1inch Cross Chain Swap

Date	March 2024
Auditors	George Kobakhidze, François Legué

1 Executive Summary

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

The review was conducted over two weeks, from **March 25, 2024**, to **March 28, 2024**, by **George Kobakhidze** and **François Legué**. A total of 10 person-days were spent.

This system of contracts provides users functionality to securely sign off on providing tokens on a source chain and unlock wanted tokens on the destination chain. This is done through escrow smart contracts on each of the chains, with the process facilitated by an off-chain secret, the management of which is currently done by an offchain 1inch relay.

The code is clean and well-documented. The use cases and intended business logic are clear both from the code itself and the NatSpec. Contracts are quite modular, and the libraries are efficient, with precise usage of assembly.

No critical or major issues were found, though there are improvements to be made and edge cases to be reconsidered.

2 Scope

Our review focused on the commit hash 1inch/cross-chain-swap@`b17cdb7`. 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. Tokens can't be moved without secrets in appropriate phases of the escrow.
4. Malicious takers or makers can't drain or leave the counter-party without their side of the order being fulfilled.

3 Recommendations

3.1 Standardize Solidity Versions Acknowledged

Resolution

The client mentioned that they prefer to keep the public interfaces and libraries with the floating pragma directive to provide flexibility when importing them into other projects.

Description

The Solidity versions in this repository are different between escrow contracts (`Escrow.sol`, `EscrowFactory.sol`, etc.) that use `pragma solidity 0.8.23` and the libraries (`Clones.sol`, `ImmutablesLib.sol`, etc.) that use `pragma solidity ^0.8.20`.

Recommendation

It would be best to have the whole repo follow the same Solidity version.

3.2 Separate Out Helper Functions Into Their Own Library or Use From an Existing One Acknowledged

Description

The `Escrow.sol` contract file contains functions that are not specific to the business logic of escrows. Specifically, these are: `_uniTransfer`:

contracts/Escrow.sol:L73-L82

```
/**
 * @dev Transfers ERC20 or native tokens to the recipient.
 */
function _uniTransfer(address token, address to, uint256 amount) internal {
    if (token == address(0)) {
        _ethTransfer(to, amount);
    } else {
        IERC20(token).safeTransfer(to, amount);
    }
}
```

`_ethTransfer`:

contracts/Escrow.sol:L84-L90

```
/**
 * @dev Transfers native tokens to the recipient.
 */
function _ethTransfer(address to, uint256 amount) internal {
    (bool success,) = to.call{ value: amount }("");
    if (!success) revert NativeTokenSendingFailure();
}
```

`_keccakBytes32`:

contracts/Escrow.sol:L102-L112

```
/**
 * @dev Computes the Keccak-256 hash of the secret.
 * @param secret The secret that unlocks the escrow.
 * @return ret The computed hash.
 */
function _keccakBytes32(bytes32 secret) private pure returns (bytes32 ret) {
    assembly ("memory-safe") {
        mstore(0, secret)
        ret := keccak256(0, 0x20)
    }
}
```

While it is not a problem to have such functions to exist within the contract, the logic itself could be isolated outside of the contract within a library. For example, the `_uniTransfer` function could simply call a `uniTransfer` function from the 1inch `UniERC20` library, if the logic is appropriate:

lib/solidity-utils/contracts/libraries/UniERC20.sol:L40-L57

```
/// @dev `token` transfer `to` `amount`.
/// Note that this function does nothing in case of zero amount.
function uniTransfer(
    IERC20 token,
    address payable to,
    uint256 amount
) internal {
    if (amount > 0) {
        if (isETH(token)) {
            if (address(this).balance < amount) revert InsufficientBalance();
            // solhint-disable-next-line avoid-low-level-calls
            (bool success, ) = to.call{value: amount, gas: _RAW_CALL_GAS_LIMIT}("");
            if (!success) revert ETHTransferFailed();
        } else {
            token.safeTransfer(to, amount);
        }
    }
}
```

Recommendation

In general, it is recommended to avoid creating such custom logic for contracts and instead specify it in libraries made just for them, and then import those libraries. This would improve maintainability for future versions.

