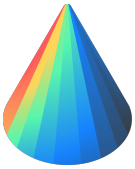
PUBLIC

Code Assessment

of the Conic Protocol Smart Contracts

December 22, 2023

Produced for



CONIC

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Conic with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Conic Protocol according to Scope to support you in forming an opinion on their security risks.

Conic implements Omnipools for Curve that allow to deposit a single asset into multiple Curve pools. The exposure to different Curve pools is changed in fixed time intervals by Governance vote.

The most critical subjects covered in our audit are functional correctness, oracle security and internal accounting. Security regarding all aforementioned subjects is high.

Functional correctness is good. Issues like Execution of wrong governance change and some smaller problems have been adequately fixed.

Newly created pools allowed Endless rebalancing due to a flaw in the handling of oracle prices. This has been addresses by rebalancing rewards being activated by governance as long as this is done in a correct manner considering TVL of the pool and CNC price.

The internal accounting of some tokenomics contracts was flawed due to Reward double counting and Wrong accounting in Bonding. These issues have also been addressed.

It should be noted that the security of funds is dependent on parameters like the imbalance buffers of the Curve oracle. These must be chosen with care (considering Curve pool fees, the share of a Conic pool's Curve LP tokens etc.) to avoid the possibility of arbitrage opportunities.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	2
• Code Corrected	2
High-Severity Findings	2
• Code Corrected	2
Medium-Severity Findings	11
• Code Corrected	9
• Risk Accepted	2
Low-Severity Findings	17
• Code Corrected	13
• Code Partially Corrected	1
• Risk Accepted	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Conic Protocol repository based on the documentation files.

The following files are in scope:

- /contracts/BaseConicPool.sol
- /contracts/ConicPool.sol
- /contracts/ConicEthPool.sol
- /contracts/LpToken.sol
- /contracts/RewardManager.sol
- /contracts/Controller.sol
- /contracts/ConvexHandler.sol
- /contracts/CurveHandler.sol
- /contracts/CurveRegistryCache.sol
- /contracts/adapters/CurveAdapter.sol
- /contracts/access/GovernanceProxy.sol
- /contracts/access/SimpleAccessControl.sol
- /contracts/oracles/ChainlinkOracle.sol
- /contracts/oracles/CurveLPOracle.sol
- /contracts/oracles/GenericOracle.sol
- /contracts/tokenomics/CNCLockerV3.sol
- /contracts/tokenomics/CNCMintingRebalancingRewardsHandler.sol
- /contracts/tokenomics/InflationManager.sol
- /contracts/tokenomics/LpTokenStaker.sol
- /contracts/tokenomics/Bonding.sol
- /contracts/zaps/EthZap.sol
- /libraries/ArrayExtensions.sol
- /libraries/CurvePoolUtils.sol
- /libraries/MerkleProof.sol
- /libraries/ScaledMath.sol
- /libraries/Types.sol

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note

1	09 October 2023	a411b13335bc847ab009009a374f03f147e18f28	Initial Version
2	01 December 2023	90470671992514a47592af04b1adf9a5ce44df86	Second Version
3	18 December 2023	75cfad6de09827479c98ee974aba247cc5a9321d	Third Version
4	22 December 2023	6474507abc7c00b9683e415d121c793f9e5ff04a	Fourth Version

For the solidity smart contracts, the compiler version `0.8.17` was chosen.

2.1.1 Excluded from scope

Third-party libraries, tests, and other files not listed above are excluded from the scope of this review.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Conic offers Omnipools, liquidity pools that optimally allocate a single underlying asset as liquidity for several Curve pools in order to derive the most yield in term of fees and liquidity mining rewards.

Multiple Conic pools exist, each one has a single underlying (ex. USDC, USDT, crvUSD, ETH). The underlying supplied by the user is supplied one-sided to Curve pools selected by governance, in a ratio selected by governance. Curve LP tokens that are received by the protocol are staked in Convex reward pools which gives rights to a share of CRV and CVX rewards (Convex Finance allows CRV holders to pool their CRV stakes in the Curve Vote Escrow under a single account, which rewards Convex Finance with a big boost for CRV emitted by Curve Liquidity Gauges).

Users supplying liquidity to Conic receive Conic LP tokens, which can be redeemed for the supplied underlying, or which can be staked in a special staking contract. Staking Conic LP tokens entitles the staking user to a share of the CRV and CVX rewards received by Conic. It also entitles the staker to a share of the CNC token emission, the Conic governance token, which is directly emitted by Conic.

Each Conic pool allocates the supplied underlying as liquidity in multiple Curve pools. The target amount allocated per Curve pool, as a percentage of the total amount of underlying, is decided by governance according to the results of a vote occurring in fixed time intervals (with a target of two weeks). When the actual allocation to Curve pools deviates from the target allocation due to result of a voted weight change, a Curve pool LP token value increase or the depegging of one of the underlying assets in a Curve pool, new deposits and withdrawals will contribute to the rebalancing of the Conic pool. To provide an economical incentive to the rebalancing of Conic pools, rewards are enabled after a weight change or a depeg rebalancing, awarding CNC to users who reduce the overall underlying deviation in a mechanism similar to Dutch auctions where each second after the update, the amount of emitted CNC increases.

2.3 Conic pools

The main functionality of Conic pools is implemented in *BaseConicPool*. *ConicPool* and *ConicEthPool* derive from it, with *ConicPool* being used when the underlying is an ERC20 stablecoin, while *ConicEthPool* is used with *WETH* as underlying.

2.3.1 Depositing and Withdrawing

Holders of the underlying token for a Conic pool can acquire a share of the pool by depositing the underlying through the `deposit()` or `depositFor()` functions. The provided amount of underlying is transferred from the user to the pool, and deposited to under-allocated Curve pools. The total underlying value of the Conic pool is estimated to compare the target allocation of each Curve pool with their current allocated values. In under-allocated pools, the underlying is unilaterally added as liquidity up to the target allocation when rebalancing rewards are active, or up to target allocation plus a deviation tolerance when rebalancing rewards are not active. The Curve LP tokens obtained by the Conic pool are staked in Convex finance in order to earn CRV and CVX rewards. The total value of the Conic pool is computed before and after adding liquidity to the Curve pools. The LP tokens supply for the Conic pool is increased proportionally to the increase in value, or to the supplied underlying, whichever is the least, so that positive price slippage benefits existing liquidity providers, and negative slippage is paid by the current depositor. Querying of the price oracle for Curve LP Tokens, before and after the deposit, ensures that Curve pools are balanced, within the defined imbalance margins (explained later in this section). The Conic LP token amount minted to the depositor is optionally staked in the *LpTokenStaker*. Finally, if rebalancing rewards are active, the deviations to the target allocations before and after the deposit are compared, and an amount of CNC proportional to the reduction is minted to the depositor as a reward.

Conic LP holders can withdraw and receive the underlying token back. The underlying is withdrawn from Curve pools with over-allocated capital. As with depositing, positive slippage benefits the totality of liquidity providers, while negative slippage costs are paid by the withdrawer. A slippage protection should be specified when calling `withdraw()` or `unstakeAndWithdraw()`. Rebalancing rewards are not awarded when withdrawing from a Conic pool.

2.3.2 Safety checks for reentrancy in ETH pools

Curve pools containing ETH or WETH might be subject to read-only reentrancies. An attacker's operation on an ETH Curve pool can generate a callback to the attacker while the Curve pool is in an invalid state. For example, `remove_liquidity()` might call back to the attacker while the native token is transferred. At this point, the LP tokens have already been burned but some tokens are yet to be transferred out, leaving an imbalance between token balances and the total supply of the pool's LP token. Therefore, if the attacker interacts with systems that rely on the state of such a Curve pool during the callback, those systems can be tricked into reading invalid values when querying `totalSupply()`, `balances()`, `price_oracle()`, and more. While newer Curve pools implement reentrancy protection on `view` methods, or disallow low level calls, querying the state of generic Curve pools requires extra care in detecting whether the call happens in the context of a reentrancy.

