OpenZeppelin | security

Cross Chain Swaps Diff Audit

September 9, 2024

Table of Contents

Summary

Type	DeFi
Timeline	From 2024-07-15 To 2024-07-20
Languages	Solidity

Total Issues	11 (6 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	5 (3 resolved)
Notes & Additional Information	5 (2 resolved)

Scope

We performed a diff audit of the 1inch/cross-chain-swap repository at commit b3acc0f against commit 49e31a8.

In scope were the following files:

```
./contracts/
├── BaseEscrow.sol
├── BaseEscrowFactory.sol
├── Escrow.sol
├── EscrowDst.sol
├── EscrowFactory.sol
├── EscrowSrc.sol
├── MerkleStorageInvalidator.sol
├── interfaces
│   ├── IBaseEscrow.sol
│   ├── IEscrow.sol
│   ├── IEscrowDst.sol
│   ├── IEscrowFactory.sol
│   ├── IEscrowSrc.sol
│   └── IMerkleStorageInvalidator.sol
├── libraries
│   ├── ImmutablesLib.sol
│   ├── ProxyHashLib.sol
│   └── TimelocksLib.sol
└── zkSync
    ├── EscrowDstZkSync.sol
    ├── EscrowFactoryZkSync.sol
    ├── EscrowSrcZkSync.sol
    ├── EscrowZkSync.sol
    ├── MinimalProxyZkSync.sol
    └── ZkSyncLib.sol
```

System Overview

The Cross-Chain Swap Protocol allows users to swap ERC-20 tokens from a certain source chain to ERC-20 tokens or native tokens on a certain destination chain. In addition to the functionalities described in the previous audit report, the system contains the following updates.

Partial Fills

Makers are able to express whether their order may be settled within multiple different fills instead of a single fill by a single resolver. Multi-fill orders are divided into a user-chosen number of equal portions which are assigned off-chain to resolvers to fill.

For the partial filling of an order, a separate set of `EscrowSrc` and `EscrowDst` contracts is created. For each portion, a secret is created by the user and distributed to the resolvers through an off-chain observer, which can then be used to create a pair of escrow contracts. These secrets are stored in a Merkle tree for each order and resolvers fill each portion of the order they validate. By doing this, they "use up" these secrets while proving that they exist in the Merkle tree.

zkSync Compatibility

The new version of the Cross-Chain Swap Protocol supports deployment on zkSync Era. zkSync Era is a Zero Knowledge (ZK) rollup that supports generalized EVM compatibility for the Ethereum blockchain. Thanks to this EVM compatibility, Solidity contracts written for the canonical EVM can be run on Layer 2 with only a few minor differences.

The escrow contracts incorporate changes to address one of these differences. More specifically, as the address derivation is different on zkSync Era, the deployment and computation of the minimal proxy's bytecode has been modified to support zkSync Era. As part of this diff audit, we assessed whether these adaptations of the escrow contracts result in the same behavior within the zkSync ecosystem compared to the escrow contracts deployed on the Ethereum mainnet.

General Refactorings

The codebase has been refactored aimed at supporting the partial fills and zkSync compatibility. In addition, some minor issues identified in the previous audit were addressed as part of this new scope. In order to facilitate efficient off-chain tracking of the contract state by off-chain tools, additional events have been added to the codebase.

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 abuse or misuse of the system by the privileged roles is quickly detected and prevented. Furthermore, the general recommendations listed in the previous audit report also remain applicable to the current scope.

- Ensure that all integration pieces are correctly implemented across all required chains. This foundational step is crucial for maintaining system integrity and preventing operational issues.
- Conduct a thorough internal "compatibility review" for all relevant dependencies (e.g., limit-order-protocol) that will be deployed on zkSync and upon which the cross-chain swap protocol relies. This review should validate that the contracts function as intended and that all of the specified functionalities do not utilize any features which act differently on zkSync Era compared to mainnet EVM.
- Update the diagram in the whitepaper to include the `SrcPublicWithdrawal` phase. This addition will provide a clearer understanding of the process and ensure that all phases are adequately documented.
- Enhance documentation to thoroughly explain all the steps involved in executing multi-fill orders. Clear and comprehensive documentation will assist users in understanding and correctly utilizing this functionality as well as successfully performing integrations with other protocols.
- Assume tokens are well-behaved and ensure that the 1inch team and resolvers conduct thorough due diligence when reviewing tokens before filling orders. Specifically, be cautious with tokens that have hooks (e.g., `ERC-777` tokens), tokens which are pausable, and tokens which charge fees, as these characteristics might lead to unintended behavior in the escrow contracts.
- 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 to provide extra

incentive for third parties to go ahead and finalize or cancel the trade if the original resolver is not acting properly. Consider evaluating the calculation of the safety deposit amount for partial fill orders. 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.

