OpenZeppelin | security

# Cross Chain Swaps Audit


April 5, 2024

# Table of Contents

# Summary

| | |
|---|---|
| Type | DeFi |
| Timeline | From 2024-03-04<br>To 2024-03-18 |
| Languages | Solidity |

| | |
|---|---|
| Total Issues | 6 (4 resolved) |
| Critical Severity Issues | 0 (0 resolved) |
| High Severity Issues | 0 (0 resolved) |
| Medium Severity Issues | 0 (0 resolved) |
| Low Severity Issues | 6 (4 resolved) |
| Notes & Additional Information | 0 (0 resolved) |

# Scope

We audited the 1inch/cross-chain-swap repository at commit 5158cb8.

In scope were the following files:

```
./contracts/
├── Escrow.sol
├── EscrowDst.sol
├── EscrowFactory.sol
├── EscrowSrc.sol
├── interfaces
│   ├── IEscrow.sol
│   ├── IEscrowFactory.sol
│   └── IEscrowSrc.sol
├── libraries
│   ├── Clones.sol
│   ├── ImmutablesLib.sol
│   └── TimelocksLib.sol
```

# System Overview

The Cross-Chain Swap Protocol allows users to swap ERC-20 tokens from a certain source chain with ERC-20 tokens (or the native tokens) at a different destination chain. The basic operating principle is that funds are escrowed on both chains and then are released after a certain period of time when a "secret" is revealed. This secret is the pre-image of a keccak256 hash, which is stored in each escrow contract as a "hashlock".

A set of privileged users called "resolvers" are able to obtain the secret during a specific time window, after which it is possible to release the destination chain funds to the user (maker) and the source funds to the resolver (taker). The resolvers must be whitelisted, which entails going through a KYC/KYB process and having an active stake in the 1inch system. This whitelisting setup incentivizes prompt and well-behaved cross-chain order filling from said users.

Secrets are generated and encrypted by the frontend and then the relayer will forward them to various resolvers. At first, the secret can only be utilised by the selected resolver (winner of the auction) who has successfully created and funded the escrow contracts on both chains. This resolver is responsible for finalizing the order. If this resolver fails to fulfill the order, then eventually a public withdraw phase is entered. In this phase, any other user or resolver holding the secret (which will be disclosed by the relayer to all resolvers or read on-chain if the original resolver completes one leg of the trade) can finalize the destination side of the swap. This means that the end-user receives their tokens from the swap.

Only the chosen resolver may withdraw tokens from the source side of the swap, but it is assumed that resolvers are technically competent and have a high level of liveness and ability to send transactions. However, there is a time window in which anyone can cancel the source escrow, releasing the escrowed funds back to the original user (maker). The taker is also able to rescue funds accidentally sent to both contracts after a delay which should be enough to complete or cancel the swap on both chains.

To encourage the swap to be finalized on both chains in a timely manner, a "security deposit" of native tokens is included on each chain. Once the swap is finalized or cancelled, the security deposit is transferred to the `msg.sender` to compensate them for gas expenses and incentivize them to move the trade forward. If the secret is never broadcast, the swap will eventually time out, after which both sides can be cancelled and the funds will be returned to their original owners.

## Privileged Roles

There are two main privileged roles in the protocol:

- The relayer: An off-chain component developed and maintained by 1inch. It manages the secret disclosure to the relevant on-chain parties. This actor holds all current, encrypted secrets for all orders and also has the power to disclose them incorrectly or hold them for a longer time than expected (effectively causing a DoS on the protocol) if it becomes compromised. Security around this component is paramount for the overall safety of the system.
- Resolvers: Winners of the auctions will be able to act as takers of the cross-chain order and have the responsibility to finalize the order on both chains. They are expected to act diligently, promptly, and with liveness regarding the time windows created for the swap.

# Security Model and Trust Assumptions

The protocol contains several trust assumptions, especially revolving around both privileged roles.

