SMART CONTRACT AUDIT REPORT

for

LongFIL

PeckShield
April 23, 2023

Document Properties

Client	LongFIL
Title	Smart Contract Audit Report
Target	LongFIL
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	April 23, 2023	Xuxian Jiang	Final Release
1.0-rc	April 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 designed documents and related smart contract source codes of the LongFIL 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 codes and designed documents, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is, in overall, solid and secure. It can also 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 LongFIL

LongFIL is a Decentralized Finance (DeFi) platform that serves the Filecoin ecosystem. It aims to emphasize high liquidity and security so that investors can subscribe and redeem their investment including interest anytime anywhere. The investment is guarded by the true rewards received by Filecoin Storage Providers. The rewards transferring to LongFIL platform instantly is executed by smart contract that best ensures the security of investments. The protocol is designed to provide the optimum financial services and to bring Filecoin into true DeFi. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The LongFIL Protocol

Item	Description
Name	LongFIL
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 23, 2023

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

- https://github.com/longfil/long-voucher.git (436cf37)

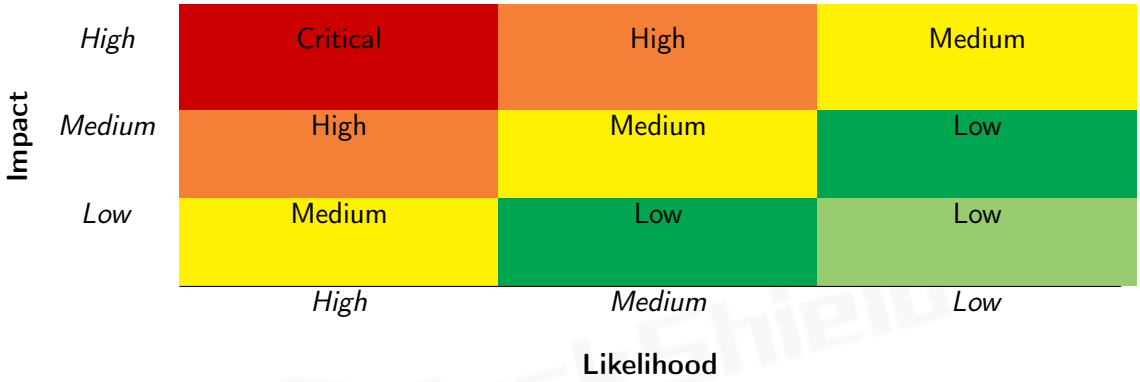
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/longfil/long-voucher.git (3c3049b)

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.

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

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) [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 LongFIL implementation. 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	2	■■
Informational	0	
Total	3	

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 vulnerabilitity and 2 low-severity vulnerabilities.

Table 2.1: Key LongFIL Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Denial-of-Service in ERC3525	Coding Practices	Confirmed
PVE-002	Low	Improved Generation With Intended Events	Coding Practices	Resolved
PVE-003	Medium	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 Possible Denial-of-Service in ERC3525

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ERC3525
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

The LongFIL protocol makes use of the ERC3525 specification to accommodate the semi-fungible nature of the voucher tokens, which is a new type of digital asset with various characteristics. While examining the underlying ERC3525 implementation, we notice it may exhibit an unbounded loop to cause out-of-gas issue.

To elaborate, we show below the related _clearApprovedValues() routine. As the name indicates, this routine clears the approved values for the given tokenId. However, it comes to our attention that the internal for-loop may be unbounded as the tokenData.valueApprovals.length can be manipulated by an untrusted user. Note that another routine _approveValue() shares the same issue. The unbounded loop may cause issues in (unlikely) cases for voucher transfers and approvals.

```
471     function _clearApprovedValues(uint256 tokenId_) internal virtual {
472         TokenData storage tokenData = _allTokens[_allTokensIndex[tokenId_]];
473         uint256 length = tokenData.valueApprovals.length;
474         for (uint256 i = 0; i < length; i++) {
475             address approval = tokenData.valueApprovals[i];
476             delete _approvedValues[tokenId_][approval];
477         }
478         delete tokenData.valueApprovals;
479     }
```

Listing 3.1: ERC3525::_clearApprovedValues()

Recommendation Revise the above logic to avoid the use of unbounded loops for token approvals and transfers.

