1inch Cross Chain Swap v2

Date	July 2024
Auditors	George Kobakhidze, François Legué

1 Executive Summary

This report presents the results of our engagement with **1inch** to review **1inch Cross-chain swap v2**.

The review was conducted over two weeks, from **July 20, 2024** to **July 26, 2024**, by **George Kobakhidze** and **François Legué**. A total of 10 person-days were spent.

This system builds on the v1 of the cross-chain swap system by introducing a few features:

- Code adjusted for deployment on zkSync with relevant proxy adjustments
- Introduction of partial fills
- Refactor of code for further modularization and optimization

The code is clean and updated to use the latest industry standards. Due to the complexity of some code in new features, additional comments and clarifications to variables may be valuable.

No major flaws were identified with the design, though the increased complexity regarding partial fill secret management may be a cause for caution.

2 Scope

Our review focused on the commit hash 1inch/cross-chain-swap@3911835. 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. Correctness of the implementation, consistent 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. Secrets are not re-usable.
4. Partial fill indices are calculated correctly.
5. zkSync deployment-specific code adjustment is done correctly.
6. Correctness of Merkle Tree usage/implementation.

3 System Overview

Most notably, the partial fills system utilizes an approach of sharding out the secret key for the swap into many secrets and binding them together via a Merkle Tree with each secret's associated index and the final secret at the root. Each secret represents a portion of the tokens allocated in the order. For example, an order of `1000` tokens to be swapped may have 10 parts with a secret for each one of them:

- a secret for 1-100 tokens with an index of 0
- a secret for 101-200 tokens with an index of 1
- and so on until 901-1000 tokens. At the end, there is a final secret that is left for the final order, which would take all the remaining tokens in the order.

So for an order with `N` parts, there would be `N+1` secrets.

The additional secrets per order require the introduction of a callback `takerInteraction` function that is invoked before the `EscrowFactory` contract initiates the cloning of the escrow contracts (in `postInteraction()`). After validating the submitted proofs in accordance with the Merkle Tree root hash, the `lastValidated` state variable on the `EscrowFactory` contract (via the inherited `MerkleStorageInvalidator` contract) is updated to reflect the last used index and its associated secret. This state variable helps to invalidate any used index and its related secret.

The `postInteraction` callback will then recompute the index based on the remaining making amount, the original making amount, and the current making amount for a partial fill. If the index matches with the one stored in the lastValidated state variable, then the `secretHash` (also known as `hashlock` of the escrow contract) is retrieved from the `lastValidated` state variable. This secret hash is used to deploy the contract that will hold the assets.

The process will then be similar to the previous version, i.e:

- 1inch backend will monitor the creation of Source and Destination escrow contract;
- 1inch backend will monitor assets in respective contracts;
- The secret hash is released to resolvers in accordance with the defined timeline.

4 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.

The overall security posture of the contracts hasn't changed from the previous version. However, the new features do imply some new security assumptions and properties.

4.1 Partial Fill Secrets

The secrets in partial fills work together via a Merkle Tree and its associated proofs. To successfully validate a tree, a leaf, and proof of its inclusion in the tree it would require to properly sort and carefully construct the hashes. This

may be an error-prone process, though the 1inch team is well-experienced with providing complex calldata via its services.

There is also a dependency here to utilize a correctly implemented Merkle Proof validator, in this case the OZ's `MerkleProof`, as Merkle Trees can be subject to pre-image attacks and adjacent issues, though none are present here.

4.2 zkSync Properties

As is evident in the code, the zkSync chain introduces changes to how an EVM would work. Some of the opcodes that are critical in this system are different as well:

- `CREATE2` bytecode handling. While the EVM directly uses the bytecode for deployment, zkSync uses the hash of the bytecode and requires the compiler to be aware of the bytecode in advance.
- Address Derivation. zkSync uses distinct methods involving a prefix and bytecode hash, resulting in different addresses compared to Ethereum.
- `block.timestamp`. Though the current version of zkSync era works the same as usual on the EVM, in previous versions the timestamp referred to the L1 block time, not the L2 block.