4 System Overview

The 1inch cross-chain swap solution aims to ease the swap of assets from one chain to another. The architecture of this solution is based on three actors:

- the Maker who initiates and emits an intent to swap an asset for another asset on a different chain;
- the Taker/Resolver who fulfills the Maker order;
- the Relayer, which is an off-chain component that interacts with the Taker (also known as Resolver).

The cross-chain swap is composed of the following steps.

4.1 Cross-chain swap steps

Order creation

The cross-chain swap solution is based on the 1inch limit order protocol. The Maker can emit an intent to swap an asset to another sending specific information to the 1inch backend (the Relayer). A secret associated with this order is encrypted for each Resolver.

Order fulfillment

A Dutch auction is performed between all Resolvers/Takers until one Resolver decides to fill the order.

The Resolver will then have to create two escrow contracts on both chains of the swap.

It first creates the escrow contract on the source chain. It uses the limit-order-protocol to move the funds from the Maker to this escrow contract and also adds a safetyDeposit in native token.

Then, it creates the escrow contract on the destination chain where it deposits the token wanted by the Maker. There are deposited in exchange for the Maker's tokens deposited in the escrow contract on the source chain. A safetyDeposit is also deposited in native tokens in these contract.

Off-chain processing

The 1inch Relayer ensures that the escrows in both chains contain the required amounts of tokens. When all conditions are fulfilled, the Relayer releases the secret to all Resolvers.

Withdrawal

The Resolver that was designed as being the Taker can use the revealed secret to:

- send the Resolver's asset from the escrow on destination to the maker;
- retrieve the maker's assets from the escrow on the source chain to himself.

4.2 Mechanisms and specificities

To successfully achieve the swap, a list of mechanisms and specificities comes into play.

Escrows address computation

The deployment of escrow on the source and destination chains is based on `create2` opcode. Using an escrow factory and a specific salt, the Resolver will compute and deposit the safetyDeposit and the Maker/Taker funds to the corresponding addresses. The salt that is used to compute these addresses depends on Immutable structures. These structure contain all information about the order (maker, taker, token, amount, etc), the safetyDeposit and the timelocks.

Timelocks

As being executed on two different chains, the swap is not atomic and is done in multiple steps. A timelock structure contains one timestamp (`DeployedAt`) and 6 periods that are relative to the timestamp. These periods are used to identify the current state/stage of the swap. These periods protect the Resolver from the Maker to retrieve his funds from the source chain escrow while being able to retrieve the funds from the destination escrow.

They also prevent abuses and help recover the funds in case the requirements of the swap are not fulfilled.

Timelocks are also used to give the Taker a period (called private phase) where it is the only one able to execute the swap (on source and destination chain) and retrieve back its safetyDeposit.

Then, a public phase gives anyone the right to either cancel the escrow and return the Maker's assets on the source chain or perform the withdrawal of the assets to the Maker's address on the destination chain.

Hashlock

The hashlock is the hash of a secret that is associated to a swap. The secret used to compute the hashlock is released by the Relayer when all swap requirements are fulfilled. This is the mechanisms that gives the Taker the right to unlock the assets.

