SMART CONTRACT AUDIT REPORT

for

1inch Aggregation (v6)

Prepared By: Xiaomi Huang

PeckShield

October 22, 2023

Document Properties

Client	1inch Protocol
Title	Smart Contract Audit Report
Target	1inch Aggregation
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Colin Zhong, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	October 22, 2023	Xuxian Jiang	Final Release
1.0-rc	October 22, 2023	Xuxian Jiang	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 smart contract source code of the `1inch Aggregation (v6)` 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 Aggregation

The `1inch Aggregation` protocol is a `DeFi` aggregator and a decentralized exchange with smart routing. The core protocol connects a large number of decentralized platforms in order to minimize price slippage and find the optimal trade for the users. The audited smart contracts are designed to create a universal exchange for tokens. Major changes within the scope are numerous gas optimization, additional external integration, as well as the limit order support. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of 1inch Aggregation

Item	Description
Name	1inch Protocol
Website	https://app.1inch.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 22, 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/1inch-contract.git (83db150)
- https://github.com/1inch/limit-order-protocol.git (d02b865)

And here is the commit ID after fixes for the issue found in the audit have been checked in::

- https://github.com/1inch/1inch-contract.git (TBD)
- https://github.com/1inch/limit-order-protocol.git (TBD)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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) [5], 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

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.

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 Aggregation (v6)` 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	0	
Low	1	■
Informational	2	■ ■
Total	3	

We have so far identified a potential issue for improvement: accommodate the possible idiosyncrasy about ERC20-related `approve()`. More information can be found in the next subsection, and its detailed discussions can be found in Section 3.

2.2 Key Findings

Overall, the smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issue (shown in Table 2.1), including 1 low-severity vulnerability and 2 informational recommendations.

Table 2.1: Key 1inch Aggregation Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Simplified Curve Swap Logic in UnoswapRouter	Coding Practices	
PVE-002	Informational	Improved Gas Efficiency in UnoswapRouter::uniswapV3SwapCallback()	Coding Practices	
PVE-003	Low	Missing _ALLOW_MULTIPLE_FILLS_FLAG Enforcement in OrderMixin	Business Logic	

Besides recommending specific countermeasure to mitigate the issue, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Simplified Curve Swap Logic in UnoswapRouter

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `UnoswapRouter`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

The `1inch aggregation` protocol provides a seamless integration with `Curve` that provides relatively deep liquidity for stablecoin exchanges. In the analysis of the `Curve` support, we notice current implementation may be improved.

To elaborate, we show below the code snippet from the related `_curfe()` routine. We notice the necessity of approving the `fromToken` to the swap pool with the intended swap amount (line 558) when there is no callback alternative (line 557). With that, we can simplify the logic by elevating the `if`-condition of `iszero(hasCallback)` (line 557) to the above `if`-condition (line 550) as below: `and(iszero(hasCallback), or(iszero(useEth), and(useEth, eq(toToken, _WETH))))`.

```
535     function _curfe(
536         address recipient,
537         uint256 amount,
538         uint256 minReturn,
539         Address dex
540     ) internal returns(uint256 ret) {
541         assembly ("memory-safe") {  // solhint-disable-line no-inline-assembly
542             ...
543             let toToken
544             {  // Stack too deep
545                 let toSelectorOffset := and(shr(_CURVE_TO_COINS_SELECTOR_OFFSET, dex),
                        _CURVE_TO_COINS_SELECTOR_MASK)
546                 let toTokenIndex := and(shr(_CURVE_TO_COINS_ARG_OFFSET, dex),
                        _CURVE_TO_COINS_ARG_MASK)
547                 toToken := curveCoins(pool, toSelectorOffset, toTokenIndex)
```

```
            }

            if or(iszero(useEth), and(useEth, eq(toToken, _WETH))) {
                let fromSelectorOffset := and(shr(_CURVE_FROM_COINS_SELECTOR_OFFSET, dex), _CURVE_FROM_COINS_SELECTOR_MASK)
                let fromTokenIndex := and(shr(_CURVE_FROM_COINS_ARG_OFFSET, dex), _CURVE_FROM_COINS_ARG_MASK)
                let fromToken := curveCoins(pool, fromSelectorOffset, fromTokenIndex)
                if eq(fromToken, 0xeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee) {
                    fromToken := _WETH
                }
                if iszero(hasCallback) {
                    asmApprove(fromToken, pool, amount, mload(0x40))
                }
            }
            ...
        }
    }
```

Listing 3.1: `UnoswapRouter::_curfe()`

Recommendation Revise the above routine for improved gas efficiency.

Status

3.2 Improved Gas Efficiency in UnoswapRouter::uniswapV3SwapCallback()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `UnoswapRouter`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

The `1inch aggregation` protocol also provides a seamless integration with `UniswapV3`, the third iteration of the `Uniswap` protocol with unique concentrated liquidity feature. While reviewing the `UniswapV3` support, we notice the callback helper can be improved.

