SMART CONTRACT AUDIT REPORT

for

LooksRare Aggregator

Prepared By: Xiaomi Huang

PeckShield

March 11, 2023

Document Properties

Client	LooksRare
Title	Smart Contract Audit Report
Target	LooksRare Aggregator
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	March 11, 2023	Xiaotao Wu	Final Release
1.0-rc	March 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 Aggregator` feature, 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 Aggregator

`LooksRare Aggregator` is a set of smart contracts that accept trade data from multiple marketplaces or liquidity sources and execute the trades in a single on-chain transaction. This audit focuses on the `LooksRareV2Proxy` contract, which will support the `LooksRare Exchange V2` protocol. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The LooksRare Aggregator

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

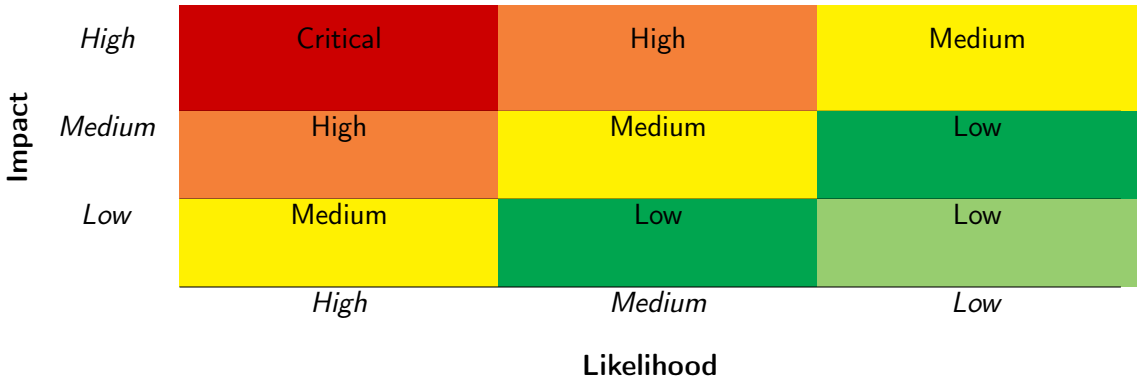
In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audit only covers the following contracts: `LooksRareAggregator.sol`, `ERC20EnabledLooksRareAggregator.sol`, and `LooksRareV2Proxy.sol`.

- https://github.com/LooksRare/contracts-aggregator.git (bdde0f3)

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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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 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

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

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.

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 `LooksRare Aggregator` feature. 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	1	■
Informational	1	■
Total	2	

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 low-severity vulnerability and 1 informational suggestion.

Table 2.1: Key LooksRare Aggregator Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-002	Low	Trust Issue of Admin Keys	Security Features	Mitigated

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 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LooksRareAggregator`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
194     /**
195     * @dev Approve the passed address to spend the specified amount of tokens on behalf
            of msg.sender.
196     * @param _spender The address which will spend the funds.
197     * @param _value The amount of tokens to be spent.
198     */
199     function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201         // To change the approve amount you first have to reduce the addresses`
202         //  allowance to zero by calling `approve(_spender, 0)` if it is not
203         //  already 0 to mitigate the race condition described here:
204         //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205         require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
```

```
207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }
```

Listing 3.1: USDT Token Contract

Because of that, a normal call to approve() with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to approve() twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

In the following, we use the approve() routine as an example. If the USDT token is supported as currency, the execution of approve() may revert if the old allowance and the new allowance to be configured are both non-zero.

```
161     /**
162      * @notice Approve marketplaces to transfer ERC20 tokens from the aggregator
163      * @param currency The ERC20 token address to approve
164      * @param marketplace The marketplace address to approve
165      * @param amount The amount of ERC20 token to approve
166      */
167     function approve(
168         address currency,
169         address marketplace,
170         uint256 amount
171     ) external onlyOwner {
172         _executeERC20Approve(currency, marketplace, amount);
173     }
```

Listing 3.2: LooksRareAggregator::approve()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related approve(). An example revision is shown as follows:

```
161     /**
162      * @notice Approve marketplaces to transfer ERC20 tokens from the aggregator
163      * @param currency The ERC20 token address to approve
164      * @param marketplace The marketplace address to approve
165      * @param amount The amount of ERC20 token to approve
166      */
167     function approve(
168         address currency,
169         address marketplace,
170         uint256 amount
171     ) external onlyOwner {
172         _executeERC20Approve(currency, marketplace, 0);
173         _executeERC20Approve(currency, marketplace, amount);
174     }
```

Listing 3.3: LooksRareAggregator::approve()

Status This issue has been resolved as the team confirms that they will call `approve()` manually to reset the allowance to zero if there exists such a conflict.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LooksRareAggregator`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the `LooksRare Aggregator` feature, there is a privileged accounts, i.e., `owner`. This account plays a critical role in governing and regulating the protocol-wide operations (e.g., add/delete marketplace proxy's address and function selector, approve marketplaces to transfer `ERC20` tokens from the aggregator, etc.). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `LooksRareAggregator` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

```
139     /**
140      * @notice Enable calling the specified proxy's trade function
141      * @dev Must be called by the current owner
142      * @param proxy The marketplace proxy's address
143      * @param selector The marketplace proxy's function selector
144      */
145     function addFunction(address proxy, bytes4 selector) external onlyOwner {
146         _proxyFunctionSelectors[proxy][selector] = 1;
147         emit FunctionAdded(proxy, selector);
148     }
149
150     /**
151      * @notice Disable calling the specified proxy's trade function
152      * @dev Must be called by the current owner
153      * @param proxy The marketplace proxy's address
154      * @param selector The marketplace proxy's function selector
155      */
156     function removeFunction(address proxy, bytes4 selector) external onlyOwner {
157         delete _proxyFunctionSelectors[proxy][selector];
158         emit FunctionRemoved(proxy, selector);
159     }
160
161     /**
162      * @notice Approve marketplaces to transfer ERC20 tokens from the aggregator
163      * @param currency The ERC20 token address to approve
164      * @param marketplace The marketplace address to approve
```

```
165      * @param amount The amount of ERC20 token to approve
166      */
167     function approve(
168         address currency,
169         address marketplace,
170         uint256 amount
171     ) external onlyOwner {
172         _executeERC20Approve(currency, marketplace, amount);
173     }
```

Listing 3.4: Example Privileged Operations in `LooksRareAggregator`

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 mitigated as the team confirms that the privileged `owner` account is a multi-sig.

4 Conclusion

In this audit, we have analyzed the LooksRare Aggregator design and implementation. LooksRare Aggregator is a set of smart contracts that accept trade data from multiple marketplaces or liquidity sources and execute the trades in a single on-chain transaction. This audit focuses on the LooksRareV2Proxy contract, which will support the LooksRare Exchange V2 protocol. During the audit, we note that the current code base is well structured and neatly organized.

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 CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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