SMART CONTRACT AUDIT REPORT

for

1inch Limit Order Settlement

Prepared By: Xiaomi Huang

PeckShield

December 22, 2022

Document Properties

Client	1inch Protocol
Title	Smart Contract Audit Report
Target	1inch Limit Order Settlement
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 22, 2022	Luck Hu	Final Release
1.0-rc	November 15, 2022	Luck Hu	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 4
- 1.1 About 1inch Limit Order Settlement 4
- 1.2 About PeckShield 5
- 1.3 Methodology 5
- 1.4 Disclaimer 6

2 Findings 10
- 2.1 Summary 10
- 2.2 Key Findings 11

3 Detailed Results 12
- 3.1 Proper Assignment of delegatedShare As defaultFarm.token 12
- 3.2 Improved Validation of Possible returndata from call() 13
- 3.3 Removal of Redundant Code 14
- 3.4 Trust Issue of Admin Keys 16

4 Conclusion 17

References 18

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `1inch Limit Order Settlement` 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About 1inch Limit Order Settlement

The `1inch Limit Order` protocol is designed with the mission to avoid potential front-run when swapping tokens. In the protocol, a swap is created as a limit order to swap maker token for taker token. The order can be filled by any resolver from a whitelist who has staked a certain amount of `st1inch` tokens and is ranked among `top-N` of all the registered stakers. The `Limit Order Settlement` contact implements the settlement rules. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of 1inch Limit Order Settlement

Item	Description
Name	1inch Protocol
Website	https://app.1inch.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 22, 2022

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit.

- https://github.com/1inch/erc20-pods.git (50b192a)

- https://github.com/1inch/delegating.git (c8f4202)
- https://github.com/1inch/limit-order-settlement.git (117fac4b)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/1inch/erc20-pods.git (fc7ff90)
- https://github.com/1inch/delegating.git (f9acf81)
- https://github.com/1inch/limit-order-settlement.git (271f41b)

1.2 About PeckShield

PeckShield Inc. [9] 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

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [8]:

- 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 classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of 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 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) [7], 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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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 Logic	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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `1inch Limit Order Settlement` smart contracts. 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 logic, 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	2	■■
Low	2	■■
Informational	0	
Total	4	

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 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key 1inch Limit Order Settlement Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper Assignment of delegatedShare As defaultFarm.token	Business Logic	Fixed
PVE-002	Medium	Improved Validation of Possible Returns from call()	Coding Practices	Fixed
PVE-003	Low	Removal of Redundant Code	Coding Practices	Fixed
PVE-004	Low	Trust Issue of Admin Keys	Security Features	Mitigated

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 Proper Assignment of delegatedShare As defaultFarm.token

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `RewardableDelegationPod`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the `1inch Limit Order` protocol, the delegation system provides users with a series of delegation contracts each of which has their specific topic for delegation. The delegation topic for gasless-swap resolving will deploy extra token for every resolver that registers in delegation token. This resolver-specific token will represent shares of users who have delegated their `voting power` to this resolver. Resolvers will be able to run farms on top of their specific tokens to reward all the delegators proportionally to their stakes.

To elaborate, we show below the code snippet of the `register()` routine, which is used for the resolver to register in the delegation token. It creates a new `DelegatedShare` to represent the delegation shares, and accepts a `defaultFarm` to run the resolver's farms on top of the `DelegatedShare`. However, we notice that the `defaultFarm` is a special kind of pod whose `token` member shall be the same as the new `DelegatedShare`. If this is not satisfied, it will revert in the `onlyToken()` check (line 33), hence the call to the `defaultFarm->updateBalances()` from the `DelegatedShare` will revert. Based on this, we suggest to update the `defaultFarm.token` with the new created `DelegatedShare` or add a validation to ensure the `defaultFarm.token` is the same as the new `DelegatedShare`.

```
    function register(string memory name, string memory symbol,
        uint256 maxUserFarms, address defaultFarm)
        external onlyNotRegistered returns(IDelegatedShare token)
    {
        token = new DelegatedShare(name, symbol, maxUserFarms);
        registration[msg.sender] = token;
```

```
        _delegateeTokens.add(address(token));
        if (defaultFarm != address(0)) {
            defaultFarms[msg.sender] = defaultFarm;
        }
    }
```

Listing 3.1: RewardableDelegationPod:: register ()

```
    function updateBalances(address from, address to, uint256 amount)
        public virtual onlyToken {
        if (from == address(0)) {
            _mint(delegated[to], amount);
            return;}
        ...
    }
```

Listing 3.2: BasicDelegationPod::updateBalances()

Note the same issue is also applicable to the `register()/setDefaultFarm()` routines.

Recommendation Revisit the above mentioned `register()` routine to properly update the `defaultFarm.token` with the new `DelegatedShare`, or add a validation to ensure the `defaultFarm.token` is the same as the new `DelegatedShare`.

Status The issue has been fixed by these commits: `bf551de` and `c34c954`.