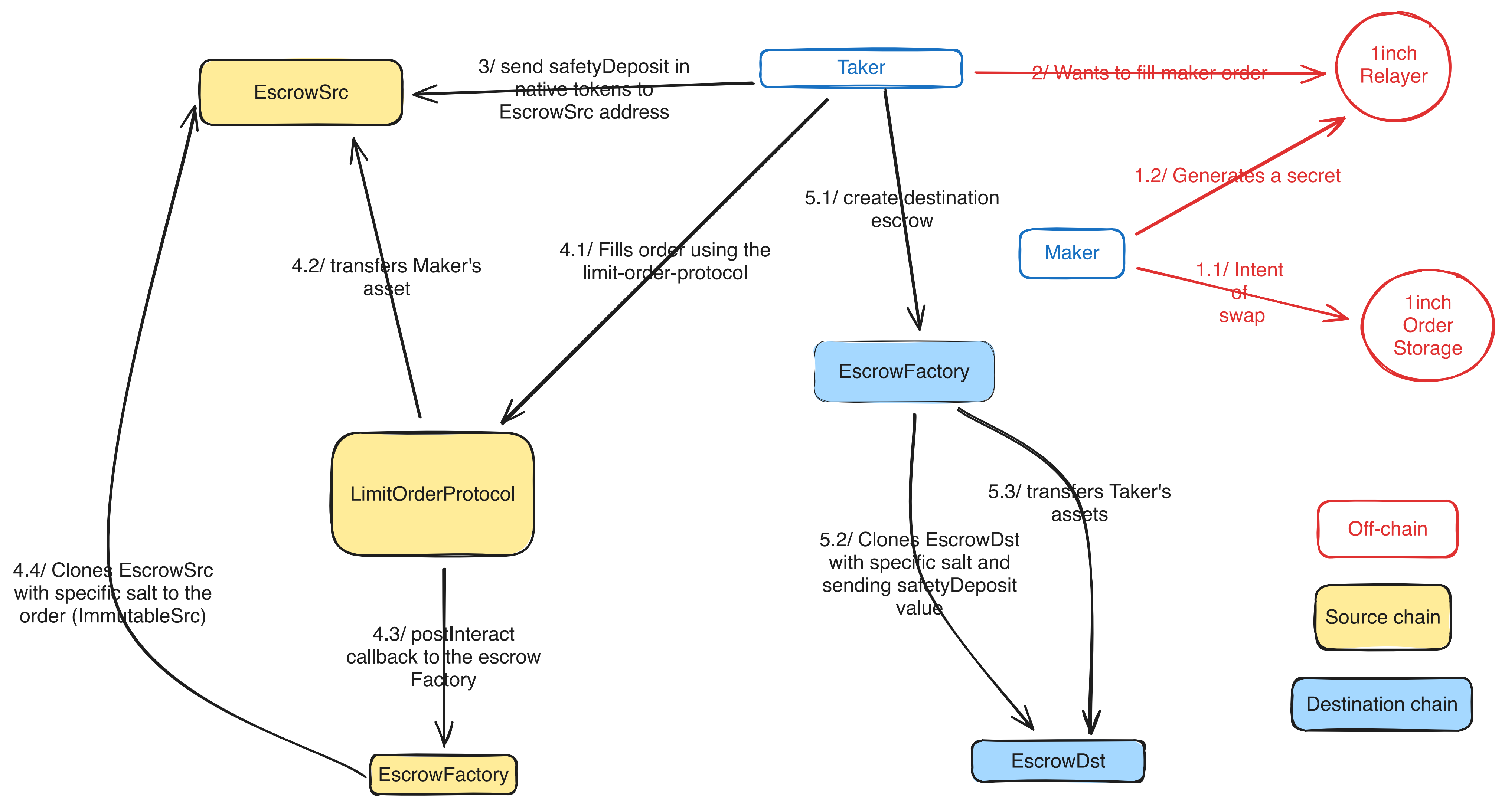
safetyDeposit

The safetyDeposit incentivizes anyone to perform actions in the lifecycle of a swap. In the licecycle, the Resolver has a private phase period where it is the only one to do action. In case of failure, this safetyDeposit incentivizes anyone to finalize or cancel a swap.