Considering timestamps and contract cloning functionality are crucial for this system, it would be important to keep an eye out for any other changes zkSync may introduce in future upgrades.

5 Findings

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 Reusable Secrets in Partial Fills Major ✓ Fixed

Resolution

Fixed with commit fcf3a8d1d4922e95af1205473619cd533dc75b76 and PR96. These fixes enforce the use of the extension parameter that cannot be tampered to verify the submitted proofs. A new check is introduced to validate that the previously used index (or current, i.e. before the order happens) is different from the index of the secret to be used to fill the order. This verifies that indices, and therefore secrets, can't be re-used.

Description

This new version of the 1inch cross-chain swap v2, amongst other features, allows partial filling of cross-chain orders. The architecture remains the same and relies on Resolvers that will partially fill orders by creating contracts on the source and destination chains. Maker's assets being sent on the source chain and Taker's assets on the destination chain.

The partial fill functionality relies on callbacks (`takerInteraction()` and the `postInteraction()` functions) that will be invoked by the Limit Order Protocol. These functions are within the `EscrowFactory` contract.

A partially fillable order is split into several parts, each with a specific secret. When requirements are met (assets are in source and destination contracts), the 1inch backend reveals the secret that permits the unlocking of the assets.

To keep track of the last partially filled order, a state variable `lastValidated`, associated with a specific Maker's order, stores the last used index and secret. These values serve as a reference to invalidate the reuse of an already used index and secret.

contracts/MerkleStorageInvalidator.sol:L20-L21

```
/// @notice See {IMerkleStorageInvalidator-lastValidated}.
mapping(bytes32 => LastValidated) public lastValidated;
```

takerInteraction

This function uses a Merkle Tree to ensure that the submitted `secretHash` and index are valid and not tampered with.

contracts/MerkleStorageInvalidator.sol:L58

```
if (!proof.verify(root, keccak256(abi.encodePacked(idx, secretHash)))) rev
```

Another check verifies that the submitted index is not below the last validated index. **This ensures that an index cannot be reused.**

contracts/MerkleStorageInvalidator.sol:L57

```
if (idx < lastValidated[key].index) revert InvalidIndex();
```

After these checks, the callback updates the `lastValidated` state variable with the validated index and `secretHash`.

postInteraction

This second callback relies on the data validation during the `takerInteraction` callback and trusts the `lastValidated` state variable. It will recompute the index based on the remaining making amount, the original making amount, and the current making amount for a partial fill. If the index matches the one stored in the `lastValidated` state variable, then the `secretHash` (also known as `hashlock`) is retrieved. This secret hash is used to deploy the contract holding the assets.

contracts/BaseEscrowFactory.sol:L83-L87

```
uint256 calculatedIndex = (order.makingAmount - remainingMakingAmount + ma
if (
    (calculatedIndex + 1 != validated.index) &&
    (calculatedIndex + 2 != validated.index || remainingMakingAmount != ma
) revert InvalidSecretIndex();
```

There are two problems in this implementation:

1. The parameter (`interaction`) submitted to the first callback (`takerInteraction`) is not signed by any authority. A malicious Resolver can craft any Merkle Tree and submit a root hash, proofs, and index that would bypass all the logic in the function (the index should be greater than the last one);
2. In the Limit Order Protocol, the `takerInteraction` callback is bypassable by submitting an `interaction` parameter whose length is below 19 bytes. A malicious Resolver would be able to bypass the whole Merkle Tree logic that is meant to update the `lastValidated` state variable.

lib/limit-order-protocol/contracts/OrderMixin.sol:L378-L383

```
if (interaction.length > 19) {
    // proceed only if interaction length is enough to store address
    ITakerInteraction(address(bytes20(interaction))).takerInteraction(
        order, extension, orderHash, msg.sender, makingAmount, takingAmoun
    );
}
```

These two problems enable a malicious Resolver to reuse an old `secret` while partially filling an order that falls in an already used index. The secret being already revealed for a previous partial fill, the Resolver could:

- initiate a new partial fill that would fall under the same index;
- Limit Order Protocol would move the assets from the Maker to the source chain contract;
- wait until the private withdrawal timeline;
- reuse the previously revealed secret to withdraw/steal the assets from the maker without sending any asset to the destination chain.

