Smart Contract Security Audit Report

1inch

1. Contents

2. General Information

This report contains information about the results of the security audit of the 1inch (hereafter referred to as "Customer") smart contracts, conducted by Decurity in the period from 03/04/2024 to 03/15/2024.

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 2e0fafdddb59ba40d8d346b51765910abd3474ed). The re-tests were done for the commits 5158cb87c4338007fc5a0aa2df0cbc992953d191 and 13b76764fc83b85b1672abd9fac60ee87dbd210f.

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 haven't discovered any exploitable security issues.

The 1inch team has given feedback for the suggested changes and an explanation for the underlying code.

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of March 15, 2024.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
validateImmutables won't work on zkSync	contracts/Escrow.sol	**High**	Acknowledged
Incorrect modifier used	contracts/EscrowDst.sol	**Medium**	Fixed
Integer overflow in TimelocksLib	contracts/libraries/TimelocksLib.sol	**Medium**	Fixed
Lack of event emitting	contracts/EscrowSrc.sol contracts/EscrowDst.sol	**Low**	Fixed
Maker can withdraw from EscrowDst	contracts/EscrowDst.sol	**Low**	Fixed
An incorrect deployment timestamp may be returned	contracts/libraries/TimelocksLib.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. validateImmutables won't work on zkSync

Risk Level: High

Status: Won't fix. We will not deploy contracts that use the Create2.computeAddress library function to any chain that has a different implementation of `create2`. These are EscrowFactory (`addressOfEscrowSrc/addressOfEscrowDst`) and Escrow (`_validateImmutables`).

Contracts:

- contracts/Escrow.sol

Description:

There is a `_validateImmutables()` function that checks the passed immutable value. This function uses the OpenZeppelin's `Create2.computeAddress()` method to calculate the address and compares it to the current address.

This will work fine as long as the chain has the default algorithm for calculating addresses in the `create2` opcode.

For example, the zkSync chain that has a modified `create2` opcode:

```
export function create2Address(sender: Address, bytecodeHash: BytesLike, salt:
BytesLike, input: BytesLike) {
  const prefix =
ethers.utils.keccak256(ethers.utils.toUtf8Bytes("zksyncCreate2"));
  const inputHash = ethers.utils.keccak256(input);
  const addressBytes = ethers.utils.keccak256(ethers.utils.concat([prefix,
ethers.utils.zeroPad(sender, 32), salt, bytecodeHash, inputHash])).slice(26);
  return ethers.utils.getAddress(addressBytes);
}
```

In this way, all the EscrowDst contracts that will be deployed on the zkSync chain won't work because the `_validateImmutables()` will revert.

This will lead to the fact that all funds sent to these contracts being stuck there.

Remediation:

Consider modifying the library for chains with different `create2` opcode behaviour.

References:

- https://docs.zksync.io/build/developer-reference/differences-with-ethereum.html#create-create2

5.2. Incorrect modifier used

Risk Level: Medium

Status: Introduced in commit 5158cb. Fixed in commit 13b767.

Contracts:

- contracts/EscrowDst.sol

Location: Lines: 55. Function: `publicWithdraw`.

Description:

The update introduced the `publicWithdraw` function that is used to withdraw funds from `EscrowDst` contract when `DstPublicWithdrawal` timestamp is passed.

```
contracts/EscrowDst.sol:
  51:     function publicWithdraw(bytes32 secret, Immutables calldata
immutables)
  52:         external
  53:         onlyValidImmutables(immutables)
  54:         onlyValidSecret(secret, immutables)
  55:
onlyAfter(immutables.timelocks.get(TimelocksLib.Stage.DstPublicWithdrawal))
  56:     {
```

The `onlyAfter` modifier checks that public withdrawal period is started but it's not checks that it finished.

It may be a problem if relayer didn't share secret with resolvers and taker didn't withdraw their funds from `EscrowDst` via `cancel` function before private cancellation period started on the `EscrowSrc` contract. In that case maker will be able to call `publicWithdraw` function on the `EscrowDst` contract and retrieve his funds from `EscrowSrc` contact.

Remediation:

Consider using `onlyBetween` modifier instead of `onlyAfter`.

5.3. Integer overflow in TimelocksLib

Risk Level: Medium

Status: Fixed in the pull request.

Contracts:

- contracts/libraries/TimelocksLib.sol

Description:

The _get function of the `TimelocksLib` library calculates the values by adding a specific value to the base date which is stored in the lower bits. The base value is announced as a uint256 variable, and the resulting value is truncated to fit in 32 bits. As a result, there're no integer overflow guards which makes it possible to pass a specially crafted large value in any offset which will end up in the past.

