SMART CONTRACT AUDIT REPORT

for

OpenLeverage Protocol

Prepared By: Yiqun Chen

PeckShield

December 18, 2021

Document Properties

Client	OpenLeverage
Title	Smart Contract Audit Report
Target	OpenLeverage
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 18, 2021	Xuxian Jiang	Final Release
1.0-rc1	December 15, 2021	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
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 OpenLeverage 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	December 18, 2021

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 (3c0ab75)

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 (bb3aa57)

1.2 About PeckShield

PeckShield Inc. [14] 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 [13]:

- 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) [12], 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	2	■■
Low	4	■■■■
Informational	0	
Total	6	

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 2 medium-severity vulnerabilities and 4 low-severity vulnerabilities.

Table 2.1: Key OpenLeverage Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improper Funding Source In XOLE::_deposit_for()	Business Logic	Fixed
PVE-002	Medium	Improper TotalSupplyCheckPoints in XOLE	Business Logic	Fixed
PVE-003	Low	Oversized Rewards May Lock All Pool Stakes	Numeric Errors	Fixed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practice	Fixed
PVE-005	Low	Possible Insurance Reduction in Liquidation	Time and State	Mitigated
PVE-006	Medium	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 Improper Funding Source In XOLE::_deposit_for()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `XOLE`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `OpenLeverage` protocol allows users to obtain the governance `xOLE` tokens by locking `OLE` tokens. While reviewing the current locking logic, we notice the key helper routine `_depositFor()` needs to be revised.

To elaborate, we show below the implementation of this `_deposit_for()` helper routine. In fact, it is an internal function to perform deposit and lock `OLE` for a user. This routine has a number of arguments and the first one `_addr` is the address to receive the `xOLE` balance. It comes to our attention that the `_addr` address is also the one to actually provide the assets, `assert(IERC20(oleToken).transferFrom(_addr, address(this), _value))` (line 292). In fact, the `msg.sender` should be the one to provide the assets for locking! Otherwise, this function may be abused to lock `xOLE` tokens from users who have approved the locking contract before without their notice.

```
280     function _deposit_for(address _addr, uint256 _value, uint256 unlock_time,
            LockedBalance memory _locked, int128 _type) internal updateReward(msg.sender) {
281         uint256 locked_before = totalLocked;
282         totalLocked = locked_before.add(_value);
283         // Adding to existing lock, or if a lock is expired - creating a new one
284         _locked.amount = _locked.amount.add(_value);

286         if (unlock_time != 0) {
287             _locked.end = unlock_time;
288         }
289         locked[_addr] = _locked;
```

```
        if (_value != 0) {
            assert(IERC20(oleToken).transferFrom(_addr, address(this), _value));
        }

        uint calExtraValue = _value;
        // only increase unlock time
        if (_value == 0) {
            _burn(_addr);
            calExtraValue = locked[_addr].amount;
        }
        uint weekCount = locked[_addr].end.sub(block.timestamp).div(WEEK);
        if (weekCount > 1) {
            uint extraToken = calExtraValue.mul(oneWeekExtraRaise).mul(weekCount - 1).
                div(10000);
            _mint(_addr, calExtraValue + extraToken);
        } else {
            _mint(_addr, calExtraValue);
        }
        emit Deposit(_addr, _value, _locked.end, _type, block.timestamp);
    }
```

Listing 3.1: XOLE::_deposit_for()

In addition, the above function has a modifier `updateReward(msg.sender)` that timely updates the reward for the given `msg.sender`. However, given the possibility that the given `_addr` may be different from `msg.sender`, it is also suggested to revise the above modifier to `updateReward(_addr)`.

Recommendation Revise the above helper routine to use the right funding source to transfer the assets for locking. Also properly adjust the associated modifier.

Status The issue has been fixed in the following commit: 024945b.

3.2 Improper TotalSupplyCheckPoints in XOLE

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `XOLE`
- Category: Business Logic [9]
- CWE subcategory: CWE-770 [5]

Description

As mentioned in Section 3.1, the `OpenLeverage` protocol allows users to obtain the governance `xOLE` tokens by locking `OLE` tokens. The governance token also comes with the checkpoint feature that allows to query the total supply at a specific block number. Our examination shows that the current checkpoint implementation logic can be further improved.