Here is a practical example to illustrate the vulnerability:

1. A maker intends to swap 1000 tokens (partially fillable);
2. 1inch set 5 secrets associated with 5 ticks of 200 tokens each;
3. A resolver partially fills a secret tick: 210 tokens, leaving 190 tokens in the next tick;

4. This invalidates the secret for that last tick from being used via the logic in the `MerkleStorageInvalidator` contract (`takerInteraction` callback);
5. The secret is invalidated and does not allow anyone to get the remaining 190 tokens in this tick
6. The logic in `MerkleStorageInvalidator` is bypassable from `OrderMixin` (interaction parameter not checked, or interaction length below 19 bytes)
7. The Resolver can then reuse the same secret, falling into the same tick (index), by partially filling 190 tokens
8. The escrow factory reuses the same secret to create the source escrow
9. The Resolver can steal the funds when the private withdrawal phase begins

It is to highlight that this vulnerability is exploitable by the Resolvers that are whitelisted by 1inch.

Recommendation

We recommend preventing the root hash parameter from being tampered with. We recommend also forcing the taker's interaction when the order is partially fillable.

5.2 Order Cancellation Does Not Re-Validate the Secret.

Minor Acknowledged

Resolution
Acknowledged as part of the design.

Description

When a taker goes through the cross-chain swap process and fills an order, they have to use a hash of a secret that's used up to deploy the escrows. This is registered in the `MerkleStorageInvalidator` :

contracts/MerkleStorageInvalidator.sol:L59

```
lastValidated[key] = LastValidated(idx + 1, secretHash);
```

However, escrows may end up getting cancelled, but that doesn't re-validate the secrets used to partially fill the order. The contract just sends back the tokens.

- Source Escrow:

contracts/EscrowSrc.sol:L118-L126

```
/**
 * @dev Transfers ERC20 tokens to the maker and native tokens to the caller.
 * @param immutables The immutable values used to deploy the clone contract.
 */
function _cancel(Immutables calldata immutables) internal onlyValidImmutab
    IERC20(immutables.token.get()).safeTransfer(immutables.maker.get(), im
    _ethTransfer(msg.sender, immutables.safetyDeposit);
    emit EscrowCancelled();
}
```

- Destination Escrow:

contracts/EscrowDst.sol:L55-L69

```
/**
 * @notice See {IBaseEscrow-cancel}.
 * @dev The function works on the time interval highlighted with capital let
 * ---- contract deployed --/-- finality --/-- private withdrawal --/-- publ
 */
function cancel(Immutables calldata immutables)
    external
    onlyTaker(immutables)
    onlyValidImmutables(immutables)
    onlyAfter(immutables.timelocks.get(TimelocksLib.Stage.DstCancellation)
{
    _uniTransfer(immutables.token.get(), immutables.taker.get(), immutable
    _ethTransfer(msg.sender, immutables.safetyDeposit);
    emit EscrowCancelled();
}
```

Recommendation

Consider making a process to work around the cancellation which would re-validate the secrets, as they are not revealed in the cancellation process. However, it is understood that there is a similar situation with regular cross-chain swap orders, and the maker would have to sign a new order. Moreover, the takers are whitelisted resolvers who are unlikely to abuse the system with fake fills that get cancelled to impact the user experience.

5.3 Weak Secrets Generation Minor ✓ Fixed

Resolution

The client mentioned that they will use cryptographically secure random to generate secrets in production.

Description

The `secretHash`, also known as the `hashlock` is the core secret needed to release/withdraw the assets from the escrow contracts on the source and destination chain.

In the current tests, the generation of this sensitive secret is weak and predictable:

test/integration/MerkleStorageInvalidator.t.sol:L24-L29

```
    for (uint256 i = 0; i < SECRETS_AMOUNT; i++) {
        hashedSecrets[i] = keccak256(abi.encodePacked(i));
        hashedPairs[i] = keccak256(abi.encodePacked(i, hashedSecrets[i]));
    }
    root = merkle.getRoot(hashedPairs);
}
```

If this pattern is used to generate secrets in production, a malicious Resolver could find it and steal the maker's assets without sending the exchanged asset to the destination chain.