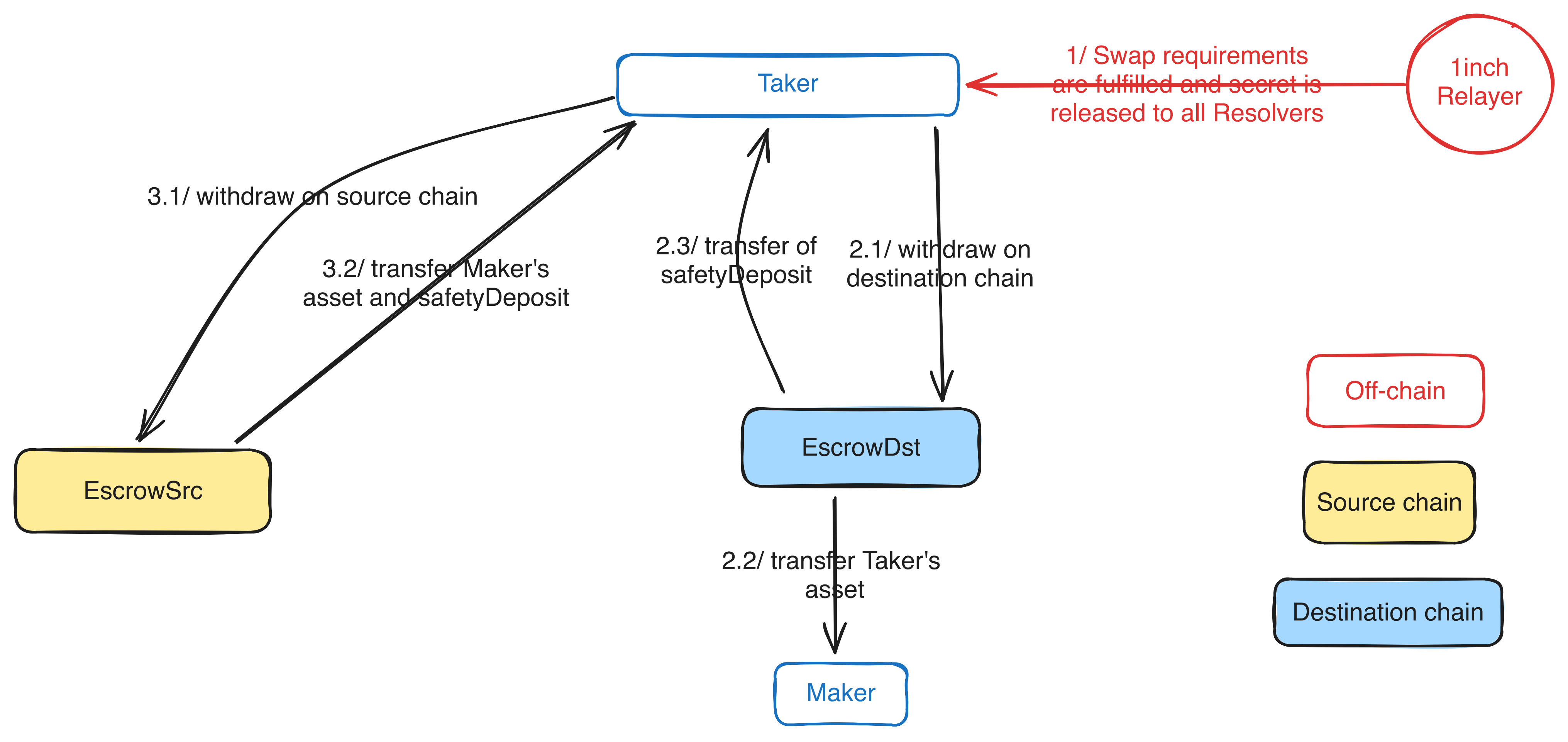
4.3 Diagrams

The following diagrams are high-level representations of cross-chain swap in normal conditions.

Initialization and deposit phase


EscrowSrc
3/ send safetyDeposit in native tokens to EscrowSrc address
Taker
2/ Wants to fill maker order
1inch Relayer
1.2/ Generates a secret
5.1/ create destination escrow
Maker
1.1/ Intent of swap
1inch Order Storage
4.2/ transfers Maker's asset
4.1/ Fills order using the limit-order-protocol
EscrowFactory
LimitOrderProtocol
5.3/ transfers Taker's assets
5.2/ Clones EscrowDst with specific salt and sending safetyDeposit value
4.4/ Clones EscrowSrc with specific salt to the order (ImmutableSrc)
4.3/ postInteract callback to the escrow Factory
EscrowFactory
EscrowDst
Off-chain
Source chain
Destination chain

Withdrawal phase


Taker
1/ Swap requirements are fulfilled and secret is released to all Resolvers
1inch Relayer
3.1/ withdraw on source chain
3.2/ transfer Maker's asset and safetyDeposit
2.3/ transfer of safetyDeposit
2.1/ withdraw on destination chain
EscrowSrc
EscrowDst
2.2/ transfer Taker's asset
Maker
Off-chain
Source chain
Destination chain

5 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.