Regarding the relayer:

- It is expected to diligently verify that the `dstEscrow` contract has been deployed after the `srcEscrow` contract, with the correct `Immutables` data and the correct safety deposit amount, before ever releasing the secret to the resolver. Otherwise, it is possible for a resolver to deploy a `dstEscrow` contract using arbitrary `Timelock` values that are against the maker's best interests. For instance, all time-deltas could be set to zero to ensure that the escrow can only be cancelled by the resolver himself with no risk of losing the funds or the safety deposit.
- It is expected to only disclose the secret when both escrow contracts are within the valid window where the resolver can successfully withdraw the amounts on both chains.
- It is expected disclose the secret to all resolvers simultaneously, including the resolver that won the auction.
- It is expected that the safety deposit amounts are coherent on a per-chain basis to account for withdrawal and cancellation gas fees, gas price spikes, and an extra incentive for third parties to go ahead and finalize or cancel the trade if the original

resolver is not acting correctly. It should also be large enough so that the resolver stands to lose a relevant amount of money when not acting in a prompt manner.

Regarding all resolvers within the system:

- They are expected to have high technical competency and familiarity with the protocol and smart contracts, and possess ample funds on all involved chains to pay for gas fees.
- They are expected to have a high level of liveness and ability to broadcast transactions at any time.

Specifically for resolvers who win the auction:

- They are expected to deploy both escrow contracts in a timely manner without altering any parameters at the destination chain or trying to mess with the deployment time in order to gain an advantage.

Specifically for other resolvers not selected to fill a specific trade:

- They should be vigilant on orders not filled in a timely manner so that they are promptly finalized or cancelled if necessary. The bounty received for doing so will be the safety deposit from the selected resolver.

Regarding general protocol operations:

- It is assumed that for any stage within all timelocks, `DeployedAt + <stage>` never exceeds `uint32(max)`, which corresponds to a timestamp in the year 2105.
- It is assumed that the functionality of the `limit-order-protocol` and `limit-order-settlement` protocols is correct as documented.
- It is assumed that data is formatted correctly when entering the `EscrowFactory._postInteraction` function within the audited repository.
- It is assumed that the ordering of the different timestamps for an order are as follows, where `<` indicates `before < after`: `SrcWithdrawal < DstWithdrawal < DstPublicWithdrawal < DstCancellation <= SrcCancellation < SrcPublicCancellation`.
- It is assumed that `RESCUE_DELAY` is significantly larger than the delta for `SrcPublicCancellation` and `DstCancellation`.

It is also assumed that this protocol will only be used on chains which have enough decentralization to avoid reorgs which may reverse fund movements for these swaps. It is assumed that 1inch will wait for enough confirmations, on a chain-by-chain basis, before confirming the existence of swap contracts and releasing secrets to finalize the swap. It is

assumed that the whitelist will be actively managed by 1inch and resolvers who misbehave will be immediately removed from the whitelist to prevent further abuse.

# General Recommendations

Aside from concrete recommendations made in the section below, there are general practices that can be followed to ensure that system assumptions are upheld and quickly detect and prevent abuse or misuse of the system by the privileged roles.

Within the user interface:

- 1inch should warn or prevent users from initiating swaps which send native tokens to a contract address on the "dst" chain. These contracts may execute code when receiving tokens from the output of a swap.
- 1inch should warn or prevent users from conducting swaps with tokens which are not trusted. Any tokens which are swapped may execute arbitrary logic. Keep in mind that a token may behave differently on one chain compared to another as it is a different contract on both chains.

During secret creation:

- Generate secrets by incorporating multiple sources of entropy in one or more air-gapped machines.
- Avoid entropy generation via known flawed or simplistic mechanisms. For example, avoid the Python `random()` function as it is simplistic, and avoid psuedorandom functions which involve a seed that could be guessed.
- Constantly refresh the entropy pool with new sources of entropy and retire old ones periodically. Do so at non-fixed intervals.
- Never disclose the technical details of entropy generation publicly.
- Maintain an internal "rainbow table" which is constantly being added to. Seek out on-chain and off-chain sources of pre-image and hash pairings. Ensure that no secrets which are in this rainbow table are ever used for creating order hashlocks. Ensure all secrets which are used for orders are added to an air-gapped version of this table or keep those secrets air-gapped until they have been revealed to resolvers.
- Avoid using human-readable secrets such as English words. Using English words that fit within a `bytes32` would narrow the set of potential secrets, making them easier to guess.

