SMART CONTRACT AUDIT REPORT

for

Convex-Frax Staking

Prepared By: Patrick Lou

PeckShield

April 19, 2022

Document Properties

Client	Convex Finance
Title	Smart Contract Audit Report
Target	Convex-Frax Staking
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 19, 2022	Shulin Bie	Final Release
1.0-rc	April 18, 2022	Shulin Bie	Release Candidate

Contact

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

Name	Patrick Lou
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 Convex-Frax Staking 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 Convex-Frax Staking

The Convex-Frax Staking protocol allows the user to trustlessly stake positions on the Frax Finance gauge system while borrowing the Convex's boosting power (via veFXS). The Convex-Frax Staking protocol creates unique proxy-based vaults for each user and allows the owning user in control of the funds without any risk of admin controls gaining the unauthorized access. Meanwhile, these proxies are then given the permission to share in using the Convex's veFXS which increases the farming efficiency.

Table 1.1: Basic Information of Convex-Frax Staking

Item	Description
Target	Convex-Frax Staking
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	April 19, 2022

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

- https://github.com/convex-eth/frax-cvx-platform.git (2f8573e)

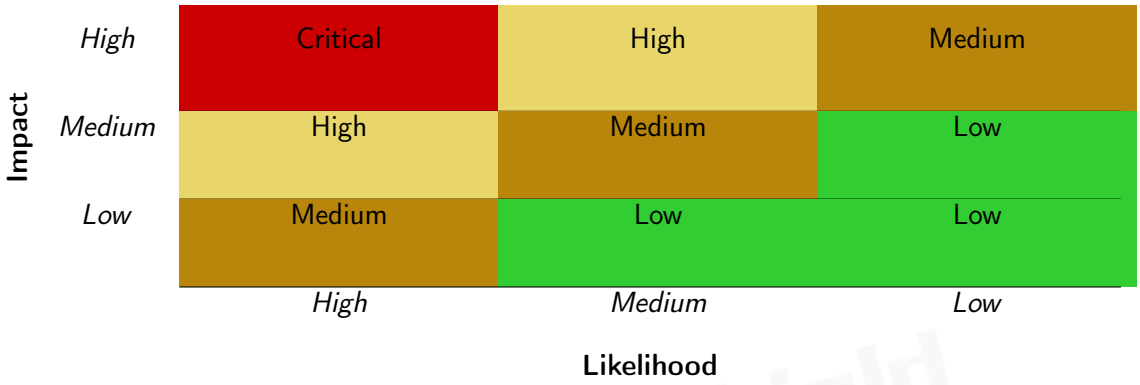
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/convex-eth/frax-cvx-platform.git (cc62dfa)

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



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

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

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) [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 `Convex-Frax Staking` 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	0	
Low	3	■■■
Informational	1	■
Total	4	

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 3 low-severity vulnerabilities and 1 informational recommendation.

Table 2.1: Key Convex-Frax Staking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Type Inconsistency Of IVoteEscrow::locked()	Business Logic	Confirmed
PVE-002	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-003	Informational	Meaningful Events For Important State Changes	Coding Practices	Fixed
PVE-004	Low	Revisited Reentrancy Protection In Current Implementation	Time and State	Fixed

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 Type Inconsistency Of IVoteEscrow::locked()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FxsDepositor`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

In the `Convex-Frax Staking` protocol, the `FxsDepositor` contract is one of the main entries for interaction with users, which accepts the deposits of the supported `fxs` token and re-deposits the assets to the `Frax Finance's Vote Escrow Pool`. In particular, the privileged `initialLock()` routine is designed to initiate the `fxs` token staking. While examining its logic, we notice there is a type inconsistency vulnerability in its calling external interfaces.