5.1 Actors

- Makers. Anyone can be a maker in this system who intends to provide tokens on the source chain in return for some other tokens on the destination chain.
- Takers. Whitelisted users that act as resolvers in the 1inch order system which in this case would require the takers to provide liquidity on the destination chain of the order, while taking in the tokens from the maker on the source chain. They are also required to provide safety deposits to the escrow contracts on both chains to provide insurance for the system that the taker would act in a fair and timely manner.
- 1inch off-chain infrastructure. This is the system that automates the 1inch order system and, therefore, facilitates the cross-chain swap and escrow creation as well. It is currently responsible for assisting makers and takers in creating the right data structures, hashes, signatures, and through the relayer - secret management and reveal.

5.2 Trust Model

In any system, it's important to identify what trust is expected/required between various actors. For this audit, we established the following trust model:

- Makers. In theory, the makers are not trusted to act one way or another. They sign off on the order, and during the order execution their tokens are automatically pulled in by the contracts. Moreover, many of the parameters for the order are provided by the 1inch servers (backend and/or frontend). However, in the event the system is used in a permissionless manner without the 1inch relayer, the makers could provide their own order parameters, which in the current system do allow for some malicious activity, namely within the time interval parameters for different phases of the escrow, as is discussed in one of the issues in this report. Therefore, takers and other participants of this system should take note and double-check any parameters and order details given to them, specifically by the makers as it would be in their interest to game the system.
- Takers. These whitelisted entities perform the bulk of the smart contract operations in this system. They are trusted, and indeed even provide assurances for that trust in the form of safety deposits, to not act maliciously and do so in a timely manner. Otherwise, their deposits may be taken away by other users. As the takers start the escrow process, they also initiate the pull of the maker's tokens to the escrow, locking them for a period. Consequently, in the intended scenario the takers are trusted not to abuse this lock without fulfilling the other part of the order. This is discussed in one of the issues in the report.
- 1inch off-chain infrastructure. As mentioned above, in the current system the off-chain servers are trusted to assist with generating a lot of the parameters needed for the system to function. Indeed, even creating the necessary data structures to create the escrow clones is normally done by the 1inch dApp for the takers. More importantly, the relayer part of the infrastructure safeguards the secret that unlocks the tokens and releases it to the takers upon verifying that the escrows are funded. In short, the relayer and the rest of the 1inch servers currently play a critical role in the system and are trusted to perform it correctly. Still, it is important to mention that the system may function without the relayer or any auxillary servers in a permissionless way if the makers and takers decide to pursue p2p communication for the orders.

5.3 Security Properties

The following is a non-exhaustive list of security properties that were verified in this audit:

- Secret Management & Security. The escrow contracts are critically dependent on the secrets created by the maker at the beginning of the process. If the storage and release of these secrets are compromised, a malicious taker may unlock source tokens without even funding the destination tokens, leaving the maker with nothing. It is imperative the secrets are neither weak nor managed insecurely.

6 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.

6.1 Lack of Sanity Checks for `timelocks` Medium Acknowledged

Resolution

The client mentioned that the Relayer and Resolver should validate timelocks before proceeding with swap. Even in p2p swap, Resolver is expected to be technically competent and be able to implement sanity checks for timelocks on their side. The client is also considering adding an example of such checks to `ResolverMock`.

