AstraSec

1inch Cross-chain Swap V2

Security Audit Report

August 12, 2024

Contents

1 | Introduction

1.1 1inch Cross-chain Swap V2

`1inch Cross-chain Swap` introduces a two-party atomic swap mechanism, optimized for EVM-compatible chains with an option to execute a swap by a single party as an ordinary limit order. This reduces centralization, improves efficiency and security over existing bridging methods and enables direct transactions between users and market makers (resolvers) across chains. The audited `1inch Cross-chain Swap V2` builds upon the V1 by adding the partial fill order functionality, making transactions more flexible and efficient.

1.2 Source Code

The following source code was reviewed during the audit:

- https://github.com/1inch/cross-chain-swap.git
- Commit ID: b3acc0f

And this is the final version representing all fixes implemented for the issues identified in the audit:

- https://github.com/1inch/cross-chain-swap.git
- Commit ID: ef6956a

2 | Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the `1inch Cross-chain Swap V2` project Throughout this audit, we identified a total of 2 issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	-	-	-	-
High	-	-	-	-
Medium	1	-	-	1
Low	-	-	-	-
Informational	1	1	-	-
Undetermined	-	-	-	-

3 Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

~~M-1~~	Potential Hashlock Replay in BaseEscrowFactory::_postInteraction()
I-1	Enforcing Consistent Solidity Compiler Version

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

Severity	Description
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and remediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

3.3 Vulnerability Details

[M-1] Potential Hashlock Replay in BaseEscrowFactory::_postInteraction()

Target	Category	IMPACT	LIKELIHOOD	STATUS
BaseEscrowFactory.sol	Business Logic	Medium	Medium	Addressed

In the `1inch Cross-chain Swap V2` protocol, the partial fill order mechanism requires the maker to provide multiple secrets to generate different secret hashes for different escrows, with the Merkle root of the secret hashes stored in the signed order information. During the order matching process, the `BaseEscrowFactory::_postInteraction()` function creates the escrow contract on the source chain to hold the maker's assets.

When an order allows partial fills, the hashlock used to generate the escrow contract is taken from `lastValidated[key]` (lines 72 - 73), which should be updated in the `MerkleStorageInvalidator::takerInteraction()` function. However, if `MerkleStorageInvalidator::takerInteraction()` does not update `lastValidated[key]` correctly (line 59), the hashlock from the previous transaction may be reused.

This reuse of the hashlock is critical because the secret associated with it would have already been exposed in the previous transaction. Using a known secret for a new cross-chain swap compromises the security of the protocol.

BaseEscrowFactory::_postInteraction()

```
55 function _postInteraction(
56     IOrderMixin.Order calldata order,
57     bytes calldata extension,
58     bytes32 orderHash,
59     address taker,
60     uint256 makingAmount,
61     uint256 takingAmount,
62     uint256 remainingMakingAmount,
63     bytes calldata extraData
64 ) internal override(ResolverValidationExtension) {
65     ...
66     bytes32 hashlock;

68     if (MakerTraitsLib.allowMultipleFills(order.makerTraits)) {
69         uint256 secretsAmount = uint256(extraDataArgs.hashlock) >> 240;
70         if (secretsAmount < 2) revert InvalidSecretsAmount();
71         bytes32 key = keccak256(abi.encodePacked(orderHash, uint240(uint256(
               extraDataArgs.hashlock))));
72         LastValidated memory validated = lastValidated[key];
73         hashlock = validated.leaf;
```

```
        uint256 calculatedIndex = (order.makingAmount - remainingMakingAmount +
            makingAmount - 1) * secretsAmount / order.makingAmount;
        if (
            (calculatedIndex + 1 != validated.index) &&
            (calculatedIndex + 2 != validated.index  remainingMakingAmount !=
                makingAmount)
        ) revert InvalidSecretIndex();
    } else {
        hashlock = extraDataArgs.hashlock;
    }

    ...
}
```

MerkleStorageInvalidator::takerInteraction()

```
function takerInteraction(
    IOrderMixin.Order calldata /* order */,
    bytes calldata /* extension */,
    bytes32 orderHash,
    address /* taker */,
    uint256 /* makingAmount */,
    uint256 /* takingAmount */,
    uint256 /* remainingMakingAmount */,
    bytes calldata extraData
) external onlyLOP {
    (
        bytes32 root,
        bytes32[] memory proof,
        uint256 idx,
        bytes32 secretHash
    ) = abi.decode(extraData, (bytes32, bytes32[], uint256, bytes32));
    bytes32 key = keccak256(abi.encodePacked(orderHash, uint240(uint256(root))))
        ;
    if (idx < lastValidated[key].index) revert InvalidIndex();
    if (!proof.verify(root, keccak256(abi.encodePacked(idx, secretHash))))
        revert InvalidProof();
    lastValidated[key] = LastValidated(idx + 1, secretHash);
}
```

Remediation Apply necessary mechanism to prevent potential hashlock replay.

[I-1] Enforcing Consistent Solidity Compiler Version

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Coding Practices	N/A	N/A	Acknowledged

While examining the current implementation, we observe the use of different compiler versions within the implementation. This practice can introduce potential security risks and compatibility issues, as various compiler versions may implement different optimizations and security features. To ensure consistency, maintainability, and security of the code, it is recommended to standardize on a single specified compiler version. Clearly declare this version at the beginning of the contract file (e.g., pragma solidity 0.8.23;). This approach helps mitigate unforeseen errors and vulnerabilities arising from compiler version discrepancies.

Contracts	Compiler Version
IBaseEscrow.sol, IEscrow.sol, IEscrowDst.sol, IEscrowFactory.sol, IEscrowSrc.sol	^0.8.0
ImmutablesLib.sol, ProxyHashLib.sol, TimelocksLib.sol	^0.8.20
Others	0.8.23

Remediation It is recommended to use a fixed Solidity compiler version in the current implementation.

4 | Appendix

4.1 About AstraSec

AstraSec is a blockchain security company that serves to provide high-quality auditing services for blockchain-based protocols. With a team of blockchain specialists, AstraSec maintains a strong commitment to excellence and client satisfaction. The audit team members have extensive audit experience for various famous DeFi projects. AstraSec's comprehensive approach and deep blockchain understanding make it a trusted partner for the clients.

4.2 Disclaimer

The information provided in this audit report is for reference only and does not constitute any legal, financial, or investment advice. Any views, suggestions, or conclusions in the audit report are based on the limited information and conditions obtained during the audit process and may be subject to unknown risks and uncertainties. While we make every effort to ensure the accuracy and completeness of the audit report, we are not responsible for any errors or omissions in the report.

We recommend users to carefully consider the information in the audit report based on their own independent judgment and professional advice before making any decisions. We are not responsible for the consequences of the use of the audit report, including but not limited to any losses or damages resulting from reliance on the audit report.

This audit report is for reference only and should not be considered a substitute for legal documents or contracts.

4.3 Contact

Phone	+86 176 2267 4194
Email	contact@astrasec.ai
Twitter	https://twitter.com/AstraSecAI