SMART CONTRACT AUDIT REPORT

for

OpenLeverage Protocol

Prepared By: Yiqun Chen

PeckShield

September 26, 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	September 26, 2021	Xuxian Jiang	Final Release
1.0-rc1	September 20, 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	September 26, 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 (e31d971)

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 (34f4ef8)

1.2 About PeckShield

PeckShield Inc. [17] 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 [16]:

- 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) [15], 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	2	■■
Medium	2	■■
Low	6	■■■■■■
Informational	2	■■
Total	12	

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 high-severity vulnerabilities, 2 medium-severity vulnerabilities, 6 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key OpenLeverage Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Oversized Rewards May Lock All Pool Stakes	Numeric Errors	Fixed
PVE-002	Informational	Simplified Logic in getReward()	Business Logic	Fixed
PVE-003	Informational	Cancellability Of Submitted Proposals	Business Logic	Fixed
PVE-004	Low	Redundant State/Code Removal	Coding Practice	Fixed
PVE-005	Low	Improved Sanity Checks For System Parameters	Coding Practice	Fixed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-007	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practice	Fixed
PVE-008	Low	Non ERC20-Compliance Of LTokens	Coding Practices	Confirmed
PVE-009	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time And State	Fixed
PVE-010	High	Unguarded Privileged initializeUniV3() Function	Security Features	Fixed
PVE-011	Medium	Incorrect Logic of reduceInsurance()	Business Logic	Fixed
PVE-012	High	Possible Sandwich/MEV Attacks For Insurance Stealing	Time and State	Fixed

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 Oversized Rewards May Lock All Pool Stakes

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FarmingPool`
- Category: Numeric Errors [14]
- CWE subcategory: CWE-190 [3]

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 $114-120$), especially when the `rewardRate` is largely controlled by an external entity, i.e., `rewardDistribution` (through the `notifyRewardAmount()` function).

```
95      modifier updateReward(address account) {
96          rewardPerTokenStored = rewardPerToken();
97          lastUpdateTime = lastTimeRewardApplicable();
98          if (account != address(0)) {
99              rewards[account] = earned(account);
100             userRewardPerTokenPaid[account] = rewardPerTokenStored;
101         }
102         _;
103     }
104
105     function lastTimeRewardApplicable() public view returns (uint256) {
106         return Math.min(block.timestamp, periodFinish);
107     }
```

```
    function rewardPerToken() public view returns (uint256) {
        if (totalSupply() == 0) {
            return rewardPerTokenStored;
        }
        return
        rewardPerTokenStored.add(
            lastTimeRewardApplicable()
            .sub(lastUpdateTime)
            .mul(rewardRate)
            .mul(1e18)
            .div(totalSupply())
        );
    }
```

Listing 3.1: `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 `notifyRewardAmount()` greatly alleviates such concern. Currently, only the `rewardDistribution` address is able to call `notifyRewardAmount()` and this address is set by the owner. 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: 48fa94b.

3.2 Simplified Logic in getReward()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `FarmingPool`
- Category: Business Logic [11]
- CWE subcategory: CWE-770 [7]

Description

In the `FarmingPool` contract, the `getReward()` routine is intended to obtain the calling user's staking rewards. The logic is rather straightforward in calculating possible reward, which, if not zero, is then allocated to the calling (staking) user.

Our examination shows that the current implementation logic can be further optimized. In particular, the `getReward()` routine has a modifier, i.e., `updateReward(msg.sender)`, which timely updates the calling user's (earned) rewards in `rewards[msg.sender]` (line 99).

```
149     function getReward() public updateReward(msg.sender) checkStart {
150         uint256 reward = earned(msg.sender);
151         if (reward > 0) {
152             rewards[msg.sender] = 0;
153             oleToken.safeTransfer(msg.sender, reward);
154             emit RewardPaid(msg.sender, reward);
155         }
156     }
```

Listing 3.2: FarmingPool::getReward()

```
95      modifier updateReward(address account) {
96          rewardPerTokenStored = rewardPerToken();
97          lastUpdateTime = lastTimeRewardApplicable();
98          if (account != address(0)) {
99              rewards[account] = earned(account);
100             userRewardPerTokenPaid[account] = rewardPerTokenStored;
101         }
102         _;
103     }
```

Listing 3.3: FarmingPool::updateReward()

Having the modifier `updateReward()`, there is no need to re-calculate the earned reward for the caller `msg.sender`. In other words, we can simply re-use the calculated `rewards[msg.sender]` and assign it to the `reward` variable (line 150).