In particular, the actual checkpoints are maintained in the `_updateTotalSupplyCheckPoints()` routine, which is shown below. It comes to our attention that it simply adds a new checkpoint without maintaining the invariant of having at most a checkpoint for a particular block number. The presence of multiple checkpoints with the same block number could greatly affect the governance functionality.

```
    function _mint(address account, uint amount) internal {
        totalSupply = totalSupply.add(amount);
        balances[account] = balances[account].add(amount);
        emit Transfer(address(0), account, amount);
        if (delegates[account] == address(0)) {
            delegates[account] = account;
        }
        _moveDelegates(address(0), delegates[account], amount);
        _updateTotalSupplyCheckPoints();
    }

    function _burn(address account) internal {
        uint burnAmount = balances[account];
        totalSupply = totalSupply.sub(burnAmount);
        balances[account] = 0;
        emit Transfer(account, address(0), burnAmount);
        _moveDelegates(delegates[account], address(0), burnAmount);
        _updateTotalSupplyCheckPoints();
    }

    function _updateTotalSupplyCheckPoints() internal {
        uint32 blockNumber = safe32(block.number, "block number exceeds 32 bits");
        totalSupplyCheckpoints[totalSupplyNumCheckpoints] = Checkpoint(blockNumber,
            totalSupply);
        totalSupplyNumCheckpoints = totalSupplyNumCheckpoints + 1;
    }
```

Listing 3.2: `XOLE::_mint()/_burn()/_updateTotalSupplyCheckPoints()`

Recommendation Correct the above checkpoint implementation to ensure there is at most a chheckpoint for a block number.

Status This issue has been fixed in the commit: c0876b2.

3.3 Oversized Rewards May Lock All Pool Stakes

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: FarmingPool
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [2]

Description

The `OpenLeverage` protocol shares an incentivizer mechanism inspired from `Synthetix`. In this section, we focus on a routine, i.e., `rewardPerToken()`, which is responsible for calculating the reward rate for each staked token. And it is part of the `updateReward()` modifier that would be invoked up-front for almost every public function in `FarmingPool` to update and use the latest reward rate.

The reason is due to the known potential overflow pitfall when a new oversized reward amount is added into the pool. In particular, as the `rewardPerToken()` routine involves the multiplication of three `uint256` integer, it is possible for their multiplication to have an undesirable overflow (lines $76-81$), especially when the `rewardRate` is largely controlled by an external entity, i.e., `admin` (through the `notifyRewardAmount()` function).

```
54     modifier updateReward(address stakeToken, address account) {
55         uint rewardPerTokenStored = rewardPerToken(stakeToken);
56         distributions[stakeToken].rewardPerTokenStored = rewardPerTokenStored;
57         distributions[stakeToken].lastUpdateTime = lastTimeRewardApplicable(stakeToken);
58         if (account != address(0)) {
59             rewards[stakeToken][account].rewards = earned(stakeToken, account);
60             rewards[stakeToken][account].userRewardPerTokenPaid = rewardPerTokenStored;
61         }
62         _;
63     }
64
65     function lastTimeRewardApplicable(address stakeToken) public view returns (uint64) {
66         return block.timestamp > distributions[stakeToken].periodFinish ? distributions[
               stakeToken].periodFinish : (uint64)(block.timestamp);
67     }
68
69     function rewardPerToken(address stakeToken) public view returns (uint256) {
70         Distribution memory distribution = distributions[stakeToken];
71         if (distribution.totalStaked == 0) {
72             return distribution.rewardPerTokenStored;
73         }
74         uint64 lastTimeRewardApplicable = lastTimeRewardApplicable(stakeToken);
75         assert(lastTimeRewardApplicable >= distribution.lastUpdateTime);
76         return distribution.rewardPerTokenStored.add(
77             distribution.rewardRate
78             .mul(lastTimeRewardApplicable - distribution.lastUpdateTime)
```

```
            .mul(1e18)
            .div(distribution.totalStaked)
        );
    }
```

Listing 3.3: `FarmingPool::rewardPerToken()`

Apparently, this issue is made possible if the reward amount is given as the argument to `notifyRewardAmount()` such that the calculation of `rewardRate.mul(1e18)` always overflows, hence locking all deposited funds! Note that an authentication check on the caller of `onlyAdmin` greatly alleviates such concern. Apparently, if the owner is a normal address, it may put users' funds at risk. To mitigate this issue, it is necessary to have the ownership under the governance control and ensure the given reward amount will not be oversized to overflow and lock users' funds.