Description

During the creation of escrows, the `immutables` data structure is used that contains a `timelocks` object, governing the stages of the escrow:

contracts/interfaces/IEscrow.sol:L13-L23

```
interface IEscrow {
    struct Immutables {
        bytes32 orderHash;
        bytes32 hashlock;  // Hash of the secret.
        Address maker;
        Address taker;
        Address token;
        uint256 amount;
        uint256 safetyDeposit;
        Timelocks timelocks;
    }
```

contracts/libraries/TimelocksLib.sol:L26-L40

```
type Timelocks is uint256;

/**
 * @title Timelocks library for compact storage of timelocks in a uint256.
 */
library TimelocksLib {
    enum Stage {
        DeployedAt,
        SrcWithdrawal,
        SrcCancellation,
        SrcPublicCancellation,
        DstWithdrawal,
        DstPublicWithdrawal,
        DstCancellation
    }
```

As it becomes evident from the comments and the intended usage of the system, for the source escrow:

1. First source withdrawal starts.
2. Then private cancellation.
3. Then public cancellation.

Similarly, for the destination escrow:

1. First private destination withdrawal starts.
2. Then public withdrawal.
3. Then cancellation.

However, this isn't actually checked to be true. Indeed, by following only the checks in the smart contracts for the escrow contracts, a malicious maker could construct an escrow `timelocks` object such that the source public cancellation could start at the same time as the source withdrawal. In parallel, they could construct the destination `timelocks` so that the public withdrawal period starts immediately, while the cancellation starts only long after.

This would allow a malicious maker to:

1. Immediately cancel the source escrow

contracts/EscrowSrc.sol:L63-L75

```
/**
 * @notice See {IEscrow-publicCancel}.
 * @dev The function works on the time intervals highlighted with capital letters:
 * ---- contract deployed --/-- finality --/-- private withdrawal --/-- private cancellation --/-- PUBLIC CANCELLATION ----
 */
function publicCancel(Immutables calldata immutables)
    external
    onlyValidImmutables(immutables)
    onlyAfter(immutables.timelocks.get(TimelocksLib.Stage.SrcPublicCancellation))
{
    IERC20(immutables.token.get()).safeTransfer(immutables.maker.get(), immutables.amount);
    _ethTransfer(msg.sender, immutables.safetyDeposit);
}
```

2. Immediately retrieve the destination tokens

contracts/EscrowDst.sol:L44-L59

```
/**
 * @notice See {IEscrow-publicWithdraw}.
 * @dev The function works on the time intervals highlighted with capital letters:
 * ---- contract deployed --/-- finality --/-- private withdrawal --/-- PUBLIC WITHDRAWAL --/-- private cancellation ----
 */
function publicWithdraw(bytes32 secret, Immutables calldata immutables)
    external
    onlyValidImmutables(immutables)
    onlyValidSecret(secret, immutables)
    onlyAfter(immutables.timelocks.get(TimelocksLib.Stage.DstPublicWithdrawal))
    onlyBefore(immutables.timelocks.get(TimelocksLib.Stage.DstCancellation))
{
    _uniTransfer(immutables.token.get(), immutables.maker.get(), immutables.amount);
    _ethTransfer(msg.sender, immutables.safetyDeposit);
    emit SecretRevealed(secret);
}
```

This results in the taker losing **both** of their safety deposits **and** not getting the maker tokens - everything will be taken by a malicious maker.

Indeed, the only current checks for timestamp sanity are for source cancellation, **not** source public cancellation, and even that has issues as it is a user-provided timestamp parameter that is not necessarily the one that was used to create the escrows. More is discussed on this in issue 6.6.

Recommendation

From conversations with the 1inch team, it is understood that the sanity checks (along with many others) for the `timelocks` object are done off-chain by the 1inch relay, which is also whitelisted for access for the takers. However, it is possible to use these smart contracts trustlessly and in a p2p fashion. For example, a very large OTC order could be created between a well-funded potentially malicious user and a whitelisted taker/resolver, who would be well-incentivized to help with this swap to earn OTC fees. Yet, this vulnerability could leave the taker without funds.

6.2 Incorrect Bitwise Shift for Enum Stage of Value Zero Medium Acknowledged

Resolution

The client mentioned that they don't currently use this in any of the contracts and have no intentions to do so.

