SMART CONTRACT AUDIT REPORT

for

Gym Network

Prepared By: Xiaomi Huang

PeckShield

May 9, 2022

Document Properties

Client	Gym Network
Title	Smart Contract Audit Report
Target	Gym Network
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 9, 2022	Xuxian Jiang	Final Release
1.0-rc2	May 6, 2022	Xuxian Jiang	Release Candidate #2
1.0-rc1	April 28, 2022	Xuxian Jiang	Release Candidate #1

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Gym Network, 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 Gym Network

Gym Network is a decentralized application that serves as the entry to everything the crypto industry has to offer: from DeFi to the metaverse and everything in between. It is based on the BSC (now BNBChain) and it connects to the best interest rates to be found in this blockchain. Its native token gives users the possibility to participate in the governance of the system as well as special access to new features. By connecting the advantages of decentralized systems to the growth potential of affiliate marketing tools it is an innovative application bringing crypto to the masses. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Gym Network

Item	Description
Issuer	Gym Network
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 9, 2022

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

- https://gitlab.com/gymnet/gymnet-sol.git (98b1f94)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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 deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

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

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) [9], 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 `Gym Network` 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 logics, 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	3	■ ■ ■
Undetermined	1	■
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, 3 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key Gym Network Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible MEV Issues For Reduced Return	Time and State	Confirmed
PVE-002	Low	Timely Reward Dissemination Upon Rate/Weight Changes	Business Logic	Resolved
PVE-003	Undetermined	Staking Incompatibility With Deflationary Tokens	Business Logic	Confirmed
PVE-004	Low	Potential Reentrancy Risk in Gym-Farming	Time and State	Resolved
PVE-005	Medium	Trust on Admin Keys	Security Features	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Possible MEV Issues For Reduced Return

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GymToken`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

The `Gym Network` protocol has the built-in support of incentive mechanisms that allow users to provide supported assets for farming. Because of that, there is a constant need of swapping one asset to another. With that, the protocol has provided helper routines to facilitate the asset conversion.

```
    function _swapReceivedGYM() internal swapping {
        IPancakeRouter02 router = IPancakeRouter02(routerAddress);

        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = router.WETH();

        approve(routerAddress, balanceOf(address(this)));
        router.swapExactTokensForETH(
            balanceOf(address(this)),
            0,
            path,
            address(this),
            block.timestamp
        );

        uint256 balance = address(this).balance;
        (bool sent,) = managementAddress.call{value: balance}("");
        require(sent, "Failed to send BNB");
    }
```

Listing 3.1: GymToken::_swapReceivedGYM()

To elaborate, we show above one such helper routine `_swapReceivedGYM()`. We notice the conversion is routed to `UniswapV2` in order to swap one asset to another. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of conversion.

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 (e.g., slippage control) to the above front-running attack to better protect the interests of farming users.

Status This issue has been confirmed.

3.2 Timely Reward Dissemination Upon Rate/Weight Changes

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `GymFarming`, `GymVaultsBank`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, the `Gym Network` protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
```

```
    ) public onlyOwner poolExists(_pid) {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint - poolInfo[_pid].allocPoint + _allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.2: `GymFarming::set()`

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

In the same vein, if the `rewardPerBlock` rate has been updated, there is also a need to timely invoke `massUpdatePools()`.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner poolExists(_pid) {
        massUpdatePools();
        totalAllocPoint = totalAllocPoint - poolInfo[_pid].allocPoint + _allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.3: Revised `GymFarming::set()`

Status This issue has been fixed in the following commit: 63a8b55.

