

Produced for



by



Contents

1	Executive Summary	3
2	Assessment Overview	5
3	Limitations and use of report	7
4	Terminology	8
5	Findings	9
6	Resolved Findings	10
7	Notes	13

1 Executive Summary

Dear 1inch team,

Thank you for trusting us to help 1inch with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Delegation according to [Scope](#) to support you in forming an opinion on their security risks.

The project reviewed features two implementations of delegation pods for ERC20Pods.

The most critical subjects covered in our audit are functional correctness, integrability and consistency of the accounting. General subjects covered include the documentation which is non-existing. Security regarding all the aforementioned subjects is satisfactory.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

-Severity Findings	0
-Severity Findings	2
•	2
-Severity Findings	1
•	1
-Severity Findings	4
•	3
•	1



2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the `contracts` folder of the Delegation repository. No documentation for the smart contracts reviewed was available. Following files from the repository `contracts` folder were part of the assessment scope:

```
interfaces/IDelegatedShare.sol
interfaces/IDelegationPod.sol
interfaces/IRewardableDelegationPod.sol
BasicDelegationPod.sol
DelegatedShare.sol
RewardableDelegationPod.sol
```

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	22 November 2022	5b4067b3dc270ae9edff57687c1cd68ffbfa25e7	Initial Version
2	05 December 2022	9a243d64422c41b0631617466deeb85dcd92e57c	Version 2

For the solidity smart contracts, the compiler version `0.8.17` was chosen.

2.1.1 Excluded from scope

The base `ERC20Pods` contract has been reviewed separately and is not part of this report. `RewardableDelegationPod` requires a separate Pod contract handling the actual reward distribution, such a contract has not been part of this review.

2.2 System Overview

This system overview describes the initially received version () of the contracts as defined in the [Assessment Overview](#).

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Delegation features two pod implementations for ERC20 tokens supporting the `ERC20Pods` extension.

An ERC20 token implementing the pods extensions pushes changes of balances to the registered pods of the involved accounts. This allows the pod contract to seamlessly keep track of involved accounts balances.

BasicDelegationPod

This pod allows a user to delegate his balance to another account using function `delegate`. Balances delegated to an account are tracked by minting or burning ERC20 tokens accordingly for this delegatee. These tokens are used for accounting only, transfers and approvals have been disabled.

Users must register for the `BasicDelegationPod` in the `ERC20Pods` contract and call `delegate()` on the `BasicDelegationPod`.

RewardableDelegationPod

Extension of BasicDelegationPod. Allows delegates to distribute rewards to users having delegated to them. When delegating the base ERC20Pods token users receive delegation share tokens (which also supports the ERC20Pods standard). On this token they register for the delegates Rewards pod (called farm contract) which is in charge to distribute the actual rewards.

Delegates must first register within the RewardableDelegationPod to add or deploy a shares token. If an existing token is added this token must adhere to the ERC20Pods standard and give minting/burning rights to the RewardableDelegationPod. To deploy a new shares token the DelegatedShare contract, an ERC20Pods token with transfer/allowance functionality disabled is used. Only after a delegatee has register user can delegate to this delegatee.

Upon balance updates first the accounting for the delegated amount is updated (see BasicDelegationPod), in addition the user delegating gets delegation shares minted.

2.2.1 *Trust Model & Roles*

Owner: Fully trusted.

Users: Untrusted.

Delegates: Fully trusted to operate the RewardsPod as expected and to pay out rewards accordingly. If an existing shares contract is added the delegatee is responsible for the correctness.

Interacting contracts such as the Farms or the Shares token used in RewardableDelegationPod: Fully trusted.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High			
Medium			
Low			

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the [Resolved Findings](#) section. The findings are split into these different categories:

- : Related to vulnerabilities that could be exploited by malicious actors
- : Architectural shortcomings and design inefficiencies
- : Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

-Severity Findings	0
-Severity Findings	0
-Severity Findings	0
-Severity Findings	1

- [Consistency on Zero Amount Transfers](#)

5.1 Consistency on Zero Amount Transfers

The function `BasicDelegationPod._updateBalances` does not trigger mint/burn/transfer of delegation shares (an ERC20Pods token) on 0 amount. The [ERC20 standard](#) specifies Note Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event..

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the [Findings](#) section.

Below we provide a numerical overview of the identified findings, split up by their severity.

-Severity Findings	0
--------------------	---

-Severity Findings	2
--------------------	---

- [Possible Frontrunning on Registration](#)
- [Pod.updateBalances\(\) Cannot Transfer ERC20Pods](#)

-Severity Findings	1
--------------------	---

- [Allowances Not Completely Disabled](#)

-Severity Findings	3
--------------------	---

- [Broken C-E-I Pattern](#)
- [Inconsistency and Zero Address Check on register\(\)](#)
- [No Event upon Registering Delegatee](#)

6.1 Possible Frontrunning on Registration

If a delegatee already deployed its DelegatedShare contract on its own and want to register it with `register(IDelegatedShare shareToken, address defaultFarm)`, another user could front run the transaction and register the already deployed contract in place of the true delegatee, who won't be able to register the contract for itself.