Recommendation Mitigate the potential overflow risk in the `FarmingPool` contract.

Status This issue has been fixed in the commit: ae1a7d4.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
```

```
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.4: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `releaseInternal()` routine in the `OLETokenLock` contract. If the `USDT` token is supported as `token`, the unsafe version of `token.transfer(beneficiary, releaseAmount)` (line 47) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
40    function releaseInternal(address beneficiary) internal {
41        uint256 amount = token.balanceOf(address(this));
42        require(amount > 0, "no amount available");
43        uint256 releaseAmount = releaseAbleAmount(beneficiary);
44        // The transfer out limit exceeds the available limit of the account
45        require(amount >= releaseAmount, "transfer out limit exceeds ");
46        releaseVars[beneficiary].lastUpdateTime = uint128(block.timestamp > releaseVars[
              beneficiary].endTime ? releaseVars[beneficiary].endTime : block.timestamp);
47        token.transfer(beneficiary, releaseAmount);
48        emit Release(beneficiary, releaseAmount);
49    }
```

Listing 3.5: OLETokenLock::releaseInternal()

The same issue is also present in other routines, including `OpenLevV1::feesAndInsurance()`. We highlight that he `approve()`-related idiosyncrasy needs to be addressed by applying `safeApprove()` twice: the first one reduces the allowance to 0 and the second one sets the new intended allowance.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

Status This issue has been fixed in the commit: ae1a7d4.

3.5 Possible Insurance Reduction in Liquidation

- ID: PVE-005
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `OpenLevV1`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [4]

Description

The `OpenLeverage` protocol has reserved a portion of accrued fee as the insurance to cover potential loss in the liquidation process. While examining the liquidation feature, we identify a potential flashloan-assisted attack to steal current insurance funds.

```
    function liquidate(address owner, uint16 marketId, bool longToken, uint
        minOrMaxAmount, bytes memory dexData) external override nonReentrant
        onlySupportDex(dexData) {
        Types.Trade memory trade = activeTrades[owner][marketId][longToken];
        Types.MarketVars memory marketVars = toMarketVar(marketId, longToken, false);
        if (dexData.isUniV2Class()) {
            updatePriceInternal(address(marketVars.buyToken), address(marketVars.
                sellToken), dexData);
        }
        //verify
        verifyCloseOrLiquidateBefore(trade.held, trade.lastBlockNum, marketVars.dexs,
            dexData.toDexDetail());
        //controller
        (ControllerInterface(addressConfig.controller)).liquidateAllowed(marketId, msg.
            sender, trade.held, dexData);
        require(!isPositionHealthy(owner, false, trade.held, marketVars, dexData), "PIH"
            );
        Types.LiquidateVars memory liquidateVars;
        liquidateVars.dexDetail = dexData.toDexDetail();
        liquidateVars.marketId = marketId;
        liquidateVars.longToken = longToken;
        liquidateVars.fees = feesAndInsurance(owner, trade.held, address(marketVars.
            sellToken), liquidateVars.marketId);
        liquidateVars.borrowed = marketVars.buyPool.borrowBalanceCurrent(owner);
        liquidateVars.isSellAllHeld = true;
        liquidateVars.depositDecrease = trade.deposited;
        //penalty
        liquidateVars.penalty = trade.held.mul(calculateConfig.penaltyRatio).div(10000);
        if (liquidateVars.penalty > 0) {
```

