SMART CONTRACT AUDIT REPORT

for

RockX ETH Staking

Prepared By: Xiaomi Huang

PeckShield

June 12, 2022

Document Properties

Client	RockX
Title	Smart Contract Audit Report
Target	RockX ETH Staking
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 12, 2022	Xuxian Jiang	Final Release
1.0-rc1	June 10, 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 `staking` support of `RockX`, 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 RockX

`RockX` is a blockchain fintech company that helps our customers embrace `Web 3.0` effortlessly through the development of innovative products and infrastructure. It also strives to enable institutions and disruptors in the financial and internet sectors to gain seamless access to blockchain data, crypto yield products and best-in-class key management solutions in a sustainable way. This audit covers the staking support for `ETH 2.0` in allowing users to deposit any number of ethers to the staking contract, and get back equivalent value of `uniETH` token (decided by real-time exchange ratio). The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The RockX ETH Staking

Item	Description
Name	RockX
Website	https://www.rockx.com/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 12, 2022

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

- https://github.com/RockX-SG/stake.git (735f12c)

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

- https://github.com/RockX-SG/stake.git (f1d0bef)

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.

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

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 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 implementation of the `staking` support in `RockX`. 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	1	■
Medium	1	■
Low	2	■ ■
Informational	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 high-severity vulnerability, 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational suggestion.

Table 2.1: Key RockX ETH Staking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Incorrect Validator Replacement Logic in replaceValidator()	Business Logic	Resolved
PVE-002	Low	Suggested Adherence Of The Checks-Effects-Interactions Pattern	Time and State	Resolved
PVE-003	Low	Proper minToMint Enforcement in mint()	Business Logic	Resolved
PVE-004	Informational	Proper Event Emission in mint()	Status Codes	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Incorrect Validator Replacement Logic in replaceValidator()

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: RockXStaking
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The RockXStaking contract of RockX provides an external replaceValidator() function that allows for replacing a validator in case of misconfiguration. Our analysis with this function shows the current logic has a flawed implementation that needs to be corrected.

To elaborate, we show below its current implementation. By design, the current logic updates the internal accounting to remove the oldpubkey-mapped pubkeyIndices and add the new pubkey-mapped pubkeyIndices. However, there is a need to validate the pubkeyIndices of the oldpubkey is indeed present. The current implementation accidentally ensures the non-presence of oldpubkey (line 271)! Moreover, the current implementation can also be improved by validating the new pubkey does not exist, which is currently missing.

```
265    function replaceValidator(bytes calldata oldpubkey, bytes calldata pubkey, bytes
           calldata signature) external onlyRole(REGISTRY_ROLE) {
266        require(pubkey.length == PUBKEY_LENGTH, "INCONSISTENT_PUBKEY_LEN");
267        require(signature.length == SIGNATURE_LENGTH, "INCONSISTENT_SIG_LEN");
268
269        // mark old pub key to false
270        bytes32 oldPubKeyHash = keccak256(oldpubkey);
271        require(pubkeyIndices[oldPubKeyHash] == 0, "PUBKEY_NOT_EXSITS");
272        uint256 index = pubkeyIndices[oldPubKeyHash] - 1;
273        delete pubkeyIndices[oldPubKeyHash];
274
275        // set new pubkey
276        bytes32 pubkeyHash = keccak256(pubkey);
```

```
        validatorRegistry[index] = ValidatorCredential({pubkey:pubkey, signature:
            signature, stopped:false});
        pubkeyIndices[pubkeyHash] = index+1;
    }
```

Listing 3.1: `RockXStaking::replaceValidator()`

Recommendation Revise the validator replacement logic in the above routine to ensure the old one is correctly replaced by the new one.

Status This issue has been fixed in the following commit: `e0bd595`.

3.2 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RockXStaking`
- Category: Time and State [7]
- CWE subcategory: CWE-663 [3]

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 `Uniswap/Lendf.Me` hack [12].

