Zellic

1inch Farming

Smart Contract Security Assessment

March 14, 2022

Prepared for:

Anton Bukov

1inch Farming

Prepared by:

Ayaz Mammadov

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than to simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Introduction

1.1 About 1inch Farming

1inch is a DeFi conglomerate with several product offerings, including a popular liquidity aggregator. We were approached to perform a security assessment for their upcoming ERC20 extension proposal. The proposal standardizes the new extension `ERC20Farmable`, which is essentially an ERC20 token that supports yield farming.

1.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of open-source tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these "shallow" bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, etc. as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions, and summarize the associated risks; for example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines, or code quality standards.

We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, etc.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document, rather than impact alone. Thus, we may sometimes emphasize a "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their importance may differ. This varies based on numerous soft factors, such as our clients' threat model, their business needs, project timelines, etc. We aim to provide useful and actionable advice to our partners that consider their long-term goals, rather than simply a list of security issues at present.

1.3 Scope

The engagement involved a review of the following targets:

1inch-farming

Repository	https://github.com/1inch/farming
Versions	`879448d1a934f767ee2b0c5a26ee1458db5f9986`
Type	Solidity
Platform	Ethereum (and other compatible chains)

1.4 Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered as financial or investment advice.

2 Executive Summary

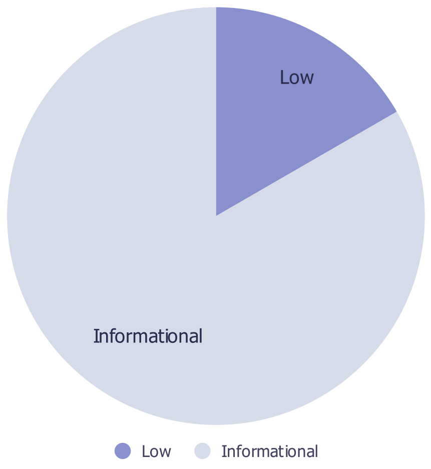
We carefully reviewed the scoped contracts and we discovered 6 findings. Fortunately, no issues in the business logic were found. Of the 6 findings, 1 was of low impact and the remaining findings were informational in nature. Overall, we applaud 1inch for their attention to detail and diligence in maintaining high code quality standards.

The ERC20Farmable project is being proposed as an efficient and thoughtful alternative to other staking contracts. It allows users to reap rewards from their holdings while still being in control of their tokens. For this audit, the contracts themselves were relatively straightforward.

During this audit, we exercised increased scrutiny towards potential code maturity issues. The 1inch farming project is to be submitted to OpenZeppelin as an ERC-20 extension. Even minor findings could have a widespread impact, as these contracts could eventually be used by a large number of downstream projects. Thus, we audited the scoped contracts to the highest level of detail.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	5



3 Detailed Findings

3.1 Insufficient validation of parameters

- **Target**: FarmingPool/Farm
- **Severity**: Low
- **Impact**: Low
- **Category**: Business Logic
- **Likelihood**: Low

Description

The function `setDistributor` verifies that the new distributor is not the old distributor, but it does not verify that the new distributor is nonzero.

```
function setDistributor(address distributor_) external onlyOwner {
    address oldDistributor = distributor;
    require(distributor_ != oldDistributor, "FP: distributor is already
    set");
    emit DistributorChanged(oldDistributor, distributor_);
    distributor = distributor_;
}
```

Impact

If an incorrect/default input is supplied to any of these functions, it will result in the loss of funds and/or control over the mentioned farms.

Recommendations

Add zero checks to the affected functions.

Remediation

The issue has been acknowledged by 1inch. Their official response is reproduced below:

> This behavior is by design:
> a) Owner may change the distributor any time.
> b) in case an owner wants to stop a farm from starting new farmings, they may set the address to zero.

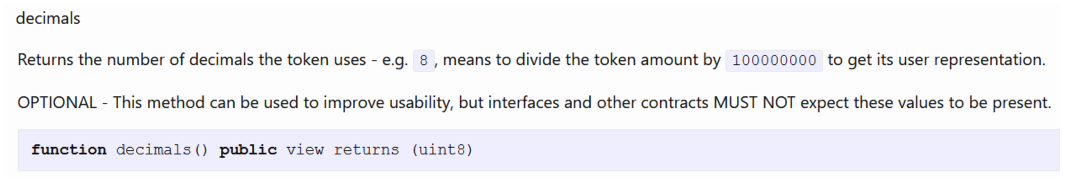
3.2 ERC20 decimals() method may be unimplemented

- **Target**: FarmingPool
- **Severity**: Informational
- **Impact**: Informational
- **Category**: Business Logic
- **Likelihood**: n/a

Description

FarmingPool provides a public method to get the number of decimals of the `stakingToken` which calls the `decimals` method of the underlying `IERC20` token. However, the EIP-20 standard declares that the decimals method is optional and that other contracts and interfaces should not rely on it being present.

