SMART CONTRACT AUDIT REPORT

for

1inch Farming

Prepared By: Xiaomi Huang

PeckShield

February 15, 2023

Document Properties

Client	1inch Protocol
Title	Smart Contract Audit Report
Target	1inch Farming
Version	2.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
2.0	February 15, 2023	Jing Wang	Final Release
1.0	February 12, 2022	Jing Wang	Final Release
1.0-rc	Jananry 30, 2022	Jing Wang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `1inch farming` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About 1inch farming

`1inch farming` protocol is a decentralized farming protocol that provides two ways to have incentives. The user could stake specific tokens to get continuously distributed rewards, or own the derived tokens to have farming without the need of staking tokens into the pool. Note that the second approach allows the user to have multiple farms simultaneously. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of 1inch Farming

Item	Description
Name	1inch Protocol
Website	https://app.1inch.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 15, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/1inch/farming.git (9c24d91)

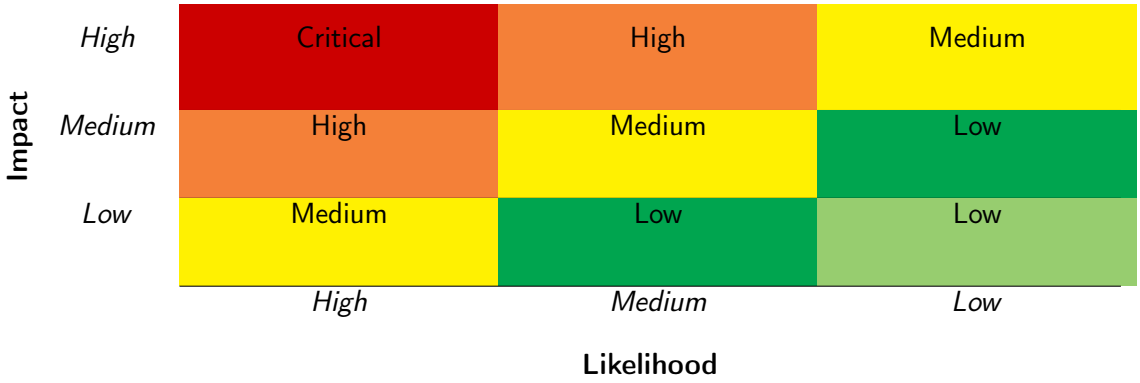
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/1inch/farming.git (1b8d7ae)

1.2 About PeckShield

PeckShield Inc. [8] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [6], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `1inch farming` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	2	■■
Informational	1	■
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 low-severity vulnerabilities and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatibility with Deflationary Tokens	Business Logic	Confirmed
PVE-002	Informational	Lack of Emitting Meaningful Events	Coding Practices	Confirmed
PVE-003	Low	Suggested Addition of rescueToken() to BaseFarm	Coding Practices	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Incompatibility with Deflationary Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FarmingPool`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In the `1inch farming` protocol, the `FarmingPool` contract is designed to take users' assets and deliver rewards depending on their deposit amount. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract makes the use of `safeTransferFrom()` or `safeTransfer()` routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
39      function deposit(uint256 amount) external override {
40          _mint(msg.sender, amount);
41          stakingToken.safeTransferFrom(msg.sender, address(this), amount);
42      }

44      function withdraw(uint256 amount) public override {
45          _burn(msg.sender, amount);
46          stakingToken.safeTransfer(msg.sender, amount);
47      }
```

Listing 3.1: `FarmingPool::deposit()/withdraw()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer.

As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary. Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Staking` for support.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate.

Status This issue has been confirmed. The team clarifies that they won't support deflationary tokens.

3.2 Lack of Emitting Meaningful Events

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

As mentioned before, the `ERC20Farmable` contract allows the user to have multiple farms simultaneously and receive rewards depending on their deposit amount. While examining the events that reflect the farming and rewards changes, we notice there is a lack of emitting related events that reflect important state changes. Specifically, when `userFarms[account]` or `info.corrections[account]`

is being updated, there is no respective event being emitted to reflect the change (line 39, 47 and 59).

```
38      function farm(address farm_) external override {
39          require(_userFarms[msg.sender].add(farm_), "ERC20Farmable: already farming");
40
41          uint256 balance = balanceOf(msg.sender);
42          infos[farm_].updateBalances(farmedPerToken(farm_), address(0), msg.sender,
                balance, false, true);
43          farmTotalSupply[farm_] += balance;
44      }
45
46      function exit(address farm_) external override {
47          require(_userFarms[msg.sender].remove(address(farm_)), "ERC20Farmable: already
                exited");
48
49          uint256 balance = balanceOf(msg.sender);
50          infos[farm_].updateBalances(farmedPerToken(farm_), msg.sender, address(0),
                balance, true, false);
51          farmTotalSupply[farm_] -= balance;
52      }
53
54      function claim(address farm_) external override {
55          uint256 fpt = farmedPerToken(farm_);
56          uint256 balance = farmBalanceOf(farm_, msg.sender);
57          uint256 amount = infos[farm_].farmed(msg.sender, balance, fpt);
58          if (amount > 0) {
59              infos[farm_].eraseFarmed(msg.sender, balance, fpt);
60              IFarm(farm_).claimFor(msg.sender, amount);
61          }
62      }
```

Listing 3.2: `ERC20Farmable::farm()`, `exit()`and `claim()`

Note another routine `FarmingPool::claim()` shares the same issue.

Recommendation Properly emit the related event when the `farm()`, `exit()` or `claim()` is being updated.

Status This issue has been confirmed by the team.

3.3 Suggested Addition of rescueToken() to BaseFarm

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseFarm`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1099 [1]

Description

By design, the `1inch farming` protocol supports multiple `BaseFarm` contracts and holds various types of reward assets. From past experience with current popular `DeFi` protocols, e.g., `YFI/Curve`, we notice that there is always non-trivial possibilities that non-related tokens may be accidentally sent to the pool contract(s). Moreover, the `BaseFarm` contract allows dynamic adjustment of `finished`, `reward` and `duration` of a farming pool, which inevitably introduces the possibility of having leftover amount in `BaseFarm` contracts. To avoid unnecessary loss of protocol users, we suggest to add the support of rescuing remaining reward tokens. This is a design choice for the benefit of protocol users.

Recommendation Add the support of rescuing remaining tokens in `BaseFarm`. An example addition is shown below:

```
    function rescueToken(address _token, uint256 _amount) external onlyOwner {
        require(_token != stakingToken, "Should not withdraw staking Token");
        IERC20(_token).safeTransfer(owner(), _amount);
        emit Recovered(_token, _amount);
    }
```

Listing 3.3: BaseFarm::rescueToken()

Status This issue has been confirmed by the team and they will consider adding it.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `1inch farming` protocol, which is a decentralized farming protocol that provides two ways to have incentives. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[5] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[6] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[7] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[8] PeckShield. PeckShield Inc. https://www.peckshield.com.