



SMART CONTRACT AUDIT REPORT

for

LooksRare Staking Protocol



Prepared By: Yiqun Chen

PeckShield
January 30, 2022

Document Properties

Client	LooksRare
Title	Smart Contract Audit Report
Target	LooksRare Staking
Version	1.0
Author	Xuxian Jiang
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 30, 2022	Xuxian Jiang	Final Release
1.0-rc	January 25, 2022	Yiqun Chen	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About LooksRare	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Timely Update Reward During Reward Rate Changes	11
3.2	Oversized Rewards May Lock All Pool Stakes	12
4	Conclusion	14
	References	15

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the LooksRare Staking 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

The LooksRare token ("LOOKS") is the protocol token from the LooksRare ecosystem. It is an ERC-20 token deployed on the Ethereum blockchain. LOOKS supply was pre-minted with 200 million tokens transferred at the deployment and has a supply cap of 1 billion tokens. The LOOKS token's emission is controlled by the TokenDistributor contract that handles the issuance of freshly minted LOOKS tokens to the TokenDistributor and TokenSplitter contracts. In the FeeSharingSystem contract, LOOKS token holders can deposit LOOKS that are auto-compounded at each user interaction. This smart contract is an additional layer on top of the TokenDistributor contract. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of LooksRare Staking

Item	Description
Name	LooksRare
Website	https://looksrare.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 30, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the scope of this audit only includes the contracts of `FeeSharingSystem.sol`, `LooksRareToken.sol` and `TokenDistributor.sol`.

- <https://github.com/LooksRare/looksrare-contracts.git> (8a3c388)

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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

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.

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

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 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 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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the LooksRare Staking 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 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	1	
Low	1	
Informational	0	
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 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 and 1 low-severity vulnerability.

Table 2.1: Key LooksRare Staking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Timely Update Reward During Reward Rate Changes	Business Logic	Confirmed
PVE-002	Medium	Oversized Rewards May Lock All Pool Stakes	Numeric Errors	Resolved

Besides 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 Timely Update Reward During Reward Rate Changes

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: FeeSharingSystem
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

The FeeSharingSystem contract provides an incentive mechanism that rewards the staking of supported asset (LooksRare token) with the rewardToken. Staking users are rewarded in proportional to their share of tokens in the reward pool.

The rate of the reward in the reward pool can be dynamically adjusted via updateRewards(). When analyzing the reward rate update routine updateRewards(), we notice the need of timely invoking it in harvest()/deposit()/withdraw()/withdrawAll() to update the reward distribution before the new rate of reward becomes effective. If the reward distribution is not immediately updated before changing the reward rate, certain situations may be crafted to create an unfair reward distribution.

```

20     function updateRewards(uint256 reward, uint256 rewardDurationInBlocks) external
        onlyOwner {
21         // Adjust the current reward per block
22         if (block.number >= periodEndBlock) {
23             currentRewardPerBlock = reward / rewardDurationInBlocks;
24         } else {
25             currentRewardPerBlock =
26                 (reward + ((periodEndBlock - block.number) * currentRewardPerBlock)) /
27                 rewardDurationInBlocks;
28         }
29
30         lastUpdateBlock = block.number;
31         periodEndBlock = block.number + rewardDurationInBlocks;
32
33         emit NewRewardPeriod(rewardDurationInBlocks, currentRewardPerBlock, reward);

```

Listing 3.1: FeeSharingSystem::updateRewards()

Recommendation Timely invoke `harvest()/deposit()/withdraw()/withdrawAll()` functions before calling the `updateRewards()` function.

Status Team has acknowledged this issue and plans to only (1) call the `updateRewards()` function after the previous reward period has ended and at least 1 user action (i.e., `deposit/withdraw/harvest`) has been conducted OR (2) in the same block prior to the `updateRewards()` function being called.

3.2 Oversized Rewards May Lock All Pool Stakes

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: FeeSharingSystem
- Category: Numeric Errors [4]
- CWE subcategory: CWE-190 [1]

Description

In this section, we continue to examine the `FeeSharingSystem` logic and focus on the `_rewardPerToken()` routine. This routine is responsible for calculating the reward rate for each staked token.

Our analysis leads to the discovery of a potential pitfall when a new oversized reward amount is added into the pool. In particular, as the `_rewardPerToken()` routine involves the multiplication of three `uint256` integer, it is possible for their multiplication to have an undesirable overflow (lines 279 – 280), especially when the `currentRewardPerBlock` is largely controlled by an external entity (through the `updateRewards()` function).

```

182     function updateRewards(uint256 reward, uint256 rewardDurationInBlocks) external
        onlyOwner {
183         // Adjust the current reward per block
184         if (block.number >= periodEndBlock) {
185             currentRewardPerBlock = reward / rewardDurationInBlocks;
186         } else {
187             currentRewardPerBlock =
188                 (reward + ((periodEndBlock - block.number) * currentRewardPerBlock)) /
189                 rewardDurationInBlocks;
190         }
191
192         lastUpdateBlock = block.number;
193         periodEndBlock = block.number + rewardDurationInBlocks;
194
195         emit NewRewardPeriod(rewardDurationInBlocks, currentRewardPerBlock, reward);

```

196 }

Listing 3.2: FeeSharingSystem::updateReward()

```
272     function _rewardPerToken() internal view returns (uint256) {
273         if (totalShares == 0) {
274             return rewardPerTokenStored;
275         }
276
277         return
278             rewardPerTokenStored +
279             (( _lastRewardBlock() - lastUpdateBlock) * (currentRewardPerBlock *
280                 PRECISION_FACTOR)) /
281             totalShares;
282     }
```

Listing 3.3: FeeSharingSystem::_rewardPerToken()

This issue is made possible if the reward amount is given as the argument to `updateRewards()` such that the calculation of `currentRewardPerBlock.mul(1e18)` always overflows, hence locking all deposited funds. Note that an authentication check on the caller of `updateRewards()` greatly alleviates such concern. Currently, only the `owner` address is able to call `updateRewards()`. Apparently, if the owner is a normal address, it may put users' funds at risk. To mitigate this issue, it is important to transfer the ownership to the governance and ensure the given reward amount will not be oversized to overflow and lock users' funds.

Recommendation Ensure the reward amount is appropriate, without resulting in overflowing and locking users' funds.

Status The team has confirmed that the `updateRewards()` function can only be called by the `FeeSharingSetter` contract, which could make sure the overflow doesn't happen.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `LooksRare Staking` protocol, which provides an incentive mechanism that rewards the staking of supported asset (`LooksRare` token) with the `rewardToken`. 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-190: Integer Overflow or Wraparound. <https://cwe.mitre.org/data/definitions/190.html>.
- [2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [3] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [4] MITRE. CWE CATEGORY: Numeric Errors. <https://cwe.mitre.org/data/definitions/189.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>.