To elaborate, we show below the code snippet from the related `uniswapV3SwapCallback()` routine. The code snippet is executed after validating the caller from the intended `UniswapV3` pool for the purpose of transfering the swap amount into the pool. We notice the `success` variable is further checked by calling the `extcodesize(_PERMIT2)` to ensure the `_PERMIT2` contract is indeed deployed and functioning. However, since it is already deployed, this call can be considered as redundant and is suggested for removal.

```
        switch usePermit2
        case 1 {
            // permit2.transferFrom(payer, msg.sender, amount, token);
            mstore(emptyPtr, selectors)
            emptyPtr := add(emptyPtr, _PERMIT2_TRANSFER_FROM_SELECTOR_OFFSET)
            mstore(add(emptyPtr, 0x04), payer)
            mstore(add(emptyPtr, 0x24), caller())
            mstore(add(emptyPtr, 0x44), amount)
            mstore(add(emptyPtr, 0x64), token)
            let success := call(gas(), _PERMIT2, 0, emptyPtr, 0x84, 0, 0)
            if success {
                success := gt(extcodesize(_PERMIT2), 0)
            }
            if iszero(success) {
                mstore(0, 0
                    xc3f9d33200000000000000000000000000000000000000000000000000000000)
                    // Permit2TransferFromFailed()
                revert(0, 4)
            }
        }
        case 0 {
            // IERC20(token.get()).safeTransferFrom(payer, msg.sender, amount);
            mstore(emptyPtr, selectors)
            emptyPtr := add(emptyPtr, _TRANSFER_FROM_SELECTOR_OFFSET)
            mstore(add(emptyPtr, 0x04), payer)
            mstore(add(emptyPtr, 0x24), caller())
            mstore(add(emptyPtr, 0x44), amount)
            safeERC20(token, 0, emptyPtr, 0x64, 0x20)
        }
```

Listing 3.2: `UnoswapRouter::uniswapV3SwapCallback()`

Recommendation Improve the above routine by removing redundant code.

Status

3.3 Missing _ALLOW_MULTIPLE_FILLS_FLAG Enforcement in OrderMixin

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OrderMixin`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `1inch aggregation` protocol is no exception. Specifically, if we examine the `allowMultipleFills` contract, it has defined a number of maker-related traits, such as `_NO_PARTIAL_FILLS_FLAG` and `_ALLOW_MULTIPLE_FILLS_FLAG`. In the following, we show the corresponding routines that examine the related status.

```
    /**
     * @notice Determines if the order allows partial fills.
     * @dev If the _NO_PARTIAL_FILLS_FLAG is not set in the makerTraits, then the order allows partial fills.
     * @param makerTraits The traits of the maker, determining their preferences for the order.
     * @return result A boolean indicating whether the maker allows partial fills.
     */
    function allowPartialFills(MakerTraits makerTraits) internal pure returns (bool) {
        return (MakerTraits.unwrap(makerTraits) & _NO_PARTIAL_FILLS_FLAG) == 0;
    }

    /**
     * @notice Determines if the order allows multiple fills.
     * @dev If the _ALLOW_MULTIPLE_FILLS_FLAG is set in the makerTraits, then the maker allows multiple fills.
     * @param makerTraits The traits of the maker, determining their preferences for the order.
     * @return result A boolean indicating whether the maker allows multiple fills.
     */
    function allowMultipleFills(MakerTraits makerTraits) internal pure returns (bool) {
        return (MakerTraits.unwrap(makerTraits) & _ALLOW_MULTIPLE_FILLS_FLAG) != 0;
    }
```

Listing 3.3: MakerTraitsLib::allowPartialFills() and MakerTraitsLib::allowMultipleFills()

These parameters define various aspects of the trading preference and need to be honored in the order execution. However, our analysis shows that the `_NO_PARTIAL_FILLS_FLAG` flag is properly enforced while the `_ALLOW_MULTIPLE_FILLS_FLAG` is not.

To elaborate, we show below the related `_fillContractOrder()` routine. Note this routine properly validates the order signature and applies order permit on the first fill. However, it needs to detect non-first fills and validate against the `_ALLOW_MULTIPLE_FILLS_FLAG` flag as follows: `if (!order.makerTraits.allowMultipleFills()&& remainingMakingAmount != order.makingAmount)revert MultipleFillNotAllowed()`.

```
    function _fillContractOrder(
        IOrderMixin.Order calldata order,
        bytes calldata signature,
        uint256 amount,
        TakerTraits takerTraits,
        address target,
        bytes calldata extension,
        bytes calldata interaction
    ) private returns(uint256 makingAmount, uint256 takingAmount, bytes32 orderHash) {
        // Check signature and apply order permit only on the first fill
        orderHash = order.hash(_domainSeparatorV4());
        uint256 remainingMakingAmount = _checkRemainingMakingAmount(order, orderHash);
        if (remainingMakingAmount == order.makingAmount) {
            if (!ECDSA.isValidSignature(order.maker.get(), orderHash, signature)) revert
                BadSignature();
        }

        (makingAmount, takingAmount) = _fill(order, orderHash, remainingMakingAmount,
            amount, takerTraits, target, extension, interaction);
    }
```

Listing 3.4: OrderMixin::_fillContractOrder()

Recommendation Revise the `_fill*()`-related routines to properly honor the `_ALLOW_MULTIPLE_FILLS_FLAG` flag.

Status

4 Conclusion

In this audit, we have analyzed the design and implementation of the `1inch Aggregation (v6)` protocol. The `1inch Aggregation` protocol is a `DeFi` aggregator and a decentralized exchange with smart routing. The core protocol connects a large number of decentralized platforms in order to minimize price slippage and find the optimal trade for the users. The audited `1inch Aggregation (v6)` implements an upgrade for the old one and the major changes within the scope are numerous gas optimization, additional external integration, as well as the limit order support. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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