PUBLIC

Code Assessment

of the Farming Smart Contracts

March 29, 2022

Produced for

by

Contents

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 Farming according to Scope to support you in forming an opinion on their security risks.

1inch implements two types of farming contracts. While the first one is a traditional farming contract where tokens need to be deposited for reward eligibility, the second one is as ERC-20 library contract which has farming capabilities built-in and, thus, allows for participating in multiple farms without requiring individual deposits in each one.

The most critical subjects covered in our audit are functional correctness, dependency on external contracts, and precision of arithmetic operations. Security regarding all the aforementioned subjects is high.

The general subjects covered are usage as a library, code complexity, documentation, specification, and gas efficiency. In general, these subjects are satisfactory. However, specification and documentation are non-existing, see Insufficient documentation, while code complexity is high due to complex control flows. That makes understanding the system and integrating with it difficult.

In summary, we find that the codebase provides an good level of security.

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.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	0
Low-Severity Findings	10
• Code Corrected	7
• Code Partially Corrected	1
• Risk Accepted	1
• Acknowledged	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 Farming repository based on the documentation files. All files in `contracts` are in scope.

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

V	Date	Commit Hash	Note
1	17 January 2022	d191cc14b526e2eab09c266580b84116b2630da8	Initial Version
2	14 February 2022	879448d1a934f767ee2b0c5a26ee1458db5f9986	Second Version
3	28 March 2022	f41b029b8da4819c28129d1d06995033cf82b7a7	Final Version

For the solidity smart contracts, the compiler version `0.8.9` was chosen. In the second iteration the compiler version has been changed to `0.8.11`. In the third iteration the compiler version has been changed to `0.8.12`.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

1inch offers a set of contracts for farming ERC-20 tokens. First, 1inch implements a farming pool which allows users to lock a staking token to accrue a reward token over time. Second, an ERC-20 extension having farming capabilities built-in is provided.

2.2.1 General

- Farmable token: An ERC-20 token that allows token holders to claim some reward tokens from farms. A farmable token contract defines the logic of how rewards are split among users.
- Farms: Farms hold the reward token funds and define the functionality on how to distribute the token to the contract of the farmable token. A farm defines the logic of how to release rewards to farmable tokens.

2.2.2 Farming Pool

FarmingPool is both a farmable token and a farm.

- The farmable token (FarmingPool) is always one-to-one exchangeable (`deposit()` and `withdraw()`) with a staking token. Hence, to be eligible for rewards users must lock a pre-defined staking token in the FarmingPool contract. However, the share of the totally released reward amount is defined as a function of the amount staked over the staking time. Users can always claim their rewards with `claim()` or when calling `exit()` which also withdraws all the staking token funds the user deposited.
- FarmingPool is a farm itself. Farming must be initialized through `startFarming()` by the reward distributor which specifies the parameters of how much additional reward will be released over time

to all the token holders. `_getFarmedSinceCheckpointScaled()` defines the distribution to be linearly increasing over time.

2.2.3 Farmable ERC-20 and Farms

- ERC20Farmable is an extension to the ERC-20 standard introducing native farming capabilities to tokens and is intended to be used as a library. Instead of having another staking token, the staking token is the ERC20Farmable itself. Hence, there is no need to lock tokens into one farming contract and, thus, ERC20Farmable allows to participate in multiple farms at once. Since anyone could create a token farm, users must explicitly join a farm through `farm()` to be eligible for its rewards (farming with all the ERC20Farmable balance). Similarly, users would need to exit a farm with `exit()`. In contrast to FarmingPool, rewards are not claimed automatically but can also be claimed with `claim()` at any point in time. For example, with such an approach LP tokens could inherit from ERC20Farmable to allow incentives coming from multiple third parties (e.g. AMM LP token pairs) without locking the token into one contract.
- Farm implements the same logic for distribution to the farmable token contract. However, farm serves as a template but could be replaced with different logic in terms of distribution. However, a Farm must always implement the `claimFor` method to allow the farmable token to pull funds from the farm.

2.2.4 Trust Model & Roles

User: Not trusted.

Farm Owner and active farms: Not trusted. Anyone could deploy farming contracts. We further assume that farms return values that are in the same base as defined by the FarmAccounting library.

Farmable token deployer: Trusted. User holds this token and interacts with its ecosystem. Hence, the ecosystem of the farmable token is trusted.

2.2.5 Changes in version 2:

Besides the fixes, functions have been renamed. Most interestingly:

- `ERC20Farmable.farm()` has been renamed to `ERC20Farmable.join()`
- `ERC20Farmable.exit()` has been renamed to `ERC20Farmable.quit()`

Additionally, batched operations have been introduced for quitting and claiming in ERC20Farmable.