- For partial fill orders, when generating the Merkle tree of secrets, ensure that every index of each possible secret is generated. It may be tempting to shorten the Merkle tree (and thereby shorten the Merkle proving process) by not including all indices in the tree. However, in this case, if there is some problem with resolvers, such as downtime or unexpected lack of funds, any indices which do not exist in the tree will limit the ability of the order to be filled in parts. This may lead to an order being "stuck" unnecessarily.
- For partial fill orders, ensure that indices are revealed in order. In the case that indices are revealed out-of-order, resolvers may block other resolvers by "skipping" their assigned indices, leading to confusion and potentially to orders not being filled as expected.

High Severity

H-01 Usage of Native Tokens on Destination Chain Can Be Abused to Steal Safety Deposit

If the token for a cross-chain swap on the destination chain is `address(0)` (native token), it is possible to steal the safety deposit on the destination chain and block resolver funds temporarily. An attacker can achieve this by deploying a contract that does not accept the native token, pre-calculating its deployment address, and specifying it as the recipient. The contract can contain logic that checks the `tx.origin` of the transaction which sends native tokens to it, reverting whenever `tx.origin` is not the attacker's EOA.

When the resolver deploys `EscrowDst`, the `safetyDeposit` will be escrowed during the cross-chain swap. The attacker can then deploy the contract to the pre-calculated address, causing the subsequent native token transfer to revert when initiated by an address aside from theirs. The attacker can then await the public withdrawal phase, receiving the expected funds and taking the resolver's safety deposit. Note that if the attacker possesses a resolver address, they may configure the contract to only accept native tokens when that resolver is the original sender. However, this is unlikely as that resolver will be KYC'ed and may face consequences for such an attack. A proof of concept for this attack can be found here.

The attacker takes almost no risk as even if they do not receive the safety deposit, they still receive the swap tokens. In addition, the resolvers may struggle to detect this setup before it is too late as it will appear to be a normal transfer to an EOA until the contract is deployed. Moreover, this attack can be leveraged to temporarily lock up the safety deposits of resolvers by spamming the cross-chain swap protocol with malicious swaps. Resolvers have to wait until the cancellation period before they can retrieve their safety deposits.

Consider informing resolvers of this risk and instructing them to check the recipient of cross-chain swaps of native tokens, and only initiate a cross-chain swap if they trust the recipient. For example, EOAs which have a transaction history or contracts with no risk of this attack. Alternatively, consider disallowing cross-chain swaps which send native tokens or utilizing `CREATE` + `SELFDESTRUCT` to force the sending of native tokens (e.g., as done here).

Update: *Partially resolved in pull request #100 at commit 394a0d5.*

> The current version is using a low-level call to send the native tokens to the recipient on the destination chain. This ensures the current context does not revert in case the subcontext, executing the `receive` function of the recipient, reverts. However, an attacker could still force the current context to revert by implementing a "gas-intensive" loop in the `receive` function of the recipient contract. This might result in a transaction that requires more gas than the current block gas limit. Relatedly, if the gas consumed by the recipient contract is very high, it might become unprofitable for the taker to execute the transaction as the received `safetyDeposit` does not cover the gas costs.

Update: *Resolved in pull request #105 at commit 9d316f2.*

> The current version restricts the public withdrawal and cancel functions to access token holders. Only resolvers going through a KYC procedure will receive this access token. This resolves the safety deposit theft issue is as `publicWithdraw` can only be called by KYC'ed entities, which disincentives malicious behaviour.

Low Severity

L-01 Functions Are Updating the State Without Event Emissions

Within `MerkleStorageInvalidator.sol`, the `takerInteraction` function is updating the `lastValidated` mapping without an event emission.

Consider emitting events whenever there are state changes to make the codebase less error-prone and improve its readability.

