SMART CONTRACT AUDIT REPORT

for

LooksRare Exchange V2

Prepared By: Xiaomi Huang

PeckShield

January 11, 2023

Document Properties

Client	LooksRare
Title	Smart Contract Audit Report
Target	LooksRare Exchange V2
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 11, 2023	Xiaotao Wu	Final Release
1.0-rc	January 9, 2023	Xiaotao Wu	Release Candidate #1

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 `LooksRare Exchange V2` 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 LooksRare Exchange V2

`LooksRare Exchange V2` protocol is built on a hybrid off-chain/on-chain system where taker orders match maker orders to execute trades between `ETH`/`ERC20` and `NFTs`. Taker orders take liquidity from the orderbook while maker orders add liquidity to the offchain orderbook. A `bid` user spends fungible tokens to acquire `NFT` assets. An `ask` user sells `NFT` assets for fungible tokens. It is a non-custodial exchange where orders require approval of the tokens (fungible and non-fungible) being transferred in a trade. Off-chain orders (`Maker orders`) are based on `EIP712` signatures which are stored off-chain. A trade consists of a bilateral exchange of either an `ETH` or an `ERC20` token against one or multiple `NFT` tokens (e.g. `ERC721`, `ERC1155`) with a specific amount at a specific price. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The LooksRare Exchange V2

Item	Description
Name	LooksRare
Website	https://looksrare.org/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 11, 2023

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

- https://github.com/LooksRare/contracts-exchange-v2.git (48283c2)

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

- https://github.com/LooksRare/contracts-exchange-v2.git (ad67592)

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 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 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 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.

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 1.3: 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

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 `LooksRare Exchange V2` 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	1	■
Low	1	■
Informational	2	■ ■
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 that 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 1 medium-severity vulnerability, 1 low-severity vulnerability, and 2 informational suggestions.

Table 2.1: Key LooksRare Exchange V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Implementation Logic in StrategyItemIdsRange	Business Logic	Fixed
PVE-002	Informational	Missed Sanity Checks in StrategyFloor-FromChainlink	Coding Practices	Fixed
PVE-003	Informational	Improved Sanity Checks in transfer-BatchItemsAcrossCollections()	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Besides recommending specific countermeasures to mitigate these issues, 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 Improved Implementation Logic in StrategyItemIdsRange

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

Description

The `LooksRare Exchange V2` protocol supports bundles so that users can submit maker bid or maker ask orders on a set of item IDs with specific amounts (that must be > 0). One example is the `StrategyItemIdsRange` contract. This strategy contract allows a bidder to select an item ID range (e.g. 1-100) and a seller can fulfill the order with any tokens within the specified ID range. While reviewing the logic of the `StrategyItemIdsRange::executeStrategyWithTakerAsk()` function, which validates the order under the context of the chosen strategy, we notice the current implementation logic can be improved.

To elaborate, we show below the related code snippet. It comes to our attention that the item IDs provided by the taker may not in the item ID range filled by the bidder. However, all the item IDs provided by the taker will be transferred to the bidder if the `desiredAmount` from the bidder side is met (lines 93-94). Thus the taker side will suffer asset losses if the item IDs provided by the taker do not fall into the item ID range provided by the bidder.

```
25     function executeStrategyWithTakerAsk(
26         OrderStructs.TakerAsk calldata takerAsk,
27         OrderStructs.MakerBid calldata makerBid
28     )
29         external
30         pure
31         returns (uint256 price, uint256[] memory itemIds, uint256[] memory amounts, bool
               isNonceInvalidated)
32     {
```

```
        if (makerBid.itemIds.length != 2  makerBid.amounts.length != 1) {
            revert OrderInvalid();
        }

        uint256 minItemId = makerBid.itemIds[0];
        uint256 maxItemId = makerBid.itemIds[1];

        if (minItemId >= maxItemId) {
            revert OrderInvalid();
        }

        uint256 desiredAmount = makerBid.amounts[0];
        uint256 totalOfferedAmount;
        uint256 lastItemId;
        uint256 length = takerAsk.itemIds.length;

        for (uint256 i; i < length; ) {
            uint256 offeredItemId = takerAsk.itemIds[i];
            // Force the client to sort the item ids in ascending order,
            // in order to prevent taker ask from providing duplicated
            // item ids
            if (offeredItemId <= lastItemId) {
                if (i != 0) {
                    revert OrderInvalid();
                }
            }

            uint256 amount = takerAsk.amounts[i];

            if (amount != 1) {
                if (amount == 0) {
                    revert OrderInvalid();
                }
                if (makerBid.assetType == 0) {
                    revert OrderInvalid();
                }
            }

            if (offeredItemId >= minItemId) {
                if (offeredItemId <= maxItemId) {
                    totalOfferedAmount += amount;
                }
            }

            lastItemId = offeredItemId;

            unchecked {
                ++i;
            }
        }

        if (totalOfferedAmount != desiredAmount) {
```

```
            revert OrderInvalid();
        }

        if (makerBid.maxPrice != takerAsk.minPrice) {
            revert OrderInvalid();
        }

        price = makerBid.maxPrice;
        itemIds = takerAsk.itemIds;
        amounts = takerAsk.amounts;
        isNonceInvalidated = true;
    }
```