Conic implements read-only reentrancy detection through the `isReentrantCall()` method of the *CurveHandler*. `isReentrantCall()` first compares the code hash of the specified Curve pool with two known versions that implement reentrancy guards in the `price_oracle()` view method. The `price_oracle()` method is queried not for its result but simply to check for reversing execution which indicates a reentrancy. For Curve pools not implementing those two known versions, a slightly more complex way of detecting reentrancy is needed. A call to `exchange()` is executed on the pool, with 0 tokens actually exchanged. `exchange()` however always implements reentrancy guards. If the call reverts after having consumed less than 5000 gas, it is assumed that it reverted because of the reentrancy guard and a reentrancy is therefore detected. If it reverts after having consumed more gas, or it does not revert at all, the reentrancy guard was not triggered, and no reentrancy is therefore detected. This reentrancy detection is implemented before estimating the prices for all Curve pools included in the ETH Conic pool.

2.3.3 Flashloan protection

Conic pools LP tokens cannot be minted and burned during the same block. This prevents flashloans from being used to mint arbitrary amounts of LP tokens for the purpose of pool manipulation. LP tokens implement a tainting mechanism such that even if transferred, they cannot be burned if the sender minted / burned in the same block. A lower threshold must be exceeded by the transfer amount before tainting of the receiver in order to prevent inexpensive DoS attacks against liquidity providers who want to deposit or withdraw.

2.3.4 Curve pools

Conic supports Curve pools where pegged assets are traded. The assets in the pool must be stable (or roughly stable) with respect to the Conic pool underlying. Both StableSwap Curve pools and CryptoSwap pools are supported, with the condition that CryptoSwap pools must contain assets whose price is closely correlated (e.g., ETH and cbETH). Curve MetaPools are also supported, with the condition that each base token is pegged to the Conic underlying. Inclusion of new Curve pools in a Conic pool is subject to a Governance vote and can be vetoed by a trusted role. The allocation target of the Curve pools backing a Conic pool is subject to a Governance vote in fixed intervals.

2.3.5 Price oracles

Price oracles are a fundamental component of the Conic system. They are required for pricing the LP tokens of the Curve pools as well as underlying tokens. When depositing or withdrawing to / from a Conic pool, the value of the liquidity of Curve pools that the Conic pool holds need to be evaluated before and after the operation so that the correct number of Conic LP tokens is minted (depositing) or the right amount of underlying is paid out (withdrawing).

CurveLpOracle computes the value of a Curve LP token by evaluating the total value of the Curve pool as the sum of the balances of each token times the value of each token. The total value is then divided by the supply of Curve LP tokens to get the value per LP token. The value of the individual tokens contained in Curve pools is returned by the *ChainlinkOracle*. The computation detailed above has to be robust against pool manipulation attacks as the total value of a Curve pool can be inflated by unbalancing it. The Conic price oracles for Curve LP tokens therefore assert that the Curve pools are balanced. That is, they trade within a close range of the price specified by Chainlink oracles.

Both *CurveLpOracle* and *ChainlinkOracle* are combined into a *GenericOracle* which also gives the Governance the ability to add additional, custom oracles per token.

2.3.6 Depeg protection

Conic pools implement a protection mechanism in case one of the tokens present in one of the Curve pools loses its peg to the Conic pool's underlying token. The effect of a depeg in a Curve pool can be loss of value for the liquidity providers due to impermanent loss, so the Conic pool implements a mechanism for setting the target allocation to that pool to 0 and incentivizing users to immediately remove liquidity.

A depeg is identified when the value of the LP token of a Curve pool incurs a difference of more than 3% (custom values may also be set per Conic pool) with respect to the cached price, stored during the latest weight update. If a depeg is detected but the underlying asset's value has changed by more than double this threshold since it was last cached, the depeg of the Curve pool is attributed to the Conic pool underlying. In that case no action is taken. if the conditions are met, `handleDepeggedCurvePool()` allows any user to reset a given Curve pool to 0 weight. When a depeg is successfully detected, rebalancing rewards are enabled to incentivize withdrawing from the affected pool. In contrast to rebalancing rewards of as pool weight update, the reward Dutch auction does not start from this event but from the last weight update, leading to instant rewards in most cases.

`handleDepeggedCurvePool()` implements the same reentrancy protection as `deposit()` and `withdraw()` which prevents the value of Curve LP tokens in ETH pools from being manipulated through a reentrancy.

2.3.7 Weight updates and rebalancing rewards

Curve pool weights within a Conic pool are updated in fixed intervals, following a governance vote, through the `updateWeights()` privileged function that can be called by the *GovernanceProxy*. If, after a weight update, the total deviation of pool allocations exceeds the configured maximum deviation (`maxDeviation`), rebalancing rewards are activated.

Rebalancing rewards are emitted as CNC tokens, proportionally to the reduction in allocation deviation following a call to `deposit()` or `rebalance()`, and proportionally to the time elapsed since the last weight update. The `rebalance()` function of the *CNCMintingRebalancingRewardsHandler* performs both deposit and withdraw and compares the deviation before the two operations to the deviation after. It overrides the Conic pool LP token flashloan protection, so it can be triggered by flashloans. The reward amount per dollar rebalanced increments as time passes since the weight update, linearly going from 0 to a maximum value reached after 21 days. The proportional coefficient `cncRebalancingRewardPerDollarPerSecond` can be set by governance in *CNCMintingRebalancingRewardsHandler*. A maximum of 1.9M CNC tokens can be minted as rebalancing rewards.

2.3.8 Access control and privileged functions

Conic pools implement two roles, `controller` and `owner`. The `controller` role is assigned to the Conic *Controller*, and is able to pause a Conic pool, shut down a Conic pool, and update the weights. The `owner` role is assigned directly to the *GovernanceProxy*. Governance has access to guarded functions that allow the configuration of Conic pools, such as setting maximum deviation parameters, setting the depeg threshold, and adding and removing Curve pools.

2.4 LP Token staking and rewards distribution

Conic LP tokens accrue the fees earned in the underlying Curve pools but they do not directly accrue CRV and CVX rewards earned by the Curve LP tokens staked in Convex. Liquidity providers earn their share of CVX and CRV rewards from Convex plus CNC rewards from Conic by staking their Conic pool LP tokens in the *LpTokenStaker*.

Staking entitles the LP to a share of the rewards of the Conic pool proportional to the staked amount. The *LpTokenStaker* is coupled to *RewardManager* contracts which are deployed for every pool. The *RewardManager* keeps track of CRV and CVX rewards earned by a Conic pool on Convex and attributes them fairly among staking liquidity providers. The *RewardManager* also tracks the *CNC* tokens awarded by Conic to its pools. A portion of the *CRV* and *CVX* rewards can be forwarded to users who lock *CNC* in *CNCLockerV3* if enabled by governance.

The *RewardManager* also handles extra rewards awarded by Curve or Convex, in the form of other tokens (e.g. *LDO*). The extra rewards are swapped on predefined Curve pools, or on SushiSwap, for CNC and the CNC is accrued to the total rewards.

2.4.1 CNC emissions

Beside rebalancing rewards, which are awarded for Conic pool rebalancing in CNC, CNC is also minted at a fixed time rate in every inflation rate period of 365 days and distributed among the Conic pools according to their relative total value. That is, every year a fixed amount of CNC rewards are minted and distributed among the pools according to their relative weight during that year. This mechanism is implemented in the *InflationManager*. Every year, the CNC emissions that are awarded to Conic pools decrease by 60%, starting with an initial value of 1.5M CNC per year.

2.4.2 Checkpointing logic

To keep track of the rewards awarded by each user, a mechanism is used similar to the accounting in Synthetix' *StakingRewards*, or Curve's *Gauges*. At the Conic pool level, a global index keeps track of the amount of reward tokens awarded for every staked token. Every time reward tokens are accrued, this index is incremented by the amount awarded divided by the total amount staked. Each user has its own checkpoint of the global index, which provides information on the last time awards have been accrued to the user claimable balance. Every time the staked balance of a user changes, the difference between the current global index and the user checkpointed index is multiplied with the previous user stake, incrementing the user's claimable balance. The user checkpointed index is then updated to the current global index.

2.4.3 Time boost

The contract *LpTokenStaker* keeps track of a boost for each user which can be queried through `getBoost()`. The boost is composed of a time boost, and a stake boost component. The stake boost grows proportionally to the user's share of the total staked amount. The time boost component grows from 0.1 to 1 over a period of 30 days. The two boost amounts are multiplied, and the result is clipped between 1 and 10 to get the total boost. This boost is used as a multiplying factor when evaluating the voting power of a user in *CNCLockerV3*.

Users that do not stake any LP tokens receive a boost of 0.1.