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

> *Noted. We assume that it is possible to obtain data based on existing events.*

L-02 Sensitive Dependence on `block.timestamp` in `deploySrc` Flow

The `ResolverMock.deploySrc` function is designed as an example for setting up an `EscrowSrc` contract. The function first computes the address of the `EscrowSrc` to be created, then sends funds to that address, and finally deploys the escrow contract. If this were not the case, and instead these actions happened non-atomically, the `EscrowSrc` contract likely would not receive these funds. This is because the `EscrowSrc` deployment depends on the hash of `immutables` to determine its address, with `immutables` containing a reference to the `block.timestamp` in which it was deployed. The result of this is that the address of the `EscrowSrc` contract is not knowable for certain until after the transaction deploying it has been mined.

In a real-world blockchain environment, there are many reasons why the timestamp of a transaction is not knowable. For example, block ordering manipulation by miners, or an exceptionally fast block production time combined with high latency from the deployer, may make it difficult to target an exact block. Within the `ResolverMock` contract, the `block.timestamp` is queried and used when atomically funding and creating an `EscrowSrc` contract. However, the name "mock" suggests that this is not intended to be a part of the production codebase.

Consider standardizing the `ResolverMock` contract into a part of the production codebase. Additionally, consider documenting its functionality and noting the sensitive dependence on `block.timestamp` in the documentation. Consider renaming it, potentially to something like `Resolver` or `ResolverBase`. Doing these things will ensure that the `ResolverMock` contract is treated as an integral part of the codebase rather than an optional helper contract. Finally, consider requiring that resolvers implement this contract exactly as it is, with no modifications. This will ensure that all resolvers behave predictably, and make it easier for 1inch to verify resolver behavior by simply checking the bytecode of deployed resolver contracts. Note that the consequences of mis-implementing the `deploySrc` function could be sending funds to addresses from where they will not be recoverable.

Alternatively, consider refactoring the flow in which `EscrowSrc` contracts are deployed such that the resolver does not need to be aware of the `block.timestamp` to correctly compute the address of the `EscrowSrc`. This could be accomplished, for example, by removing the `block.timestamp` from the `salt` used to deploy the `EscrowSrc` or by implementing a hook to fund `EscrowSrc` within `_postInteraction`.

***Update:** Resolved in pull request #97 at commit 6d90144.*

L-03 Shortened Merkle Proof Attack in `takerInteraction`

The `takerInteraction` function in `MerkleStorageInvalidator` is called by the taker before calling the "post interaction". The function aims to update the index and secret hash for further use in `_postInteraction` within `BaseEscrowFactory`. Before saving the passed values, the Merkle proof is validated using `abi.encodePacked(idx, secretHash)` as the leaf, resulting in a total length of 64 bytes.

Since the intermediate nodes of the Merkle tree are also 64 bytes long using the same hash function, this setup allows for a potential shortened Merkle proof attack. Intermediate tree nodes can be passed as `idx` and `secretHash`, and the proof length can be reduced accordingly, still producing a valid proof. A secret gist was created to demonstrate the attack scenario. This opens up the possibility for an attack, though it is difficult to exploit since `idx` is checked for validity in `_postInteraction`.

Consider strictly checking the path length of the proof against the number of leaves to ensure the integrity of the Merkle proof validation process.

Update: *Resolved in pull request #95 at commit c1d7e3e.*

L-04 Assembly Block Marked as `memory-safe` Violates Solidity Memory Model

In the `hash` function of the `ImmutablesLib` library and in the `computeAddressZkSync` function of the `ZkSyncLib` library, assembly blocks are used that are marked as `memory-safe` but violate the Solidity memory model.

The issue arises because the free memory pointer (FMP) is used to write data to the memory location it points to, but the FMP is not incremented once the data is saved. Consequently, the FMP continues to point to a memory location that has already been used. This can result in the compiler using the memory to optimize processes which may affect data that has been placed there or cause the compiler to use the data stored in the assembly block. Such issues can lead to unnoticed errors.

Consider incrementing the FMP by the amount of memory used.

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

> *Not an issue. We do not consider free memory to be zeroed out. Solidity documentation warns about the possibility of this situation.*

L-05 Timelocks Library Returns Invalid Time for `DeployedAt` Stage

In the `get` function of the `TimelocksLib` library, requesting `DeployedAt` returns an incorrect time value of `deployedAt * 2`. The issue arises because the bit shift calculation and the subsequent addition result in an incorrect doubling of the `deployedAt` value.

