore 0 means order doesn't exist and 1 means order was filled
mapping(bytes32 => uint256) private _remaining;

constructor(IWETH weth) {
    _WETH = weth;
}
```

Normalizing this process across all smart contracts in the 1inch system could help avoid accidental mistakes when the deployer could forget to first edit the unoswap contract to have the correct address.

4.2 Selfdestruct may be removed as an opcode in future Minor

Resolution

The 1inch team acknowledged and noted.

The `AggregationRouterV5` contract implements a function called `destroy` that calls a `selfdestuct` on the contract with the `msg.sender` as the argument, that is checked by the `onlyOwner` modifier on the function.

1inch-contract/contracts/AggregationRouterV5.sol:L35-L37

```
function destroy() external onlyOwner {
    selfdestruct(payable(msg.sender));
}
```

However, there are discussions currently around removing the `selfdestruct` functionality from the EVM altogether with various motivations and rationale provided, such as this being not possible with Verkle trees and it being a requirement for stateless. Link to the EIP is below: https://eips.ethereum.org/EIPS/eip-4758

It appears that the suggested remediation of this functionality per the EIP-4758 will not significantly change the results, for example all of the funds will still be sent to the specified address, but the destruction of the actual contract will not occur. So this is just an advisory note for the 1inch team to notify of this potential change in the future.

5 Limit Order Protocol

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major 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.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 Malicious maker can take more takers funds than taker expected Major

Resolution

Remediated as per the 1inch team in 1inch/limit-order-protocol@ `9ddc086` by adding a check that reverts when `actualTakingAmount > takingAmount`

`OrderMixin` contract allows users to match makers(sellers) and takers(buyers) in an orderbook-like manner. One additional feature this contract has is that both makers and takers are allowed to integrate hooks into their orders to better react to market conditions and manage funds on the fly. Two out of many of these hooks are called: `_getMakingAmount` and `_getTakingAmount`. Those particular hooks allow the maker to dynamically respond to the making or taking amounts supplied by the taker. Essentially they allow overriding the rate that was initially set by the maker when creating an order up to a certain extent. To make sure that the newly suggested maker rate is reasonable taker also provides a `threshold` value or in other words the minimum amount of assets the taker is going to be fine receiving.

Generally speaking, the maker can override the taking amount offered to the taker if the buyer passed a specific making amount in the fill transaction and vice versa. But there is one special case where the maker will be able to override both, which when done right will force the taker to spend an amount larger than the one intended. Specifically, this happens when the taker passed the desired taking amount and the maker returns a suggested making amount that is larger than the remaining order size. In this case, the making amount is being set to the remaining order amount and the taking is being recomputed.

limit-order-protocol/contracts/OrderMixin.sol:L214-L217

```
actualMakingAmount = _getMakingAmount(order.getMakingAmount(), order.takingAmount, actualTakingAmount, order.makingAmount, remai
if (actualMakingAmount > remainingMakingAmount) {
    actualMakingAmount = remainingMakingAmount;
    actualTakingAmount = _getTakingAmount(order.getTakingAmount(), order.makingAmount, actualMakingAmount, order.takingAmount, r
```

Essentially this allows the maker to override the taker amount and as long as the maker keeps the price intact or changed within a certain threshold like described in this issue, they can take all taking tokens of the buyer up to an amount of the token balance or approval limit whatever comes first.

Consider the following example scenario:

- The maker has a large order to sell 100 ETH on the order book for 100 DAI each.
- The taker then wants to partially fill the order and buy as much ETH as 100 DAI will buy. At the same time taker has 100,000 DAI in the wallet.

When taker tries to fill this order taker passes the `takingAmount` to be 100. Since OrderMixin received the taking amount we go this route:

limit-order-protocol/contracts/OrderMixin.sol:L214

```
actualMakingAmount = _getMakingAmount(order.getMakingAmount(), order.takingAmount, actualTakingAmount, order.makingAmount, remai
```

However, note that when executing the `_getMakingAmount()` function, it first evaluates the `order.getMakingAmount()` argument which is evaluated as `bytes calldata _getter` within the function.

limit-order-protocol/contracts/OrderMixin.sol:L324-L337