According to the excerpt from the EIP-20 standard:



decimals

Returns the number of decimals the token uses - e.g. `8`, means to divide the token amount by `100000000` to get its user representation.

OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present.

```
function decimals() public view returns (uint8)
```

Excerpt from the EIP-20 standard

Impact

The function could revert or return incorrect data. This may pose a composability risk for other contracts that try to interact with the farming pool.

Recommendations

Add documentation stipulating that the `decimals` method is required, or that the implementation may be unreliable.

Remediation

The issue has been acknowledged by 1inch, and they will add natspec documentation when it will be implemented.

3.3 Undocumented code

- **Target**: Multiple contracts
- **Severity**: Low
- **Impact**: Informational
- **Category**: Code Maturity
- **Likelihood**: n/a

Description

The methods in the contracts `FarmingPool`, `IFarmingPool`, `ERC20Farmable`, `Farm`, `IFarm`, `UserAccounting`, `FarmAccounting`, `IERC20Farmable` lack documentation in general. There are few or no code comments available.

Impact

This is a source of developer confusion and a general coding hazard. Lack of documentation, or unclear documentation, is a major pathway to future bugs. It is best practice to document all code. Documentation also helps third-party developers integrate with the platform, and helps any potential auditors more quickly and thoroughly assess the code. Since there are plans to eventually merge the contracts into Open-Zeppelin, a widespread community library, the code should be as mature as possible.

Recommendations

Document the functions in the affected contracts so that the purpose, preconditions, and semantics are clearly explained. Return values and function arguments should be detailed to help prevent mistakes when calling the functions.

Remediation

The issue has been acknowledged by 1inch, and they will add additional documentation.

3.4 Internal discrepancy between function access control

- **Target**: Farm, FarmingPool
- **Severity**: Low
- **Impact**: Informational
- **Category**: Code Maturity
- **Likelihood**: n/a

Description

The functions `_updateCheckpoint` in `Farm` and `FarmingPool` both have the `private` access control modifier. However, when passed to the `startFarming` function as a callback, the parameter type is labeled as `internal`.

Impact

A manual review found no security issues with the current implementation. However, while there is no immediate impact, inconsistencies like these can make the code confusing and difficult to reason about, which could lead to future bugs. Since there are plans to eventually merge the contracts into OpenZeppelin, a widespread community library, the code should be as mature as possible.

Recommendations

Modify the function `_updateCheckpoint` to be an internal function if this was not a deliberate design decision.

Remediation

The issue was fixed by 1inch in commit e513e429.

3.5 Some methods are not exposed by their interface

- **Target**: IFarm, IFarmingPool
- **Severity**: Low
- **Impact**: Informational
- **Category**: Code Maturity
- **Likelihood**: n/a

Description

The interfaces `IFarm` and `IFarmingPool` do not expose the following methods from their concrete implementation:

`IFarm.sol`

- `startFarming  (Farm.sol)`

`IFarmingPool.sol`

- `startFarming  (FarmingPool.sol)`
- `decimals      (FarmingPool.sol)`

The interfaces also do not expose the `onlyOwner` `setDistributor` function, but we assume this is part of the intended design.

Impact

Consumers of this interface will not be able to call the unexposed methods.

Recommendations

If this is not the intended design, add the methods to the interface declarations.

Remediation

The issue was fixed by 1inch in commit 29bf4aed.

3.6 Insufficient test suite code coverage

- **Target**: Multiple contracts
- **Severity**: Low
- **Impact**: Informational
- **Category**: Code Maturity
- **Likelihood**: n/a

Description

Several functions in the smart contract are not covered by any unit or integration tests, to the best of our knowledge. We examined the tests `ERC20Farmable.js` and `FarmingPool.js`. The following functions do not have test coverage:

`ERC20Farmable.sol`

- `farmBalanceOf`
- `userIsFarming`
- `userFarmsCount`
- `userFarmsAt`
- `quitAll`
- `claimAll`
- `updateCheckpoint`

Because correctness is so critically important when developing smart contracts, we always recommend that projects strive for 100% code coverage. Testing is an essential part of the software development lifecycle. No matter how simple a function may be, untested code is always prone to bugs.

Recommendations

Expand the test suite so that all functions are covered by unit or integration tests.

Remediation

The issue has been acknowledged by 1inch, and they will add additional tests.