3.3 Staking Incompatibility With Deflationary Tokens

- ID: PVE-003
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `GymFarming`, `GymVaultsBank`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

In the `Gym Network` protocol, the `GymFarming` 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) public poolExists(_pid) {
        updatePool(_pid);
        poolInfo[_pid].lpToken.safeTransferFrom(msg.sender, address(this), _amount);
        _deposit(_pid, _amount, msg.sender);
    }

    function _deposit(
        uint256 _pid,
        uint256 _amount,
        address _from
    ) private {
        UserInfo storage user = userInfo[_pid][_from];
        _harvest(_pid, _from);
        user.amount += _amount;
        user.rewardDebt = (user.amount * poolInfo[_pid].accRewardPerShare) / 1e18;
        emit Deposit(_from, _pid, _amount);
    }
```

Listing 3.4: `GymFarming::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.accRewardPerShare` via dividing the reward by `lpSupply`, where the `lpSupply` is derived from `pool.lpToken.balanceOf(address(this))` (line 354). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may yield a huge `pool.accRewardPerShare` as the final result, which dramatically inflates the pool's reward.

```
349     function updatePool(uint256 _pid) public {
350         PoolInfo storage pool = poolInfo[_pid];
351         if (block.number <= pool.lastRewardBlock) {
352             return;
353         }
354         uint256 lpSupply = pool.lpToken.balanceOf(address(this));
355         if (lpSupply == 0) {
356             pool.lastRewardBlock = block.number;
357             return;
358         }
359         uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
360         uint256 reward = (multiplier * pool.allocPoint) / totalAllocPoint;
361         pool.accRewardPerShare = pool.accRewardPerShare + ((reward * 1e18) / lpSupply);
362         pool.lastRewardBlock = block.number;
363     }
```

Listing 3.5: `GymFarming::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 `Gym Network` 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.

3.4 Potential Reentrancy Risk in GymFarming

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GymFarming`
- 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` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `GymFarming` as an example, the `withdraw()` 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 508) starts before effecting the update on the internal state (lines $510-511$), 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.

```
502     function withdraw(uint256 _pid, uint256 _amount) public poolExists(_pid) {
503         PoolInfo storage pool = poolInfo[_pid];
504         UserInfo storage user = userInfo[_pid][msg.sender];
505         require(user.amount >= _amount, "withdraw: not good");
506         updatePool(_pid);
507         uint256 pending = (user.amount * pool.accRewardPerShare) / 1e18 - user.
                rewardDebt;
508         safeRewardTransfer(msg.sender, pending);
509         emit Harvest(msg.sender, _pid, pending);
510         user.amount -= _amount;
511         user.rewardDebt = (user.amount * pool.accRewardPerShare) / 1e18;
512         pool.lpToken.safeTransfer(address(msg.sender), _amount);
513         emit Withdraw(msg.sender, _pid, _amount);
514     }
```

Listing 3.6: `GymFarming::withdraw()`

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

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

Status This issue has been fixed in the following commit: 19391c3.

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

The `Gym Network` protocol has a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., pool addition, reward adjustment, and parameter setting). It also has the privilege to control or govern the flow of assets among various protocol components. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function updateTaxOnSell(uint8 _newTaxValue) public onlyOwner {
        require(_newTaxValue <= 80, "GymToken::_adminFunctions: Tax cannot be greater
            than 80");
        taxOnSell = _newTaxValue;
    }

    function updateTaxOnPurchase(uint8 _newTaxValue) public onlyOwner {
        require(_newTaxValue <= 80, "GymToken::_adminFunctions: Tax cannot be greater
            than 80");
        taxOnPurchase = _newTaxValue;
    }

    function updateDevTax(uint8 _newTaxValue) public onlyOwner {
        require(_newTaxValue <= 80, "GymToken::_adminFunctions: Tax cannot be greater
            than 80");
        devFundTax = _newTaxValue;
    }

    function updateLimitPeriod(uint256 _newlimit) public onlyOwner {
        limitPeriod = _newlimit;
    }
```

```
    function updateDexAddress(address _dex, bool _isDex) public onlyOwner {
        isDex[_dex] = _isDex;
        _isLimitExcempt[_dex] = true;
    }
```

Listing 3.7: Example Privileged Operations in `GymNetwork`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the `owner` privilege to the intended `DAO`-like governance contract. And 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.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Gym Network`, which is a decentralized application that serves as the entry to everything the crypto industry has to offer: from `DeFi` to the `metaverse` and everything in between. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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

References

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

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

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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 CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

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

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