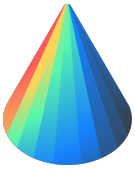
PUBLIC

Code Assessment

of the Crypto Pool Update Smart Contracts

September 03, 2024

Produced for



CONIC

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Conic,

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 Crypto Pool Update according to Scope to support you in forming an opinion on their security risks.

Conic implements a new oracle for pricing LP tokens of Curve Crypto pools. Additionally, Conic implements a new contract for determining the share of CNC rewards that is distributed to each Conic pool.

The most critical subjects covered in our audit are functional correctness and resistance to oracle manipulations. The contracts are functionally correct and are, in most cases, resistant against oracle manipulations under the assumptions that:

1. Curve's `price_oracle()` cannot be manipulated to a lower value during a maximum of 2 blocks.
2. Curve pool imbalances are efficiently arbitraged every block.
3. `CryptoPoolOracle` is not used for StableSwap pools.
4. The underlying Curve pools experience regular usage.

However, some certain edge conditions can enable oracle manipulation attacks that are able to extract value: Oracle manipulation during withdrawal. Conic, for now, accepts this risk and tries to find an optimal solution.

In summary, we find that the codebase provides an improvable 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	0
High-Severity Findings	0
Medium-Severity Findings	1
• Code Partially Corrected	1
Low-Severity Findings	4
• Code Corrected	2
• Risk Accepted	2

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 Crypto Pool Update repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	11 July 2024	d0870eb522ccf4d345ce336394aaaa8bccad8a1b	Initial Version
2	14 August 2024	860c08429274949a23e49b783d89158c16d683fc	Second Version
3	30 August 2024	5430462ca10b53ed246bd51136f2eb5cdfd1e483	Final Version

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

The following contracts were in scope of the review:

1. contracts/tokenomics/InflationManagerV2.sol
2. contracts/oracles/CryptoPoolOracle.sol

In Version 2, the following contracts have been added to the scope:

1. contracts/ConicCryptoPool.sol
2. contracts/LpTokenCrypto.sol

2.1.1 Excluded from scope

Excluded from scope are any other contracts including third party libraries and external interactions.

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 liquidity pools that allocate deposits of a single underlying token to various Curve pools and deposit the resulting LP tokens into Convex. Additionally, Conic offers reward contracts that handle the rewards paid by Curve, Convex and Conic itself.

Formerly, only deposits into Curve Stable pools were possible. This report examines a new update to the protocol that enables deposits into Curve Crypto pools. Additionally, it examines a new distribution method for the Conic rewards. To see a system overview of the general Conic protocol, please refer to our Conic Protocol Report.

2.2.1 CryptoPoolOracle

Conic determines the value of deposited funds by multiplying the LP balance of each underlying Curve pool with an oracle price. For Stable pools, this is oracle price is determined by the `CurveLPOracle` which calculates the LP value in the following way:

$$\frac{\sum_{i=0}^{n} balance(i) * oraclePrice(i)}{totalSupply}$$

As this calculation is easily manipulatable, the `CurveLPOracle` reverts if token prices on a given Curve pool deviate from the Chainlink prices of the respective tokens by more than a pre-defined *imbalance buffer*. Since Stable pools allow trading in a rather wide range without incurring much slippage, this system works fine as long as the imbalance buffers are chosen correctly for the respective *A* values of each Curve pool.

Crypto pools, however, incur more slippage during regular trading, which would result in regular Denial of Service if the `CurveLPOracle` were used. To support Crypto pools, the new `CryptoPoolOracle` has been created. Instead of computing the LP prices directly from pool balances, it uses the `lp_price()` function of Curve pools (pools without that function are not supported). The `lp_price()`, which states the price of an LP token denominated in token 0 of a pool, is then simply multiplied with the Chainlink oracle price of token 0. If no Chainlink oracle price of token 0 is available, it converts the LP price to a denomination of one of the other underlying tokens and multiplies it with that token's Chainlink oracle price:

$$\frac{pool.lp\_price() * chainlink.latestRoundData(pool.coins(i))}{pool.price\_oracle(i-1)}$$

`lp_price()` is dependent on the `price_oracle()` function which, in most Curve pools, is an EMA of a the pool's token prices denominated in token 0.

2.2.2 InflationManagerV2

Depositors on a Conic pool get Conic LP tokens in return. These tokens can be staked in the `LpTokenStaker` contract to receive rewards. Besides CRV and CVX, these rewards consist of CNC, Conic's governance token. To determine the amount of CNC that are rewarded, the `LpTokenStaker` fetches an inflation rate from the `InflationManager` contract for each existing Conic pool. The total inflation rate is divided equally between pools as per their total value.

