SMART CONTRACT AUDIT REPORT

for

Conic Finance

Prepared By: Xiaomi Huang

PeckShield

February 3, 2023

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	February 3, 2023	Stephen Bie	Final Release
1.0-rc	December 29, 2022	Stephen Bie	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 Conic Finance 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 Conic Finance

The Conic Finance protocol introduces Conic Omnipools, which allocate liquidity in a single asset across multiple Curve Pools, hence giving the Conic liquidity providers exposure to multiple Curve Pools through a single Conic LP token. All Curve LP tokens are automatically staked on Convex to earn CVX and CRV as reward. Additionally, the Conic LP token holders can earn CNC (i.e., Conic Finance DAO token) as well.

Table 1.1: Basic Information of Conic Finance

Item	Description
Target	Conic Finance
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	February 3, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that we assume the Conic Finance protocol uses a trusted price oracle with timely market price feeds for supported assets and the Curve LP token price formula is sound.

- https://github.com/ConicFinance/protocol_audit.git (6314eed)

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

- https://github.com/ConicFinance/protocol_audit.git (bebada5)

The SHA256 Hash of each audited file is as follows:

- contracts/ConicPool.sol
  Audit SHA256 Hash: 42b50d2c3154178383569e3de81d4328d31693685c10684ee961828afacfe802
  Fixed SHA256 Hash: 5f185de891d99b737d998bed1fd95e3f1295854cd6b4b1b3eb2ebe4947f49159
- contracts/Controller.sol
  Audit SHA256 Hash: ed4d00190e257e9a0025d38648981f11b1e9dd710987c8f30ad2d42676c7fd0b
  Fixed SHA256 Hash: 0f37715285db954ad042ad7e4351c6560bf1b16eda2a6b7f42d76248fad9239a
- contracts/ConvexHandler.sol
  Audit SHA256 Hash: 1544c198b3bb08879549766f98032bc203233fac85ac19abb789b53fbf472330
  Fixed SHA256 Hash: 1544c198b3bb08879549766f98032bc203233fac85ac19abb789b53fbf472330
- contracts/CurveHandler.sol
  Audit SHA256 Hash: c4bcc5c8e28f320dd8c9d151c75070da925ad6cd8fbc01358bb6a9e0d9dbab5f
  Fixed SHA256 Hash: 86b9fbcc1c3a5b08122fcd441010111abbadfb2ab3799e5ed390899c9762f9cf
- contracts/CurveRegistryCache.sol
  Audit SHA256 Hash: 17e7fd624802c0eed6170143caa7571add105368ec0ab5bff8bb83b03fa485e6
  Fixed SHA256 Hash: f00793a2700e287148770dc80e201b5f265c6d77ef2a2a34b1e00e98f84d5f0f
- contracts/LpToken.sol
  Audit SHA256 Hash: 617e45039daa47985d8f28ee5159282382a110a89844d5053b3026c64496e304
  Fixed SHA256 Hash: 617e45039daa47985d8f28ee5159282382a110a89844d5053b3026c64496e304
- contracts/RewardManager.sol
  Audit SHA256 Hash: 22fc994c5fe5ff91c457cac778714be1790321cbd801cd26b759f8462086de68
  Fixed SHA256 Hash: cf8e6b56036d693b0f23b722d5f4257915d8ae0d3e182b2c3a2235e91a7b9055
- contracts/access/GovernanceProxy.sol
  Audit SHA256 Hash: 2a417ffaebd494846e5fb548b6c1cd8225d62d80325f0ca588a3c210f2fb7258
  Fixed SHA256 Hash: 2a417ffaebd494846e5fb548b6c1cd8225d62d80325f0ca588a3c210f2fb7258
- contracts/access/SimpleAccessControl.sol
  Audit SHA256 Hash: aa1e55a836c7dfafa0f49d7395be865a9ae8ae7b9d3e39c28805f6c3776a89d2
  Fixed SHA256 Hash: aa1e55a836c7dfafa0f49d7395be865a9ae8ae7b9d3e39c28805f6c3776a89d2