Recommendation Avoid the duplicated calculation of the caller's reward in `getReward()`, which also leads to (small) beneficial reduction of associated gas cost.

```
149     function getReward() public updateReward(msg.sender) checkStart {
150         uint256 reward = rewards[msg.sender];
151         if (reward > 0) {
152             rewards[msg.sender] = 0;
153             oleToken.safeTransfer(msg.sender, reward);
154             emit RewardPaid(msg.sender, reward);
155         }
156     }
```

Listing 3.4: Revised FarmingPool::getReward()

Status This issue has been fixed in the commit: 48fa94b.

3.3 Cancellability Of Submitted Proposals

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GovernorAlpha`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [8]

Description

The `OpenLeverage` protocol adopts the governance implementation from `Compound` by accordingly adjusting its governance token and related parameters, e.g., `quorumVotes()` and `proposalThreshold()`. In this section, we examine the proposal life-cycle and notice the submitted proposals cannot be canceled.

Specifically, a proposal can be in the following eight states: `Pending`, `Active`, `Canceled`, `Defeated`, `Succeeded`, `Queued`, `Expired`, and `Executed`. And the `GovernorAlpha` contract provides a public function `state()` to query the current state of a given `proposalId`. It comes to our attention that while the proposal has the member field to show whether it is canceled, the current implementation does not support to cancel a submitted proposal.

```
    function state(uint proposalId) public view returns (ProposalState) {
        require(proposalCount >= proposalId && proposalId > 0, "GovernorAlpha::state:
            invalid proposal id");
        Proposal storage proposal = proposals[proposalId];
        if (proposal.canceled) {
            return ProposalState.Canceled;
        } else if (block.number <= proposal.startBlock) {
            return ProposalState.Pending;
        } else if (block.number <= proposal.endBlock) {
            return ProposalState.Active;
        } else if (proposal.forVotes <= proposal.againstVotes   proposal.forVotes <
            quorumVotes(proposal.startBlock)) {
            return ProposalState.Defeated;
        } else if (proposal.eta == 0) {
            return ProposalState.Succeeded;
        } else if (proposal.executed) {
            return ProposalState.Executed;
        } else if (block.timestamp >= add256(proposal.eta, timelock.GRACE_PERIOD())) {
            return ProposalState.Expired;
        } else {
            return ProposalState.Queued;
        }
    }
```

Listing 3.5: StakingDAO::withdraw()

Recommendation Revisit the proposal lifecycle and add the support of canceling an ongoing proposal.

Status This issue has been fixed in the commit: 48fa94b.

3.4 Redundant State/Code Removal

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: OLETokenLock
- Category: Coding Practices [10]
- CWE subcategory: CWE-1126 [2]

Description

The OpenLeverage protocol makes good use of a number of reference contracts, such as ERC20, SafeMath , and ReentrancyGuard, to facilitate its code implementation and organization. For example, the OpenLevV1 smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the OLETokenLock contract, the constructor routine takes a number of input arguments. However, the last argument delegateTo is currently not used and can be safely removed.

```
    constructor(IOLEToken token_, address[] memory beneficiaries, uint256[] memory
        amounts, uint128[] memory startTimes, uint128[] memory endTimes, address
        delegateTo) {
        require(beneficiaries.length == amounts.length
        && beneficiaries.length == startTimes.length
            && beneficiaries.length == endTimes.length, "array length must be same");
        token = token_;
        for (uint i = 0; i < beneficiaries.length; i++) {
            address beneficiary = beneficiaries[i];
            releaseVars[beneficiary] = ReleaseVar(beneficiary, 0, amounts[i], startTimes
                [i], endTimes[i]);
        }
    }
```

Listing 3.6: OLETokenLock::constructor()

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been fixed in the commit: 48fa94b.