We notice an occasion where the `checks-effects-interactions` principle is violated. Using the `RockXStaking` as an example, the `withdrawManagerFee()` function (see the code snippet below) is provided to externally interact with a given `to` address to transfer assets. However, the invocation of an external address requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external address (line 357) starts before effecting the update on internal state (line 358 – 360), hence violating the principle. In this particular case, if the external address is an contract with certain hidden logic that may be capable of launching `re-entrancy` via the very same `withdrawManagerFee()` function. Note that there is no harm that may be caused to current protocol. However, it is still suggested to follow the known `checks-effects-interactions` best practice.

```
    function withdrawManagerFee(uint256 amount, address to) external nonReentrant
        onlyRole(MANAGER_ROLE)  {
        require(amount <= accountedManagerRevenue, "WITHDRAW_EXCEEDED_MANAGER_REVENUE");
        require(amount <= _currentEthersReceived(), "INSUFFICIENT_ETHERS");
        payable(to).sendValue(amount);
        accountedBalance -= int256(amount);
        // track manager's revenue
        accountedManagerRevenue -= amount;
        emit ManagerFeeWithdrawed(amount, to);
    }
```

Listing 3.2: `RockXStaking::withdrawManagerFee()`

In the meantime, we should mention that 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`.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice.

Status This issue has been fixed in the following commit: `e0bd595`.

3.3 Proper minToMint Enforcement in mint()

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

Description

The `RockXStaking` contract allows to deposit any number of ethers to the staking contract, and get back equivalent value of `uniETH` token (decided by real-time exchange ratio). In addition, the contract allows the user to specify the minimum `uniETH` amount (`minToMint`) that will be returned. While examining the related exchange logic, we observe the `minToMint` enforcement needs to be improved.

To elaborate, we show below the related `mint()` function. It comes to our attention that the `minToMint` parameter is enforced to ensure `require(toMint >= minToMint)` (line 669), which may not represent the actual amount of the returned `uniETH`. In other words, the current enforcement only ensures that the exchange ratio is at least 1. However, the proper enforcement requires the use of the final `toMint` of `totalXETH * msg.value / totalEthers` (line 672)!

```
    function mint(uint256 minToMint, uint256 deadline) external payable nonReentrant whenNotPaused {
        require(block.timestamp < deadline, "TRANSACTION_EXPIRED");
        require(msg.value > 0, "MINT_ZERO");

        // track balance
        _balanceIncrease(msg.value);

        // mint xETH while keeping the exchange ratio invariant
        uint256 totalXETH = IERC20(xETHAddress).totalSupply();
        uint256 totalEthers = currentReserve();
        uint256 toMint = 1 * msg.value;  // default exchange ratio 1:1
        require(toMint >= minToMint, "EXCHANGE_RATIO_MISMATCH");

        if (totalEthers > 0) { // avert division overflow
            toMint = totalXETH * msg.value / totalEthers;
        }
        // mint xETH
        IMintableContract(xETHAddress).mint(msg.sender, toMint);
        totalPending += msg.value;

        // spin up n nodes
        uint256 numValidators = totalPending / DEPOSIT_SIZE;
        for (uint256 i = 0;i<numValidators;i++) {
            if (nextValidatorId < validatorRegistry.length) {
                _spinup();
            }
        }
    }
```

Listing 3.3: `RockXStaking::mint()`

Recommendation Revisit the above logic to properly enforce the `minToMint`. An example revision is shown in the following:

```
    function mint(uint256 minToMint, uint256 deadline) external payable nonReentrant whenNotPaused {
        require(block.timestamp < deadline, "TRANSACTION_EXPIRED");
        require(msg.value > 0, "MINT_ZERO");

        // track balance
        _balanceIncrease(msg.value);

        // mint xETH while keeping the exchange ratio invariant
        uint256 totalXETH = IERC20(xETHAddress).totalSupply();
        uint256 totalEthers = currentReserve();
        uint256 toMint = 1 * msg.value;  // default exchange ratio 1:1

        if (totalEthers > 0) { // avert division overflow
            toMint = totalXETH * msg.value / totalEthers;
        }

```

```
        require(toMint >= minToMint, "EXCHANGE_RATIO_MISMATCH");

        // mint xETH
        IMintableContract(xETHAddress).mint(msg.sender, toMint);
        totalPending += msg.value;

        // spin up n nodes
        uint256 numValidators = totalPending / DEPOSIT_SIZE;
        for (uint256 i = 0;i<numValidators;i++) {
            if (nextValidatorId < validatorRegistry.length) {
                _spinup();
            }
        }
    }
```

Listing 3.4: Revised `RockXStaking::mint()`

Status This issue has been fixed in the following commit: `e0bd595`.