Status The issue has been confirmed.

3.2 Improved Generation With Intended Events

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Recommendation`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `Recommendation` contract as an example. This contract has a public function that is used to bind a specific `referrer`. While examining the event that reflects the `referrer` binding, we notice the emitted event contains incorrect information. Specifically, the `Bind` event is defined as `event Bind(address indexed referrer, address referral, uint256 atBlock)` with `referrer` and `referral` as the first and second argument accordingly. However, the resulting event is emitted with these two arguments out of order.

```
97      function bind(address referrer, uint8 v, bytes32 r, bytes32 s) external {
98          require(isReferrer(referrer), "missing qualification");
100         address referral = ECDSAUpgradeable.recover(
101             _hashTypedDataV4(
102                 keccak256(abi.encode(RECOMMENDATION_TYPEHASH, referrer))
103             ),
104             v,
105             r,
106             s
107         );
108         require(referrer != referral, "illegal referral");
109         require(!_existsReferralData(referral), "already bind");
111         // update storage
112         ReferralData storage recommendation = _referralReferralData[referral];
113         recommendation.referrer = referrer;
```

```
114         recommendation.bindAt = block.number;
116         emit Bind(referral, referrer, block.number);
117     }
```

Listing 3.2: `Recommendation::bind()`

Recommendation Properly emit the respective event when a `referrer` is being binded.

Status This issue has been fixed in the following commit: `3c3049b`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the `LongFIL` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configure protocol parameters and assign various roles). In the following, we use the `ProductCenter` contract as an example and show the representative functions potentially affected by the privileges of the owner.

```
51     function setInterestRate(
52         uint256 productId,
53         address interestRate_
54     ) external onlyRole(OPERATOR_ROLE) {
55         require(interestRate_ != address(0), Errors.ZERO_ADDRESS);
56         _requireExistsProduct(productId);
57
58         ProductParameters storage parameters = _allProducts[_allProductsIndex[productId
               ]].parameters;
59         require(
60             block.number < parameters.beginSubscriptionBlock
61                 block.number >= parameters.endSubscriptionBlock,
62             Errors.INVALID_PRODUCT_STAGE
63         );
64
65         address oldInterestRate = address(parameters.interestRate);
66         parameters.interestRate = IInterestRate(interestRate_);
67
68         emit InterestRateChanged(productId, oldInterestRate, interestRate_);
69     }
70
```

```
71      function setRecommendationCenter(address recommendationCenter_) external onlyRole(
            ADMIN_ROLE) {
72          require(recommendationCenter == address(0), Errors.RECOMMENDATION_CENTER);
73          require(recommendationCenter_ != address(0), Errors.ZERO_ADDRESS);
74
75          // test
76          IRecommendationCenter(recommendationCenter_).beforeEquitiesTransfer(address(this
                ), 0, address(0), address(0), 0, 0, 0);
77
78          // set storage
79          recommendationCenter = recommendationCenter_;
80
81          emit SetRecommendationCenter(recommendationCenter_);
82      }
```

Listing 3.3: Example Privileged Operations in the `ProductCenter` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the privileged account may also be a counter-party risk to the protocol users. It is worrisome if the privileged 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 accounts 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 confirmed and the team plans to use a multi-sig to manage the admin account.

4 Conclusion

In this audit, we have analyzed the LongFIL protocol design and implementation. LongFIL is a Decentralized Finance (DeFi) platform that serves the Filecoin ecosystem. It aims to emphasize high liquidity and security so that investors can subscribe and redeem their investment including interest anytime anywhere. The investment is guarded by the true rewards received by Filecoin Storage Providers. The rewards transferring to LongFIL platform instantly is executed by smart contract that best ensures the security of investments. The protocol is designed to provide the optimum financial services and to bring Filecoin into true DeFi. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that 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.