To elaborate, we show below the related code snippet of the contracts. In the `initialLock()` routine, the `locked()` routine of the `escrow` specified contract is called (line 51) to retrieve the amount of the `fxs` token staked via `FxsDepositor`. After further analysis, it comes to our attention that its interface declaration (i.e., `function locked(address)external view returns(uint256)`) is inconsistent with its definition (i.e., `locked: public(HashMap[address, LockedBalance])`) in the returned value field.

```
47      function initialLock() external{
48          require(msg.sender==feeManager, "!auth");
49
50          uint256 vefxs = IERC20(escrow).balanceOf(staker);
51          uint256 locked = IVoteEscrow(escrow).locked(staker);
52          if(vefxs == 0  vefxs == locked){
53              uint256 unlockAt = block.timestamp + MAXTIME;
54              uint256 unlockInWeeks = (unlockAt/WEEK)*WEEK;
55
56              //release old lock if exists
57              IStaker(staker).release();
58              //create new lock
```

```
59              uint256 fxsBalanceStaker = IERC20(fxs).balanceOf(staker);
60              IStaker(staker).createLock(fxsBalanceStaker, unlockAt);
61              unlockTime = unlockInWeeks;
62          }
63      }
```

Listing 3.1: `FxsDepositor::initialLock()`

```
4   interface IVoteEscrow {
5       function locked(address) external view returns(uint256);
6       function locked__end(address) external view returns(uint256);
7       function create_lock(uint256, uint256) external;
8       function increase_amount(uint256) external;
9       function increase_unlock_time(uint256) external;
10      function withdraw() external;
11      function checkpoint() external;
12      function smart_wallet_checker() external view returns (address);
13  }
```

Listing 3.2: `IVoteEscrow`

```
59  struct LockedBalance:
60      amount: int128
61      end: uint256
62
63  ...
64
65  locked: public(HashMap[address, LockedBalance])
```

Listing 3.3: `veFXS`

Recommendation Correct the above-mentioned type inconsistency.

Status The issue has been confirmed by the team.

3.2 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StakingProxyERC20/StakingProxyUniV3`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1109 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible 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."*

```
64      function transfer(address _to, uint _value) returns (bool) {
65          //Default assumes totalSupply can't be over max (2^256 - 1).
66          if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
67              balances[msg.sender] -= _value;
68              balances[_to] += _value;
69              Transfer(msg.sender, _to, _value);
70              return true;
71          } else { return false; }
72      }
73
74      function transferFrom(address _from, address _to, uint _value) returns (bool) {
75          if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
                balances[_to] + _value >= balances[_to]) {
76              balances[_to] += _value;
77              balances[_from] -= _value;
78              allowed[_from][msg.sender] -= _value;
79              Transfer(_from, _to, _value);
80              return true;
81          } else { return false; }
82      }
```

Listing 3.4: ZRX.sol

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

In the following, we show the StakingProxyERC20::_transferTokens() routine. If the USDT token is supported as _tokens[i], the unsafe version of IERC20(_tokens[i]).transfer(msg.sender, bal) (line 230) may revert as there is no return value in the USDT token contract's transfer() implementation. We may intend to replace IERC20(_tokens[i]).transfer(msg.sender, bal) (line 230) with safeTransfer().

```
224     function _transferTokens(address[] memory _tokens) internal{
225         //transfer all tokens
226         for(uint256 i = 0; i < _tokens.length; i++){
227             if(_tokens[i] != fxs){
```

```
228                 uint256 bal = IERC20(_tokens[i]).balanceOf(address(this));
229                 if(bal > 0){
230                     IERC20(_tokens[i]).transfer(msg.sender, bal);
231                 }
232             }
233         }
234     }
```

Listing 3.5: StakingProxyERC20::_transferTokens()

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

Status The issue has been addressed by the following commits: 263f096 and cc62dfa.

3.3 Meaningful Events For Important State Changes

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [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.

While examining the events that reflect the protocol dynamics, we notice there are several privileged routines that lack meaningful events to reflect their changes. In the following, we show several representative routines.

```
50     //set owner
51     function setOwner(address _owner) external onlyOwner{
52         owner = _owner;
53     }
54
55     //set fee queue, a contract fees are moved to when claiming
56     function setFeeQueue(address _queue) external onlyOwner{
57         feeQueue = _queue;
58     }
59
60     //set who can call claim fees, 0x0 address will allow anyone to call
```

```
61      function setFeeClaimer(address _claimer) external onlyOwner{
62          feeclaimer = _claimer;
63      }
64
65      //set a reward manager address that controls extra reward contracts for each pool
66      function setRewardManager(address _rmanager) external onlyOwner{
67          rewardManager = _rmanager;
68      }
69
70      //shutdown this contract.
71      function shutdownSystem() external onlyOwner{
72          //This version of booster does not require any special steps before shutting
                down
73          //and can just immediately be set.
74          isShutdown = true;
75      }
```

Listing 3.6: `Booster`

With that, we suggest to emit meaningful events in these privileged routines. Also, the key event information is better `indexed`. Note each emitted event is represented as a topic that usually consists of the signature (from a `keccak256` hash) of the event name and the types (`uint256`, `string`, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the key information is typically queried, it is better treated as a topic, hence the need of being `indexed`.