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	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	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:

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

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	3

5.1 Inefficient Transfer Hook

Design Low Version 1 Risk Accepted

The internal function `ERC20Farmable.beforeTokenTransfer` is called before any transfer logic is executed. Assuming that user A is farming on `n` and user B is farming on `m` farms without any overlap in the sets, then,

- `m+n` addresses are loaded from storage at the very beginning,
- `m+n` external calls are made,
- `m+n` storage writes to `corrections`,
- and more reads and writes.

Furthermore, `m` and `n` are not limited. A token transfer could end up being very expensive without the user noticing. Hence, token transfers could easily fail by running out of gas.

Risk accepted:

1inch accepts the risk.

5.2 Insufficient Documentation

Design Low Version 1 Acknowledged

Documentation helps users, developers and others to understand a system in a shorter amount of time. Especially if the code is to be used as a library by other developers, it could help these to prevent errors. Otherwise, no assumptions on the code and its behavior can be made.

Currently, code is undocumented. Furthermore, no behavioral description is provided on what to expect from a function call. For example, it is undocumented how the libraries handle errors in external contracts.

Acknowledged:

1inch replied:

```
Will be improved in the future.
```

5.3 Lack of Events

Design | Low | Version 1 | Code Partially Corrected

Typically, events help track the state of the smart contract. Some functions, such as `startFarming`, emit events while others do not emit any event. Some examples lacking event emissions are:

- `ERC20Farmable.farm()`
- `ERC20Farmable.claim()` and `FarmingPool.claim()`
- `ERC20Farmable.exit()`
- Public checkpointing functions
- `BaseFarm.setDistributor()`

Code partially corrected:

An event has been added only for `setDistributor()`.

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.

Critical-Severity Findings	0
High-Severity Findings	1

- Gas Griefing Code Corrected

Medium-Severity Findings	0
Low-Severity Findings	7

- Commented Code Code Corrected
- Farms Rely on Token to Checkpoint Code Corrected
- Gas Inefficiencies Code Corrected
- Ineffective period Check Code Corrected
- Introduction of Batched Operations Code Corrected
- Usage as a Library Code Corrected
- farmingCheckpoint() Has No Functionality Code Corrected

6.1 Gas Griefing

Security High Version 1 Code Corrected

ERC20Farmable calls farm contracts in every call to `farmedPerToken()` to query information with `IFarm.farmedSinceCheckpointScaled()` on how many rewards have been released so far. Even though that call is handled with a `try/catch` block to prevent the target contract from reverting maliciously, it is still possible that the farm consumes all gas.

1. A malicious farm honeypots users into joining.
2. The malicious farm contract is upgraded through an upgradeability pattern.
3. Every call to `farmedSinceCheckpointScaled()` consumes all gas.

Now, following is not possible:

- any ERC20Farmable transfer from an affected user
- any ERC20Farmable transfer to an affected user
- exiting the malicious farm
- Claiming from the malicious farm

Ultimately, tokens will be locked for affected users.

Code corrected:

The call to `IFarm.famedSinceCheckpointScaled()` now has a gas limit. If the gas limit of 200000 is exceeded, the failure is handled by behaving equivalently to a revert in the farm contract.

Additionally, the static-call was wrapped inside an assembly block to prevent the return data bomb issue in the Solidity compiler (documented here: https://github.com/ethereum/solidity/issues/12306).

6.2 Commented Code

Design Low Version 1 Code Corrected

`ERC20Farmable._getFarmedSinceCheckpointscaled` contains commented code. Removing the code could help keep the code cleaner such that it is easier to understand.

Code corrected:

Commented out code has been replaced by calls to on `onError()`.

6.3 Farms Rely on Token to Checkpoint

Design Low Version 1 Code Corrected

`Farm._updateFarmingState()` calls `checkpoint()` of an external ERC20Farmable contract. Then, the ERC20Farmable contract calls `Farm.farmingCheckpoint()`. However, a malicious ERC20Farmable implementation could purposefully leave out the call to `Farm.farmingCheckpoint()`. Hence, the farm checkpoints could remain without updates.

Code corrected:

`farmingCheckpoint` has been removed from the farm contracts. Hence, there is no need to call it.