```
function _getMakingAmount(
    bytes calldata getter,
    uint256 orderTakingAmount,
    uint256 requestedTakingAmount,
    uint256 orderMakingAmount,
    uint256 remainingMakingAmount,
    bytes32 orderHash
) private view returns(uint256) {
    if (getter.length == 0) {
        // Linear proportion
        return getMakingAmount(orderMakingAmount, orderTakingAmount, requestedTakingAmount);
    }
    return _callGetter(getter, orderTakingAmount, requestedTakingAmount, orderMakingAmount, remainingMakingAmount, orderHash);
}
```

That is because the `Order` struct that is made and signed by the maker actually contains the necessary bytes within it that can be decoded to construct a target and calldata for static calls, which in this case are supposed to be used to return the making asset amounts that the maker determines to be appropriate, as seen in the comments under the `uint256 offsets` part of the struct.

limit-order-protocol/contracts/OrderLib.sol:L7-L27

```
library OrderLib {
    struct Order {
        uint256 salt;
        address makerAsset;
        address takerAsset;
        address maker;
        address receiver;
        address allowedSender;  // equals to Zero address on public orders
        uint256 makingAmount;
        uint256 takingAmount;
        uint256 offsets;
        // bytes makerAssetData;
        // bytes takerAssetData;
        // bytes getMakingAmount; // this.staticcall(abi.encodePacked(bytes, swapTakerAmount)) => (swapMakerAmount)
        // bytes getTakingAmount; // this.staticcall(abi.encodePacked(bytes, swapMakerAmount)) => (swapTakerAmount)
        // bytes predicate;       // this.staticcall(bytes) => (bool)
        // bytes permit;          // On first fill: permit.1.call(abi.encodePacked(permit.selector, permit.2))
        // bytes preInteraction;
        // bytes postInteraction;
        bytes interactions; // concat(makerAssetData, takerAssetData, getMakingAmount, getTakingAmount, predicate, permit, preInte
    }
```

Finally, if these bytes indeed contain data (i.e. length>0), they are passed to the `_callGetter()` function that asks the previously mentioned target for the data.

limit-order-protocol/contracts/OrderMixin.sol:L354-L377

```
    function _callGetter(
        bytes calldata getter,
        uint256 orderExpectedAmount,
        uint256 requestedAmount,
        uint256 orderResultAmount,
        uint256 remainingMakingAmount,
        bytes32 orderHash
    ) private view returns(uint256) {
        if (getter.length == 1) {
            if (OrderLib.getterIsFrozen(getter)) {
                // On "x" getter calldata only exact amount is allowed
                if (requestedAmount != orderExpectedAmount) revert WrongAmount();
                return orderResultAmount;
            } else {
                revert WrongGetter();
            }
        } else {
            (address target, bytes calldata data) = getter.decodeTargetAndCalldata();
            (bool success, bytes memory result) = target.staticcall(abi.encodePacked(data, requestedAmount, remainingMakingAmoun
            if (!success || result.length != 32) revert GetAmountCallFailed();
            return abi.decode(result, (uint256));
        }
    }
}
```

However, since the getter is set in the `Order` struct, and the `Order` is set by the maker, the getter itself is entirely under the maker's control and can return whatever the maker wants, with no regard for the taker's passed `actualTakingAmount` or any arguments at all for that matter. So, in our example, the return value could be 100.1 ETH, i.e. just above the total order size. That will get us on the route of recomputing the taking amount since 100.1 is over the 100ETH remaining in the order.

limit-order-protocol/contracts/OrderMixin.sol:L217

```
actualTakingAmount = _getTakingAmount(order.getTakingAmount(), order.makingAmount, actualMakingAmount, order.takingAmount, remai
```

This branch will set the `actualMakingAmount` to 100ETH and then the malicious maker will say the `actualTakingAmount` is 10000 DAI, this can be done via the `_getTakingAmount` static call in the same exact way as the making amount was manipulated.

Then the threshold check would look like this as defined by its formula:

limit-order-protocol/contracts/OrderMixin.sol:L222

