SMART CONTRACT AUDIT REPORT

for

Equilibria

Prepared By: Xiaomi Huang

PeckShield
May 15, 2023

Document Properties

Client	Equilibria
Title	Smart Contract Audit Report
Target	Equilibria
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Patrick Lou, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 15, 2023	Xuxian Jiang	Final Release
1.0-rc	May 10, 2023	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `Equilibria` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Equilibria

The `Equilibria Finance` is designed exclusively for `$PENDLE` holders and liquidity providers, offering an easy-to-use platform to maximize the profits. It leverages the `veToken/boosted yield model` adopted by `Pendle Finance` to provide a boosted yield for `LPs` and extra reward to `PENDLE` holders with a tokenized version of `vePENDLE`, `ePENDLE`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Equilibria

Item	Description
Name	Equilibria
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 15, 2023

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

- https://github.com/eqbtech/equilibria-contracts.git (e613513)

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

- https://github.com/eqbtech/equilibria-contracts.git (c2dc827e)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

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) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Equilibria` protocol, implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	4	■ ■ ■ ■
Informational	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 4 low-severity vulnerabilities.

Table 2.1: Key Equilibria Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Simplified getReward() Logic in BaseRewardPool	Business Logic	Resolved
PVE-002	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Resolved
PVE-003	Low	Inconsistent Pool Shutdown Logic in PendleBoosterBaseUpg	Business Logic	Resolved
PVE-004	Low	Staking Incompatibility With Deflationary Tokens in EqbMasterChef	Business Logic	Confirmed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Beside 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 Simplified getReward() Logic in BaseRewardPool

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BaseRewardPool, PendleCampaign
- Category: Business Logic [6]
- CWE subcategory: CWE-770 [3]

Description

The Equilibria protocol has a BaseRewardPool contract, which incentivizes staking users with supported protocol rewards. The logic is based on the popular StakeReward contract and there is getReward () routine that allows for querying 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(_account), which timely updates the calling user's (earned) rewards in userRewards[_account][rewardToken].rewards (line 116).

```
    function getReward(
        address _account
    ) public override updateReward(_account) {
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            address rewardToken = rewardTokens[i];
            uint256 reward = earned(_account, rewardToken);
            if (reward > 0) {
                userRewards[_account][rewardToken].rewards = 0;
                rewardToken.safeTransferToken(_account, reward);
                IPendleBooster(booster).rewardClaimed(
                    pid,
                    _account,
                    rewardToken,
                    reward
                );
```

```
264                 emit RewardPaid(_account, rewardToken, reward);
265             }
266         }
267     }
```

Listing 3.1: BaseRewardPool::getReward()

```
    modifier updateReward(address _account) {
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            address rewardToken = rewardTokens[i];
            Reward storage reward = rewards[rewardToken];
            reward.rewardPerTokenStored = rewardPerToken(rewardToken);
            reward.lastUpdateTime = lastTimeRewardApplicable(rewardToken);

            UserReward storage userReward = userRewards[_account][rewardToken];
            userReward.rewards = earned(_account, rewardToken);
            userReward.userRewardPerTokenPaid = rewards[rewardToken]
                .rewardPerTokenStored;
        }

        userAmountTime[_account] = getUserAmountTime(_account);
        userLastTime[_account] = block.timestamp;

        _;
    }
```

Listing 3.2: BaseRewardPool::updateReward()

Having the modifier `updateReward()`, there is no need to re-calculate the earned reward for the staking user. In other words, we can simply re-use the calculated `userRewards[_account][rewardToken].rewards` and assign it to the `reward` variable (line 254).

Recommendation Avoid the duplicated calculation of the caller's reward in `getReward()`, which also leads to (small) beneficial reduction of associated gas cost. Note another routine `PendleCampaign::claim()` shares the same issue.

Status This issue has been fixed in the commit: c2dc827.

3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `EqbMasterChef`
- Category: Time and State [7]
- CWE subcategory: CWE-663 [2]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [12] exploit, and the `Uniswap/Lendf.Me` hack [11].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `EqbMasterChef` as an example, the `emergencyWithdraw()` 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`.

Apparently, the interaction with the external contract (line 305) starts before effecting the update on the internal state (lines 307-308), 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.

```
302     function emergencyWithdraw(uint256 _pid) external {
303         PoolInfo storage pool = poolInfo[_pid];
304         UserInfo storage user = userInfo[_pid][msg.sender];
305         pool.lpToken.safeTransfer(address(msg.sender), user.amount);
306         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
307         user.amount = 0;
308         user.rewardDebt = 0;
309 
310         //extra rewards
311         IRewarder _rewarder = pool.rewarder;
312         if (address(_rewarder) != address(0)) {
313             _rewarder.onReward(_pid, msg.sender, msg.sender, 0, 0);
314         }
315     }
```

Listing 3.3: `EqbMasterChef::emergencyWithdraw()`

Note that other routines share the same issue, including `deposit()` and `withdraw()` from the same contract.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been fixed in the commit: c2dc827.