Description

The issue lies in the function `get()` where the bitshift operation is performed. Enumeration values in Solidity start from 0, and the value is multiplied by 32 to perform the shift operation. The `Stage` enum has a `DeployedAt` value of 0. When this value is passed to the `get()` function for bitshifting, the multiplication operation results in a zero bitshift, i.e., no shift at all. This causes the result to be an unintended doubling of the `data` variable when it is supposed to be an addition of the original `data` and the shifted `data`.

contracts/libraries/TimelocksLib.sol:L63-L73

```
/**
 * @notice Returns the timelock value for the given stage.
 * @param timelocks The timelocks to get the value from.
 * @param stage The stage to get the value for.
 * @return The timelock value for the given stage.
 */
function get(Timelocks timelocks, Stage stage) internal pure returns (uint256) {
    uint256 data = Timelocks.unwrap(timelocks);
    uint256 bitShift = uint256(stage) * 32;
    return uint32(data) + uint32(data >> bitShift);
}
```

Recommendation

To fix this issue, perform special handling of the scenario when the `DeployedAt` stage is requested in the `get()` function. For example, simply immediately return `uint32(data)`.

6.3 Maker Orders Can Be Invalidated by a Malicious Taker/Resolver Medium Acknowledged

Resolution

The client mentioned that all Resolvers are motivated to behave well as they are bounded by their contract with 1inch. They use legal mechanism instead of economic mechanism.

Description

A cross-chain swap is composed of multiple phases. The Maker creates an intent of the order, signs it, and sends it to the Relayer of 1inch. With the current design, a malicious Resolver can perform a griefing attack against the Maker by pretending to fulfill his order while never doing so.

The attack would consist of the following steps:

1. Signal to the Relayer the willingness to fill the Maker's order;

2. The Resolver/Taker then creates the Escrow on the source chain and initiates the `fillOrder()` call to the limit-order-protocol.
3. The Resolver then **never** deploys any contracts on the destination chain; the secret is then never revealed.
4. The Resolver waits and calls the `cancel()` function of the `EscrowSrc` contract on the source chain.
5. The Resolver gets back his `safetyDeposit` and the maker's assets are returned

This is possible because the current design defines a private period where the Resolver is the only one to be able to process the action. While this is protection against a malicious Maker, it could be exploited by a malicious Resolver to cancel the swap order and get back the safetyDeposit.

Recommendation

An economic mechanism should punish/slash Resolvers on malicious actions.

6.4 Missing Events Minor Acknowledged

Resolution

The client mentioned that only in the case of `EscrowDst.publicWithdraw` this event can actually provide useful information to the Resolver. The client assumes that a possible need for getting swap status can be fulfilled by the ERC-20 `Transfer` event. `_from` can be associated with an escrow address and `_to` with either Maker or Taker.

Description

Some of the functions that are critical to the business logic flow don't have events emitted. Consider implementing events for:

- all instances of `cancel` functions
- all instances of `withdraw` functions
- `publicCancel`

Similarly, `SecretRevealed` event is emitted during the execution of `EscrowDst.publicWithdraw` function, while it is not emitted during the execution of `EscrowSrc.withdraw` or `EscrowSrc.withdrawTo` functions, both of which also reveal the `bytes32 secret` parameter.

Recommendation

Consider implementing more events.

6.5 Interfaces Inconsistent With Implementation Minor Acknowledged

Resolution

The client mentioned that If they add the virtual functions withdraw and cancel to the base abstract contract, they will have to add override to the mentioned implementation functions. The client believes that this reduces readability, so he decided to mention the need to implement withdraw and cancel in derived contracts (link to PR).

Description

Some of the interfaces provided don't reflect the full external functionality of contracts.

Source Escrow:

`IEscrowSrc` is missing:

- `cancel`
- `withdraw`

Destination Escrow:

`IEscrowDst` is missing:

- `cancel`
- `withdraw`

Base Escrow:

`IEscrow` refers to functions which are **not** in the base contract :

- `cancel`
- `withdraw`

As can be seen above, it appears that the functions missing from interfaces of source and destination escrow implementations are instead available in the `IEscrow` interface, the implementation of which doesn't have them.