```
if (actualMakingAmount * takingAmount < thresholdAmount * actualTakingAmount) revert MakingAmountTooLow();
```

- ActualMakingAmount - 100ETH
- ActualTakingAmount - 10000DAI
- threshold - 1 ETH
- takingAmount - 100 DAI

then: `100ETH * 100DAI < 1ETH*10000DAI` This condition will be false so we will pass this check.

Then we proceed to taker interaction. Assuming the taker did not pass any interaction, the `actualTakingAmount` will not change.

Then we proceed to exchange tokens between maker and taker in the amount of `actualTakingAmount` and `actualMakingAmount`.

The scenario allows the maker to take the taker's funds up to an amount of taker's approval or balance. Essentially while taker wanted to only spend 100 DAI, potentially they ended up spending much more. This paired with infinite approvals that are currently enabled on the 1inch UI could lead to funds being lost.

While this does not introduce a price discrepancy this attack can be profitable to the malicious actor. The attacker could put an order to sell a large amount of new not trustworthy tokens for sale who's supply the attacker controls. Then after a short marketing campaign when people will cautiously try to buy a small amount of those tokens for let's say a small amount of USDC due to this bug attacker could drain all of their USDC.

We advise that 1inch team treats this issue with extra care since a similar issue is present in a currently deployed production version of 1inch OrderMixin. One potential solution to this bug is introducing a global threshold that would represent by how much the actual taking amount can differ from the taker provided taking amount.

5.2 Invalidating users orders Medium

1inch team has implemented a more streamlined version of the order book that is called `OrderRFQMixin`. This version has no hooks and is meant to be more straightforward than the main order book contract.

One significant difference between those contracts is that the RFQ version invalidates the orders even after they have been only partially filled.

limit-order-protocol/contracts/OrderRFQMixin.sol:L197-L203

```
{  // Stack too deep
    uint256 info = order.info;
    // Check time expiration
    uint256 expiration = uint128(info) >> 64;
    if (expiration != 0 && block.timestamp > expiration) revert OrderExpired(); // solhint-disable-line not-rely-on-time
    _invalidateOrder(maker, info, 0);
}
```

Since makers have to sign the orders, only makers can place the remainder of the original order as a new one. Given that information, an attacker could take all the orders and fill them with 1 wei of taking assets. While this will cost an attacker gas, on some chains it would be possible to make the operations of the protocol unreliable and impractical for makers.

One way to fix that without making significant changes to the logic is to introduce a threshold that will determine the smallest taking amount for each order. That could be a percent of the taking amount specified in the order. This change will make the attack more expensive and less likely to happen.

5.3 Reentrancy potential issue for contracts building on top RFQOrderMixin Minor

Resolution

Remediated as per the 1inch team in 1inch/limit-order-protocol@`d3957fe` by forwarding a limited amount of gas to guard against complex execution at the target but still allow for smart contract receipt that may require a bit more gas than usual EOA receipts.

The `RFQOrderMixin` contract is used to facilitate transfer of assets in RFQ orders between makers and takers. Naturally, one of such possible assets could be the native coin of the chain, such as ETH. In order to perform these transfers, the contract currently utilizes the `target.call(){value:X}` method to transfer X ETH to the target address. However, this also calls into the target address and opens up arbitrary code execution that could lead to significant problems, that often times result in a reentrancy attack.

limit-order-protocol/contracts/OrderRFQMixin.sol:L233

```
(bool success, ) = target.call{value: makingAmount}("");  // solhint-disable-line avoid-low-level-calls
```

While 1inch's RFQOrderMixin contract doesn't have a clear reentrancy attack vector, other smart contract systems that might utilize 1inch RFQ orders will have to handle a potential reentrancy due to this problem. The impact for downstream systems could be critical.

This could be changed to `.transfer()` or `.send()` methods of transferring ETH, or at least heavily noted in documentation for any and all developers who may fork/utilize this code so reentrancy risks are made aware of.

This does not seem to be a general-purpose use library for other systems, so likelihood of this issue happening isn't as high as in issue 6.4, so the severity is lower.

5.4 Order cancellation event spam in orderMixin Minor

The 1inch Limit Order protocol's contracts utilize mechanisms to allow creation of orders without posting the orders on chain. Indeed, the orders are created by signing `Order` struct hashes off-chain by the maker, and then having takers pass the signatures associated with those hashes to fill in those orders. However, a maker needs to be able to cancel their order if they change their mind, which would require them to execute an on-chain transaction marking that order hash as invalid:

limit-order-protocol/contracts/OrderMixin.sol:L113-L121