A new `InflationManagerV2` contract has been created that will replace the existing `InflationManager`. Instead of automatically setting each pool's share of the inflation according to the pool's value, the contract exposes a function `setPoolWeights()` that allows the owner (the Conic governance) to set the share manually.

2.2.3 Changes in Version 2

In VERSON2, the following changes have been added:

1. The `ConicCryptoPool` implementation.
2. The `LpTokenCrypto` for crypto pools.

`ConicCryptoPool` is a new contract that inherits from `BaseConicPool` and disables price caching and asset depegging. `LpTokenCrypto` is a new contract that inherits from `LpToken` and enables tainting over multiple blocks.

2.2.4 Roles & Trust Model

`CryptoPoolOracle` is set by the owner of the `GenericOracle` contract, either by replacing the existing LP oracle or by setting a custom oracle for each LP token that should be priced by this oracle. The owner is fully trusted to set the right oracle contract. Misconfigurations can result in losses.

The owner of the `InflationManagerV2` can set the weights of each Conic pool arbitrarily. The owner is partially trusted to set fair weights. Apart from decreasing the CNC rewards of certain Conic pools, no harm can be done.

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	1

- Oracle Manipulation During Withdrawal Code Partially Corrected Risk Accepted

Low-Severity Findings	2

- Boost Can Be Manipulated Through Read-Only Reentrancy Risk Accepted
- Unfair Unbalanced Pool Withdrawals Risk Accepted

5.1 Oracle Manipulation During Withdrawal

Security Medium Version 1 Code Partially Corrected Risk Accepted

CS-COIM-001

Withdrawals are performed in a way that balances out value distribution among underlying Curve pools according to the chosen weights. The actual distribution of withdrawals to the Curve pools is, however, subject to the LP price of the pools. Consider the following example:

1. Fees are assumed to be zero for simplification.
2. Curve pool #1 holds 1000A / 1000B liquidity, an LP price of 1 and a supply of 2000 LP tokens.
3. Curve pool #2 holds 1000A / 1000C liquidity, an LP price of 1 and a supply of 2000 LP tokens.
4. All tokens have a price of 1.
5. A Conic pool is comprised of Curve pool #1 and #2 with underlying A and [50, 50] weights. It holds 1000 LP tokens of each pool and a total supply of 2000 Conic LP tokens. Max deviation is 0.
6. A user that wants to withdraw 1000 Conic LP tokens, withdraws exactly 500 LP tokens from each Curve pool (assuming no slippage for simplification).
7. If the user performs an oracle manipulation on pool #1 that increases the LP price to 2 and then withdraws 1000 LP tokens, they withdraw 625 LP tokens from pool #1 and 250 LP tokens from pool #2. This is unprofitable but shifts the weights.
8. However, if the user backruns a weight update to [100, 0], they withdraw 1000 LP tokens from pool #2 and an additional 250 LP tokens from pool #1.

LP prices of Curve pools can be manipulated over 2 blocks as the `price_oracle()` is an EMA. A manipulation over 2 blocks can be achieved by increasing the price extremely in the last transaction of a block and then trading back in the first transaction of the next block. Such attacks are possible for

validators: A validator can know two epochs in advance which block they will propose. They can then get access to the previous block's last transaction via Flashbots or similar services.

Code partially corrected:

Conic has introduced a new LP token `LPTokenCrypto` that allows to extend the tainting mechanism to more than 1 block. This helps against attacks where the attacker deposits only after they see a weight update in the mempool. However, if an attacker has deposited some other time before, the attack is still possible.

Additionally, it should be noted that, depending on the size of the manipulation, it takes some additional blocks for a manipulated Curve pool's oracle price to return to normal. A suitable taint period should therefore be chosen carefully.

Risk accepted:

Conic states that the remaining issue requires a more thorough solution and thus accepts the risk until such a solution is found.

5.2 Boost Can Be Manipulated Through Read-Only Reentrancy

Security | Low | Version 1 | Risk Accepted

CS-COIM-002

`LpTokenStaker.updateBoost()` can be manipulated through read-only reentrancy. This can be exploited since the call might rely on `CryptoPoolOracle` which depends on the curve pool contract when calling `lp_price()`.