Recommendation

It would be recommended to reconcile the interfaces with the actual implementations. Consider implementing virtual functions in the `Escrow` base contract for `cancel` and `withdraw` functions to keep the base abstract contract in sync with the interfaces.

6.6 Minor Issues With Destination Escrow Timestamp Checks Minor Acknowledged

Description

The function `createDstEscrow` accepts a parameter `srcCancellationTimestamp` which is meant to provide the source escrow cancellation timestamp. It is then checked against the timestamp stored in the destination `immutables` to provide some protection for the taker:

contracts/EscrowFactory.sol:L126-L129

```
IEscrow.Immutables memory immutables = dstImmutables;
immutables.timelocks = immutables.timelocks.setDeployedAt(block.timestamp);
// Check that the escrow cancellation will start not later than the cancellation time on the source chain.
if (immutables.timelocks.get(TimelocksLib.Stage.DstCancellation) > srcCancellationTimestamp) revert InvalidCreationTime();
```

However, as is evident by this line itself, the destination escrow already uses `timelocks` in its `immutables` that may contain the necessary information. If we take a look at the `TimelocksLib.sol`, we can see that indeed the timelocks object should have all the relevant stages and their data in it:

contracts/libraries/TimelocksLib.sol:L31-L40

```
library TimelocksLib {
    enum Stage {
        DeployedAt,
        SrcWithdrawal,
        SrcCancellation,
        SrcPublicCancellation,
        DstWithdrawal,
        DstPublicWithdrawal,
        DstCancellation
    }
```

It may be better to just query the `timelocks` object and check against `timelocks.get(TimelocksLib.Stage.SrcCancellation)` directly. Moreover, it would be prudent to also check against `timelocks.get(TimelocksLib.Stage.SrcPublicCancellation)` as well, as that cancellation is currently not guaranteed to be after the `SrcCancellation` period within the smart contract checks.

It is understood from conversations with the 1inch team that currently the destination `immutables` don't get all the timestamps passed to them, as they don't make use of all of them. However, that would mean the `timelocks` object would be different between the two escrows, which may not be expected when constructing one manually. The result of this would be a hash of the `immutables` data structure that may be different from what one would expect, which then would create the destination escrow contract at an address that a user may not expect as well.

Recommendation

To avoid potential issues and to ensure uniformity between the source and destination escrows, it is recommended to use the same `timelocks` object when creating the escrows. Consequently, rather than accepting `srcCancellationTimestamp` as a parameter for `createDstEscrow`, retrieve the relevant timestamp directly from the escrow's immutables. Finally, consider checking against `SrcPublicCancellation` as well.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
contracts/Escrow.sol	`8eafdefb7b649a8eaf5c0c65cfe186f3d9e9d913`
contracts/EscrowDst.sol	`51d41753cd987c41130e8e39df0a9c9bf7238d74`
contracts/EscrowFactory.sol	`f7a11904d6c1e964d49e9148743c4cf9c37e1f82`
contracts/EscrowSrc.sol	`a2c4aca783126c15c663aee9c0a042c2fc19bbcd`
contracts/interfaces/IEscrow.sol	`6119af0271d42a6b8f56ae556f2ca1786163deaf`
contracts/interfaces/IEscrowDst.sol	`bdb7c7d34a36bde70a3d644b5800b99b7a5fefe1`
contracts/interfaces/IEscrowFactory.sol	`ae9fa30d133a2e0231c9bdf5af2148d550c7e04a`
contracts/interfaces/IEscrowSrc.sol	`16a38077ec28b7493ab15476f6ecca1dab942a1d`
contracts/libraries/Clones.sol	`f3451766e41bed8ab0318cc027245ae5f0313d99`
contracts/libraries/ImmutablesLib.sol	`3bfbc11a14bbf036306afa2311e9630675e92418`
contracts/libraries/TimelocksLib.sol	`d25b5dc5a6cbf25c8e3e03d3d9a159f871c4ece0`

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.