```
function cancelOrder(OrderLib.Order calldata order) external returns(uint256 orderRemaining, bytes32 orderHash) {
    if (order.maker != msg.sender) revert AccessDenied();

    orderHash = hashOrder(order);
    orderRemaining = _remaining[orderHash];
    if (orderRemaining == _ORDER_FILLED) revert AlreadyFilled();
    emit OrderCanceled(msg.sender, orderHash, orderRemaining);
    _remaining[orderHash] = _ORDER_FILLED;
}
```

Unfortunately, since the OrderMixin contract is not aware of order hashes before interacting with them for the first time, it can not verify that the order was actually ever seriously present or intended to be executed. As a result, this would allow users to cancel non-existent orders and create event spam. While this would be costly to the spammer, it would nonetheless be possible.

The impact of this would need systems that rely on the `OrderCanceled` event log to be aware of potential spam attacks with fake order cancellation and not use them, for example, for analytics, potential volume forecasting, tracking order created -> order cancelled metrics and so on.

5.5 Order book slippage Minor

`OrderMixin` contract allows users to match makers and takers in an orderbook-like manner. One additional feature this contract has is that both makers and takers are allowed to integrate hooks into their orders to better react to market conditions and manage funds on the fly. Two of these hooks are called: `_getMakingAmount` and `_getTakingAmount` .

When trying to fill an order taker is required to provide either the making amount or taking amount as well as the threshold or in other words the minimum amount of assets the taker is going to be fine receiving. During the fill transaction, an order maker is given the opportunity to update the offer by the means of the `_getMakingAmount` and `_getTakingAmount` . A threshold checks are then used in order to make sure that the updated values are within taker's acceptable bounds:

limit-order-protocol/contracts/OrderMixin.sol:L222

```
if (actualMakingAmount * takingAmount < thresholdAmount * actualTakingAmount) revert MakingAmountTooLow();
```

limit-order-protocol/contracts/OrderMixin.sol:L212

```
if (actualTakingAmount * makingAmount > thresholdAmount * actualMakingAmount) revert TakingAmountTooHigh();
```

It is reasonable to assume that if the maker knows the threshold the taker selected, the maker will attempt to update the making or taking amount to maximize profits. While `_getMakingAmount` and `_getTakingAmount` do not pass the threshold selected by the taker directly, it is still possible for the maker to obtain this information and act accordingly.

1. A malicious maker could listen to the mempool and wait for a transaction that is meant to fill his order obtaining the threshold value.
2. Maker would then update the state of the contract that responds to the static call of the `_getMakingAmount` and `_getTakingAmount` hooks.

If the maker is using FlashBots or a similar service, the maker can ensure that the above actions are performed before the transaction that would fill the order.

While there is no good way to alleviate this issue given the current design we believe it is important to be aware of this issue and allow the 1inch users to know that some analogy of slippage is still possible even on the orderbook-like system. This will allow them to choose tighter and more secure threshold values.

6 Solidity Utils

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major 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.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

6.1 ECDSA library has a vulnerability for signature malleability of EIP-2098 compact signatures Major

Resolution

Remediated as per the 1inch team in 1inch/solidity-utils@ `166353b` by adding a warning note in the comments of the library code.

The 1inch ECDSA library supports several types of signatures and forms in which they could be provided. However, for compact signatures there is a recently found malleability attack vector. Specifically, the issue arises when contracts use transaction replay protection through signature uniqueness (i.e. by marking it as used). While this may not be the case in the scope of other contracts of this audit, this ECDSA library is meant to be a general use library so it should be fixed so as to not mislead others who might use this.

For more details and context, find below the advisory notice and fix in the OpenZeppelin's ECDSA library:
https://github.com/OpenZeppelin/openzeppelin-contracts/security/advisories/GHSA-4h98-2769-gh6h
OpenZeppelin/openzeppelin-contracts@ `d693d89`

6.2 Ethereum reimbursements sent to an incorrect address Medium

Resolution

Remediated as per the 1inch team as of 1inch/solidity-utils@ `6b1a3df` by adding the correct recipient of the refund.

1inch team has written a library called `UniERC20` that extends the traditional ERC20 standard to also support eth transfers seamlessly. In the case of the `uniTransferFrom` function call, the library checks that the `msg.value` of the transaction is bigger or equal to the amount passed in the function argument. If the `msg.value` is larger than the amount required, the difference, or extra funds, should be sent to the sender. In the actual implementation Instead of returning the funds to the sender, extra funds are actually sent to the destination.

solidity-utils/contracts/libraries/UniERC20.sol:L59-L65