- contracts/oracles/ChainlinkOracle.sol
  Audit SHA256 Hash: 43da06d72fab4b4eb4b0b966711b82265cc62bdfb8cb47bd1d78a5f484601b66
  Fixed SHA256 Hash: 43da06d72fab4b4eb4b0b966711b82265cc62bdfb8cb47bd1d78a5f484601b66
- contracts/oracles/CurveLPOracle.sol
  Audit SHA256 Hash: e2d6c72015331ad3b33fb958e4e05eb2ad75da03dc9411f3adbfa169bda93e4d
  Fixed SHA256 Hash: e2d6c72015331ad3b33fb958e4e05eb2ad75da03dc9411f3adbfa169bda93e4d
- contracts/oracles/DerivativeOracle.sol
  Audit SHA256 Hash: f227870b065d688e0836b7364a6cbe9a6d65a5425c83d1633f717a866bfdc649
  Fixed SHA256 Hash: 7874a27b7fbdf5008bd277a9743b58e77e0c1f28c5227ed4019d4e9c4b8998b5
- contracts/oracles/GenericOracle.sol
  Audit SHA256 Hash: 1bd1ee606f7b22f156ef205bd9e60ad7e2fc98c778a79fc26035708a162fba3a
  Fixed SHA256 Hash: 1bd1ee606f7b22f156ef205bd9e60ad7e2fc98c778a79fc26035708a162fba3a
- contracts/tokenomics/CNCLockerV2.sol
  Audit SHA256 Hash: b147e613136b72e542974fb948902c21e1ff904ba9f6f1281934a78845c554ca
  Fixed SHA256 Hash: 1e3f1e242889faa26b69d87a10d96937c5bfdae790eddac1c87b60e36384d068
- contracts/tokenomics/CNCMintingRebalancingRewardsHandler.sol
  Audit SHA256 Hash: 5af25c0612ddc3c6732b6cac02d105b4e5582185904f0c7c7510512991afc75e
  Fixed SHA256 Hash: 5af25c0612ddc3c6732b6cac02d105b4e5582185904f0c7c7510512991afc75e
- contracts/tokenomics/CNCToken.sol
  Audit SHA256 Hash: a10ebbe60ced6eddf4fd3963334b2e219346d17db26ab1e46a01ee871b9d83c8
  Fixed SHA256 Hash: a10ebbe60ced6eddf4fd3963334b2e219346d17db26ab1e46a01ee871b9d83c8
- contracts/tokenomics/InflationManager.sol
  Audit SHA256 Hash: 28780cc3631f377192d9dc0d94d0eee315ecbb263fa36e3d94692c7d231e2507
  Fixed SHA256 Hash: a5a2bc889f72613a543f44e842ce4b1695f5feac0aab7a5a062bce5772873948
- contracts/tokenomics/LpTokenStaker.sol
  Audit SHA256 Hash: 0ed49a717a0d35be67b185716f046e1c3165ed7f8e14e6bd1c03e14bc31f68ff
  Fixed SHA256 Hash: a6b9fa46d5c24807ac5b3bf6fb1ab9bac39121d473ebe57e4e277b88be0b0e01
- libraries/CurveLPTokenPricing.sol
  Audit SHA256 Hash: 1e3b12c23d34851752eb91a1d3b834b7b0ac2e2aed51859b2a67e72317b4f480
  Fixed SHA256 Hash: 1e3b12c23d34851752eb91a1d3b834b7b0ac2e2aed51859b2a67e72317b4f480

- libraries/CurvePoolUtils.sol
  Audit SHA256 Hash: e65169fff367ae3612b356451c2574cff6e538e52d64c5d8394f5bdd588977a8
  Fixed SHA256 Hash: e65169fff367ae3612b356451c2574cff6e538e52d64c5d8394f5bdd588977a8
- libraries/MerkleProof.sol
  Audit SHA256 Hash: d963ee0e5e5549f042fe1747fe838543d9b277a043d59de43ffcc8fbd7d0c7a2
  Fixed SHA256 Hash: d963ee0e5e5549f042fe1747fe838543d9b277a043d59de43ffcc8fbd7d0c7a2
- libraries/ScaledMath.sol
  Audit SHA256 Hash: 8d81b27cd8d8963cb2d98b852ea40b363be7d4e3f213b7f41216dd0ba7bcf02b
  Fixed SHA256 Hash: 8d81b27cd8d8963cb2d98b852ea40b363be7d4e3f213b7f41216dd0ba7bcf02b
- libraries/SquareRoot.sol
  Audit SHA256 Hash: b5d4184ff84b3992faeafa0a48e6e430d9da56c2daff68315c89e2550ffb623b
  Fixed SHA256 Hash: b5d4184ff84b3992faeafa0a48e6e430d9da56c2daff68315c89e2550ffb623b

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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 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) [7], 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.

