SMART CONTRACT AUDIT REPORT

for

LooksRare Protocol

Prepared By: Patrick Lou

PeckShield

March 22, 2022

Document Properties

Client	LooksRare
Title	Smart Contract Audit Report
Target	LooksRare Protocol
Version	1.0
Author	Patrick Lou
Auditors	Patrick Lou, Xuxian Jiang
Reviewed by	Xiaotao Wu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 22, 2022	Patrick Lou	Final Release

Contact

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

Name	Patrick Lou
Phone	+86 156 0639 2692
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `AggregatorFeeSharingWithUniswapV3` contract, 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 Protocol

The `LooksRare` token ("`LOOKS`") is the protocol token from the `LooksRare` ecosystem, which has a `FeeSharingSystem` contract. In this contract, `LOOKS` token holders can deposit `LOOKS` that are auto-compounded at each user interaction. This audited smart contract is able be used to harvest the pending reward token (`WETH`), sell them for `LOOKS` via `UniswapV3`, and then deposit the `LOOKS` to `FeeSharingSystem` for staking. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of LooksRare Protocol

Item	Description
Name	LooksRare
Website	https://looksrare.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 22, 2022

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

- https://github.com/LooksRare/looksrare-contracts/blob/main/contracts/contracts/tokenStaking

/AggregatorFeeSharingWithUniswapV3.sol (0e8e9cd)

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

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

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 `AggregatorFeeSharingWithUniswapV3` smart contract. 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	0	
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 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 informational recommendation.

Table 2.1: Key LooksRare Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Safe-Version Replacement With safeApprove()	Coding Practices	Resolved
PVE-002	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Safe-Version Replacement With safeApprove()

- ID: PVE-001
- Severity: Informational
- Likelihood: NA
- Impact: NA
- Target: AggregatorFeeSharingWithUniswapV3
- 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 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).

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

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
```

```
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

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

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.

More importantly, the `approve()` function does not have a return value. However, the `IERC20` interface has defined the following `approve()` interface with a `bool` return value: `function approve(address spender, uint256 amount)external returns (bool)`. As a result, the call to `approve()` may expect a return value. With the lack of return value of `USDT`'s `approve()`, the call will be unfortunately reverted.

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `transfer()/transferFrom()` as well, i.e., `safeTransfer()/safeTransferFrom()`. While reviewing the current `AggregatorFeeSharingWithUniswapV3` contract, it comes to our attention that while `safeTransfer()/safeTransferFrom()` has been used, the `approve()` is still being used instead of the safe version `safeApprove()`

In the following, we show the `constructor()` routine in the `AggregatorFeeSharingWithUniswapV3` contract. If the `USDT` token is supported as `rewardTokenAddress`, the unsafe version of `IERC20(rewardTokenAddress).approve(_uniswapRouter, type(uint256).max)` (line 91) may revert as there is no return value in the `USDT` token contract's `approve()` implementation (but the `IERC20` interface expects a return value)!

```
80     constructor(address _feeSharingSystem, address _uniswapRouter) {
81         address looksRareTokenAddress = address(FeeSharingSystem(_feeSharingSystem).
               looksRareToken());
82         address rewardTokenAddress = address(FeeSharingSystem(_feeSharingSystem).
               rewardToken());

84         looksRareToken = IERC20(looksRareTokenAddress);
85         rewardToken = IERC20(rewardTokenAddress);

87         feeSharingSystem = FeeSharingSystem(_feeSharingSystem);
88         uniswapRouter = ISwapRouter(_uniswapRouter);

90         IERC20(looksRareTokenAddress).approve(_feeSharingSystem, type(uint256).max);
```

```
        IERC20(rewardTokenAddress).approve(_uniswapRouter, type(uint256).max);

        tradingFeeUniswapV3 = 3000;
        MINIMUM_DEPOSIT_LOOKS = FeeSharingSystem(_feeSharingSystem).PRECISION_FACTOR();
    }
```

Listing 3.2: AggregatorFeeSharingWithUniswapV3::constructor()

Similarly, `safeApprove()` is suggested to be used for both the `checkAndAdjustLOOKSTokenAllowanceIfRequired()` and the `checkAndAdjustRewardTokenAllowanceIfRequired()` function.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`. And there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Status The issue has been resolved as the team confirms that only `ERC20`-compliant tokens are used in the current protocol.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `AggregatorFeeSharingWithUniswapV3`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the `LooksRare` protocol, there is a privileged `user` account that plays a critical role in governing and regulating the system-wide operations (e.g., system parameters setting and harvest control). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

To elaborate, we show below example privileged routines in the `AggregatorFeeSharingWithUniswapV3` contract. These routines allow the `owner` account to enable/disable harvest, adjust trading fee for `UniswapV3`, pause deposit of the contract, etc.

```
    function startHarvest() external onlyOwner {
        canHarvest = true;

        emit HarvestStart();
    }

```

```
    function stopHarvest() external onlyOwner {
        canHarvest = false;

        emit HarvestStop();
    }

    function updateMinPriceOfLOOKSInWETH(uint256 _newMinPriceLOOKSInWETH) external
        onlyOwner {
        minPriceLOOKSInWETH = _newMinPriceLOOKSInWETH;

        emit NewMinimumPriceOfLOOKSInWETH(_newMinPriceLOOKSInWETH);
    }

    function updateTradingFeeUniswapV3(uint24 _newTradingFeeUniswapV3) external
        onlyOwner {
        require(
            _newTradingFeeUniswapV3 == 10000  _newTradingFeeUniswapV3 == 3000
                _newTradingFeeUniswapV3 == 500,
            "Owner: Fee invalid"
        );

        tradingFeeUniswapV3 = _newTradingFeeUniswapV3;

        emit NewTradingFeeUniswapV3(_newTradingFeeUniswapV3);
    }

    function updateThresholdAmount(uint256 _newThresholdAmount) external onlyOwner {
        thresholdAmount = _newThresholdAmount;

        emit NewThresholdAmount(_newThresholdAmount);
    }

    function pause() external onlyOwner whenNotPaused {
        _pause();
    }
```

Listing 3.3: `PresaleContract::Multiple Functions`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the `owner` may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

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. The team will use a multisig contract for the privileged `owner` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `AggregatorFeeSharingWithUniswapV3` smart contract, which is part of the `LooksRare` protocol to the harvest pending reward token (`WETH`), sell them for `LOOKS` via `UniswapV3`, and then deposit the `LOOKS` to `FeeSharingSystem` for staking.

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-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.