Risk accepted:

Conic accepted the risk stating:

```
This attack would require substantial capital and time commitment (including locking of governance tokens). We therefore accept the risk.
```

5.3 Unfair Unbalanced Pool Withdrawals

Design | Low | Version 1 | Risk Accepted

CS-COIM-005

`BaseConicPool.withdraw()` determines the amount that should be withdrawn by calculating the pool value with Curve's `lp_price()` (if `CryptoPoolOracle` is used). When the Curve pool is unbalanced in favor of the underlying token, the actual withdrawn amount is greater than the calculated amount. The withdrawing user, however, does not receive any benefits of this re-balancing as their withdrawal amount is capped:

```
uint256 underlyingWithdrawn_ = _min(
    underlying.balanceOf(address(this)),
    underlyingToReceive_
);
```

The remaining balance stays on the contract and now increases the total value of the pool, giving subsequent withdrawers higher payouts.

Risk accepted:

Conic will not address this issue to avoid creating potential new attack vectors but plans to display a warning to users in the UI.

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	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	2

- Missing Base Implementations Code Corrected
- Missing Sanity Check Code Corrected

Informational Findings	4

- Gas Optimizations Code Corrected
- currentPoolWeights Has No Initial Value Code Corrected
- Unused Variable Code Corrected
- Redundant Checks Code Corrected

6.1 Missing Base Implementations

Correctness Low Version 1 Code Corrected

CS-COIM-003

This review was conducted on the implementation of the `CryptoPoolOracle` contract. It is used to provide some pricing for future usage of Curve's Crypto pools. However, this future usage requires some additional contracts and/or changes to be implemented. This includes, for example, a contract inheriting from the `BaseConicPool` contract that disables price caching and asset depegging and enables reentrancy checks when necessary.

Code corrected:

`ConicCryptoPool` contract was added to the codebase. It inherits from the `BaseConicPool` contract and disables price caching and asset depegging. It also enables reentrancy checks by default.

6.2 Missing Sanity Check

Correctness Low Version 1 Code Corrected

CS-COIM-004

The caller of `InflationManagerV2.setPoolWeights()` can freely choose the `pools` list. It can be any arbitrary list of Conic pools containing duplicates, as long as the size is equal to the amount of active pools.

If such duplicates exist in the array, it is possible that the invariant `totalPoolWeight == ScaledMath.ONE` is violated.

Code corrected:

The code now iterates over the list of active pools obtained from the controller instead of a user-supplied list.

6.3 Gas Optimizations

Informational **Version 2** **Code Corrected**

CS-COIM-009

`InflationManagerV2.setPoolWeights()` uses two `for` loops to check for duplicates in the input array, which is inefficient. Also, in this case, the supposed array composition is known in advance and thus can be fetched instead of supplied by the user (`Controller.listActivePools()`).

Code corrected:

`setPoolWeights()` now iterates over the already fetched active pools.

6.4 Redundant Checks

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

CS-COIM-006

Some checks exist in the contracts that are not required / redundant:

1. `InflationManagerV2.updatePoolWeights()` performs `runSanityChecks()` but does not call any Curve functions.
2. `InflationManagerV2._executeInflationRateUpdate()` calls `updatePoolWeights()` but the call is only required when a pool is shut down.

Code corrected:

1. `runSanityChecks()` is removed from `InflationManagerV2.updatePoolWeights()`.
2. `updatePoolWeights()` is removed from `InflationManagerV2._executeInflationRateUpdate()` and replaced by a simple checkpoint of each pool.

6.5 Unused Variable

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

CS-COIM-007

`InflationManagerV2.totalLpInflationMinted` is declared but never used.

Code corrected:

The unused variable has been removed.

6.6 `currentPoolWeights` Has No Initial Value

Informational | Version 1 | Code Corrected

CS-COIM-008

`InflationManagerV2.currentPoolWeights` is a mapping from pool address to the pool's weight. These weights are, however, neither set in the initializer nor the constructor. Doc comments of `initializeInflationData()` indicate that the function should be called in the same transaction the inflation manager is updated in the `Controller`. However, it is not mentioned that the pool weights also should be set in that same transaction as they are not copied from the old manager.

Code corrected:

`InflationManagerV2.initializeInflationData()` now also sets the pool weights (equally distributed). Note that this reverts if the number of active pools is 0

7 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.

7.1 Deposit DoS

Note Version 1

On deposit, pools determine the amount of LP tokens a user receives by dividing the supplied amount by the exchange rate. The exchange rate is calculated as a sum of each underlying Curve pool's LP balance with its LP price. It is possible for threat actors to manipulate the LP price upwards (slightly in one block or more extremely over two blocks). Such manipulations will result in depositors receiving noticeably less LP tokens. The depositors now have the choice of either losing money (by setting a low slippage protection) or not depositing at all until the LP price has normalized.

This risk is, however, rather low due to the high cost of such a manipulation.