Recommendation

Consider using randomness to generate the secrets to prevent any Resolver from guessing it.

5.4 Partial Fills of a Secret's Tick Force the Next Fill to Take in the Remainder of the First Tick. Acknowledged

Resolution
Acknowledged as part of the design.

Description

The issue highlighted here stems from the handling of partial orders in the contract. Specifically, when a buyer decides to fulfil an order, even for just `1` token in the tick, the secret associated with it is used up and can't be provided again as can be seen in the `MerkleStorageInvalidator`:

contracts/MerkleStorageInvalidator.sol:L57

```
if (idx < lastValidated[key].index) revert InvalidIndex();
```

This could lead to user experience issues, as another buyer, who wants to fulfil the remaining order, would have to go through the entire first tick – potentially making it less attractive.

Recommendation

It is understood that this is part of the design and may be too complex to implement another solution, such as allowing several different secrets for the same tick, which would introduce a lot more proofs in the calldata. However, it may be worth making the number of secrets large so that the impact of having to fill a large amount for each order's tick is minimized.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
contracts/BaseEscrow.sol	`f93821ad8a1547d9f9dee885a41415ce5e5afbf7`
contracts/BaseEscrowFactory.sol	`2939d787c64ddf26e4564cd5960568d839f4476f`
contracts/Escrow.sol	`af7bf5f70ede26ac067450c383d6e4a51a809ade`
contracts/EscrowDst.sol	`13ba8910b94eb8d123a8a4bc2cdd95d122cfc64c`
contracts/EscrowFactory.sol	`c5df58d29ab8a4b11b8447ca2f4855f5ecc48cf2`
contracts/EscrowSrc.sol	`7e58a93f7d9d4ff56774a0b5fa4f68bdab0b3528`
contracts/MerkleStorageInvalidator.sol	`3c63200609ab1aa8730edfb5c8cd897a378bed5e`
contracts/interfaces/IBaseEscrow.sol	`2504f9a081741deb127f6cb269506336115039ec`
contracts/interfaces/IEscrow.sol	`3c3287be40470942a5752894e4cc116bcbfac633`
contracts/interfaces/IEscrowDst.sol	`09d76a47583c35b7ccf1f99f23e240fb9ce204a0`
contracts/interfaces/IEscrowFactory.sol	`555de1c55c4266bfee8c183485bc7e2fac044294`
contracts/interfaces/IEscrowSrc.sol	`02f1ee678796650d6c45ee0a5ca7090a1d3cc4be`
contracts/interfaces/IMerkleStorageInvalidator.sol	`36dde1a3999d292d4e067d0b8360fd52a6ed3073`
contracts/libraries/ImmutablesLib.sol	`1a5d808d170f658ab5cbf8bd6b84130e1e768214`
contracts/libraries/ProxyHashLib.sol	`f4e4406094ac6abb638a6def2b5442bfc90b0f71`
contracts/libraries/TimelocksLib.sol	`4e1e1eb4c8a5f4d57e6bfff4febcd6a035c249a6`
contracts/zkSync/EscrowDstZkSync.sol	`8006fdcd0f28f1df2191a1c2e1b94dea08f2eeb6`

File	SHA-1 hash
contracts/zkSync/EscrowFactoryZkSync.sol	`fd16771af124f5a15ec02ca7e47fd7577db2fbab`
contracts/zkSync/EscrowSrcZkSync.sol	`4dc56387685f2680c33993e771ee5fb676138439`
contracts/zkSync/EscrowZkSync.sol	`49a0fe587beaf0479ae674f6e9f1af9b2f105186`
contracts/zkSync/MinimalProxyZkSync.sol	`da8a1c77cf84f6e7e6aabc2b0af7da9590fbd1a5`
contracts/zkSync/ZkSyncLib.sol	`b0a4e58e5f5ff380e6974ccc7eb532ea19fdeb8e`

Appendix 2 - Disclosure

Consensys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated

otherwise, by Consensys and CD.