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

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the Conic Finance 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	1	■
High	2	■■
Medium	3	■■■
Low	1	■
Total	7	

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 critical-severity vulnerability, 2 high-severity vulnerabilities, 3 medium-severity vulnerabilities, and 1 low-severity vulnerability.

Table 2.1: Key Conic Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Improper Logic of ConicPool::withdraw()	Business Logic	Fixed
PVE-002	Medium	Improper Logic of LpTokenStaker::unstakeFor()	Business Logic	Fixed
PVE-003	High	Improper Logic of CNCLockerV2::executeAvailableUnlocks()	Business Logic	Fixed
PVE-004	High	Improper Logic of CNCLockerV2::claimAirdropBoost()	Business Logic	Fixed
PVE-005	Medium	Revisited Slippage Control in claimPoolEarningsAndSellRewardTokens()	Business Logic	Fixed
PVE-006	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-007	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 Improper Logic of ConicPool::withdraw()

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `ConicPool`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `Conic Finance` protocol implements a so-called `Conic Omnipools`, which allocate liquidity in a single asset across multiple `Curve Pools`, hence giving the `Conic` liquidity providers exposure to multiple `Curve Pools` through a single `Conic LP` token. All `Curve LP` tokens are automatically staked on `Convex` for farming. The `ConicPool` contract accepts the deposits of the supported underlying token. In particular, one entry routine, i.e., `withdraw()`, is used by the liquidity providers to withdraw the underlying token. While examining its logic, we observe the current implementation should be improved.

To elaborate, we show below the related code snippet of the `ConicPool` contract. Inside the `withdraw()` routine, if the underlying token in the `Conic Pool` cannot cover the withdrawal, the shortfall needs to be withdrawn from its corresponding `Curve Pools` (lines 332 - 334). When calculating the underlying token amount that the user can receive eventually (line 339), we observe the local `underlyingBalanceDiff_` variable is incorrectly used, which just represents the shortfall amount withdrawn from the `Curve Pools` (line 338). Given this, we suggest to improve the implementation as below: `uint256 underlyingWithdrawn_ = _min(underlyingBalanceAfter_, underlyingToReceive_)` (line 339).

Moreover, it comes to our attention that the local `underlyingBalanceAfter_` variable is incorrectly used to update the `_cachedTotalUnderlying` storage variable, which is designed to store the total underlying token amount (including the `Curve LP` tokens) that the `Conic Pool` holds. This improper implementation needs to be improved as well.

```
316     function withdraw(uint256 conicLpAmount, uint256 minUnderlyingReceived)
317         public
318         override
319         returns (uint256)
320     {
321         // Preparing Withdrawals
322         ILpToken lpToken_ = lpToken;
323         require(lpToken_.balanceOf(msg.sender) >= conicLpAmount, "insufficient balance")
                ;
324         IERC20 underlying_ = underlying;
325         uint256 underlyingBalanceBefore_ = underlying.balanceOf(address(this));
326 
327         // Processing Withdrawals
328         (
329             uint256 totalUnderlying_,
330             uint256[] memory allocatedPerPool
331         ) = _getTotalAndPerPoolUnderlying();
332         uint256 underlyingToReceive_ = conicLpAmount.mulDown(_exchangeRate(
                totalUnderlying_));
333         uint256 underlyingToWithdraw_ = underlyingToReceive_ - underlyingBalanceBefore_;
334         _withdrawFromCurve(totalUnderlying_, allocatedPerPool, underlyingToWithdraw_);
335 
336         // Sending Underlying and burning LP Tokens
337         uint256 underlyingBalanceAfter_ = underlying_.balanceOf(address(this));
338         uint256 underlyingBalanceDiff_ = underlyingBalanceAfter_ -
                underlyingBalanceBefore_;
339         uint256 underlyingWithdrawn_ = _min(underlyingBalanceDiff_, underlyingToReceive_
                );
340         require(underlyingWithdrawn_ >= minUnderlyingReceived, "too much slippage");
341         lpToken_.burn(msg.sender, conicLpAmount);
342         underlying_.safeTransfer(msg.sender, underlyingWithdrawn_);
343 
344         _cachedTotalUnderlying = underlyingBalanceAfter_;
345         _cacheUpdatedTimestamp = block.timestamp;
346 
347         emit Withdraw(msg.sender, underlyingWithdrawn_);
348         return underlyingWithdrawn_;
349     }
```

Listing 3.1: `ConicPool::withdraw()`

Recommendation Correct the implementation of the `withdraw()` routine as above-mentioned.