Listing 3.1: `StrategyItemIdsRange::executeStrategyWithTakerAsk()`

Recommendation Only transfer the item IDs from the taker to the bidder which are in the item ID range provided by the bidder.

Status This issue has been fixed in the following PR: `287`.

3.2 Missed Sanity Checks in StrategyFloorFromChainlink

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `StrategyFloorFromChainlink`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `executeFixedDiscountStrategyWithTakerAsk()` function of the `StrategyFloorFromChainlink` contract looks at the bidder's desired execution price in ETH (floor - discount) as well as the maximum execution price, and chooses the lower price. The strategy validation will be passed if the `minPrice` provided by the taker is no more than the bidder's desired execution price. While reviewing its logic, we notice the current implementation fails to validate the `itemIds` of the given arguments in `takerAsk` and `makerBid`.

To elaborate, we show below the code snippet of the `executeFixedDiscountStrategyWithTakerAsk()` function. Since the item ID sent to the bidder is determined by the `itemIds` provided by the taker (line 201), the item ID wanted by the bidder side may not be equal to the item ID offered by the taker side.

```
160    function executeFixedDiscountStrategyWithTakerAsk(
161        OrderStructs.TakerAsk calldata takerAsk,
162        OrderStructs.MakerBid calldata makerBid
```

```
    )
        external
        view
        returns (uint256 price, uint256[] memory itemIds, uint256[] memory amounts, bool
            isNonceInvalidated)
    {
        if (makerBid.currency != WETH) {
            revert WrongCurrency();
        }

        if (
            takerAsk.itemIds.length != 1
            takerAsk.amounts.length != 1
            takerAsk.amounts[0] != 1
            makerBid.amounts.length != 1
            makerBid.amounts[0] != 1
        ) {
            revert OrderInvalid();
        }

        uint256 floorPrice = _getFloorPrice(makerBid.collection);
        uint256 discountAmount = abi.decode(makerBid.additionalParameters, (uint256));

        if (floorPrice <= discountAmount) {
            revert DiscountGreaterThanFloorPrice();
        }

        uint256 desiredPrice = floorPrice - discountAmount;

        if (desiredPrice >= makerBid.maxPrice) {
            price = makerBid.maxPrice;
        } else {
            price = desiredPrice;
        }

        if (takerAsk.minPrice > price) {
            revert AskTooHigh();
        }

        itemIds = takerAsk.itemIds;
        amounts = takerAsk.amounts;
        isNonceInvalidated = true;
    }
```

Listing 3.2: StrategyFloorFromChainlink::executeFixedDiscountStrategyWithTakerAsk()

Note a similar issue also exists in the `executeBasisPointsDiscountStrategyWithTakerAsk()` routine of the same contract.

Recommendation Add necessary sanity checks for the above mentioned functions.

Status This issue has been addressed as the `LooksRare` teams confirms that these two functions are used for collection orders: 289.

3.3 Improved Sanity Checks in transferBatchItemsAcrossCollections()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `TransferManager`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In the `TransferManager` contract, the `transferBatchItemsAcrossCollections()` function is designed to facilitate the transfers of items across an array of collections that can be both `ERC721` and `ERC1155`. While reviewing the implementation of this routine, we notice that it can benefit from additional sanity checks.

