SMART CONTRACT AUDIT REPORT

for

LombardFi Protocol

Prepared By: Xiaomi Huang

PeckShield

November 23, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0-rc	November 23, 2022	Luck Hu	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 design document and related smart contract source code of the `LombardFi` 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About LombardFi Protocol

`LombardFi` is a permission-less, reputation-based liquidity protocol facilitating fixed-term borrowing of all the assets on the `Ethereum` network. It aims to bridge the gap between institutional investors, deploying sophisticated trading infrastructure and strategies on one hand, and the passive crypto investors on another. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of LombardFi Protocol

Item	Description
Name	LombardFi
Website	https://lombard.fi/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 23, 2022

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

- https://github.com/lombardfi/peckshield-audit.git (ff2deb8)

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

- https://github.com/lombardfi/peckshield-audit.git (TBD)

1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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 is considered safe regarding the check item. For any discovered issue, we might further deploy

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

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) [8], 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 `LombardFi` smart contracts. 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	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 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 recommendations.

Table 2.1: Key LombardFi Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Removal of Unused Ownable/ReentrancyGuard	Coding Practices	
PVE-002	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	
PVE-003	Informational	Suggested immutable Usage for Gas Efficiency	Coding Practices	
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	

Beside 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 Removal of Unused Ownable/ReentrancyGuard

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pool/ChainlinkOracleAdapter`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

LombardFi Protocol makes use of a number of reference libraries and contracts, such as `SafeERC20`, `Ownable`, and `ReentrancyGuard`, to facilitate the protocol implementation and organization. For example, the `Router` smart contract has so far imported at least three reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `Pool` contract, it imports the `ReentrancyGuard` contract from `OpenZeppelin`. However, it comes to our attention that the `nonReentrant` modifier is not used anywhere in the `Pool` contract. Hence the import of the `ReentrancyGuard` can be safely removed. Similarly, the import of the `Ownable` in the `ChainlinkOracleAdapter` contract can be safely removed as well, as the `onlyOwner` modifier is not used at all in the `ChainlinkOracleAdapter` contract.

```
contract Pool is IPool, ReentrancyGuard {
using SafeERC20 for IERC20;

/**
 * @notice Address of the borrower.
 */
address public borrower;
...
}
```

Listing 3.1: Pool.sol

Recommendation Consider the removal of the unused reference contracts.

Status

3.2 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `PoolFactory/Router`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `LombardFi` protocol, the `Router` contract is designed to be the main entry point for interaction with users. In particular, one entry routine, i.e., `deposit()`, transfers the `lentAsset` from the user to the specified pool and records the corresponding deposit amount in the pool. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of `LombardFi`. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
  function deposit(uint256 _pid, uint256 _amt)
  external
  nonZero(_amt)
  whenNotPaused
  nonReentrant
{
  IPool pool = _getPool(_pid);
  ...
  // Transfer lent asset from lender to pool and perform accounting.
  address lentAsset = pool.lentAsset();
  IERC20(lentAsset).safeTransferFrom(msg.sender, address(pool), _amt);
  pool.deposit(msg.sender, _amt);

  emit Deposit(_pid, lentAsset, _amt);
}
```

Listing 3.2: Router::deposit()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the `Router` contract before and after the `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted to be the collateral tokens. In fact, the protocol is indeed in the position to effectively regulate the set of assets that can be used as collaterals. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Note the same issue is also applicable to the `PoolFactory::createPool()/Router::borrow()/Router::repay()` routines.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status

3.3 Suggested immutable Usage for Gas Efficiency

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PoolFactory`
- Category: Coding Practices [6]
- CWE subcategory: CWE-561 [3]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the `LombardFi` protocol, we observe there are several variables that need not to be updated dynamically. They can be declared as `immutable` for gas efficiency. For example, the `router` state variable in the `PoolFactory` contract can be declared as `immutable`, as it doesn't need to be updated dynamically.

```
contract PoolFactory is IPoolFactory, Ownable, ReentrancyGuard {
    using SafeERC20 for IERC20;

    /**
     * @notice Address of Router contract.
     */
    address public router;
    ...
}
```

Listing 3.3: `PoolFactory.sol`

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `LombardFi` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the system-wide operations (e.g., set the price oracles for the supported assets). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `Router` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in `Router` allow for the `owner` to set the `poolFactory` which is used to retrieve the pool per id, set the `oracleManager` which is used to retrieve assets prices, set `treasury` address which is used to receive the borrow fee, pause/unpause the normal operations, etc.

```
    function setFactory(IPoolFactory _poolFactory)
        external
        nonZeroAddress(address(_poolFactory))
        onlyOwner
    {
        poolFactory = _poolFactory;
        emit FactorySet(msg.sender, address(_poolFactory));
    }

    /**
     * @notice Set the OracleManager implementation address. Callable only by the owner.
     * @dev Throws an error if the address is the zero address.
     * @param _oracleManager The new implementation.
     */
    function setOracleManager(IOracleManager _oracleManager)
        external
        nonZeroAddress(address(_oracleManager))
        onlyOwner
    {
        oracleManager = _oracleManager;
        emit OracleManagerSet(msg.sender, address(_oracleManager));
    }

    /**
     * @notice Set the treasury address. Callable only by the owner.
     * @dev Throws an error if the address is the zero address.
     * @param _treasury The new recipient.
     */
    function setTreasury(address _treasury)
        external
        nonZeroAddress(_treasury)
        onlyOwner
    {
        treasury = _treasury;
        emit TreasurySet(msg.sender, _treasury);
    }

    /**
     * @notice Pause the contract. Callable only by the owner.
     */
    function pause() external onlyOwner {
        _pause();
    }

    /**
     * @notice Unpause the contract. Callable only by the owner.
     * @dev The contract must be paused to unpause it.
     */
```

```
    function unpause() external onlyOwner {
        _unpause();
    }
```

Listing 3.4: Example Privileged Operations in the `Router` Contract

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 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

4 Conclusion

In this audit, we have analyzed the design and implementation of the `LombardFi` protocol. `LombardFi` is a permissionless, reputation-based liquidity protocol facilitating fixed-term borrowing of all the assets on the `Ethereum` network. It aims to bridge the gap between institutional investors, deploying sophisticated trading infrastructure and strategies on one hand, and the passive crypto investors on another. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[3] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.