Status The issue has been addressed by the following commits: `e6d952e` and `09fff2e`.

3.2 Improper Logic of LpTokenStaker::unstakeFor()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LpTokenStaker`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In the `Conic Finance` protocol, the `LpTokenStaker` contract is one of the main entries for interaction with users, which allows the user to stake the `Conic LP` tokens for farming. In particular, one entry routine, i.e., `unstakeFor()`, is used by the staker to withdraw the staked `Conic Lp` tokens. While examining its logic, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `LpTokenStaker` contract. By design, the `unstakeFor()` routine allows the staker to specify the recipient via the input `account` parameter. Inside the `unstakeFor()` routine, the statement of `IConicPool(conicPool).rewardManager().accountCheckpoint(account)` (line 79) is executed to update the reward timely. However, we observe the recipient's (i.e., `account`) reward is updated instead of the `msg.sender`'s. Given this, we suggest to improve the implementation as below: `IConicPool(conicPool).rewardManager().accountCheckpoint(msg.sender)` (line 79).

```
71      function unstakeFor(
72          uint256 amount,
73          address conicPool,
74          address account
75      ) public override {
76          require(controller.isPool(conicPool), "not a conic pool");
77          require(stakedPerUser[msg.sender][conicPool] >= amount, "not enough staked");
78          // Checkpoint all inflation logic
79          IConicPool(conicPool).rewardManager().accountCheckpoint(account);
80          _stakerCheckpoint(msg.sender, 0);
81          // Actual unstaking
82          ILpToken lpToken = IConicPool(conicPool).lpToken();
83          stakedPerUser[msg.sender][conicPool] -= amount;
84          stakedPerPool[conicPool] -= amount;
85          lpToken.safeTransfer(account, amount);
86      }
```

Listing 3.2: `LpTokenStaker::unstakeFor()`

Recommendation Correct the implementation of the `unstakeFor()` routine as above-mentioned.

Status The issue has been addressed by the following commit: `4ff4316`.

3.3 Improper Logic of CNCLockerV2::executeAvailableUnlocks()

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `CNCLockerV2`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `CNCLockerV2` contract allows the user to lock their `CNC` token to get a balance of vote-locked `CNC` (i.e., `vlCNC`) to participate in community governance and potential protocol fee distribution. It implements a boost mechanism to increase the `vlCNC` balance. Usually, the longer the lock time is, the larger the boost factor is. In particular, one entry routine, i.e., `executeAvailableUnlocks()`, is designed to withdraw the unlocked `CNC` token. While examining its logic, we observe the current implementation should be improved.

To elaborate, we show below the related code snippet of the `CNCLockerV2` contract. Inside the `executeAvailableUnlocks()` routine, it iterates through all the locks of `msg.sender` and filters out the expired locks. After that, the `totalLocked` (storing the total locked `CNC` token in the contract) and `lockedBalance[msg.sender]` (storing the `msg.sender`'s locked `CNC` token) are updated accordingly. However, we notice the `totalBoosted` and `lockedBoosted[msg.sender]` are not timely updated, which directly undermines the assumption of the design.

```
136     /// @notice Process all expired locks of msg.sender and withdraw unlocked CNC.
137     function executeAvailableUnlocks() external override returns (uint256) {
138         _feeCheckpoint(msg.sender);
139         uint256 sumUnlockable = 0;
140         VoteLock[] storage _pending = voteLocks[msg.sender];
141         uint256 i = _pending.length;
142         while (i > 0) {
143             i = i - 1;
144 
145             if (isShutdown) {
146                 sumUnlockable += _pending[i].amount;
147                 _pending[i] = _pending[_pending.length - 1];
148                 _pending.pop();
149             } else if (_pending[i].unlockTime <= block.timestamp) {
150                 sumUnlockable += _pending[i].amount;
151                 _pending[i] = _pending[_pending.length - 1];
152                 _pending.pop();
153             }
154         }
155         totalLocked -= sumUnlockable;
```

```
156         lockedBalance[msg.sender] -= sumUnlockable;
157         cncToken.safeTransfer(msg.sender, sumUnlockable);
158         emit UnlockExecuted(msg.sender, sumUnlockable);
159         return sumUnlockable;
160     }
```

Listing 3.3: `CNCLockerV2::executeAvailableUnlocks()`

Recommendation Timely update the `totalBoosted` and `lockedBoosted[msg.sender]` inside the `executeAvailableUnlocks()` routine.