This can become problematic if the DelegatedShare contract already has some accounting done.

Code corrected:

The `register(IDelegatedShare shareToken, address defaultFarm)` function has been removed.

6.2 Pod.updateBalances() Cannot Transfer ERC20Pods

`Pod.updateBalances()` cannot transfer (including mint or burn) tokens of another ERC20Pods (with at least one pod involved): Using the default implementation of ERC20Pods the call to `updateBalances()` of a pod is executed with `_POD_CALL_GAS_LIMIT` amount of gas. Currently this value is hardcoded to `200_000`. A transfer of an ERC20Pods within `updateBalances()` would trigger `_updateBalances()` of this ERC20Pods. The current call executing with this amount of gas cannot forward another `200_000` gas and hence the execution reverts.

```
function _updateBalances(address pod, address from, address to, uint256 amount) private {
    bytes4 selector = IPod.updateBalances.selector;
    bytes4 exception = InsufficientGas.selector;

    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)

        if lt(div(mul(gas(), 63), 64), _POD_CALL_GAS_LIMIT) {
            mstore(0, exception)
            revert(0, 4)
        }
        pop(call(_POD_CALL_GAS_LIMIT, pod, 0, ptr, 0x64, 0, 0))
    }
}
```

The design of RewardableDelegationPod however requires this and hence cannot work.

Within RewardableDelegationPod.updateBalances() the call to the DelegatedShare token (which is ERC20Pods, and the accounts are connected to at least the farm pod) is wrapped within try/catch. Despite the function itself being annotated with best effort of having consistent shares the accounting is totally off. A transfer of the underlying ERC20Pod which triggers RewardableDelegationPod.updateBalances() will never successfully execute registration[_delegate].burn(from, amount)/ registration[_delegate].mint(from, amount). These calls only succeed when updateBalances() is called with sufficient gas, e.g. using DelegatedShare.addPod().

Code corrected:

The root of the issue has been addressed in ERC20Pods. The amount of gas for each of the calls in ERC20Pods._updateBalances() is no longer hardcoded in ERC20Pods but passed as constructor parameter. The ERC20Pods that is DelegatedShare has a fixed 100_000 gas for each of the callbacks. When two ERC20Pods are stacked like in ERC20Pods->RewardableDelegationPod->ERC20Pods``, developers must be careful to set the correct amount of gas in each of them for the system to work.

The function RewardableDelegationPod.updateBalances has been updated to call mint()/burn() without try/catch blocks so that every call to DelegatedShare.mint()/burn() must be successful.

6.3 Allowances Not Completely Disabled

BasicDelegationPod overwrites and inhibits functions transfer, transferFrom and approve. The increaseAllowance and decreaseAllowance functions inherited from OpenZeppelin's ERC20 implementation are not overridden and hence can be used.

Code corrected:

The functions increaseAllowance and decreaseAllowance have been explicitly disabled.

6.4 Broken C-E-I Pattern

The check-effects-interaction pattern is in function `BasicDelegationPod.delegate`. The delegated mapping is updated after a possible contract interaction upon `_updateAccountingOnDelegate`, this could lead to reentrancy or other unexpected behaviors.

Code corrected:

The mapping update and event have been moved before the call to `_updateAccountingOnDelegate`.

6.5 Inconsistency and Zero Address Check on `register()`

In function `register(IDelegatedShare shareToken, address defaultFarm)`, it is possible to provide `shareToken=address(0)`, this would allow one user to add a default farm for the zero address, and to call on of the `register()` functions once again, which should not be possible.

Code corrected:

The `register(IDelegatedShare shareToken, address defaultFarm)` function has been removed.

6.6 No Event upon Registering Delegatee

Events are used to be informed of or to keep track of transactions changing the state of a contract. Generally, any important state change should emit an event.

Both functions used to register delegatees do not emit an event, hence for an observer it's hard to track new delegatees.

Code corrected:

Two events `RegisterDelegatee` and `DefaultFarmSet` have been added, and are emitted resp. when a new delegatee registers, and when a default farm is added.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Users Must Add Farm if Default Farm Is Updated

The deployed DelegatedShare contracts may not have a farm associated with them directly. If a farm is added later on, the users must either re-delegate or manually add the farm Pod on the DelegatedShare contract themselves.

It's possible for an user to remove himself from the default farm using `DelegatedShare.remove/removeAll()` but still keep delegating to this delegatee. Users must be careful and understand the consequences of their actions.