2.5 Vote locking and bonding

Holders of CNC tokens can lock them in *CNCLockerV3* in order to receive part of the *RewardManager* fees and be entitled to voting power for Conic governance in snapshots. CNC can be locked for a period between 120 and 240 days, earning the locking user a boost between 1 and 1.5. This boost is multiplied with an optional airdropped boost. The total boosted amount for the lock, which contributes to the voting power of the user when queried through `balanceOf()`, is the locked amount of *CNC* multiplied with the boost. This amount is further multiplied with the *LpTokenStaker* boost. The airdropped boost can be claimed by selected users for the first 182 days of the contract, and its value is a multiplier between 1 and 3.5. The airdropped boost can only be claimed in the first 182 days, but can be used indefinitely in the future, on the next lock created by the claiming user.

Multiple locks can be created per user, through the methods `lock()` and `lockFor()`. Locks can be extended by setting the `relock_` flag in `lock()` or `lockFor()` and through functions `relock()` and `relockMultiple()`. Expired locks can be unlocked through `executeAvailableUnlocks()`, `executeAvailableUnlocksFor()`, and `executeUnlocks()`.

Locks that are not unlocked by their owner after a grace period of 28 since their expiration can be removed in an unpermissioned way through `kick()`, returning the locked amount to the owner minus a kick penalty of 10% of the lock value up to 1000 CNC. This is needed because unlockable balances still accrue fees until they are actively removed.

2.5.1 Bonding

The *Bonding* contract implements a multi-epoch Dutch action wherein LP tokens for the crvUSD Conic pool can be used to buy CNC. The amount of CNC which is initially present in the *Bonding* contract is sold in equal amounts in a defined number of epochs. The starting price is determined as the last price at which CNC was acquired during the preceding epoch, multiplied by a factor. The price decreases linearly during an epoch from the starting price down to 0. The proceeds of the auction, the LP token for the crvUSD Conic pool, are gradually distributed to users over the next epoch proportionally to their `totalRewardsBoost()` in *CNCLockerV3*.

2.6 Changes in Version 2

Version 2 of the protocol introduced the following changes:

1. Rebalancing rewards start at 0 and increase with a different factor when a pool is marked as depegged.
2. The inflation rate is no longer automatically updated on reward claims.
3. *Bonding* requires a minimum amount (up to 1,000 tokens).
4. *Bonding* can claim earnings of staked LP tokens and send them to the debt pool via the new method `claimFeesForDebtPool()`.
5. *CNCLockerV3* requires a minimum lock amount and allows no more than 10 locks per account.
6. *CNCLockerV3* contains a new function `batchKick()` for kicking multiple locks at once.
7. A `feeRecipient` is introduced so that fees can be sent to other contracts apart from *CNCLockerV3*.

2.7 Governance and Trust model

Contracts of the Conic codebase implementing an `onlyOwner` modifier are assumed to be owned by the Conic *GovernanceProxy*. Voting for Conic governance happens off-chain through a snapshot mechanism. Governance decisions are then relayed on-chain by a trusted wallet holding the `GOVERNANCE_ROLE` in *GovernanceProxy*. The `GOVERNANCE_ROLE` user has the right to `requestChange()`, `cancelChange()` and `grantRole()` / `revokeRole()`. Another trusted role is `VETO_ROLE` which has the right to `cancelChange()`.

In the *GovernanceProxy*, a *change* is a list of external calls. A selector mapping is queried with the function selectors in the external calls returning the delay for every selector (0 by default). The delay of the overall change is the maximum of the delays of each external call. If the delay is 0, the calls are immediately executed. Otherwise, the change is recorded in storage for future execution. After the delay has elapsed, an unpermissioned call to `executeChange()` will perform the external calls and mark the change as executed. While a change is pending, that is it has not been executed yet, users with the `VETO_ROLE` are allowed to cancel it.

We assume that care is taken when setting parameters in the protocol as the following parameters are critical and could lead to loss of funds if not set carefully:

- `customImbalanceBuffers` in *CurveLpOracle*.
- `extraRewardsCurvePool` in *`RewardManager*.
- `customOracles` in *GenericOracle*.

It is also important that the deployment process of Conic pools involves the deployment of *ConicEthPool* over *ConicPool* whenever there is a possibility that a Curve pool with underlying ETH or WETH can be added.

We further assume that `GOVERNANCE_ROLE` and `VETO_ROLE` are multi-sig accounts that are distributed between parties that can be expected to act in the best interest of the protocol.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	2

- Exchange Rate Rounding Errors Risk Accepted
- No Governance Default Delay Risk Accepted

Low-Severity Findings	4

- Instant Rewards Risk Accepted
- Missing Checks Code Partially Corrected Risk Accepted
- No Reward Checkpoint When Unstaking Risk Accepted
- Stale Oracle Price Means Token Is Not Supported Risk Accepted

5.1 Exchange Rate Rounding Errors

Design Medium Version 1 Risk Accepted

CS-CCP-006

`BaseConicPool.deposit()` allows users to add funds to the protocol while no shares are minted. On a new pool, a user can donate some tokens to the contract before calling `deposit()` with an `underlyingAmount` of 0. The donated tokens are then added to the contract holdings while no shares are minted for the user. After that, the user can deposit 1 wei of tokens, minting them exactly 1 wei of shares.

The exchange rate is skewed:

```
totalUnderlying_.divDown(lpSupply);
```

Since the amount of deposited tokens by other users is divided by this exchange rate to determine the amount of minted shares, the results can include large rounding errors. Users that are not depositing multiples of the initially deposited amount will incur slippage (up to 100%) which results in either Denial of Service or, if they choose a loose slippage parameter, loss of funds.

For example, a donation of 10,000 USDC and a subsequent deposit of another user of 15,000 USDC would result in the second user getting only 1 wei of LP tokens, thus losing 2,500 USDC to the first user.

It is also possible to burn LP tokens without decreasing the underlying in `withdraw()`.

Risk accepted:

Conic accepts the risk claiming that all pools will be atomically seeded by the team on deployment. In that case, the mentioned attack is not possible.

5.2 No Governance Default Delay

Security | Medium | Version 1 | Risk Accepted

CS-CCP-008

Conic governance decisions are computed off-chain. Based on the results, a multi-sig address with the `GOVERNANCE_ROLE` on the `GovernanceProxy` can then request changes and execute them. Depending on the function signatures these changes are going to call, a delay is invoked so that a separate multi-sig address (`VETO_ROLE`) that belong to different entities can veto the change.

However, no default delay is enforced which means that the `GOVERNANCE_ROLE` can perform any actions that have not explicitly been marked directly, evading any possible vetos.

Risk accepted:

Conic accepts the risk with the following statement:

> We do not want a delay for all the functions. The community and veto multisig can easily check which functions have a delay and which does not.

5.3 Instant Rewards

Design | Low | Version 1 | Risk Accepted

CS-CCP-018

`RewardManager` performs reward calculations and actual reward claiming in separate steps. Only when certain conditions are met, rewards are actually claimed. This approach is, however, flawed for Convex' extra rewards as the reward calculation is only performed during the *claim* step here. If claiming has not occurred for a longer period, the accrued extra rewards are added to the earned rewards of all users in bulk the next time they are claimed.

User that have deposited LP tokens to the `LpTokenStaker` can call `RewardManager.claimPoolEarningsAndSellRewardTokens()` directly after staking and instantly receive some CNC rewards in this case.

Risk accepted:

Conic accepts the risk with the following statement:

> Very few pools have extra rewards (not any that we currently support) and the chances of these rewards becoming an important part of the APR is low enough for us to accept this risk.

5.4 Missing Checks

Correctness | Low | Version 1 | Code Partially Corrected | Risk Accepted

CS-CCP-021

The protocol is missing some checks that could potentially lead to a problematic state:

- `RewardManager.addExtraReward()` checks that the added reward token is not an LP token of one of the Curve pools of the associated LP tokens. If, however, a Curve pool is added to the Conic pool at a later stage, its LP token might have already been added.
- `RewardManager.addExtraReward()` does not check whether there is a valid SushiSwap or Curve pool for a given reward token.
- Neither `RewardManager.setExtraRewardsCurvePool()` nor `_swapRewardTokenForWeth()` check whether a given Curve pool actually holds the asset that is going to be swapped on it.
- `RewardManager.removeExtaReward()` does not check that the specified argument is successfully removed from the extra rewards list.
- `Bonding.startBonding()` does not check whether an `epochPriceIncreaseFactor` is set. Since there exists a minimum for the factor, it should be set before starting the bonding period.