The resolvers who don't account for the overflow and only check the absolute values of the timelocks may be tricked to accept an order with undesired parameters.

A malicious maker can specify a low value in `_SRC_WITHDRAWAL_OFFSET` bits of the Timelock value and a large value in the `_DST_FINALITY_OFFSET`, `_DST_WITHDRAWAL_OFFSET`, and `_SRC_CANCELLATION_OFFSET` bits. Then they can immediately withdraw the funds on both sides and receive their own funds, the funds of the resolver, and both safety deposits.

The following test case can be added to `Escrow.t.sol`:

```
    function test_Pwn() public {
        vm.warp(1710159521); // make it real, it's 0 in foundry

        srcTimelocks = SrcTimelocks({ finality: 120, withdrawal: 1, cancel:
2584807817 });
        dstTimelocks = DstTimelocks({ finality: 2584807817, withdrawal:
2584807817, publicWithdrawal: 1 });
        _setTimelocks();

        // deploy escrow
        (
            IOrderMixin.Order memory order,
            bytes32 orderHash,
            bytes memory extraData,
            /* bytes memory extension */,
            IEscrow srcClone,
            IEscrow.Immutables memory immutables
        ) = _prepareDataSrc(SECRET, MAKING_AMOUNT, TAKING_AMOUNT,
```

```
SRC_SAFETY_DEPOSIT, DST_SAFETY_DEPOSIT, address(0), true);

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

        // dst
        (IEscrow.Immutables memory immutablesDst, uint256
srcCancellationTimestamp, IEscrow dstClone) = _prepareDataDst(
            SECRET, TAKING_AMOUNT, alice.addr, bob.addr, address(dai)
        );

        // deploy escrow
        vm.prank(bob.addr);
        escrowFactory.createDstEscrow{ value: DST_SAFETY_DEPOSIT
}(immutablesDst, srcCancellationTimestamp);

        vm.prank(address(limitOrderProtocol));
        escrowFactory.postInteraction(
            order,
            "", // extension
            orderHash,
            bob.addr, // taker
            MAKING_AMOUNT,
            TAKING_AMOUNT,
            0, // remainingMakingAmount
            extraData
        );

        uint256 balanceAliceSrc = usdc.balanceOf(alice.addr);
        uint256 balanceAliceDst = dai.balanceOf(alice.addr);
        uint256 balanceAliceNative = alice.addr.balance;

        // withdraw
        vm.startPrank(alice.addr);
        dstClone.withdraw(SECRET, immutablesDst);
        skip(1); // 1 second later because
dst.publicWithdrawal=src.withdrawal=1
        srcClone.cancel(immutables);

        assertEq(alice.addr.balance, balanceAliceNative + SRC_SAFETY_DEPOSIT +
DST_SAFETY_DEPOSIT);
        assertEq(usdc.balanceOf(alice.addr), balanceAliceSrc + MAKING_AMOUNT);
        assertEq(dai.balanceOf(alice.addr), balanceAliceDst + TAKING_AMOUNT);
    }
```

Remediation:

Implement the overflow check in `_get`.

5.4. Lack of event emitting

Risk Level: Low

Status: Fixed in the pull request.

Contracts:

- contracts/EscrowSrc.sol
- contracts/EscrowDst.sol

Description:

There are no events emitted in the `withdraw()`, `cancel()` functions in the `EscrowSrc` and `EscrowDst` contracts.

Remediation:

Emit events for all important actions. Specifically, emit an event containing the secret to make it easier for the resolvers to extract the secret in case if a maker initiates the action.

5.5. Maker can withdraw from EscrowDst

Risk Level: Low

Status: Fixed in the commit 80f734.

Contracts:

- contracts/EscrowDst.sol

Description:

There is a `withdraw()` function in the `EscrowDst` contract that withdraws the `amount` to the maker, which should be called by the resolver when the relayer verifies that the swap is correct.

Unfortunately, if something goes wrong and the swap does not happen, the maker can initiate the swap themselves and receive the tokens from `EscrowDst` with the resolvers' safety deposit.

If the resolver does not extract the secret from the maker's transaction to `EscrowDst` on time, before the private cancellation starts, then the maker will receive their tokens on the src chain (`EscrowSrc`).

Remediation:

Consider mentioning this case in the documentation so that resolvers are warned about and emit events with secrets to make it easier for them to extract the secret from the chain.

Additionally, provide a testing suite for validating the orders.

5.6. An incorrect deployment timestamp may be returned

Risk Level: Info

Status: Introduced in commit [5158cb](5158cb).

Contracts:

- contracts/libraries/TimelocksLib.sol

Description:

The update introduced a refactored version of the `TimelocksLib` librariy which is used for compact storage of timelocks in a `uint256`. It 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.

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.