Note that other routines, i.e., `cvxFxsToken::setOperator()`, `FeeRegistry::setFees()/setDepositAddress()`, `FxsDepositor::setFeeManager()/setFees()`, `MultiRewards::setActive()/addReward()/approveRewardDistributor()`, `PoolRegistry::setOperator()/setRewardImplementation()/setRewardActiveOnCreation()`, `FraxVoterProxy::setOwner()/setOperator()/setDepositor()`, also can benefit from the improvement.

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

Status The issue has been addressed by the following commit: `b0bfe71`.

3.4 Revisited Reentrancy Protection In Current Implementation

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Time and State [7]
- CWE subcategory: CWE-682 [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 [12] exploit, and the recent Uniswap/Lendf.Me hack [11].

In the StakingProxyUniV3 contract, we notice there is a routine (i.e., lockAdditional()) that has potential reentrancy risk. To elaborate, we show below the related code snippet of the StakingProxyUniV3::lockAdditional() routine. In the lockAdditional() routine, we notice the IERC20(token0).safeTransferFrom(msg.sender, stakingAddress, _token0_amt) is called (line 100) to transfer the token0 to the StakingProxyUniV3 contract. If the token0 faithfully implements the ERC777-like standard, then the lockAdditional() routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

```
93      function lockAdditional(uint256 _token_id, uint256 _token0_amt, uint256 _token1_amt)
            external onlyOwner{
94          uint256 userLiq = IFraxFarmUniV3(stakingAddress).lockedLiquidityOf(address(this)
                );
95
96          if(_token_id > 0 && _token0_amt > 0 && _token1_amt > 0){
97              address token0 = IFraxFarmUniV3(stakingAddress).uni_token0();
98              address token1 = IFraxFarmUniV3(stakingAddress).uni_token1();
99              //pull tokens directly to staking address
100             IERC20(token0).safeTransferFrom(msg.sender, stakingAddress, _token0_amt);
101             IERC20(token1).safeTransferFrom(msg.sender, stakingAddress, _token1_amt);
102
103             //add stake - use balance of override,  min in is ignored when doing so
104             IFraxFarmUniV3(stakingAddress).lockAdditional(_token_id, _token0_amt,
                    _token1_amt, 0, 0, true);
105         }
106
107         //if rewards are active, checkpoint
108         if(IRewards(rewards).active()){
109             userLiq = IFraxFarmUniV3(stakingAddress).lockedLiquidityOf(address(this)) -
                    userLiq;
```

```
110             IRewards(rewards).deposit(owner,userLiq);
111         }
112     }
```

Listing 3.7: StakingProxyUniV3::lockAdditional()

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when transfer() or transferFrom() actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering tokensToSend() and tokensReceived() hooks. Consequently, any transfer() or transferFrom() of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In our case, the above hook can be planted in IERC20(token0).safeTransferFrom(msg.sender, stakingAddress, _token0_amt) (line 100). By doing so, we can effectively keep userLiq intact (used for the calculation of the increased liquidity at line 109). With a lower userLiq, the re-entered StakingProxyUniV3::lockAdditional() is able to obtain more rewards. It can be repeated to exploit this vulnerability for gains.

Note that other routines, i.e., StakingProxyUniV3::stakeLocked()/withdrawLocked()/getReward()/recoverERC721(), StakingProxyERC20::stakeLocked()/lockAdditional()/withdrawLocked()/getReward(), MultiRewards::deposit()/withdraw()/getReward(), and FxsDepositor::lockFxs()/deposit(), can also benefit from the reentrancy protection.

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks.

Status The issue has been addressed by the following commit: acc77c6.

4 | Conclusion

In this audit, we have analyzed the Convex-Frax Staking design and implementation. The Convex-Frax Staking protocol allows the user to trustlessly stake positions on the Frax Finance gauge system while borrowing the Convex's boosting power via veFXS. The Convex-Frax Staking protocol creates unique proxy vaults for each user which only they can control. This keeps the user in control of their funds without any risk of admin controls gaining access. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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