Code partially corrected:

`Bonding.startBonding()` now checks if the `epochPriceIncreaseFactor` has already been set.

Risk accepted:

Conic accepts the risk for all other missing checks with the following statement:

> We accept the risk for the extra rewards.

5.5 No Reward Checkpoint When Unstaking

Design | Low | Version 1 | Risk Accepted

CS-CCP-022

`LpTokenStaker.unstakeFor()` does not call `RewardManager.accountCheckpoint()` when the staker is shut down. Users unstaking on such a shut down staking contract will lose their rewards since the last checkpoint but users that call `accountCheckpoint()` before unstaking will keep their rewards.

Risk accpted:

Conic accepts the risk with the following statement:

> This is an extremely rare event, in which case we will inform our users beforehand so that they claim their rewards.

5.6 Stale Oracle Price Means Token Is Not Supported

Correctness | Low | Version 1 | Risk Accepted

CS-CCP-024

`ChainlinkOracle.isTokenSupported()` calls `getUSDPrice()` to determine whether a token is supported by the oracle. If the price is stale (older than the heartbeat), the function erroneously returns `false`.

Risk accepted:

Conic accepts the risk with the following statement:

> We only use isTokenSupported in two places:
>
> 1. When adding a new curve pool to the registry. If it fails here, we can retry later.
> 2. When claiming extra token rewards. In the unlikely event that it fails here, we accept the slippage risk when swapping extra rewards.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	2

- Endless Rebalancing Code Corrected
- Execution of Wrong Governance Change Code Corrected

High-Severity Findings	2

- Depeg Due to Oracle Manipulation Code Corrected
- Wrong Accounting in Bonding Code Corrected

Medium-Severity Findings	9

- Bonding lastCncPrice Manipulation With Leftover Dust Code Corrected
- Extra Reward Tokens Not Sent to RewardManager Code Corrected
- Higher Imbalance Tolerance in Metapools Code Corrected
- Incomplete Pool Balance Check Code Corrected
- Oracle Price Manipulation Code Corrected
- Reward Double Counting Code Corrected
- Slippage Losses Are Socialized Code Corrected
- Weight Update Rounding Errors Code Corrected
- Wrong Denomination of Deviation Delta Code Corrected

Low-Severity Findings	13

- Possible Zeroed Pool Weight Increase Code Corrected
- Reward Factor Override Code Corrected
- Boost After Shutdown Code Corrected
- CNCLockerV3 Lock Squatting Code Corrected
- Claimable Rewards Potentially Wrong Code Corrected
- Enabled Fee Not Reset Code Corrected
- Endless Loop Code Corrected
- Lock Spam DoS Code Corrected
- Minimum Tainted Transfer Amount Can Be Circumvented Code Corrected
- Rebalancing Reward After Depeg Code Corrected
- Unreachable Imbalance Buffers Code Corrected
- Wrong TVL Factor Code Corrected
- Wrong Time to Full Boost Code Corrected

Informational Findings	9

- removeDuplicates Not Working With 0 Elements Code Corrected
- Fees Without Locked CNC Specification Changed
- Exchange Rate Race Condition Code Corrected
- Interface Differences Code Corrected
- Ambiguous Naming Code Corrected
- Typographical Errors Code Corrected
- Rebalancing Reward Formula Mismatch Specification Changed
- Missing Events Code Corrected
- Shadowed Variables Code Corrected

6.1 Endless Rebalancing

Design Critical Version 1 Code Corrected

CS-CCP-001

As detailed in Depeg due to oracle manipulation, Curve pools can be depegged wilfully at any time. On new Conic pools, or pools with very low TVL, this is even more problematic because an infinite amount of rebalancing rewards can be claimed.

This is possible by donation of Curve LP tokens to the Conic pool. When the attacker is the only liquidity provider on a Conic pool, or if they hold most of the LP tokens, all donated value is given straight back to them. Therefore, the donations are free (or almost free), enabling this attack:

1. The attacker deposits to an empty Conic pool.
2. The attacker depegs one of the underlying Curve pools, enabling rebalancing rewards.
3. The attacker rebalances the Conic pool to *almost* the `maxDeviation` threshold, so that the rebalancing rewards are still active.
4. The attacker adds liquidity directly to the Curve pool, sends these tokens to the Conic pool.
5. The attacker repeats **step 3** and **step 4** as often as possible.
6. The attacker withdraws their LP tokens from the Conic pool (in the next block).

As long as the attacker gains more CNC rewards per iteration than they lose to other Conic LPs (only relevant if they are not the only LP), the attack is profitable and can be performed indefinitely, allowing them to mint CNC up to the `_MAX_REBALANCING_REWARDS`. The only cost is the amount of tokens needed to increase the LP token price of a given Curve pool by the depeg threshold. Therefore, small Curve pools are more vulnerable.

Code corrected:

A new (off-chain) threshold for a pool's TVL is introduced which has to be passed before rebalancing rewards are activated. In practice, this is done via governance change that calls the function `BaseConicPool.setRebalancingRewardsEnabled()`. Additionally, rebalancing rewards now start from 0 after a pool has been marked as depegged. This ensures that the attack does not become instantly profitable. With a reward factor of 10, the attacker has to wait 1.4 days to achieve the same result as before.

6.2 Execution of Wrong Governance Change

Correctness | Critical | Version 1 | Code Corrected

CS-CCP-002

`GovernanceProxy.executeChange()` gets a storage pointer to the change corresponding to a given ID, deletes the change from storage and then tries to execute the change.

The change is deleted in the following way:

```
pendingChanges[index] = pendingChanges[pendingChanges.length - 1];
pendingChanges.pop();
```

If the change is the last one in the `pendingChanges` array, then nothing is executed at all. If the change is any other change, the last change in `pendingChanges` will be executed instead of the correct one.

Code corrected:

A given pending change is now deleted after all calls have been performed. To ensure that the change cannot re-execute itself, a new state `Executing` has been introduced that is set over the duration of the calls. Only `Pending` changes can be executed.

6.3 Depeg Due to Oracle Manipulation

Correctness | High | Version 1 | Code Corrected

CS-CCP-003

Deposit and withdraw functions of ETH pools are protected against reentrancy from Curve pools which disables the ability of attackers to manipulate the `totalSupply()` function of Curve and therefore the manipulation of the `CurveLPOracle`. Furthermore, the oracle checks that a given Curve pool is balanced by comparing the Chainlink oracle prices of the underlying tokens with the actual price of the tokens on the Curve pool (using `get_dy()`). This ensures that an attacker cannot perform large trades on the pool before calling Conic, which would also skew the LP token price.

There exists, however, another possibility: Fee accrual on the Curve pool that results in the LP token price becoming (permanently) inflated. While this is not a problem for deposits and withdrawals, the mechanism can be used to call `BaseConicPool.handleDepeggedCurvePool()` and set the weight of the pool to 0. This automatically enables rebalancing rewards. An attacker can then rebalance the pool and gain the rebalancing rewards. The fees can be accrued with a single, large, bi-directional trade.

Since `handleDepeggedCurvePool()` does not set the timestamp for pool weight updates, the reward for rebalancing is instantly available. If a pool is depegged right before a weight update (which is estimated to happen around every 14 days), the reward can be as high as 280 CNC per 10.000 USD value rebalanced.

Consider the following example:

1. A Conic pool exists that contains two Curve pools with 3 assets holding 100k tokens each. Weights are `[0.5, 0.5]`.
2. An attacker (iteratively) adds 900k tokens liquidity per asset to the first Curve pool (by adding the Conic pool's underlying via Conic and the rest via Curve). The attacker also has to add liquidity to the other Curve pool to ensure that everything keeps balanced. These tokens can be withdrawn again later.
3. The first Curve pool's value is now roughly 3M. With the aforementioned fee donation attack, the attacker increases the value of the pool to 3.09M.

4. The attacker now depegs the first Curve pool and rebalances 900k on Conic, netting them ~25k CNC tokens.
5. The attacker withdraws liquidity from Conic and Curve. They get back up to 90% of the donated 90k fees to Curve (depending on the Curve pool setup) as they hold 90% of the liquidity of the pool.
6. Depending on the amount of fees the attacker gets back, and the current market value of CNC, this attack becomes profitable.

The attack can be scaled infinitely with sufficient holdings and also becomes more profitable. Rebalancing and liquidity provision to Curve can be done with flash loans while the liquidity provision to Conic requires capital as a deposit cannot be withdrawn in the same block.