Status The issue has been addressed by the following commit: `cb4b57a`.

3.4 Improper Logic of CNCLockerV2::claimAirdropBoost()

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `CNCLockerV2`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.3, the `CNCLockerV2` contract implements a boost mechanism to increase the `vlCNC` balance. Usually, the boost factor is related to lock time. The longer the lock time is, the larger the boost factor is. Especially, a user that locks `CNC` for `vlCNC` may receive an airdrop boost. The `claimAirdropBoost()` routine is designed to claim the airdrop boost. While examining its logic, we observe the current implementation should be improved.

To elaborate, we show below the related code snippet of the `CNCLockerV2` contract. Inside the `claimAirdropBoost()` routine, the requirement of `require(_airdroppedBoost[claimer] == 0, "already claimed")` (line 311) is executed to prevent the user from claiming the airdrop boost repeatedly. However, after further analysis, we observe the `_airdroppedBoost` is reset (line 104) inside the `lock()` routine. That is to say, the user has capability to claim the airdrop boost repeatedly.

```
305     function claimAirdropBoost(
306         address claimer,
307         uint256 amount,
308         MerkleProof.Proof calldata proof
309     ) external override {
310         require(block.timestamp < airdropEndTime, "airdrop ended");
311         require(_airdroppedBoost[claimer] == 0, "already claimed");
312         bytes32 node = keccak256(abi.encodePacked(claimer, amount));
313         require(proof.isValid(node, merkleRoot), "invalid proof");
314         _airdroppedBoost[claimer] = amount;
```

```
315         emit AirdropBoostClaimed(claimer, amount);
316     }
```

Listing 3.4: CNCLockerV2::claimAirdropBoost()

```
89      function lock(
90          uint256 amount,
91          uint128 lockTime,
92          bool relock_
93      ) public override {
94          require(!isShutdown, "locker suspended");
95          require((_MIN_LOCK_TIME <= lockTime) && (lockTime <= _MAX_LOCK_TIME), "lock time
                invalid");
96          _feeCheckpoint(msg.sender);
97          cncToken.safeTransferFrom(msg.sender, address(this), amount);
98
99          uint128 boost = computeBoost(lockTime);
100
101         uint256 airdropBoost_ = airdropBoost(msg.sender);
102         if (airdropBoost_ > 0) {
103             boost = boost.mulDownUint128(uint128(airdropBoost_));
104             _airdroppedBoost[msg.sender] = 0;
105         }
106
107         ...
108     }
```

Listing 3.5: CNCLockerV2::lock()

Recommendation Improve the sanity check in the claimAirdropBoost() routine to prevent the user from claiming the airdrop boost repeatedly.

Status The issue has been addressed by the following commit: 2132402.

3.5 Revisited Slippage Control in RewardManager::claimPoolEarningsAndSellRewardTokens()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: RewardManager
- Category: Time and State [6]
- CWE subcategory: CWE-682 [2]

Description

As mentioned in Section 3.1, all Curve LP tokens are automatically staked on Convex for farming. The RewardManager is designed to manage and distribute the rewards. In particular, one entry routine,

i.e., `claimPoolEarningsAndSellRewardTokens()`, is designed to claim all the `CVX/CRV` from `Convex` and swap all additional reward tokens to `CNC`. During the process of token swap, it does restrict possible slippage with the input `minRewardTokensCncAmount` parameter. However, it ignores the fact that the `claimPoolEarningsAndSellRewardTokens()` routine can be executed by anyone and the slippage control specified by the caller is untrusted, and is therefore still vulnerable to possible front-running attacks.

```
231     function claimPoolEarningsAndSellRewardTokens(uint256 minRewardTokensCncAmount)
232         public
233         override
234     {
235         _claimPoolEarnings();
236         _sellRewardTokens(minRewardTokensCncAmount);
237     }
238
239     function _sellRewardTokens(uint256 minCncAmount) internal {
240         uint256 extraRewardsLength_ = _extraRewards.length();
241         if (extraRewardsLength_ == 0) return;
242         uint256 cncBalanceBefore_ = CNC.balanceOf(pool);
243         for (uint256 i; i < extraRewardsLength_; i++) {
244             _swapRewardTokenForWeth(_extraRewards.at(i));
245         }
246         _swapWethForCNC();
247
248         uint256 received_ = CNC.balanceOf(pool) - cncBalanceBefore_;
249         require(received_ >= minCncAmount, "received less than minCncAmount");
250
251         uint256 _totalStaked = lpTokenStaker.getBalanceForPool(pool);
252         if (_totalStaked > 0) _updateEarned(_CNC_KEY, received_, _totalStaked);
253         emit SoldRewardTokens(received_);
254     }
```

