SMART CONTRACT AUDIT REPORT

for

OpenLeverage OLE V2

Prepared By: Xiaomi Huang

PeckShield

November 2, 2023

Document Properties

Client	OpenLeverage
Title	Smart Contract Audit Report
Target	OLEv2
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Final

Version Info

Version	Date	Author(s)	Description
1.0	November 2, 2023	Xuxian Jiang	Final Release
1.0-rc1	October 31, 2023	Xuxian Jiang	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 design document and related smart contract source code of the `OpenLeverage OLE V2` 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 OpenLeverage OLE V2

The `OpenLeverage` protocol is a permissionless margin trading protocol that enables traders or other applications to be long or short on any trading pair on `DEXs` efficiently and securely. This audit covers the upgrading of the `OLEV1` token to the `OLEV2` token, which implements `LayerZero`'s `OFT` token standard. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of OLEv2

Item	Description
Name	OpenLeverage
Website	https://openleverage.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 2, 2023

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

- https://github.com/OpenLeverageDev/ole-v2-contracts.git (72b3535)

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

- https://github.com/OpenLeverageDev/ole-v2-contracts.git (7fc89e9)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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) [7], 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 `OpenLeverage OLE V2` 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	0	
Low	3	🟦🟦🟦
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 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 3 low-severity vulnerabilities.

Table 2.1: Key OLEv2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Reentrancy Protection in RewardVault	Time and State	Resolved
PVE-002	Low	Improved Early Exit Withdraw Logic in RewardDistributor	Business Logic	Resolved
PVE-003	Low	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 Suggested Reentrancy Protection in RewardVault

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardVault`
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [11] exploit, and the `Uniswap/Lendf.Me` hack [10].

We notice there are occasions where the reentrancy protection can be improved. Using the `RewardVault` as an example, the `newTranche()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 66) start before effecting the update on internal states. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
64      function newTranche(uint256 total, IERC20 token, uint64 startTime, uint64 endTime,
            uint128 ruleFlag) external payable {
65          require(startTime > block.timestamp && endTime > startTime && total > 0 &&
                ruleFlag > 0, "Incorrect inputs");
66          uint256 _transferIn = transferIn(msg.sender, token, total);
67          uint256 _trancheId = ++trancheIdx;
```

```
        uint64 expireTime = endTime + defaultExpireDuration;
        tranches[_trancheId] = Tranche(_INIT_MERKLE_ROOT, startTime, endTime, expireTime
            , _transferIn, 0, 0, 0, 0, msg.sender, token);
        emit TrancheAdded(_trancheId, startTime, endTime, expireTime, _transferIn, msg.
            sender, token, ruleFlag);
    }
```

Listing 3.1: `RewardVault::newTranche()`

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`, it is important to take precautions to thwart possible `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`. Note another routine `updateTranche()` can be similarly improved.

Status The issue has been fixed by this commit: `7fc89e9`.

3.2 Improved Early Exit Withdraw Logic in RewardDistributor

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardDistributor`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The audited protocol has a core `RewardDistributor` contract that is designed to distribute rewards. The rewards for distribution may subject to certain lockup period. While it allows the user to make early withdrawal, there is a penalty imposed for the withdrawal. Our analysis on the early withdrawal logic shows it can be improved.

Specifically, if we examine the `_earlyExitWithdrawable()` implementation, it computes the penalty as `penalty = locked * penaltyFactor / PERCENT_DIVISOR` where `penaltyFactor` is calculated as `(endTime - block.timestamp)* epoch.penaltyAdd / epoch.vestDuration + epoch.penaltyBase` (line 242). As a result, it is possible to compute `penalty` to be larger than `locked`. If it does occur, the execution may be reverted due to the next arithmetic underflow (line 245).

```
236    function _earlyExitWithdrawable(Reward memory reward, uint256 epochId) internal view
           returns (uint256 withdrawable, uint256 penalty) {
237        Epoch memory epoch = epochs[epochId];
238        uint256 releaseable = _releaseable(reward, epoch);
239        withdrawable = releaseable - reward.withdrawn;
```

```
        // cal penalty
        uint256 endTime = reward.vestStartTime + epoch.vestDuration;
        uint256 penaltyFactor = (endTime - block.timestamp) * epoch.penaltyAdd / epoch.
            vestDuration + epoch.penaltyBase;
        uint256 locked = reward.amount - releaseable;
        penalty = locked * penaltyFactor / PERCENT_DIVISOR;
        withdrawable += locked - penalty;
        return (withdrawable, penalty);
    }
```

Listing 3.2: `RewardDistributor::_earlyExitWithdrawable()`

Recommendation Revise the above function to ensure the computed `penaltyFactor` is always smaller than `PERCENT_DIVISOR`.

Status The issue has been fixed by this commit: `7fc89e9`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the audited protocol, there is a privileged `admin` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and pool adjustment). 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 their related privileged accesses in current contracts.

```
    function setExpireDuration(uint64 _defaultExpireDuration) external onlyAdmin {
        require (_defaultExpireDuration > 0, "Incorrect inputs");
        defaultExpireDuration = _defaultExpireDuration;
    }

    function setDistributor(address _distributor) external onlyAdmin {
        distributor = _distributor;
    }

    /// @notice Only the distributor can set the tranche reward distribute info.
    /// @dev If the reward is not distributed for some reason, the merkle root will be
        set with 1.
    /// @param _undistributed The reward of not distributed.
    /// @param _distributed The reward of distributed.
    /// @param _tax tax fund to admin.
    /// @param _merkleRoot reward tree info.
    function setTrancheTree(uint256 _trancheId, uint256 _undistributed, uint256
        _distributed, uint256 _tax, bytes32 _merkleRoot) external {
        require(msg.sender == distributor, "caller must be distributor");
        Tranche storage tranche = tranches[_trancheId];
        require(tranche.endTime < block.timestamp, 'Not end');
        require(_undistributed + _distributed + _tax == tranche.total, 'Incorrect inputs
            ');
        tranche.unDistribute = _undistributed;
        tranche.merkleRoot = _merkleRoot;
        tranche.tax = _tax;
        taxFund[tranche.token] = taxFund[tranche.token] + _tax;
        emit TrancheTreeSet(_trancheId, _undistributed, _distributed, _tax, _merkleRoot)
            ;
    }
```

Listing 3.3: Example Privileged Operations in `RewardVault` Contract

In addition, we notice the `admin` account that is able to add new markets and configure various liquidity. Apparently, if the privileged `admin` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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 by confining the scope of the admin key.

4 Conclusion

In this audit, we have analyzed the design document and related smart contract source code of the OpenLeverage OLE V2 protocol. OpenLeverage is a permissionless margin trading protocol that enables traders or other applications to be long or short on any trading pair on DEXs efficiently and securely. This audit covers the upgrading of the OLEV1 token to the OLEV2 token, which implements LayerZero's OFT token standard. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

[6] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.