Code corrected:

Depegs are now identified by comparing Chainlink prices of all underlying tokens to their cached price. As there are no longer any LP token prices involved, there is no possibility for manipulation (except broad market manipulation).

6.4 Wrong Accounting in Bonding

Correctness | High | Version 1 | Code Corrected

CS-CCP-004

`Bonding._checkpointAccount()` calculates the already accrued stream of LP tokens that can be unstaked by multiplying the user's rewards boost with the difference of the total integral and the user's integral since the last checkpoint:

```
uint256 accountBoostedBalance = cncLocker.totalRewardsBoost(account);
perAccountStreamAccrued[account] += accountBoostedBalance.mulDown(
    streamIntegral - perAccountStreamIntegral[account]
);
perAccountStreamIntegral[account] = streamIntegral;
```

The `streamIntegral` is computed with `CNCLockerV3.totalBoosted()` amount which is the total of the locked CNC times the boosts of each user. The accrued stream of users is computed with `totalRewardsBoost()` which also contains balances of the old `CNCLockerV2` contract and does not contain CNC of locks that have already expired:

```
function totalRewardsBoost(address account) public view override returns (uint256) {
    return
        lockedBoosted[account] -
        unlockableBalanceBoosted(account) +
        ICNCVoteLocker(V2_LOCKER).balanceOf(account);
}
```

Since `totalBoosted()` is smaller than the sum of `totalRewardsBoost()` for all users, there can be more claims than could be satisfied. Consider the following scenario:

1. User 1 has a `totalRewardsBoost()` of 1000 tokens.
2. User 2 has a `totalRewardsBoost()` of 0.
3. `totalBoosted()` is 0.
4. User 2 calls `bondCncCrvUsd()` with an amount of 1000 LP tokens and gets a bonding price of 1.

5. After 2 epochs, user 1 calls `checkpointAccount()`. `streamIntegral` is set to 1. Since user 1 has a balance of 1000 tokens in `totalRewardsBoost()` but their account integral has not been set yet, `perAccountStreamAccrued` for user 1 is updated to 1000 tokens.
6. User 1 calls `claimStreamed()` and receives 1000 LP tokens.
7. User 2 can still accrue 1000 tokens in `perAccountStreamAccrued` by calling `checkpointAccount()`. But they cannot claim the stream anymore, since the 1000 LP tokens have already been unstaked.

Additionally, it is problematic that the integral calculation with `totalBoosted()` does not consider unlockable CNC as the calculation might result in an integral smaller than it should be.

Code corrected:

`Bonding` now uses the function `CNCLockerV3.totalStreamBoost()` instead of `totalRewardsBooost()` to calculate the integral of individual accounts. This function only returns the locked boost of a user which matches the calculation for the total integral.

6.5 Bonding lastCncPrice Manipulation With Leftover Dust

Security | Medium | Version 1 | Code Corrected

CS-CCP-005

During every bonding epoch, there will likely be some CNC dust left-over since it is hard to estimate the exact amount of LP tokens to bond in order to acquire all the CNC up to the last decimal. Some of this CNC dust can however be acquired just before the epoch ends for the purpose of manipulating `lastCncPrice` to be `MIN_CNC_START_PRICE`, even if the actual bonding happened at a much higher price.

Code corrected:

A `minBondingAmount` has been added that can be set to up to 1,000 LP tokens. This ensures (if set to a sensible value) that leftover dust cannot be acquired.

6.6 Extra Reward Tokens Not Sent to RewardManager

Correctness | Medium | Version 1 | Code Corrected

CS-CCP-007

`RewardManager._swapRewardTokenForWeth()` assumes the extra rewards reside on the contract. This is not true as all tokens are sent to the corresponding Conic pool and are never sent to the `RewardManager`. No approvals from Conic pools to their reward managers exist for these extra tokens.

Code corrected:

`BaseConicPool` now has a function `updateRewardSpendingApproval()` that allows to set approvals of arbitrary tokens to the `RewardManager`. It is called each time a new reward token is added.

`_sellRewardTokens()` now transfers tokens from the respective Conic pool to the `RewardManager` before they are swapped.

6.7 Higher Imbalance Tolerance in Metapools

Design | Medium | Version 1 | Code Corrected

CS-CCP-044

In `CurveLPOracle.getUSDPrice()`, the pool balancing for Metapools is checked twice, once for the wrapping Metapool, and once for the base pool. This allows both pools to be unbalanced up to the maximum threshold, which is twice the imbalance threshold that would apply to a single pool.

Code corrected:

A new `customInternalImbalanceBuffers` storage mapping has been added that allows to set custom imbalance buffers for LP tokens of a base pool. If these parameters are set in the right way, the threat can be mitigated.

6.8 Incomplete Pool Balance Check

Design | Medium | Version 1 | Code Corrected

CS-CCP-045

`CurvePoolUtils.ensurePoolBalanced()` compares Chainlink prices to the prices returned by a Curve pool's `get_dy()` function. The checks are always performed from the first asset to all other assets. In pools with more than 2 assets, this can become problematic.

Consider the following scenario:

1. A Curve pool with 3 assets holds exactly 1000 tokens per asset (perfectly balanced).
2. The Curve pool accrues 0 fees (for simplicity) and has an A parameter of 2000.
3. An attacker trades 900 tokens from asset 1 to asset 2.
4. The attacker also trades 800 tokens from asset 1 to asset 0.
5. `get_dy(0, 1)` returns ~1.04.
6. `get_dy(0, 2)` returns ~0.96.
7. `get_dy(1, 2)` returns ~0.92.

With an imbalance buffer of 4% (simply for demonstration purposes, in production this would be smaller), the pool would still be considered balanced while there is an imbalance of 8% between asset 1 and 2.

Code corrected:

The function now checks all combinations of tokens in a given pool.

6.9 Oracle Price Manipulation

Security | Medium | Version 1 | Code Corrected

CS-CCP-009

Deposit and withdraw functions in Conic ETH pools are protected against reentrancy from a Curve pool that can potentially manipulate LP token prices. This is, however, not true for some other functions.

`InflationManager.updatePoolWeights()` can be called by reentering from a Curve pool resulting in skewed pool weights as the calculation relies on prices of the `CurveLpOracle` which can be manipulated by removing liquidity from a Curve pool that holds ETH and then reentering to the function in the callback.

`BaseConicPool.handleInvalidConvexPid()` allows to manipulate the `totalDeviationAfterWeightUpdate` storage variable which is, however, not used anywhere in the code.

Furthermore, `onlyOwner` functions that are called from the `GovernanceProxy` (if they have a delay) are also principally open to this manipulation as the changes can be executed by any user. `Controller.updateWeights()` and `updateAllWeights()` can be tricked into writing wrong LP token prices into the `_cachedPrices` of `BaseConicPool`, which can then be used to set the weight of a Curve pool to 0 with `handleDepeggedCurvePool()`.

Code corrected:

All mentioned functions are now executing reentrancy checks similarly to the deposit and withdraw functions.

6.10 Reward Double Counting

Correctness | Medium | Version 1 | Code Corrected

CS-CCP-010

`RewardManager.poolCheckpoint()` accrues rewards by storing a total integral and holdings since the last checkpoint per reward token:

```
function _updateEarned(
    bytes32 key,
    uint256 holdings,
    uint256 earned,
    uint256 _totalSupply
) internal {
    _rewardsMeta[key].earnedIntegral += earned.divDown(_totalSupply);
    _rewardsMeta[key].lastHoldings = holdings;
}
```

After claiming the regular rewards, `_claimPoolEarningsAndSellRewardTokens()` claims extra rewards on Convex and swaps them for CNC. These additional CNC rewards are then added to the total integral. The last holdings, however, are not updated accordingly.

```
if (_totalStaked > 0)
    _rewardsMeta[_CNC_KEY].earnedIntegral += receivedCnc_.divDown(_totalStaked);
```

As the CNC holdings of the contract increase, but the last holdings do not, the next checkpoint will count these tokens as new rewards again and add them to the integral again:

```
cncHoldings = CNC.balanceOf(conicPool);
...
uint256 cncEarned = cncHoldings - _rewardsMeta[_CNC_KEY].lastHoldings;
```

```
...
_updateEarned(_CNC_KEY, cncHoldings, cncEarned, _totalStaked);
```

Code corrected:

`_claimPoolEarningsAndSellRewardTokens()` now correctly sets the `lastHoldings` for CNC after selling reward tokens. Additionally, if rewards have to be claimed in `claimEarnings()`, the account share of the calling user is updated again after the rewards tokens have been swapped to ensure that the user receives the extra reward in the same call.