To elaborate, we show below the code snippet of the `transferBatchItemsAcrossCollections()` function. Specifically, there is a lack of length verification for the input argument `assetTypes`. If `assetTypes.length != collectionsLength`, the `assetType = assetTypes[i]` execution may revert (line 140).

```
112     function transferBatchItemsAcrossCollections(
113         address[] calldata collections,
114         uint256[] calldata assetTypes,
115         address from,
116         address to,
117         uint256[][] calldata itemIds,
118         uint256[][] calldata amounts
119     ) external {
120         uint256 collectionsLength = collections.length;
121 
122         if (
123             collectionsLength == 0  (itemIds.length ^ collectionsLength)  (amounts.
                length ^ collectionsLength) != 0
124         ) {
125             revert WrongLengths();
126         }
127 
128         if (from != msg.sender) {
129             if (!isOperatorValidForTransfer(from, msg.sender)) {
130                 revert TransferCallerInvalid();
131             }
132         }
133 
```

```
        for (uint256 i; i < collectionsLength; ) {
            uint256 itemIdsLengthForSingleCollection = itemIds[i].length;
            if (itemIdsLengthForSingleCollection == 0  amounts[i].length !=
                itemIdsLengthForSingleCollection) {
                revert WrongLengths();
            }

            uint256 assetType = assetTypes[i];
            if (assetType == 0) {
                for (uint256 j; j < itemIdsLengthForSingleCollection; ) {
                    _executeERC721TransferFrom(collections[i], from, to, itemIds[i][j]);
                    unchecked {
                        ++j;
                    }
                }
            } else if (assetType == 1) {
                _executeERC1155SafeBatchTransferFrom(collections[i], from, to, itemIds[i
                    ], amounts[i]);
            } else {
                revert WrongAssetType(assetType);
            }

            unchecked {
                ++i;
            }
        }
    }
```

Listing 3.3: TransferManager:: transferBatchItemsAcrossCollections ()

Recommendation Add necessary sanity checks to ensure `assetTypes.length == collectionsLength`
.

Status This issue has been fixed in the following commit: `pull288`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `LooksRare Exchange V2` protocol, there is a privileged accounts, i.e., `owner`. This account plays a critical role in governing and regulating the protocol-wide operations (e.g., whitelist/blacklist currency, update the maximum creator fee, update affiliate rate, add/update strategy, whitelist operator, etc.). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `ExecutionManager` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

```
    /**
     * @notice This function allows the owner to update the creator fee manager address.
     * @param newCreatorFeeManager Address of the creator fee manager
     * @dev Only callable by owner.
     */
    function updateCreatorFeeManager(address newCreatorFeeManager) external onlyOwner {
        creatorFeeManager = ICreatorFeeManager(newCreatorFeeManager);
        emit NewCreatorFeeManager(newCreatorFeeManager);
    }

    /**
     * @notice This function allows the owner to update the maximum creator fee (in basis point).
     * @param newMaxCreatorFeeBp New maximum creator fee (in basis point)
     * @dev The maximum value that can be set is 25%.
     *      Only callable by owner.
     */
    function updateMaxCreatorFeeBp(uint16 newMaxCreatorFeeBp) external onlyOwner {
        if (newMaxCreatorFeeBp > 2_500) {
            revert CreatorFeeBpTooHigh();
        }

        maxCreatorFeeBp = newMaxCreatorFeeBp;

        emit NewMaxCreatorFeeBp(newMaxCreatorFeeBp);
    }

    /**
     * @notice This function allows the owner to update the protocol fee recipient.
     * @param newProtocolFeeRecipient New protocol fee recipient address
     * @dev Only callable by owner.
     */
    function updateProtocolFeeRecipient(address newProtocolFeeRecipient) external onlyOwner {
        if (newProtocolFeeRecipient == address(0)) {
            revert NewProtocolFeeRecipientCannotBeNullAddress();
        }

        protocolFeeRecipient = newProtocolFeeRecipient;
        emit NewProtocolFeeRecipient(newProtocolFeeRecipient);
    }
```

Listing 3.4: Example Privileged Operations in `ExecutionManager`

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 changes 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 confirmed.

4 Conclusion

In this audit, we have analyzed the LooksRare Exchange V2 design and implementation. LooksRare Exchange V2 protocol is built on a hybrid off-chain/on-chain system where taker orders match maker orders to execute trades between ETH/ERC20 and NFTs. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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.