During order creation:

- The general volatility and overall market volume should be considered and positively correlated with safety deposit amounts. Gas prices are more likely to spike during times when prices of crypto assets are fluctuating more rapidly.
- Rules for creating a valid "dst" escrow contract should be available to all resolvers and should not be changed frequently. These rules should define all values as "immutables", including all stages of the `timelocks` for the "dst" escrow. The rules should also define the time window in which a valid "dst" escrow contract can be deployed.
- Resolution rules for choosing exactly one valid "dst" escrow contract should be defined and made public for all resolvers. Note that multiple contracts may be deployed with the exact same "immutables", and that these contracts need not be deployed by the specified resolver address. Generally, the valid "dst" escrow contract should be the first one deployed with the exact correct immutables (as defined by the bullet above) deployed *after* the "src" escrow contract is deployed.
- Monitor all chains where the `EscrowFactory` is deployed for sequencer or block producer liveness. If there is a consensus failure or excessive delays in block production, do not allow new orders to be created for a pre-set period of time.

During order filling:

- Monitor all transactions sent to any escrow contracts on any chain. Analyze these transactions to identify delays in sending transactions from resolvers, as well as any attempted attack vectors. Should an attack attempt be detected, all new order creation should be immediately halted until the attack attempt is dealt with.
- Monitor for all calls to `EscrowFactory.createDstEscrow` and parse the immutables used for creating these contracts. Immediately flag any deployed "dst" escrow contracts as invalid if they violate any of the rules defined in the "During order creation" section above.
- Before releasing any secrets, double-check the exact correctness of immutables for the "src" and "dst" escrow contracts. Double-check that the immutables in the "dst" escrow match the `DstImmutablesComplement` defined when creating the "src" escrow contract.
- Before releasing any secrets, validate that the native token and ERC-20 token balances of both escrow contracts are exactly correct. If they are not, ensure that this is only due to an accidental sending of extra tokens to the contract before releasing the secret. The secret should not be released if there are too few tokens in any escrow contract.
- Monitor for non-whitelisted actors who broadcast transactions that reveal secrets. Analyze transactions to identify and correlate non-whitelisted addresses with resolvers.

- It is not necessary to prevent secret reveals if there is more than one "dst" escrow contract with the correct immutables. This is important because if the secret release were prevented by this, malicious actors could create "dst" escrow contracts to prevent the fulfillment of trades.

After secret release:

- Keep a table of trade fulfillment. Record transaction IDs for both "src" and "dst" chains in which tokens were transferred to the maker and taker respectively. Validate this table again after a short wait period on each chain to validate that reorgs have not happened. Scan to identify any instances in which cancellation happened only on one chain but not the other, or instances where one side of the trade was never transferred out of the escrow contract.
- Monitor for calls to `rescueFunds`. Check that these calls only transfer funds which were accidentally transferred to some escrow contract. In rare cases, it is possible that the `rescueFunds` function could be used in the "src" escrow to take funds which should have been transferred to the `maker` during the cancellation phase.

# Low Severity

## L-01 Missing Docstrings

Throughout the codebase, there are multiple instances of code that do not have docstrings or use incorrect NatSpec syntax:

- The `RESCUE_DELAY` state variable in `Escrow.sol`
- The `FACTORY` state variable in `Escrow.sol`
- The `PROXY_BYTECODE_HASH` state variable in `Escrow.sol`
- The `ESCROW_SRC_IMPLEMENTATION` state variable in `EscrowFactory.sol`
- The `ESCROW_DST_IMPLEMENTATION` state variable in `EscrowFactory.sol`
- The `RESCUE_DELAY` function in `IEscrow.sol`
- The `FACTORY` function in `IEscrow.sol`
- The `PROXY_BYTECODE_HASH` function in `IEscrow.sol`
- The `CrosschainSwap` event in `IEscrowFactory.sol`
- The `ImmutablesLib` library in `ImmutablesLib.sol`

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

***Update:*** *Resolved in pull request #54 at commit cf9bee6.*

## L-02 Incomplete Docstrings

Throughout the codebase, there are several instances of incomplete docstrings:

- The rescueFunds, withdraw, and cancel functions in `IEscrow.sol` do not have the `immutables` parameter documented.
- The publicWithdraw function in `IEscrowDst.sol` do not have the `secret` and `immutables` parameters documented.