6.11 Slippage Losses Are Socialized

Design | Medium | Version 1 | Code Corrected

CS-CCP-040

`BaseConicPool.depositFor()` calculates the amount of LP tokens a user receives based on the value of the whole pool. When slippage is incurred, this is problematic as the depositor might receive more LP tokens than they should, resulting in a loss for all other liquidity providers. Consider the following example with simplified numbers:

1. A Conic pool contains one Curve pool with two tokens, 1000/1000 liquidity, 2000 LP total supply, token prices of 1 and an LP price of 1.
2. The Conic pool holds 1000 of the LP tokens and the Conic LP token has a total supply of 1000.
3. A user deposits 1000 token 0 to Conic.
4. Conic receives 900 LP tokens from Curve (slippage of 10%).
5. The Curve pool now holds 3000 USD value and has 2900 LP tokens total supply. Conic owns 1900 of these LP tokens.
6. Due to the slippage, the LP price of the Curve pool according to Conic now increased to 1.0345.
7. `underlyingBalanceAfter` therefore is now 1965, so the delta is 965.
8. The user now receives 965 Conic LP tokens for their deposit of 1000 tokens.
9. The Conic pool holds a total of 1900 Curve LP tokens which means the user's share of the Curve LP tokens now is 933 while the Conic pool only received 900 Curve LP tokens for the user's deposit.

Code corrected:

If the price of a Curve pool's LP token increases during a deposit, the price before the deposit is used to calculate the amount of LP tokens the user receives.

6.12 Weight Update Rounding Errors

Correctness | Medium | Version 1 | Code Corrected

CS-CCP-011

`BaseConicPool._setWeightToZero()` sets the weight of a given pool to 0 and scales the weights of all other pools accordingly to reach at a total weight of 1. This is done by computing a scale factor which involves a division. This division can result in rounding errors which will be passed to the upscaled

weights. Therefore, it is possible that the total weight after the operation is slightly smaller than 1, breaking the invariant that the sum of all weights must equal exactly 1.

In turn, this can become problematic when rebalancing rewards are active (which is the case after the function has been called) as deposit / withdrawal maximums are now calculated without `maxDeviation` gaps.

Consider the following example:

1. A Conic pool has two Curve pools with weights `[666666666666666667, 333333333333333333]`.
2. The first pool is depegged, resulting in the following weights: `[999999999999999999]`.
3. A user deposits 101 tokens with 18 decimals. `_getDepositPool()` returns a maximum amount of 100.999999999999999899 tokens that can be deposited to the Curve pool. Including the `1e2` constant in `_depositToCurve()`, the user's deposit can not completely be satisfied and the call results in a revert after a second iteration of `getDepositPool()`.

Code corrected:

`_setWeightToZero()` now adds the remaining weight to the last element that is not equal to the pool being set to 0-weight instead of multiplying its weight with the scaling factor. This ensures that all weights always sum up to 1. However, in Version 2, if the last pool had already 0-weight, it will incorrectly receive the remaining weight, setting the pool weight to a non-zero value as explained in Possible zeroed pool weight increase.

6.13 Wrong Denomination of Deviation Delta

Correctness Medium Version 1 Code Corrected

CS-CCP-012

`CNCMintingRebalancingRewardsHandler.computeRebalancingRewards()` computes rebalancing rewards with the following formula:

```
(elapsedSinceUpdate * cncRebalancingRewardPerDollarPerSecond).mulDown(
    deviationDelta.convertScale(decimals, 18)
);
```

The `cncRebalancingRewardPerDollarPerSecond` factor is per Dollar. It is therefore assumed that `deviationDelta` should be in USD denomination. This is, however, not the case as the value is in underlying.

For example, rebalanced deviation of 10.000 USDC would net ~0.833 CNC per hour, while a rebalanced deviation if 5 ETH (roughly the same value as the 10.000 USDC) would only net ~0.0004166 CNC.

Code corrected:

The formula has been corrected by multiplying the amount with the current price of the underlying token.

6.14 Possible Zeroed Pool Weight Increase

Correctness Low Version 2 Code Corrected

CS-CCP-041

`BaseConicPool._setWeightToZero()` sets the weight of the last pool that is not the pool whose weight is set to 0 to the leftover weight so that the total weights equal to exactly 1.

If this pool has already been set to 0 weight previously, the weight might increase again by some dust.

Code corrected:

The function now filters out all pools with 0 weight before performing the scaling.

6.15 Reward Factor Override

Correctness | Low | Version 2 | Code Corrected

CS-CCP-042

In `BaseConicPool.handleDepeggedCurvePool`, the `rebalancingRewardsFactor` is set even when rebalancing rewards are not activated. If rebalancing rewards have been activated before due to a weight update and the function is called on an empty pool, the reward factor is set regardless.

Code corrected:

The reward factor is now only set when rebalancing rewards are activated.

6.16 Boost After Shutdown

Correctness | Low | Version 1 | Code Corrected

CS-CCP-013

`LpTokenStaker.unstakeFrom()` calculates `_stakerCheckpoint()` even after the contract has been shut down, further increasing the boost of the users.

Code corrected:

`unstakeFrom()` now checks if the contract has been shut down before calling `_stakerCheckpoint()` (and `RewardManager.accountCheckpoint()`).

6.17 CNCLockerV3 Lock Squatting

Security | Low | Version 1 | Code Corrected

CS-CCP-014

Similarly to issue Lock spam DoS, a user who wants to avoid having its lock ever kicked (for example to use an airdropped boost indefinitely) can create a very big amount of 1 wei locks before and after the lock they wish to protect. Unlocking those locks will be very gas expensive for other users, and the cost will surpass the kicking reward. Kicking will not be possible since the gas cost of running `_getLockIndexById()` will exceed the block gas limit.

Code corrected:

A `_MIN_LOCK_AMOUNT` of 10 CNC has been introduced. Additionally, `_MAX_LOCKS` restricts the amount of locks a single account can hold to 10. It is now impossible to create enough locks for an account to be able to squat a certain lock.

6.18 Claimable Rewards Potentially Wrong

Correctness | Low | Version 1 | Code Corrected

CS-CCP-015

`RewardManager.claimableRewards()` returns 0 if the balance of a Conic pool is 0 in the `LpTokenStaker`. This is not correct if the pool already accrued some rewards and later all tokens are unstaked (for example after shutdown).

Code corrected:

`claimableRewards()` now does not return early if the balance of a Conic pool is 0 and instead returns the correct value.

6.19 Enabled Fee Not Reset

Correctness | Low | Version 1 | Code Corrected

CS-CCP-016

`RewardManager.setFeePercentage()` does not reset `feesEnabled` to `false` when the fee is set back to 0.

Code corrected:

`feesEnabled` is now set to `false` when the fee is set to 0.

6.20 Endless Loop

Correctness | Low | Version 1 | Code Corrected

CS-CCP-017

`RewardManager.poolCheckpoint()` claims rewards from Convex and the `LpTokenStaker` if certain conditions are met (either if there are not enough funds to cover fees or if the Convex cliff is approaching). If one of the conditions is met, and additionally the current `_INFLATION_RATE_PERIOD` has ended in the `InflationManager`, then the function executes an endless loop that calls back to itself (because the conditions are still met at the time of the callback) until the transaction runs out of gas. The callpath is as follows:

1. `RewardManager.poolCheckpoint()`.
2. `RewardManager._claimPoolEarningsForCliff()` (optional).
3. `RewardManager._claimPoolEarningsAndSellRewardTokens()`.
4. `RewardManager._claimPoolEarnings()`.
5. `LpTokenStaker.claimCNCRewardsForPool()`.
6. `LpTokenStaker._claimCNCRewardsForPool()`.
7. `InflationManager.executeInflationRateUpdate()`.

8. `InflationManager._executeInflationRateUpdate()`.
9. `InflationManager.updatePoolWeights()`.
10. `RewardManager.poolCheckpoint()`.

It is also worth to note that the subsequent calls of `LpTokenStaker.claimCNCRewardsForPool()` increase the amount of CNC minted every time since the `poolShares` are only reset after the call to `InflationManager.executeInflationRateUpdate()` while the shares are minted before. A fix of the issue should take this into consideration.