3.5 Improved Sanity Checks For System Parameters

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `OpenLeverage` protocol is no exception. Specifically, if we examine the `ControllerV1` and `LPool` contracts, they have defined a number of protocol-wide risk parameters, e.g., `baseRatePerBlock`, `multiplierPerBlock` and `kink`. In the following, we show an example routine that allows for their changes.

```
    function setBorrowCapFactorMantissa(uint newBorrowCapFactorMantissa) external
        override onlyAdmin {
        borrowCapFactorMantissa = newBorrowCapFactorMantissa;
    }

    function setInterestParams(uint baseRatePerBlock_, uint multiplierPerBlock_, uint
        jumpMultiplierPerBlock_, uint kink_) external override onlyAdmin {
        //accrueInterest except first
        if (baseRatePerBlock != 0) {
            accrueInterest();
        }
        baseRatePerBlock = baseRatePerBlock_;
        multiplierPerBlock = multiplierPerBlock_;
        jumpMultiplierPerBlock = jumpMultiplierPerBlock_;
        kink = kink_;
        emit NewInterestParam(baseRatePerBlock_, multiplierPerBlock_,
            jumpMultiplierPerBlock_, kink_);
    }

    function setReserveFactor(uint newReserveFactorMantissa) external override onlyAdmin
        {
        accrueInterest();
        uint oldReserveFactorMantissa = reserveFactorMantissa;
        reserveFactorMantissa = newReserveFactorMantissa;
        emit NewReserveFactor(oldReserveFactorMantissa, newReserveFactorMantissa);
    }
```

Listing 3.7: An example`setter` in `LPool`

Our result shows the update logic on the above parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large baseRatePerBlock parameter will incur unreasonably high interest for ongoing borrows.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

Status This issue has been fixed in the commit: 48fa94b.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [9]
- CWE subcategory: CWE-287 [4]

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;
    }
```

Listing 3.8: 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.

3.7 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1126 [2]

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 3.9: 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 `doTransferIn()` routine in the `LPool` contract. If the `USDT` token is supported as `underlying`, the unsafe version of `IERC20(underlying).transferFrom(from, address(this), amount)` (line 295) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
281     /**
282      * @dev Similar to EIP20 transfer, except it handles a False result from `
           transferFrom` and reverts in that case.
283      *      This will revert due to insufficient balance or insufficient allowance.
284      *      This function returns the actual amount received,
285      *      which may be less than `amount` if there is a fee attached to the transfer.
286      *
287      *      Note: This wrapper safely handles non-standard ERC-20 tokens that do not
           return a value.
288      *            See here: https://medium.com/coinmonks/missing-return-value-bug-at-
           least-130-tokens-affected-d67bf08521ca
289      */
290     function doTransferIn(address from, uint amount, bool convertWeth) internal returns
            (uint) {
291         uint balanceBefore = IERC20(underlying).balanceOf(address(this));
292         if (isWethPool && convertWeth) {
293             IWETH(underlying).deposit{value : msg.value}();
294         } else {
295             IERC20(underlying).transferFrom(from, address(this), amount);
```

```
        }
        // Calculate the amount that was *actually* transferred
        uint balanceAfter = IERC20(underlying).balanceOf(address(this));
        require(balanceAfter >= balanceBefore, "transfer overflow");
        return balanceAfter - balanceBefore;
    }
```

Listing 3.10: LPool::doTransferIn()

The same issue is also present in other routines, including `OpenLevV1::feesAndInsurance()`, and `OpenLevV1::flashSell()/flashBuy()`. 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: 48fa94b.

3.8 Non ERC20-Compliance Of LTokens

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LPool`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1126 [2]

Description

Each asset supported by the `OpenLeverage` protocol is integrated through a so-called `LPool` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `LPools` (or `LTokens`), users can earn interest through the `LPool`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `LTokens` as collateral. In the following, we examine the ERC20 compliance of these `LTokens`.

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the `LPool` contract. Specifically, the current `mint()` function allows for minting new `LTokens` into circu-

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

lation. However, it emits the following event `emit Transfer(address(this), minter, vars.mintTokens)`. To be compliant with the ERC20 specification, the event needs to be revised as `emit Transfer(address(0), minter, vars.mintTokens)`. The reason is that the ERC20 specification has explicitly stated that *"A token contract which creates new tokens SHOULD trigger a Transfer event with the _from address set to 0x0 when tokens are created."* Note the `redeem()` function shares the same issue.

In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Recommendation Revise the `LPool` implementation to ensure its ERC20-compliance.