```
if (msg.value > amount) {
    // Return remainder if exist
    unchecked {
        (bool success, ) = to.call{value: msg.value - amount}("");  // solhint-disable-line avoid-low-level-calls
        if (!success) revert ETHSendFailed();
    }
}
```

Given that this code is packed as a library and allows for easy reusability by the 1inch team and outside developers it is crucial that this logic is written well and well tested.

We recommend reconsidering reimbursing the sender when an incorrect amount is being sent because it introduces an easy-to-oversee reentrancy backdoor with `call()` that is mentioned in issue 6.4. Reverting was a default behavior in similar cases across the rest of the 1inch contracts.

If this functionality is required, a fix we could recommend is replacing the `to` with `from`. We can also suggest running a fuzzing campaign against this library.

6.3 ECDSA incorrect size provided for calldata in the static call Medium

Resolution

Remediated as per the 1inch team in 1inch/solidity-utils@ `cfdc889` by passing the correct data size.

The ECDSA library implements support for IERC1271 interfaces that verify provided signature for the data through the different `isValidSignature` functions that depend on the type of signature used.

However, the library passes an incorrect size for the calldata in the static call for signatures that are of the form `(bytes32 r, bytes32 vs)`. It should be 0xa4 (164 bytes) instead of 0xa5 (165 bytes).

solidity-utils/contracts/libraries/ECDSA.sol:L178

```
if staticcall(gas(), signer, ptr, 0xa5, 0, 0x20) {
```

The impact could vary and depends on the signature verifier. For example, it could be significant if the signature verifier performs a check on the calldatasize for this specific type of signature and reverts on incorrect sizes, thereby having valid signatures return `false` when passed to `isValidSignature`.

6.4 Re-entrancy risk in UniERC20 Medium

Resolution

Remediated as per the 1inch team in 1inch/solidity-utils@ `6b1a3df` by forwarding a limited amount of gas to guard against complex execution at the target but still allow for smart contract receipt that may require a bit more gas than usual EOA receipts.

UniERC20 is a general library for facilitating transfers of any ERC20 or native coin assets. It features gas-efficient code and could be easily integrated into large systems of contract, such as those that are used in this audit – 1inch routers and limit order protocol.

However, it also utilizes `.call(){value:X}` method of transferring chain native assets, such as ETH. This introduces a large risk in the form of re-entrancy attacks, so any system implementing this library would have to handle them. While 1inch's projects in the scope of this audit do not seem to have re-entrancy attack vectors, other projects that could be utilizing this library might. Since this is an especially efficient and convenient library, the likelihood that some other project using this suffers and then sufferring a re-entrancy attack is significant.

solidity-utils/contracts/libraries/UniERC20.sol:L45

```
(bool success, ) = to.call{value: amount}("");  // solhint-disable-line avoid-low-level-calls
```

solidity-utils/contracts/libraries/UniERC20.sol:L62