Listing 3.6: `RewardManager::claimPoolEarningsAndSellRewardTokens()`

Note another routine, i.e., `claimEarnings()`, shares the similar issue.

Recommendation Improve the above-mentioned routines by adding effective slippage control.

Status The issue has been addressed by the following commit: `7f484a4`.

3.6 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CNCLockerV2/RewardManager`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

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 `transferFrom()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transferFrom()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transferFrom()` interface with a `bool` return value. As a result, the call to `transferFrom()` may expect a return value. With the lack of return value of `USDT`'s `transferFrom()`, the call will be unfortunately reverted.

```
171     function transferFrom(address _from, address _to, uint _value) public
            onlyPayloadSize(3 * 32) {
172         var _allowance = allowed[_from][msg.sender];
173
174         // Check is not needed because sub(_allowance, _value) will already throw if
                this condition is not met
175         // if (_value > _allowance) throw;
176
177         uint fee = (_value.mul(basisPointsRate)).div(10000);
178         if (fee > maximumFee) {
179             fee = maximumFee;
180         }
181         if (_allowance < MAX_UINT) {
182             allowed[_from][msg.sender] = _allowance.sub(_value);
183         }
184         uint sendAmount = _value.sub(fee);
185         balances[_from] = balances[_from].sub(_value);
186         balances[_to] = balances[_to].add(sendAmount);
187         if (fee > 0) {
188             balances[owner] = balances[owner].add(fee);
189             Transfer(_from, owner, fee);
190         }
191         Transfer(_from, _to, sendAmount);
192     }
```

Listing 3.7: `USDT Token Contract`

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`. 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 `receiveFees()` routine in the `CNCLockerV2` contract. If the `USDT`-like token is supported as `crv`, the unsafe version of `crv.transferFrom(msg.sender, address(this), amountCrv)` (line 287) may revert as there is no return value in the `USDT`-like token contract's `transferFrom()` implementation (but the `IERC20` interface expects a return value).

```
286     function receiveFees(uint256 amountCrv, uint256 amountCvx) external override {
287         crv.transferFrom(msg.sender, address(this), amountCrv);
288         cvx.transferFrom(msg.sender, address(this), amountCvx);
289         accruedFeesIntegralCrv += amountCrv.divDown(totalBoosted);
290         accruedFeesIntegralCvx += amountCvx.divDown(totalBoosted);
291         emit FeesReceived(msg.sender, amountCrv, amountCvx);
292     }
```

Listing 3.8: `CNCLockerV2::receiveFees()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transferFrom()/approve()`. And there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Status The issue has been addressed by the following commit: `2c25389`.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Conic Finance` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., parameter setting and price oracle adjustment). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
80     function setConvexBooster(address _convexBooster) external override onlyOwner {
81         convexBooster = _convexBooster;
82     }
```

```
83
84      function setCurveHandler(address _curveHandler) external override onlyOwner {
85          curveHandler = _curveHandler;
86      }
87
88      function setConvexHandler(address _convexHandler) external override onlyOwner {
89          convexHandler = _convexHandler;
90      }
91
92      function setInflationManager(address manager) external onlyOwner {
93          inflationManager = IInflationManager(manager);
94      }
95
96      function setPriceOracle(address oracle) external override onlyOwner {
97          priceOracle = IOracle(oracle);
98      }
```

Listing 3.9: `Controller`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the `owner` is not 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 by the team. The team intends to introduce `multi-sig` and `timelock` mechanisms to mitigate this issue.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Conic Finance protocol, which introduces a so-called Conic Omnipools allocating liquidity in a single asset across multiple Curve Pools . The Conic liquidity providers are exposed to multiple Curve Pools through a single Conic LP token. All Curve LP tokens are automatically staked on Convex to earn CVX and CRV as reward. Additionally, the Conic LP token holders can earn CNC (i.e., Conic Finance DAO token) as well. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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

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

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