Status The issue has been confirmed.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Table 3.3: Additional Opt-in Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

3.9 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [12]
- CWE subcategory: CWE-663 [5]

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` [19] exploit, and the recent `Uniswap/Lendf.Me` hack [18].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `LPool` as an example, the `borrowFresh()` 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 873) start before effecting the update on internal states (lines 876 – 878), hence violating the principle. 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.

```
    function borrowFresh(address payable borrower, address payable payee, uint borrowAmount) internal sameBlock {
        /* Fail if borrow not allowed */
        (ControllerInterface(controller)).borrowAllowed(address(this), borrower, payee, borrowAmount);

        /* Fail gracefully if protocol has insufficient underlying cash */
        require(getCashPrior() >= borrowAmount, 'cash<borrow');

        BorrowLocalVars memory vars;

        /*
         * We calculate the new borrower and total borrow balances, failing on overflow:
         *  accountBorrowsNew = accountBorrows + borrowAmount
         *  totalBorrowsNew = totalBorrows + borrowAmount
         */
        (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower);
        require(vars.mathErr == MathError.NO_ERROR, 'calc acc borrows error');

        (vars.mathErr, vars.accountBorrowsNew) = addUInt(vars.accountBorrows, borrowAmount);
        require(vars.mathErr == MathError.NO_ERROR, 'calc acc borrows error');

        (vars.mathErr, vars.totalBorrowsNew) = addUInt(totalBorrows, borrowAmount);
        require(vars.mathErr == MathError.NO_ERROR, 'calc total borrows error');

        /*
         * We invoke doTransferOut for the borrower and the borrowAmount.
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken borrowAmount less of cash.
         *  doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
         */
        doTransferOut(payee, borrowAmount, false);

        /* We write the previously calculated values into storage */
        accountBorrows[borrower].principal = vars.accountBorrowsNew;
        accountBorrows[borrower].interestIndex = borrowIndex;
        totalBorrows = vars.totalBorrowsNew;

        /* We emit a Borrow event */
        emit Borrow(borrower, payee, borrowAmount, vars.accountBorrowsNew, vars.totalBorrowsNew);

        /* We call the defense hook */
    }
```

Listing 3.11: `LPool::borrowFresh()`

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. The similar issue is also present in other functions, including redeemFresh()/borrowFresh()/repayBorrowFresh()/addReserves() in other contracts, and the adherence of the checks-effects-interactions best practice is strongly recommended. We highlight that the very same issue has been exploited in a recent Cream incident [1] and therefore deserves special attention.

From another perspective, the current mitigation in applying money-market-level reentrancy protection can be strengthened by elevating the reentrancy protection at the ControllerV1-level. In addition, each individual function can be self-strengthened by following the checks-effects-interactions principle

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions principle and utilizing the necessary nonReentrant modifier to block possible re-entrancy. Also consider strengthening the reentrancy protection at the protocol-level instead of at the current money-market granularity.

Status This issue has been fixed in the commit: 48fa94b.

3.10 Unguarded Privileged initializeUniV3() Function

- ID: PVE-010
- Severity: High
- Likelihood: High
- Impact: High
- Target: UniV3Dex
- Category: Security Features [9]
- CWE subcategory: CWE-287 [4]

Description

To build a truly permissionless margin trading market, the OpenLeverage protocol chooses not to rely on oracles that pipe off-chain prices on-chain to support risk calculation. Instead, the protocol makes use of TWAP prices provided by Uniswap (or its forks). While reviewing the supported DexAggregatorV1 price oracle, we notice the integration with UniswapV3 needs revision.

To elaborate, we show below the initialization function in the UniV3Dex contract. It comes to our attention it is defined as a public, perimssion-less function. As a result, the current uniV3Factory that is queried for current pool pairs may be manipulated! To remedy it, there is a need to define this function as an internal one, not public.

```
    function initializeUniV3(
        IUniswapV3Factory _uniV3Factory
    ) public {
        uniV3Factory = _uniV3Factory;
```

```
    }
```

Listing 3.12: `UniV3Dex::initializeUniV3()`

Recommendation Make the above `initializeUniV3()` function an internal one, not public.

Status This issue has been fixed in the commit: 48fa94b.

3.11 Incorrect Logic of reduceInsurance()

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `OpenLevV1`
- Category: Business Logic [11]
- CWE subcategory: CWE-770 [7]

Description

As mentioned earlier, each asset supported by the `OpenLeverage` protocol is integrated through a so-called `LPool` contract, which acts as a lending pool and allows the assets to be borrowed for margin trading. The borrowed assets will automatically accrue the interest and 10% of interest earned on the pool and 1/3 of the transaction fees of the pair are set aside as insurance to compensate lenders if the loan fails to maintain the solvency of the pool. Furthermore, each lending pool will have 20% of generated interests set aside as a reserve. The insurance funds can be used to afford possible compensation when there is a such need.