Code corrected:

`LpTokenStaker._claimCNCRewardsForPool()` no longer calls `InflationManager.executeInflationRateUpdate()` so there is no loop anymore.

6.21 Lock Spam DoS

Design | Low | Version 1 | Code Corrected

CS-CCP-019

`CNCLockerV3.lockFor()` allows anyone to create a lock for a given account. An attacker can create a big amount of 1 wei locks on a victim account, such that if a legitimate lock is then created unlocking it becomes impossible as the cost of running `_getLockIndexById()` exceeds the block gas limit.

This attack allows an actor to effectively freeze any CNC that is to be locked by a specific user. It is extremely costly though (Around $27k in gas fees at gas price 45 Gwei and ETH value $2000).

Code corrected:

A `_MIN_LOCK_AMOUNT` of 10 CNC has been introduced. Additionally, `_MAX_LOCKS` restricts the amount of locks a single account can hold to 10. It is now impossible to create enough locks for an account to be able DoS it. However, as described in note Locking in CNCLockerV3 can potentially fail if too many locks exist , some annoyance could be caused by an attacker willing to spend 100 CNC to create 10 locks for another user.

6.22 Minimum Tainted Transfer Amount Can Be Circumvented

Security | Low | Version 1 | Code Corrected

CS-CCP-020

`LpToken` sets a flag on accounts that `mint()` or `burn()` that disables them from minting or burning again in the same block. Since LP tokens can be transferred, the flag also has to be set on all addresses that the tokens are sent to.

To prevent cheap DoS attacks on arbitrary accounts that deposit or withdraw on a Conic pool, there exists a minimum threshold. Minting / burning of less than this amount will not set the flag.

The tainting mechanism is not in use when users stake their minted tokens directly in the `LpTokenStaker`. Only when the tokens are withdrawn again, the flag is set. This can be abused to circumvent the minimum tainted transfer amount in the following way:

1. Call `BaseConicPool.deposit()` with at least the minimum tainted transfer amount to a Conic Pool and set the `stake` argument to `true`.
2. Call `lpTokenStaker.unstakeFor()` with an amount of 1 wei and the address you want to DoS.
3. In the next block, withdraw the rest if the deposited amount.

Code corrected:

The transferred amount is now passed to `LpToken.taint()` (the function called by `lpTokenStaker.unstakeFor()` to taint a transfer) and the function checks for the minimum taint amount.

6.23 Rebalancing Reward After Depeg

Correctness Low Version 1 Code Corrected

CS-CCP-023

`BaseConicPool.handleDepeggedCurvePool()` automatically enables rebalancing rewards. This is, however, not necessary if the allocation of the given pool is already below the `_MAX_USD_VALUE_FOR_REMOVING_POOL` threshold.

Code corrected:

`handleDepeggedCurvePool()` now checks if the value of a Curve pool is below the threshold and does not start rebalancing rewards in that case.

6.24 Unreachable Imbalance Buffers

Design Low Version 1 Code Corrected

CS-CCP-025

`CurvePoolUtils.ensurePoolBalanced()` compares prices of token pairs on Curve with their respective Chainlink prices using an imbalance buffer as threshold. The imbalance buffers are set for individual tokens. A pair of two tokens will only ever be compared to the imbalance buffer of input token. Depending on the Curve pool configuration, this can result in the inability to set a buffer for a certain token.

For example, in an ETH/rETH Curve pool that contains WETH as the 0-token, it is not possible to use the imbalance buffer of rETH:

```
for (uint256 i = 0; i < poolMeta.numberOfCoins - 1; i++) {
    ...
    for (uint256 j = i + 1; j < poolMeta.numberOfCoins; j++) {
        ...
        toActual = ICurvePoolV2(poolMeta.pool).get_dy(i, j, fromBalance);
        ...
        require(
            _isWithinThreshold(toExpected, toActual, poolFee, poolMeta.imbalanceBuffers[i]),
            "pool is not balanced"
        );
    }
}
```

Code corrected:

The function now uses the minumum of the imbalance buffers of each token pair.

6.25 Wrong TVL Factor

Correctness Low Version 1 Code Corrected

CS-CCP-027

Conic stated that the `TVL_FACTOR` in `LpTokenStaker` is supposed to work in a way that gives the full boost to a user that holds 20% of the total staked amount using the following calculation:

```
uint256 stakeBoost = ScaledMath.ONE +
    userStakedUSD.divDown(totalStakedUSD).mulDown(TVL_FACTOR);
```

With the given `TVL_FACTOR` of 50, the full boost of 10 is already achieved with a share of 18%. The correct `TVL_FACTOR` for 20% would be 45.

Code corrected:

The `TVL_FACTOR` has been changed to 45.

6.26 Wrong Time to Full Boost

Correctness Low Version 1 Code Corrected

CS-CCP-028

`LpTokenStaker.getTimeToFullBoost()` returns the time for a given user until their full boost is active:

```
function getTimeToFullBoost(address user) external view returns (uint256) {
    uint256 fullBoostAt_ = boosts[user].lastUpdated + INCREASE_PERIOD;
    if (fullBoostAt_ <= block.timestamp) return 0;
    return fullBoostAt_ - block.timestamp;
}
```

This calculation is not correct. For example, the function returns the full `INCREASE_PERIOD` for a user that just reached their full boost amount in the current block, while it should return 0. `lastUpdated` is the point in time when a user's boost has been updated the last time. The function is therefore only correct for users that have just created a new position.

Code corrected:

The function `getTimeToFullBoost()` has been removed.

6.27 Ambiguous Naming

Informational Version 1 Code Corrected

The following code parts contain symbols that are not precise and could be misunderstood:

- `ChainlinkOracle._getPrice()` defines a boolean argument `shouldRevert`. Contrary to the name of the argument, the function can still revert when it is set to true.
- `InflationManager.hasPoolRebalancingRewardHandlers()` allows to check a single handler address while the function name contains the word "handlers" in plural.
- `LpTokenStaker.unstakeFor()` allows a user to unstake their own tokens to a specific address. It does not, as opposed to the naming, allow a user to unstake for another address.

Code corrected:

All aforementioned function names have been changed except for the function `unstakeFor()` because it is a public interface that has already been in use before.

6.28 Exchange Rate Race Condition

Informational **Version 1** **Code Corrected**

`BaseConicPool.deposit()` allows to instantly stake the freshly minted LP tokens with the following flow:

1. Mint tokens.
2. Stake tokens on the `LpTokenStaker`.
3. Update the `_cachedTotalUnderlying`.

`LpTokenStaker.stakeFor()` calls `BaseConicPool.usdExchangeRate()` in *step 2*. It is implemented in the following way:

```
function usdExchangeRate() external view virtual override returns (uint256) {
    uint256 underlyingPrice = controller.priceOracle().getUSDPrice(address(underlying));
    return _exchangeRate(cachedTotalUnderlying()).mulDown(underlyingPrice);
}

function _exchangeRate(uint256 totalUnderlying_) internal view returns (uint256) {
    uint256 lpSupply = lpToken.totalSupply();
    if (lpSupply == 0 || totalUnderlying_ == 0) return ScaledMath.ONE;

    return totalUnderlying_.divDown(lpSupply);
}
```

As can be seen, the exchange rate is calculated by dividing the cached total underlying (which is not yet updated in the call) by the total supply of the LP token (which has already been increased due to the minting in *step 1*). The exchange rate is therefore erroneously deflated.

However, this exchange rate is used in a way that completely factors it out in this call which makes this call safe after all.

Code corrected:

`_cachedTotalUnderlying` is now updated before the calls to `LpToken.mint()` and `LpTokenStaker.stakeFor()`.

6.29 Fees Without Locked CNC

Informational | Version 1 | Specification Changed

CS-CCP-033

`RewardManager.setFeePercentage()` only allows to set a fee if the `CNCLockerV3.totalBoosted() > 0`. Once the fee is set, fees are however still accrued even if the amount of locked CNC goes down back to 0.

Specifiaction changed:

`setFeePercentage()` now no longer requires `CNCLockerV3.totalBoosted() > 0`.

6.30 Interface Differences

Informational | Version 1 | Code Corrected

CS-CCP-035

The following parts of the code have non-uniform interfaces:

- `BaseConicPool.depositFor()` allows the minted LP tokens to be instantly staked by setting a boolean parameter. `withdraw()` does not expose such a boolean parameter for unstaking LP tokens. Instead, a separate function `unstakeAndWithdraw()` must be used. This interface is non-uniform.
- `RewardManager` contains a function `accountCheckpoint()` while `Bonding` contains a function `checkpointAccount()`.