3.3 Inconsistent Pool Shutdown Logic in PendleBoosterBaseUpg

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PendleBoosterBaseUpg`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The `Equilibria` protocol has a `PendleBoosterBaseUpg` contract which allows for on-chain adjustment of reward pools. Each pool can be individually turned off or shut down. While examining the related shutdown logic, we notice the current implementation can be improved for consistency.

In the following, we use the implementation of the related `rewardClaimed()` routine. As the name indicates, this routine is designed to be called back from the associated reward contract when `$PENDLE` is received. Our analysis shows that the callback simply returns when the intended token is not `$PENDLE` or the `PendleBoosterBaseUpg` contract has been shut down. It misses another condition that each pool may be dynamically turned off. When a pool is turned off, even the `PendleBoosterBaseUpg` contract is still operational, we may still need to simply return.

```
    function rewardClaimed(
        uint256 _pid,
        address _account,
        address _token,
        uint256 _amount
    ) external override {
        PoolInfo memory pool = poolInfo[_pid];
        require(_isAllowedClaimer(pool, msg.sender), "!auth");

        if (_token != pendle  isShutdown) {
            return;
        }

        // mint eqb
        IEqbMinter(eqbMinter).mint(_account, _amount);
```

```
        if (contributor == address(0)) {
            return;
        }
        uint256 contributorAmount = _getContributorAmount(pool, _amount);
        if (contributorAmount > 0) {
            IEqbMinter(eqbMinter).mint(contributor, contributorAmount);
        }
    }
```

Listing 3.4: `PendleBoosterBaseUpg::rewardClaimed()`

Recommendation Improve the above logic to simply return when the given pool is shutdown.

Status This issue has been resolved as the team confirms it is part of design.

3.4 Staking Incompatibility With Deflationary Tokens in EqbMasterChef

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `EqbMasterChef`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

In the `Equilibria` protocol, the `EqbMasterChef` contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract using the `safeTransfer()/safeTransferFrom()` routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function deposit(uint256 _pid, uint256 _amount) external override {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        uint256 pending = 0;
        if (user.amount > 0) {
            pending =
                ((user.amount * pool.accEqbPerShare) / 1e12) -
```

```
                user.rewardDebt;
            safeRewardTransfer(msg.sender, pending);
        }
        pool.lpToken.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        user.amount = user.amount + _amount;
        user.rewardDebt = (user.amount * pool.accEqbPerShare) / 1e12;

        //extra rewards
        IRewarder _rewarder = pool.rewarder;
        if (address(_rewarder) != address(0)) {
            _rewarder.onReward(
                _pid,
                msg.sender,
                msg.sender,
                pending,
                user.amount
            );
        }

        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.5: `EqbMasterChef::deposit())`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accEqbPerShare` via dividing `eqbReward` by `lpSupply`, where the `lpSupply` is derived from `pool.lpToken.balanceOf(address(this))` (line 204). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may yield a huge `pool.accEqbPerShare` as the final result, which dramatically inflates the pool's reward.

```
199    function updatePool(uint256 _pid) public {
200        PoolInfo storage pool = poolInfo[_pid];
201        if (block.number <= pool.lastRewardBlock) {
202            return;
203        }
204        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
205        if (lpSupply == 0) {
206            pool.lastRewardBlock = block.number;
```

```
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 eqbReward = (multiplier * rewardPerBlock * pool.allocPoint) /
            totalAllocPoint;
        pool.accEqbPerShare =
            pool.accEqbPerShare +
            ((eqbReward * 1e12) / lpSupply);
        pool.lastRewardBlock = block.number;
    }
```

Listing 3.6: `EqbMasterChef::updatePool()`

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Equilibria` for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status This issue has been confirmed.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `Equilibria` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters, add/set reward pools, assign operators). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        IRewarder _rewarder,
        bool _withUpdate
    ) external onlyOwner {
        require(address(_lpToken) != address(0), "invalid _lpToken!");
        _checkDuplicate(_lpToken);
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock
            ? block.number
            : startBlock;
        totalAllocPoint = totalAllocPoint + _allocPoint;
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accEqbPerShare: 0,
                rewarder: _rewarder
            })
        );
    }

    // Update the given pool's EQB allocation point. Can only be called by the owner.
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        IRewarder _rewarder,
        bool _updateRewarder
    ) external onlyOwner {
        massUpdatePools();
        totalAllocPoint =
            totalAllocPoint -
            poolInfo[_pid].allocPoint +
            _allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
        if (_updateRewarder) {
            poolInfo[_pid].rewarder = _rewarder;
        }
    }
```

Listing 3.7: Example Privileged Operations in EqbMasterChef

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be better if the privileged account is governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 The issue has been confirmed and mitigated with a multi-sig account to manage the admin key.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Equilibria` protocol, which is designed exclusively for `$PENDLE` holders and liquidity providers, offering an easy-to-use platform to maximize the profits. It leverages the `veToken/boosted yield model` adopted by `Pendle Finance` to provide a boosted yield for `LPs` and extra reward to `PENDLE` holders with a tokenized version of `vePENDLE`, `ePENDLE`. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

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

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

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

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

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

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

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

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

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

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

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

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