```
            doTransferOut(msg.sender, marketVars.sellToken, liquidateVars.penalty);
        }
        liquidateVars.remainHeldAfterFees = trade.held.sub(liquidateVars.fees).sub(liquidateVars.penalty);
        // Check need to sell all held,base on longToken=depositToken
        if (longToken == trade.depositToken) {
            // uniV3 can't cal buy amount on chain,so get from dexdata
            if (dexData.toDex() == DexData.DEX_UNIV3) {
                liquidateVars.isSellAllHeld = dexData.toUniV3QuoteFlag();
            } else {
                liquidateVars.isSellAllHeld = calBuyAmount(address(marketVars.buyToken), address(marketVars.sellToken), liquidateVars.remainHeldAfterFees, dexData) > liquidateVars.borrowed ? false : true;
            }
        }
        // need't to sell all held
        if (!liquidateVars.isSellAllHeld) {
            liquidateVars.sellAmount = flashBuy(address(marketVars.buyToken), address(marketVars.sellToken), liquidateVars.borrowed, liquidateVars.remainHeldAfterFees, dexData);
            require(minOrMaxAmount >= liquidateVars.sellAmount, 'BLM');
            liquidateVars.receiveAmount = liquidateVars.borrowed;
            marketVars.buyPool.repayBorrowBehalf(owner, liquidateVars.borrowed);
            liquidateVars.depositReturn = liquidateVars.remainHeldAfterFees.sub(liquidateVars.sellAmount);
            doTransferOut(owner, marketVars.sellToken, liquidateVars.depositReturn);
        } else {
            liquidateVars.sellAmount = liquidateVars.remainHeldAfterFees;
            liquidateVars.receiveAmount = flashSell(address(marketVars.buyToken), address(marketVars.sellToken), liquidateVars.sellAmount, minOrMaxAmount, dexData);
            // can repay
            if (liquidateVars.receiveAmount > liquidateVars.borrowed) {
                marketVars.buyPool.repayBorrowBehalf(owner, liquidateVars.borrowed);
                // buy back depositToken
                if (longToken == trade.depositToken) {
                    liquidateVars.depositReturn = flashSell(address(marketVars.sellToken), address(marketVars.buyToken), liquidateVars.receiveAmount - liquidateVars.borrowed, 0, dexData);
                    doTransferOut(owner, marketVars.sellToken, liquidateVars.depositReturn);
                } else {
                    liquidateVars.depositReturn = liquidateVars.receiveAmount - liquidateVars.borrowed;
                    doTransferOut(owner, marketVars.buyToken, liquidateVars.depositReturn);
                }
            } else {
                uint finalRepayAmount = reduceInsurance(liquidateVars.borrowed, liquidateVars.receiveAmount, liquidateVars.marketId, liquidateVars.longToken);
```

```
253                 liquidateVars.outstandingAmount = liquidateVars.borrowed.sub(
                        finalRepayAmount);
254                 marketVars.buyPool.repayBorrowEndByOpenLev(owner, finalRepayAmount);
255             }
256         }
257         liquidateVars.token0Price = longToken ? liquidateVars.sellAmount.mul(1e18).div(
                liquidateVars.receiveAmount) : liquidateVars.receiveAmount.mul(1e18).div(
                liquidateVars.sellAmount);

259         emit Liquidation(owner, liquidateVars.marketId, longToken, trade.depositToken,
                trade.held, liquidateVars.outstandingAmount, msg.sender,
260             liquidateVars.depositDecrease, liquidateVars.depositReturn, liquidateVars.
                    fees, liquidateVars.token0Price, liquidateVars.penalty, liquidateVars.
                    dexDetail);
261         delete activeTrades[owner][marketId][longToken];
262     }
```

Listing 3.6: `OpenLevV1::liquidate()`

To elaborate, we show above the `liquidate()` routine. It implements the intended logic by taking real-time pricing from the on-chain AMM model as a reference and utilizing it in risk calculation and liquidation. Unfortunately, a flashloan-assisted manipulation may make the on-chain pricing information highly skewed. As a result, the liquidation logic can be "guided" into stealing the insurance fund. In particular, the skewed DEX pricing influences the judgment such that all held collaterals need to be sold (line 241). The actual `flashSell()` of all held collaterals (line 239) is also influenced to return the `receiveAmount` such that it is still insufficient to pay `borrowed`, hence taking the execution path (line 251). After that, the insurance funds will be used for payment.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above flashloan attack to better protect the interests of protocol users.

Status This issue has been mitigated with the risk parameter `maxLiquidationPriceDiffientRatio` to control the possible slippage.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

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 setLPoolUnAllowed(address lpool, bool unAllowed) external override
        onlyAdminOrDeveloper {
        lpoolUnAlloweds[lpool] = unAllowed;
    }

    function setSuspend(bool _uspend) external override onlyAdminOrDeveloper {
        suspend = _uspend;
    }

    function setMarketSuspend(uint marketId, bool suspend) external override
        onlyAdminOrDeveloper {
        marketSuspend[marketId] = suspend;
    }

    function setOleWethDexData(bytes memory _oleWethDexData) external override
        onlyAdminOrDeveloper {
        oleWethDexData = _oleWethDexData;
    }
```

Listing 3.7: Example Setters in the ControllerV1 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. When the protocol becomes stable, the team will transfer the admin key to a timelock for community governance.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[5] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

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

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

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

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

[10] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[11] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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