```
(bool success, ) = to.call{value: msg.value - amount}("");  // solhint-disable-line avoid-low-level-calls
```

Consider instead implementing `transfer()` or `send()` methods for transferring chain native assets, such as ETH, instead of performing a `.call()`

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
1inch-contract/contracts/AggregationRouterV5.sol	24f38359ed47453a23f887a161476c881a1dca24
1inch-contract/contracts/routers/ClipperRouter.sol	5a5bf312439739b51ee3e85a13f9073f2df9580e
1inch-contract/contracts/routers/GenericRouter.sol	8c7b36d771ec3a90d4626af5688c0ee9ba81d848
1inch-contract/contracts/routers/UnoswapRouter.sol	29fd870637291dc1b1b780f218ebba479711148c
1inch-contract/contracts/routers/UnoswapV3Router.sol	a50928ce7e7240b034c0b8ed65b718af5201a253
1inch-contract/contracts/interfaces/IAggregationExecutor.sol	af07897cc153e2f314ff76d3fa6d56054cd5bf06
1inch-contract/contracts/interfaces/IClipperExchangeInterface.sol	4f2c8506ff588d5f01089b8bffe70221174988fb
1inch-contract/contracts/interfaces/IUniswapV3Pool.sol	daf1644f03d51181fab1370d2a814b0e0ed5cd25
1inch-contract/contracts/interfaces/IUniswapV3SwapCallback.sol	f8691ae2136e8f0cd6968e55dba591a484ef71fc
1inch-contract/contracts/helpers/Errors.sol	e4cc3321a15bba97fa05bbcfcafc30500b39a6f4
limit-order-protocol/contracts/OrderLib.sol	2ed82e9c40614b92ea172f0858f8a17578209a17
limit-order-protocol/contracts/OrderMixin.sol	99a3f474b74d4594cc30a84d6dbfb663e3e1d1f8
limit-order-protocol/contracts/OrderRFQLib.sol	ff36138f4e5858b60f09624f75f54421692bacff
limit-order-protocol/contracts/OrderRFQMixin.sol	bd114c6abb67a6a06693fad13d7118f02b2eb819
limit-order-protocol/contracts/helpers/AmountCalculator.sol	cc02dc63e915bb01f2aececebaafee61de20e614
limit-order-protocol/contracts/helpers/NonceManager.sol	0fd6ae05850f817d932eaf61384727ccdc5a8d7c
limit-order-protocol/contracts/helpers/PredicateHelper.sol	633ef15d62bd8193de3aa065bea55803254aa8e6
limit-order-protocol/contracts/interfaces/IOrderMixin.sol	a4129477d3e88c7257229885d567f18659eb863e
limit-order-protocol/contracts/interfaces/NotificationReceiver.sol	b52b5f9690602759832e0f70eb376263b9e10c96
limit-order-protocol/contracts/libraries/ArgumentsDecoder.sol	3a9fef97be3af820e2950239b2a43dcac740622f
limit-order-protocol/contracts/libraries/Errors.sol	300b7c8c3ae29194260e0062635ebcbb01d0de0a
solidity-utils/contracts/EthReceiver.sol	93e2b054ef5f8839a769ef0ce7f5bbb15badcca6
solidity-utils/contracts/OnlyWethReceiver.sol	15eb42f03496738812daa0d0eb493c790dcd582d
solidity-utils/contracts/libraries/StringUtil.sol	a006835d5a8b54e5a212207e9374dddcb71f8835
solidity-utils/contracts/libraries/SafeERC20.sol	da8a249eb4318ca002ba6b83b0b7aa0ece2da8eb
solidity-utils/contracts/libraries/UniERC20.sol	af7e0c8f247c885f7bd362226d702df8f50c0a22
solidity-utils/contracts/libraries/RevertReasonForwarder.sol	f92839e53f0d9e7557f4af1f7b45d71a65eba56c
solidity-utils/contracts/libraries/ECDSA.sol	6c1da08ddbfcd7bca5a2c8694883f1899307bf72
solidity-utils/contracts/interfaces/IDaiLikePermit.sol	04e8e27f31e5ede1ecbe4c09f366828c54896928
solidity-utils/contracts/interfaces/IWETH.sol	24cd2851b4394d0d33fca955d7332c40f83ea88b

Appendix 2 - Disclosure