SMART CONTRACT AUDIT REPORT

for

LeverFi TokenSwap

Prepared By: Xiaomi Huang

PeckShield

May 11, 2022

Document Properties

Client	LeverFi
Title	Smart Contract Audit Report
Target	LeverFi TokenSwap
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	May 11, 2022	Xiaotao Wu	Final Release
1.0-rc1	May 10, 2022	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 source code of the `LeverFi TokenSwap` smart 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 LeverFi TokenSwap

`LeverFi` is an on-chain leverage trading platform using yield-bearing collateral. Trade and earn can be carried out at the same time with up to 10x leverage. The audited `TokenSwap` contract allows for swapping from one token to a different token with defined exchange ratio. The basic information of the audited feature is as follows:

Table 1.1: Basic Information of The LeverFi TokenSwap

Item	Description
Issuer	LeverFi
Website	https://www.leverfi.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 11, 2022

In the following, we show the Git repository of reviewed file and the commit hash value used in this audit. Note this audit only covers the `TokenSwap` contract.

- https://github.com/LeverFi/leverfi-token.git (c9a3807)

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

- https://github.com/LeverFi/leverfi-token.git (dfb4691)

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 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 `LeverFi TokenSwap` 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	0	
Low	1	■
Undetermined	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 undetermined issue.

Table 2.1: Key LeverFi TokenSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Sanity Checks Of System/-Function Parameters	Coding Practices	Resolved
PVE-002	Undetermined	Improved Reentrancy Protection In TokenSwap	Time and State	Resolved

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 Improved Sanity Checks Of System/Function Parameters

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

Description

In the TokenSwap contract, the tokenSwap() function is used to swap from inToken to outToken with the defined exchange ratio. While reviewing the implementation of this routine, we notice that it can benefit from additional sanity checks.

To elaborate, we show below the implementation of the tokenSwap() function. Specifically, the current implementation fails to check the given argument in inAmount. As a result, a user could tokenSwap(0) and transfer zero tokens, which is a waste of gas.

```
    // @dev swap inToken to outToken with respected exchangeRatio
    function tokenSwap(uint256 inAmount) public {

        // Transfer the inToken from the caller to the burn address
        IERC20(inTokenAddress).safeTransferFrom(
            address(msg.sender),
            burnAddress,
            inAmount
        );

        // Calculate the amount to send to the caller
        uint256 outAmount = inAmount * exchangeRatio / 1e18;

        // Transfer the outToken from the wallet to the caller
        IERC20(outTokenAddress).safeTransferFrom(
            outTokenWallet,
            address(msg.sender),
            outAmount
```

```
48          );
49
50          // Emit an event
51          emit TokenSwapped(msg.sender, inAmount, outAmount);
52      }
```

Listing 3.1: `TokenSwap::tokenSwap()`

Recommendation Validate the input arguments by ensuring `inAmount > 0` in the above `tokenSwap()` function.

Status This issue has been fixed in the following commit: `dfb4691`.

3.2 Improved Reentrancy Protection In TokenSwap

- ID: PVE-002
- Severity: Undetermined
- Likelihood: Low
- Impact: Low
- Target: `TokenSwap`
- Category: Time and State [3]
- CWE subcategory: CWE-362 [2]

Description

As mentioned in Section 3.1, the `tokenSwap()` function of the `TokenSwap` contract is used to swap from `inToken` to `outToken` with defined exchange ratio. Our analysis shows there is a potential reentrancy issue in the function.

To elaborate, we show below the code snippet of the `tokenSwap()` function. In this function, the `inToken` will be transferred from the caller to the `burnAddress` (lines 34-38) and the `outToken` will be transferred from the `outTokenWallet` to the caller (lines 44-48). If the `inToken` or the `outToken` faithfully implements the ERC777-like standard, then the `tokenSwap()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend()` and `tokensReceived()` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In our case, the above hook can be planted in `IERC20(inTokenAddress).safeTransferFrom()` (line 34) or `IERC20(outTokenAddress).safeTransferFrom()` (line 44) before the actual transfer of the underlying assets occurs. So far, we also do not know how an attacker can exploit this issue to earn profit. After internal discussion, we consider it is necessary to bring this issue up to the team. Though the implementation of the `tokenSwap()` function is well designed, we may intend to use the `ReentrancyGuard::nonReentrant` modifier to protect the `tokenSwap()` function at the whole protocol level.

```
    // @dev swap inToken to outToken with respected exchangeRatio
    function tokenSwap(uint256 inAmount) public {

        // Transfer the inToken from the caller to the burn address
        IERC20(inTokenAddress).safeTransferFrom(
            address(msg.sender),
            burnAddress,
            inAmount
        );

        // Calculate the amount to send to the caller
        uint256 outAmount = inAmount * exchangeRatio / 1e18;

        // Transfer the outToken from the wallet to the caller
        IERC20(outTokenAddress).safeTransferFrom(
            outTokenWallet,
            address(msg.sender),
            outAmount
        );

        // Emit an event
        emit TokenSwapped(msg.sender, inAmount, outAmount);
    }
```

Listing 3.2: `TokenSwap::tokenSwap()`

Recommendation Apply the non-reentrancy protection in the above-mentioned routine.

Status This issue has been fixed in the following commit: `dfb4691`.

4 Conclusion

In this audit, we have analyzed the `LeverFi TokenSwap` design and implementation. The audited `TokenSwap` contract allows for swapping from one token to a different token with the defined exchange ratio. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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.