Currently, the `get` function with `stage == DeployedAt` is not used in the codebase, so the risk is more potential than real. However, if this function is invoked in the future, it could lead to incorrect time calculations, potentially affecting any time-dependent logic or features relying on this value.

Consider revising the return statement to avoid the erroneous multiplication effect.

***Update:** Resolved in pull request #101 at commit 5f4c1f51. The `DeployedAt` stage has been removed.*

Notes & Additional Information

N-01 Floating Pragma

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

Throughout the codebase, multiple instances of floating pragma directives were identified:

- `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.

Consider using fixed pragma directives.

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

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

N-02 Lack of Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, several instances of contracts without a security contact were identified:

- The `Escrow` abstract contract
- The `EscrowDst` contract
- The `EscrowFactory` contract
- The `EscrowSrc` contract
- The `IEscrow` interface
- The `IEscrowDst` interface
- The `IEscrowFactory` interface
- The `IEscrowSrc` interface
- The `IMerkleStorageInvalidator` interface
- The `ImmutablesLib` library
- The `MerkleStorageInvalidator` contract
- The `ProxyHashLib` library
- The `TimelocksLib` library

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Resolved in pull request #98 at commit 474b704.*

N-03 Missing Named Parameters in Mapping

Since Solidity 0.8.18, developers can utilize named parameters in mappings. This means mappings can take the form of `mapping(KeyType KeyName? => ValueType ValueName?)`. This updated syntax provides a more transparent representation of a mapping's purpose.

In the `lastValidated` state variable of the `MerkleStorageInvalidator` contract, there are no named parameters in the mapping.

Consider adding named parameters to mappings in order to improve the readability and maintainability of the codebase.

Update: *Resolved in pull request #99 at commit 9e033fa.*

N-04 Unclear Naming

The `makingAmount` parameter of `_postInteraction` is too similar to `order.makingAmount` and may cause confusion.

To improve code clarity, consider renaming `makingAmount` to help distinguish it from `order.makingAmount`.

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

> *Noted. The `_postInteraction` function overrides the Limit Order Settlement function, preserving the parameter names from the protocol.*

N-05 No Masking of Addresses in `ProxyHashLib` Library

The `computeProxyBytecodeHash` function of the `ProxyHashLib` library is used to calculate the hash of the proxy bytecode in a gas-efficient way. The library uses assembly for efficiency. However, the `implementation` argument is directly copied to memory without being masked in the assembly block. Note that because the library functions are called internally in the codebase, the function calls are replaced by `JUMP` instructions but no bit cleaning operation is inserted by the compiler. Depending on the context in which the library is used, this may result in the calculation of an invalid bytecode hash. However, there are no instances in the codebase where this would pose a risk.

For example, an address with dirty upper bits could be computed using assembly, after which the assembly block could be closed and `computeProxyBytecodeHash` function is called. Since `implementation` is used with the `or` operation in the function, this may lead to an incorrect result for the bytecode hash, which would then lead to an incorrect address for `EscrowSrc` and `EscrowDst`.

A similar issue was recently identified and fixed in the `Clones` contract of the OpenZeppelin contracts library. The cross-chain swap protocol relies on this `Clones` library to deploy a minimal proxy for the escrow contracts. Therefore, we recommend ensuring that the `computeProxyBytecodeHash` function is using the same method to compute the proxy's bytecode as the one used by the `Clones` library such that there is no discrepancy between the computed and actual address of the escrow contract, which could potentially result in a loss of funds.

As the protocol is currently using an older version of the OpenZeppelin contracts library where this issue has not been fixed, it is consistent with the current implementation of the `computeProxyBytecodeHash` function. However, if this issue is to be resolved, we recommend also updating the OpenZeppelin contracts library to a newer version where this similar issue in the `Clones` library has been fixed.

As libraries can be used in a wide variety of contexts and should aim to work well in all of them, consider resolving this issue by clearing the upper bytes for the `implementation` argument in the `computeProxyBytecodeHash` function as well as updating the OpenZeppelin contracts library to a newer version.

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

> *Not an issue. We follow the rule to return clean bits, so we also expect values with clean bits as input.*

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. The current version of the protocol introduces partial fill orders and enables compatibility with zkSync Era.

One high-severity issue was identified which allowed users to steal the safety deposit of resolvers. In addition, several recommendations were made to make the codebase more robust.

The 1inch 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 following our best practices recommendations.