In the following, we examine the `reduceInsurance()` function that is designed to use the insurance funds for compensation purposes. It comes to our attention that the current implementation contains a flaw when updating the `maxCanRepayAmount` (lines 387 and 394). In particular, it needs to be computed before the respective `market.pool0Insurance` (line 386) or `market.pool1Insurance` (line 393) is updated.

```
378     function reduceInsurance(uint totalRepayment, uint remaining, uint16 marketId, bool
            longToken) internal returns (uint) {
379         uint maxCanRepayAmount = totalRepayment;
380         Types.Market storage market = markets[marketId];
381         uint needed = totalRepayment.sub(remaining);
382         if (longToken) {
383             if (market.pool0Insurance >= needed) {
384                 market.pool0Insurance = market.pool0Insurance.sub(needed);
385             } else {
386                 market.pool0Insurance = 0;
387                 maxCanRepayAmount = market.pool0Insurance.add(remaining);
388             }
389         } else {
```

```
            if (market.pool1Insurance >= needed) {
                market.pool1Insurance = market.pool1Insurance.sub(needed);
            } else {
                market.pool1Insurance = 0;
                maxCanRepayAmount = market.pool1Insurance.add(remaining);
            }
        }
        return maxCanRepayAmount;
    }
```

Listing 3.13: `OpenLevV1::reduceInsurance()`

Recommendation Correct the above `reduceInsurance()` function with the proper return value of `maxCanRepayAmount`.

Status This issue has been fixed in the commit: 48fa94b.

3.12 Possible Sandwich/MEV Attacks For Insurance Stealing

- ID: PVE-012
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `OpenLevV1`
- Category: Time and State [13]
- CWE subcategory: CWE-682 [6]

Description

As mentioned in Section 3.11, 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);
        //verify
        verifyLiquidateBefore(trade, marketVars, dexData);
        //controller
        (ControllerInterface(addressConfig.controller)).liquidateAllowed(marketId, msg.
            sender, trade.held, dexData);
        require(!isPositionHealthy(owner, false, trade.held, marketVars, dexData), "
            Position is Healthy");
        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, 'Buy amount less than min');
            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.sub(liquidateVars.borrowed), 0, dexData);
                    doTransferOut(owner, marketVars.sellToken, liquidateVars.depositReturn);

                } else {
```

```
250                     liquidateVars.depositReturn = liquidateVars.receiveAmount.sub(
                            liquidateVars.borrowed);
251                     doTransferOut(owner, marketVars.buyToken, liquidateVars.
                            depositReturn);
252                 }
253             } else {
254                 uint finalRepayAmount = reduceInsurance(liquidateVars.borrowed,
                        liquidateVars.receiveAmount, liquidateVars.marketId, liquidateVars.
                        longToken);
255                 liquidateVars.outstandingAmount = liquidateVars.borrowed.sub(
                        finalRepayAmount);
256                 marketVars.buyPool.repayBorrowEndByOpenLev(owner, finalRepayAmount);
257             }
258         }
259         liquidateVars.token0Price = longToken ? liquidateVars.sellAmount.mul(1e18).div(
                liquidateVars.receiveAmount) : liquidateVars.receiveAmount.mul(1e18).div(
                liquidateVars.sellAmount);

261         //verify
262         verifyLiquidateAfter(marketId, address(marketVars.buyToken), address(marketVars.
                sellToken), dexData);

264         emit Liquidation(owner, liquidateVars.marketId, longToken, trade.depositToken,
                trade.held, liquidateVars.outstandingAmount, msg.sender,
265             liquidateVars.depositDecrease, liquidateVars.depositReturn, liquidateVars.
                    fees, liquidateVars.token0Price, liquidateVars.penalty, liquidateVars.
                    dexDetail);
266         delete activeTrades[owner][marketId][longToken];
267     }
```

Listing 3.14: `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 227). The actual `flashSell()` of all held collaterals (line 240) is also influenced to return the `receiveAmount` such that it is still insufficient to pay `borrowed`, hence taking the execution path (line 253). 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 fixed in the commit: 48fa94b.

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] Aislinn Keely. Cream Finance Exploited in $18.8 million Flash Loan Attack. https://www.theblockcrypto.com/linked/116055/creamfinance-exploited-in-18-8-million-flash-loan-attack.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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