3.2 Improved Validation of Possible returndata from call()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `Settlement`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The `Settlement` contract interacts with the `Limit Order Protocol` contract to complete the order settlement. It may specify several interactions to settle a limit order. The `Limit Order Protocol` contract calls the `fillOrderInteraction()` to fill each interaction. While analyzing the logic in the `fillOrderInteraction()` routine, we notice it may perform some ERC20 specific operations (e.g., `approve()/transfer()`) via low level calls. However, there is a lack of proper validation for the possible `returndata` from these the low level calls.

To elaborate, we show below the code snippet of the `fillOrderInteraction()` routine. If this is the finalized interaction (`_FINALIZE_INTERACTION`), it executes the specified low level calls to the `targest[i]` (line 100). Specially, if some calls are token related, there is a need to validate the possible `returndata`. Especially if the token is not ERC20 compliant, it may return `false` for failure, not revert. In this case, if the `returndata` is `false`, it shall revert the transaction.

```
86   function fillOrderInteraction(
87     address, /* taker */
88     uint256, /* makingAmount */
89     uint256 takingAmount,
90     bytes calldata interactiveData
91 ) external returns (uint256) {
92     address interactor = _onlyLimitOrderProtocol();
93     if (interactiveData[0] == _FINALIZE_INTERACTION) {
94         (address[] calldata targets, bytes[] calldata calldatas) = _abiDecodeFinal(
             interactiveData[1:]);
95
96         uint256 length = targets.length;
97         if (length != calldatas.length) revert IncorrectCalldataParams();
98         for (uint256 i = 0; i < length; i++) {
99             // solhint-disable-next-line avoid-low-level-calls
100            (bool success, ) = targets[i].call(calldatas[i]);
101            if (!success) revert FailedExternalCall();
102        }
103    } ...
104 }
```

Listing 3.3: `Settlement::fillOrderInteraction()`

Recommendation Revisit the `fillOrderInteraction()` routine to properly validate the possible `returndata` for ERC20 token related calls, and revert the transaction if the token returns `false`.

Status The issue has been fixed by this commit: `174410f`.

3.3 Removal of Redundant Code

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `St1inch`,
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The `1inch Limit Order Settlement` makes good use of a number of reference contracts, such as `SafeERC20` and `Ownable` to facilitate its code implementation and organization. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `St1inch` contact, it defines two errors, i.e., `ZeroAddress()/BurnAmountExceedsBalance()` (lines 16-17), but never emit them anywhere. Moreover, if we examine closely the `WhitelistRegistry::_shrinkPoorest()` routine, the statement of verifying of `if(i < addresses.length)` (line 116) is contradictory with the statement of `if(i < addresses.length)` (line 109). Which means, the `if(i < addresses.length)` (line 116) is always `true`, which could safely removed.

```
13     contract St1inch is ERC20Pods, Ownable, VotingPowerCalculator, IVotable {
14     using SafeERC20 for IERC20;
15 
16     error ZeroAddress();
17     error BurnAmountExceedsBalance();
18     error ApproveDisabled();
19     error TransferDisabled();
```

Listing 3.4: St1inch.sol

```
103    function _shrinkPoorest(AddressSet.Data storage set, IVotable vtoken, uint256 size)
           private {
104        uint256 richestIndex = 0;
105        address[] memory addresses = set.items.get();
106        uint256[] memory balances = new uint256[](addresses.length);
107        for (uint256 i = 0; i < addresses.length; i++) {...}
108 
109        for (uint256 i = size; i < addresses.length; i++) {
110            if (balances[i] <= balances[richestIndex]) {
111                // Swap i-th and richest-th elements
112                (addresses[i], addresses[richestIndex]) = (addresses[richestIndex],
                       addresses[i]);
113                (balances[i], balances[richestIndex]) = (balances[richestIndex],
                       balances[i]);
114 
115                // Find new richest in first size elements
116                if (i < addresses.length) {...}
117            }
118        }
119 
120        // Remove poorest elements from set
121        for (uint256 i = 0; i < size; i++) {...}
122    }
```

Listing 3.5: WhitelistRegistry.sol

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been fixed by these commits: `233a02a` and `6ec2017`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `1inch Limit Order Settlement` contracts, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the system-wide operations (e.g., set the `feeBank`). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `Settlement` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged function in `Settlement` contact allows for the `owner` to set the `feeBank` which is expected to be the fee mechanism for resolvers to pay for using the system. The `feeBank` has the right to update the credit allowance of the resolvers in the `Settlement` contract. If the `feeBank` is set to a malicious one, the valid resolvers may can not use the system, but the faked resolvers may can use the system.

```
    function setFeeBank(address newFeeBank) external onlyOwner {
        feeBank = newFeeBank;
    }
```

Listing 3.6: Example Privileged Operations in the `Settlement` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team confirm they plan to set the `owner` to timelocked multisig.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `1inch Limit Order Settlement` contract. The `1inch Limit Order` protocol is designed with the mission to avoid potential front-run when swapping tokens. In the protocol, a swap is created as a limit order to swap maker token for taker token. The order can be filled by any resolver from a whitelist who has staked a certain amount of `st1inch` tokens and is ranked `top-N` among all the registered stakers. The `Limit Order Settlement` contact implements the settlement rules. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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

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