3.4 Proper Event Emission in mint()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `RockXStaking`
- Category: Status Codes [8]
- CWE subcategory: CWE-391 [2]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we show the set of events defined in `RockXStaking`. While examining the list of events, we notice the following three of them are defined, but not used: `RevenueWithdrawedFromValidator` (line 876), `Redeemed` (line 881), and `RedeemFromValidator` (line 882). Therefore, there is a need to properly emit them when respective operations are performed. For example, we may emit `RedeemFromValidator` within `redeemFromValidators()` and emit `Redeemed` within `_payDebts()`.

```
873    event ValidatorActivated(uint256 node_id);
874    event ValidatorStopped(uint256 stoppedCount, uint256 stoppedBalance);
875    event RevenueAccounted(uint256 amount);
876    event RevenueWithdrawedFromValidator(uint256 amount);
877    event ValidatorSlashedStopped(uint256 stoppedCount, uint256 slashed);
```

```
    event ManagerAccountSet(address account);
    event ManagerFeeSet(uint256 milli);
    event ManagerFeeWithdrawed(uint256 amount, address);
    event Redeemed(uint256 amountXETH, uint256 amountETH);
    event RedeemFromValidator(uint256 amountXETH, uint256 amountETH);
    event WithdrawCredentialSet(bytes32 withdrawCredential);
    event DebtQueued(address creditor, uint256 amountEther);
    event XETHContractSet(address addr);
    event DepositContractSet(address addr);
    event RedeemContractSet(address addr);
    event BalanceSynced(uint256 diff);
```

Listing 3.5: The Events Defined in `RockXStaking`

Recommendation Properly emit the above-mentioned events within respective functions to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status This issue has been fixed in the following commit: `e0bd595`.

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 `RockXStaking`, there is a privileged administrative account, i.e., the account with the `DEFAULT_ADMIN_ROLE` role. The administrative account plays a critical role in governing and regulating the staking-wide operations. It also has the privilege to control or govern the flow of assets within the protocol contracts. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `RockXStaking` contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
    function setManagerFeeShare(uint256 milli) external onlyRole(DEFAULT_ADMIN_ROLE)  {
        require(milli >=0 && milli <=1000, "SHARE_OUT_OF_RANGE");
        managerFeeShare = milli;

        emit ManagerFeeSet(milli);
    }

    /**
     * @dev set xETH token contract address
     */
```

```
    function setXETHContractAddress(address _xETHContract) external onlyRole(
        DEFAULT_ADMIN_ROLE) {
        xETHAddress = _xETHContract;

        emit XETHContractSet(_xETHContract);
    }

    /**
     * @dev set eth deposit contract address
     */
    function setETHDepositContract(address _ethDepositContract) external onlyRole(
        DEFAULT_ADMIN_ROLE) {
        ethDepositContract = _ethDepositContract;

        emit DepositContractSet(_ethDepositContract);
    }

    /**
     * @dev set redeem contract
     */
    function setRedeemContract(address _redeemContract) external onlyRole(
        DEFAULT_ADMIN_ROLE) {
        redeemContract = _redeemContract;

        emit RedeemContractSet(_redeemContract);
    }

    /**
     @dev set withdraw credential to receive revenue, usually this should be the
         contract itself.
     */
    function setWithdrawCredential(bytes32 withdrawalCredentials_) external onlyRole(
        DEFAULT_ADMIN_ROLE) {
        withdrawalCredentials = withdrawalCredentials_;
        emit WithdrawCredentialSet(withdrawalCredentials);
    }
```

Listing 3.6: Example Privileged Operations in `RockXStaking`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative account is a plain EOA account. 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.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changes 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 mitigated as the team confirms the use of `Aragon DAO` to use these administrative functions.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `staking` support in `RockX`, which makes it possible for anyone to access efficient and reliable mining and staking services. The staking contract allows users to deposit any number of ethers to the staking contract of `ETH 2.0`, and get back equivalent value of `uniETH` token (decided by real-time exchange ratio). The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-391: Unchecked Error Condition. https://cwe.mitre.org/data/definitions/391.html.

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