SMART CONTRACT AUDIT REPORT

for

OpenLeverage

Prepared By: Xiaomi Huang

PeckShield

November 6, 2022

Document Properties

Client	OpenLeverage
Title	Smart Contract Audit Report
Target	OpenLeverage
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 6, 2022	Xuxian Jiang	Final Release
1.0-rc1	November 2, 2021	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 **OpenLeverage** design document and related smart contract source code, 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

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. In particular, it enables margin trading with liquidity on various DEXs, hence connecting traders to trade with the most liquid decentralized markets. It is also designed to have two separated pools for each pair with different risk and interest rate parameters, allowing lenders to invest according to the risk-reward ratio. The governance token `OLE` is minted based on the protocol usage and can be used to vote and stake to get rewards and protocol privileges. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of OpenLeverage

Item	Description
Issuer	OpenLeverage
Website	https://openleverage.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 6, 2022

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

- https://github.com/OpenLeverageDev/openleverage-contracts.git (c9ce7c3)

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

- https://github.com/OpenLeverageDev/openleverage-contracts.git (115f6d0)

1.2 About PeckShield

PeckShield Inc. [8] 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 [7]:

- 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.

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

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 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) [6], 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

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `OpenLeverage` 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	1	■
Low	1	■
Informational	1	■
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 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key OpenLeverage Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited Logic in payoffTrade()	Business Logic	Resolved
PVE-002	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-003	Informational	Revisited Interest Rate Calculation in LTimePool	Business Logic	Resolved

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 Revisited Logic in payoffTrade()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OpenLevV1`
- Category: Business Logic [5]
- CWE subcategory: CWE-770 [2]

Description

The `OpenLeverage` protocol supports permissionless margin trading markets. A user may provide assets as collateral to borrow more assets from the protocol. While examining the current logic to pay off an open trade, we notice the current implementation can be improved.

To elaborate, we show below the related routine `payoffTrade()`. As the name indicates, the routine is used to close the borrow position by paying off all current debt. It comes to our attention that the method to retrieve the remaining debt after the payment is `marketVars.buyPool.borrowBalanceCurrent(msg.sender)` (line 293), which can be improved by making use of `marketVars.buyPool.borrowBalanceStored()(msg.sender)` to avoid repeated calculation of interest accrual.

```
275     function payoffTrade(uint16 marketId, bool longToken) external payable override
            nonReentrant {
276         Types.Trade storage trade = activeTrades[msg.sender][marketId][longToken];
277         bool depositToken = trade.depositToken;
278         uint deposited = trade.deposited;
279         Types.MarketVars memory marketVars = toMarketVar(longToken, false, markets[
                marketId]);
280
281         //verify
282         require(trade.held != 0 && trade.lastBlockNum != block.number, "HI0");
283         (ControllerInterface(addressConfig.controller)).closeTradeAllowed(marketId);
284         uint heldAmount = trade.held;
285         uint closeAmount = OpenLevV1Lib.shareToAmount(heldAmount, totalHelds[address(
                marketVars.sellToken)], marketVars.reserveSellToken);
286         uint borrowed = marketVars.buyPool.borrowBalanceCurrent(msg.sender);
```

```
        //first transfer token to OpenLeve, then repay to pool, two transactions with
            two tax deductions
        uint24 taxRate = taxes[marketId][address(marketVars.buyToken)][0];
        uint firstAmount = Utils.toAmountBeforeTax(borrowed, taxRate);
        uint transferAmount = transferIn(msg.sender, marketVars.buyToken, Utils.
            toAmountBeforeTax(firstAmount, taxRate), true);
        marketVars.buyPool.repayBorrowBehalf(msg.sender, transferAmount);
        require(marketVars.buyPool.borrowBalanceCurrent(msg.sender) == 0, "IRP");
        delete activeTrades[msg.sender][marketId][longToken];
        totalHelds[address(marketVars.sellToken)] = totalHelds[address(marketVars.
            sellToken)].sub(heldAmount);
        doTransferOut(msg.sender, marketVars.sellToken, closeAmount);

        emit TradeClosed(msg.sender, marketId, longToken, depositToken, heldAmount,
            deposited, heldAmount, 0, 0, 0);
    }
```

Listing 3.1: `OpenLevV1::payoffTrade()`

In addition, in order to support tokens with tax, the above logic involves two transfers, which may introduce unecessary friction or cost. The first one transfers the bespoke token to `OpenLev`, and the second one repays to the pool. It would be helpful to minimize the friction by involving only one transfer for tokens with tax.

Recommendation Revisit the above `payoffTrade()` logic to optimize the gas/tax cost. A similar gas optimization can be applied to the `OpenLevV1Lib::moveInsurance()` routine.