Code corrected:

`Bonding.checkpointAccount()` has been renamed to `Bonding.accountCheckpoint()`. The withdraw function names are kept as-is for backwards compatibility.

6.31 Missing Events

Informational | Version 1 | Code Corrected

CS-CCP-036

The following state-changing functions are not emitting events (the list is non-exhaustive):

- `RewardManager.poolCheckpoint()`.
- All functions in `SimpleAccessControl`.
- `ChainlinkOracle.setHeartbeat()`.

Code corrected:

Events have been added to most functions where it makes sense.

6.32 Rebalancing Reward Formula Mismatch

Informational | Version 1 | Specification Changed

CS-CCP-037

The doc comments of `CNCMintingRebalancingRewardsHandler.computeRebalancingRewards()` describe the following formula:

CNC = t * CNC/s * (1 - (Δdeviation / initialDeviation))

This is different to the actual implementation:

```
(elapsedSinceUpdate * cncRebalancingRewardPerDollarPerSecond).mulDown(
    deviationDelta.convertScale(decimals, 18)
);
```

The formula in the comments is also likely wrong as it would imply lower rewards the higher the deviation delta is.

Specification changed:

The formula has been updated.

6.33 Shadowed Variables

Informational | Version 1 | Code Corrected

CS-CCP-038

The `LpToken` constructor's arguments `name` and `symbol` shadow the storage variables of the `ERC20` contract.

Code corrected:

The variable names have been changed.

6.34 Typographical Errors

Informational | Version 1 | Code Corrected

CS-CCP-039

Typographical errors have been identified in the following parts of the code:

1. `BaseConicPool.depositFor()` and `shutdownPool()` define an error message that contain the word "shutdown" as a verb.
2. Doc comments of `BaseConicPool.handleInvalidConvexPid()` contain the word "shutdown" as a verb.
3. Doc comments of `BaseConicPool.handleInvalidConvexPid()` contain the word "outcomu".
4. Doc comments of `BaseConicPool.handleInvalidConvexPid()` contain the word "unilkely".
5. Doc comments of `ETH_FACTORY_POOL_CODE_HASH_1` in `CurveHandler` contain the phrase "a optimization".

6. Error string `"convex pool pid is shutdown"` in `BaseConicPool.handleInvalidConvexPid()` is inconsistent with the check performed.

Code corrected:

All errors have been fixed.

6.35 `removeDuplicates` Not Working With 0 Elements

Informational | Version 1 | Code Corrected

CS-CCP-043

`ArrayExtensions.removeDuplicates()` does not work correctly if the array passed as arguments contains the 0-address. It will be filtered out.

Code corrected:

The function now correctly checks for 0 elements.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Code Copies

Informational **Version 1**

CS-CCP-030

The project contains multiple functions that share a similar or even identical codebase. The common functionality should be refactored into separate functions to minimize the risks of future changes resulting in different functions behaving differently when they should behave the same way.

Examples are:

- `BaseConicPool._withdrawFromCurve()` and `_depositToCurve()`.
- `LpTokenStaker.poolCheckpoint()` and `claimableCnc()`.
- Multiple functions in `CNCLockerV3`.

7.2 Events Emitted on No Change

Informational **Version 1** **Code Partially Corrected**

CS-CCP-031

Some functions emit events even when no change in storage has occurred. Here are some examples:

1. `BaseConicPool.updateDepegThreshold()`.
2. `Controller.setCurveHandler()`.
3. `RewardManager.addExtraReward()`.
4. `RewardManager.removeExtraReward()`.

Code corrected:

Most of the functions have been corrected to only emit events when the state changes.

7.3 Gas Optimizations

Informational **Version 1**

CS-CCP-034

Some code parts can be optimized for better gas efficiency.

1. Redundant calls. For example:
   - `BaseConicPool.depositFor()` calls the price oracle for the underlying price. The same call is then performed in `_exchangeRate()` and potentially `_isBalanced()`.
   - In `CurveHandler._version_0_remove_liquidity_one_coin()`, the call `CurveRegistryCache.coins()` is executed in each loop iteration.

- `RewardManager.poolCheckpoint()` could send fees directly from a Conic pool to the `CNCLockerV3`.
- `CurveAdapter._stakedCurveLpBalance()` calls `IConvexHandler(controller.convexHandler()).getRewardPool()` which in turn calls `CurveRegistryCache.getRewardPool()`. This function could be called directly.
- The call to `CurveRegistryCache.nCoins()` in `CurveLpOracle.getUSDPrice()` can be omitted as the number of coins is already available.
- `GenericOracle.getUSDPrice()` calls `ChainlinkOracle.isTokenSupported()` which calls `ChainlinkOracle.getUsdPrice()`. It then proceeds to call `ChainlinkOracle.getUSDPrice()`.

2. Redundant storage reads. For example:
   - `BaseConicPool._getDepositPool()` loads all `pools` and `weights` in each iteration of `_depositToCurve()`.
   - `BaseConicPool._depositToCurve()` loads the pool address from storage when it could have just been passed back by `_getDepositPool()`.
   - `CNCLockerV3._feeCheckpoint()` loads `accruedFeesIntegralCrv` and `accruedFeesIntegralCvx` multiple times from storage.
   - `LpTokenStaker._claimCNCRewardsForPool()` loads `poolShares` from storage instead of using the return value of `checkpoint()`.
3. Redundant storage writes. For example:
   - `GovernanceProxy._endChange()` writes data to the pending change in storage before deleting it from storage.
4. Unnecessary computation. For example:
   - The loop in `BaseConicPool._getDepositPool()` does not continue if the weight of a given pool is 0.
   - The computation of `_isEthIndexFirst()` in `CurveHandler.isReentrantCall()` is irrelevant.
   - `RewardManager.poolCheckpoint()` does not set the `rewardsClaimed` flag when rewards are claimed due to being within the threshold of the Convex cliff. This results in `claimPoolEarningsAndSellRewardTokens()` potentially executing the claiming functionality two times.
   - `RewardManager.poolCheckpoint()` does not return early if no rewards have accrued.
5. Unoptimized structs in storage. For example:
   - The size of the `endedAt` field in the `Change` struct of `IGovernanceProxy` could be reduced to fit the `Status` enum into the same word.
   - The `Boost` struct in `LpTokenStaker` could be optimized to only occupy 1 word.
6. `_chainlinkOracle` and `_curveLpOracle` in `GenericOracle` can be `immutable`.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Extra Rewards Might Have 100% Slippage

Note Version 1

`RewardManager` swaps all extra rewards of a Convex pool to CNC on either SushiSwap or Curve. If the respective token is not supported by the `GenericOracle`, no slippage protection is set for these swaps. It is likely that these swaps will be arbitraged by bots.

8.2 Locking in CNCLockerV3 Can Potentially Fail if Too Many Locks Exist

Note Version 1

No more than `_MAX_LOCKS` can exist in `CNCLockerV3` for every user. If a user has created more, or if an attacker targets a user, they could be prevented from creating new locks according to their intentions. New locks can always be created by using the `relock` option of `lockFor()`, however that requires the duration to be longer than any of the existing locks.

If it is not possible to create new locks for an address, an alternative address will have to be used. If an airdrop cannot be used because of having reached `_MAX_LOCKS`, the airdrop can still be used on another address through `lockFor()`.

8.3 RewardManager Can Become Temporarily Insolvent

Note Version 1

`RewardManager` handles CVX rewards by calculating the current amount of earnings for the current Convex cliff period. If the end of the cliff period approaches, earnings are finally claimed. It is, however, possible that there are no interactions with the contract for a longer period which would result in this claim being missed before the period ends. In that case, the amount of CVX rewards in the contract are inflated as the actual claimable reward is lower than the reward that has been calculated before. It is therefore possible that not all claims can be served until the new incoming CVX reward reach the previously calculated amount of CVX rewards that *should* be in the contract.

Users that stake after this incident also do not accrue CVX rewards in favor of older stakers.

8.4 Rewards of Rebalance Function

Note Version 1

`CNCMintingRebalancingRewardsHandler.rebalance()` allows users to easily rebalance a Conic pool and earn rewards. This includes rewards for `withdraw()` which are not granted if the function is called directly.

The setup allows for additional reward handlers to be added to a Conic Pool. These reward handlers, however, will only grant rewards for deposits even when the `rebalance()` function is used.