- The withdrawTo function in `IEscrowSrc.sol` does not have `target` and `immutables` parameters documented. The publicCancel function within the same interface also does not have the `immutables` parameter documented.

Consider thoroughly documenting all functions/events (and their parameters or return values) that are part of a contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

***Update:*** *Resolved in pull request #54 at commit 49e31a8.*

## L-03 Gas Optimization

The `cancel` function within the `EscrowDst` contract can only be called by the taker who will be the recipient of both the safety deposit and the escrowed tokens.

Consider enforcing consistency and making the function a bit more gas-efficient by transferring the escrowed tokens to `msg.sender` instead of retrieving the taker address from the `immutables` data structure.

***Update:*** *Acknowledged, not resolved. The 1inch team stated:*

> *After discussions within the team, we decided not to fix this issue. The difference in gas is close to 0 (our tests showed ~3-4), with readability, it does not get any better.*

## L-04 Lack of Sensitive Event Emission

When a cross-chain order has been filled on the source chain, the `CrosschainSwap` event is emitted. However, when the `EscrowDst` contract is deployed on the destination chain, no event is emitted.

Consider emitting an event upon deployment of the `EscrowDst` contract in order for off-chain logic to be able to properly track when both contracts are already deployed and the chronological order in which they were deployed.

***Update:*** *Resolved at commit 3de7020. The old `CrosschainSwap` event has been replaced by two new events (`SrcEscrowCreated` and `DstEscrowCreated`) which are emitted when each escrow contract is deployed in order to facilitate off-chain tracking. In addition, a new `SecretRevealed` event is being emitted upon public withdrawal including the secret.*

## L-05 Floating Pragma

Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled.

Throughout the codebase, there are multiple floating pragma directives:

- `Clones.sol` has the `solidity ^0.8.20` floating pragma directive.
- `IEscrow.sol` has the `solidity ^0.8.0` floating pragma directive.
- `IEscrowDst.sol` has the `solidity ^0.8.0` floating pragma directive.
- `IEscrowFactory.sol` has the `solidity ^0.8.0` floating pragma directive.
- `IEscrowSrc.sol` has the `solidity ^0.8.0` floating pragma directive.
- `ImmutablesLib.sol` has the `solidity ^0.8.20` floating pragma directive.

Consider using fixed pragma directives.

***Update:*** *Acknowledged, not resolved. The pragma directive in `TimelocksLib` has been changed to floating (`^0.8.20`) at commit f4b6f20 and the rest have remained unchanged. The 1inch team stated:*

> *We prefer to keep the public interfaces and libraries with the floating pragma directive to provide flexibility when importing them into other projects.*

## L-06 `publicWithdraw` Can Happen During Cancellation

Within the `EscrowDst` contract, the `publicWithdraw` function is callable at any time after the `DstPublicWithdrawal` timelock stage. This means that it could potentially be called after the escrow is supposed to be cancelled, thereby allowing a withdrawal at the destination chain and a cancellation at the source chain. This also mismatches with the comment on line 49 which explicitly states that the function is not callable during the cancellation phase.

Consider utilizing the `onlyBetween` modifier to prevent this function from being called after the cancellation phase begins.

***Update:*** *Resolved in pull request #53 at commit 13b7676. The modifier has been changed so that `publicWithdraw` can only be called between `DstPublicWithdrawal` and `DstCancellation` timestamps, effectively disallowing this call during the cancellation period.*

# Conclusion

The audited version of the 1inch Cross-Chain Swap Protocol enables users to seamlessly and gaslessly perform cross-chain swaps by leveraging the existing infrastructure from 1inch Fusion and the Limit Order Protocol. It provides a simple yet elegant way to incentivize whitelisted resolvers to promptly bridge and swap user assets. We found the codebase to be very well-written, thoroughly documented, and optimized for gas consumption.

No major issues were found during the audit, although some best practice recommendations were made in order to make the codebase even more robust. There are strong trust assumptions reflected in the introduction about the off-chain component (the relayer) which should be studied carefully.

The team was very responsive throughout the engagement, providing us with the necessary insight to expedite our understanding of the changes implemented and their effect on the codebase, while being open to our best practices recommendations.