6.4 Gas Inefficiencies

Design Low Version 1 Code Corrected

In multiple locations code could be optimized to reduce gas cost. Some examples are:

- Function `UserAccounting.checkpoint()` loads the stored update time and the store farmed per token value from storage. However, to correctly call that function it is required to first call `farmedPerToken()` which also loads the same variable from storage. Hence, storage reads could be prevented.
- In function `FarmingPool.exit()` `balanceOf` is called twice. However, the second time it is called it is evident that it must be zero.
- `_beforeTokenTransfer` could exit early for self-transfers.
- Casting `period` to `uint40` when the input could have been restricted to be `uint40` in `startFarming`.

Code corrected:

The overall gas consumption has been optimized.

6.5 Ineffective `period` Check

Correctness | Low | Version 1 | Code Corrected

`FarmAccounting.startFarming()` contains following code:

```
require(period < 2**40, "FA: Period too large");
require(amount < 2**176, "FA: Amount too large");
(info.finished, info.duration, info.reward) = (uint40(block.timestamp + period), uint40(period), uint176(amount));
```

However, the first check is insufficient for `uint40(block.timestamp + period)` not to overflow.

Code corrected:

The precondition was changed to:

```
require(period + block.timestamp <= 2**40, "FA: Period too large");
```

6.6 Introduction of Batched Operations

Design | Low | Version 1 | Code Corrected

Assume a user participates in 10 farms for a farmable token. To claim all rewards the user needs to call `claim()` multiple times. Gas consumption could be reduced by allowing batched operations for ERC20Farmable.

Code corrected:

Following batched operations have been introduced:

- `claimAll`: claims on all farms
- `quitAll`: quits all farms

6.7 Usage as a Library

Design | Low | Version 1 | Code Corrected

ERC20Farmable is intended to be used as a library for farmable ERC-20 tokens. As such, some functions may require to be overridden so their functionality can be enhanced. However, no function in the supplied codebase has a `virtual` modifier, and so child contracts cannot override any method, meaning code that inherits from ERC20Farmable cannot extend its core functionality.

On the other hand, for some functions it could make sense to disallow overriding. An example could be `farmedPerToken()` which specifies the distribution among token holders. Allowing developers to modify its behaviour could lead to subtle issues that may not be caught during testing.

Assuming there is a use-case of changing the semantics of computing the farmed amount, code would require changes in several functions. First, `farmed()` would require changes. Second, `claim()` would require changes as it calls `UserAccounting.farmed()` instead of `farmed`. Hence, wrapping functionality from libraries in the abstract ERC20Farmable and using the wrapper functions internally could ease the overriding process and prevent errors.

Furthermore, it could be that some functions should not be callable by any child contracts, and there are certain state variables that should not be set in child contracts. For example, `farmTotalSupply` is a `public` variable, querying that value is helpful for users interacting with the contract. However, developers could unknowingly interleave writes to that variable in between updates to it in the internal callflow which would lead to inconsistent state. In that case, it could be helpful to have a public getter while restricting writes in child contracts.

To summarize, 1inch provides a library for staking. Since documentation is also part of writing an application library, it would be helpful to explicitly document the overridability and the visibility of functions and variables, as well as their intended use.

Code corrected:

The code has been adapted and functions have been marked as `virtual`.

6.8 `farmingCheckpoint()` Has No Functionality

Design | Low | Version 1 | Code Corrected

`FarmAccounting.farmingCheckpoint()` is empty and has no functionality. The calls to it further complicate the code. Moreover, replacing farm accounting logic through overriding is not easily possible.

Additionally, `Farm._updateFarmingState()` lacks checkpointing for a farm's state.

Code corrected:

The function has been removed.

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 Calls to Farms

Note Version 1

Note that the system interacts with untrusted farms and untrusted contracts.

Changes implemented or functionality in contracts inheriting from ERC20Farmable should ensure that there is no possibility of re-entering the ERC20Farmable contract when interacting with untrusted contracts since that could lead to possible unwanted modifications of farming state for other farms.

7.2 `farmedSinceCheckpointScaled()` Decimals

Note Version 1

Note that farms may run into issues if a Farm's `farmedSinceCheckpointScaled()` does not return a value that is in the base of `10**(18 + rewardToken.decimals())`:

- Assume the call to `farmedSinceCheckpointScaled()` returns a value in the base of `10**x`.
- Then, the call to farmedPerToken will return something in the base of `10**(x-ERC20Farmable.decimals())` which implies that `corrections` is in base of `10**x`.
- In farmed, the subtraction arguments will be both in base `10**x`. However, the result of the division will be in the base of `10**(x-18)`.

Using Farm of 1inch, will ensure that `x==18+rewardToken.decimals()`. However, if that is not the case, errors could occur.