Status This issue has been fixed in the commit: 115f6d0.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `OpenLeverage` 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 marketing 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 setAddressConfig(address controller, DexAggregatorInterface dexAggregator)
        external override onlyAdmin() {
        OpenLevV1Lib.setAddressConfigInternal(controller, dexAggregator, addressConfig);
        emit NewAddressConfig(controller, address(dexAggregator));
    }

    function setMarketConfig(uint16 marketId, uint16 feesRate, uint16 marginLimit,
        uint16 priceDiffientRatio, uint32[] memory dexs) external override onlyAdmin() {
        OpenLevV1Lib.setMarketConfigInternal(feesRate, marginLimit, priceDiffientRatio,
            dexs, markets[marketId]);
        emit NewMarketConfig(marketId, feesRate, marginLimit, priceDiffientRatio, dexs);
    }

    /// @notice List of all supporting Dexes.
    /// @param poolIndex index of insurance pool, 0 for token0, 1 for token1
    function moveInsurance(uint16 marketId, uint8 poolIndex, address to, uint amount)
        external override nonReentrant() onlyAdmin() {
        Types.Market storage market = markets[marketId];
        OpenLevV1Lib.moveInsurance(market, poolIndex, to, amount, totalHelds);
    }

    function setSupportDex(uint8 dex, bool support) public override onlyAdmin() {
        supportDexs[dex] = support;
    }

    function setTaxRate(uint16 marketId, address token, uint index, uint24 tax) external
         override onlyAdmin() {
        taxes[marketId][token][index] = tax;
    }
```

Listing 3.2: Example `Setters` in the `OpenLevV1` Contract

In addition, we notice the `admin` account that is able to add new markets and grant specified `pool0/pool1` with the access to the contract funds. 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.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 confirmed with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

3.3 Revisited Interest Rate Calculation in LTimePool

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LTimePool`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `OpenLeverage` protocol has a new `LTimePool` contract to facilitate assets borrowing. In this section, we examine the interest calculation from the use of this new `LTimePool`.

Specifically, if we examine the `totalBorrowsCurrent()` function, the latest borrow amount is computed by considering the current borrow interest rate, the elapsed timestamp, as well as current borrow amount. It comes to our attention that the current logic makes an implicit assumption of the block time is equal to 1 second, which may not be the case in the deployed blockchain.

```
    function totalBorrowsCurrent() external override view returns (uint) {
        /* Remember the initial block timestamp */
        uint currentBlockTimestamp = getBlockTimestamp();
        uint accrualBlockTimestampPrior = accrualBlockTimestamp;

        /* Short-circuit accumulating 0 interest */
        if (accrualBlockTimestampPrior == currentBlockTimestamp) {
            return totalBorrows;
        }

        /* Read the previous values out of storage */
        uint cashPrior = getCashPrior();
        uint borrowsPrior = totalBorrows;
        uint reservesPrior = totalReserves;

        /* Calculate the current borrow interest rate */
        uint borrowRateMantissa = getBorrowRateInternal(cashPrior, borrowsPrior,
            reservesPrior);
        require(borrowRateMantissa <= borrowRateMaxMantissa, "borrower rate higher");

        /* Calculate the number of timestamp elapsed since the last accrual */
        (MathError mathErr, uint blockDelta) = subUInt(currentBlockTimestamp,
            accrualBlockTimestampPrior);
        require(mathErr == MathError.NO_ERROR, "calc block delta erro");
```

```
        Exp memory simpleInterestFactor;
        uint interestAccumulated;
        uint totalBorrowsNew;

        (mathErr, simpleInterestFactor) = mulScalar(Exp({mantissa : borrowRateMantissa}), blockDelta);
        require(mathErr == MathError.NO_ERROR, 'calc interest factor error');

        (mathErr, interestAccumulated) = mulScalarTruncate(simpleInterestFactor, borrowsPrior);
        require(mathErr == MathError.NO_ERROR, 'calc interest acc error');

        (mathErr, totalBorrowsNew) = addUInt(interestAccumulated, borrowsPrior);
        require(mathErr == MathError.NO_ERROR, 'calc total borrows error');

        return totalBorrowsNew;
    }
```

Listing 3.3: `LTimePool::totalBorrowsCurrent()`

Recommendation Revisit the implicit assumption of using 1 second as the block time.

Status This issue has been resolved as the team clarifies that the `borrowRate` is defined not on the block numbers, but the timestamps.

4 Conclusion

In this audit, we have analyzed the `OpenLeverage` design and implementation. The system presents a unique, robust offering as 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. 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-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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