PUBLIC

Code Assessment

of the Curve Stablecoin Smart Contracts

January 24, 2024

Produced for

by

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 8
4 Terminology 9
5 Findings 10
6 Resolved Findings 11
7 Notes 25

1 Executive Summary

Dear Michael,

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

Curve implements a new stablecoin that is based on different mechanics to keep it stable and manage the loans.

We did not identify any critical issues. All high severity findings were resolved. Some new medium severity issues were identified in the latest review of the codebase. There are still many low severity issues open, and given a stable codebase and more time, likely many more could be found, due to the complexity of the codebase. However, assuming the more severe issues are addressed, they should be mostly benign.

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

The contracts are complex and have even more complex dependencies. We did not review the economic soundness of the contracts nor is it possible to find all the edge cases in this system. 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.

Findings	Count
Critical-Severity Findings	0
High-Severity Findings	3
• Code Corrected	3
Medium-Severity Findings	7
• Code Corrected	6
• Code Partially Corrected	1
Low-Severity Findings	20
• Code Corrected	18
• Specification Changed	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 Curve Stablecoin 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	04 October 2022	32f85fe9b06b538cce8fb3a952af4523fc9f93b1	Initial Version
2	31 October 2022	59171820b0b41510157778d49335dd3bbf06fcdf	Version 1
3	17 February 2023	475ccb7572b93a2b826f10edc2678b4aff0bfc48	Version 2
4	19 April 2023	f7a514ae24f86fc4856401826f8bc6cc207451d1	Version 3
5	7 May 2023	7b1e773877c9e9055b41db320b131626fd98faf2	Version 4
6	1 July 2023	64dc13db563ec6067c75c662ee71a285442ef638	Version 5
7	12 August 2023	5c61cdf2cb2098595ad25cb5f6cc479b3201f4bd	Version 6
8	28 August 2023	b048fc782bd80a868d4ed882b3e6b371b40c1c03	Version 7
9	11 Dec 2023	5a46bb9c1f43b7d4062127b9919e3c2ed366ad34	PegKeeperV2

For the vyper smart contracts, the compiler version `0.3.7` was chosen.

The following files were in scope:

- mpolicies/AggMonetaryPolicy.vy
- mpolicies/AggMonetaryPolicy2.vy (added in Version 5)
- price_oracles/AggregateStablePrice.vy
- price_oracles/AggregateStablePrice2.vy (added in Version 5)
- price_oracles/CryptoWithStablePriceAndChainlinkFrxeth.vy
- stabilizer/PegKeeper.vy
- AMM.vy
- Controller.vy
- ControllerFactory.vy
- Stablecoin.vy

2.1.1 Excluded from scope

Third-party dependencies, testing files, and any other files not listed above are outside 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.

Curve offers a new stablecoin backed by three core concepts:

- An AMM handling (partial) liquidations in most cases
- A peg keeper in combination with a stable swap exchange
- A Monetary policy

Stable Coin

The stablecoin contract itself is an ERC20-compliant, mintable, and burnable token. The contract has one admin, which should be the controller factory contract.

Collateral Token

To borrow stable coins, collateral needs to be deposited. Collateral contracts are assumed to be ERC20-compliant tokens with no uncommon behavior like deflation, inflation or callbacks.

AMM (LLAMMA) contract

To prevent bad debt (the collateral being worth less in stable coin than the debt in stable coin), a special-purpose AMM sells the collateral step-by-step if it falls in price against the stablecoin. The AMM differs from Uniswap in that, when the price of the collateral drops, the AMM accumulates stablecoins and vice versa. Such an AMM is able to perform liquidations automatically. Therefore, this is referred to as lending-liquidating AMM algorithm (LLAMMA). Like any AMM, the LLAMMA allows depositing liquidity, withdrawing and exchanging. But only exchanging is non-restricted. Deposits and withdrawals must to be done via the controller contract. Liquidity deposits are initially always in collateral and, when prices fall to a certain level, the collateral is exchanged for stablecoin. This kind of soft liquidation ensures that, in the end, the collateral is fully liquidated before the debt gets underwater.

Oracle contracts

Three price oracle contracts are implemented `AggregateStablePrice`, `CryptoWithStablePrice` and `EmaPriceOracle`. All contracts aggregate prices in different ways and return the collateral price.

Monetary Policy

The monetary policy contracts include two implementations. `AggMonetaryPolicy` and `ConstantMonetaryPolicy`. Both contracts return a rate. This rate is used to discount the base price of the LLAMMA contract. This mechanism is implicitly acting like a loan interest rate by discounting the base price of the LAMMA. `ConstantMonetaryPolicy` will always return the rate that is currently set by the admin contract. `AggMonetaryPolicy` dynamically calculates a rate by weighting the oracle price with the aggregated debt of the peg keeper contracts in relation to the debt of the controller and a target debt ratio.

Peg Keeper

The Peg Keepers can add and withdraw one-sided liquidity to stable swap exchanges to push or pull the price up or down. The peg keeper assumes that prices should always be 1:1 to the pegged asset. Hence, they act when the balances in the pool are not equally distributed. The peg keeper checks ex-post that the action's impact did not change the price in an unfavorable direction (pushed the price over the 1:1 ratio in the wrong direction).

Stable Swap

The stable swap contract is the latest version of the common stable swap pool. It allows the trading of two assets that should stay in a very small price range.

Controller

The controller is the entry point for users to get a loan and manage their debt positions. Additionally, the contract also allows users to liquidate either themselves or other users with bad debt. The controller contract is the admin of the corresponding LLAMMA contract.

Controller Factory

The Controller Factory manages the deployment of new markets (consisting of a controller and a LLAMMA), the monetary policy and the peg keepers. It oversees and manages the stablecoin minting and, hence, the limits of each debt controller. The factory has the admin role in the stablecoin. The factory's admin is also the admin of the controllers.

Assumptions and thread model:

- Collateral token contracts are assumed to be ERC20-compliant tokens with no uncommon behavior like deflation, inflation, or callbacks.
- All permissioned roles are fully trusted.
- All token balances are smaller than `2**127`.
- We assume that the LLAMMA contract's admin functions are only accessed via the controller.
- All stablecoins in the pools upon which the Peg Keeper is acting are 1:1 and do not lose their peg.
- It won't be necessary to loop over more than `MAX_SKIP_TICKS`.

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

- Manipulable Price Calculation in AggregateStablePrice Method Code Partially Corrected

Low-Severity Findings	0

5.1 Manipulable Price Calculation in AggregateStablePrice Method

Security Medium Version 1 Code Partially Corrected

CS-CRVUSD-004

The `price()` function in the `AggregateStablePrice` contract calculates the price of the stablecoin based on the total supply of stableswap pools.

```
pool_supply: uint256 = price_pair.pool.totalSupply()
```

It is possible to manipulate this value, as a malicious actor could significantly change the total supply of pools by using a large amount of capital (obtained for example with a flashloan). This manipulation could alter the computed stablecoin price between the range of the stableswap pool with the lowest price to the stableswap pool with the greatest price. Given the function's role in determining the price used by the main price oracle, the pegkeepers, and the monetary policies, this may represent a risk.

Code partially corrected:

The new `AggregateStablePrice2` contract implements an exponential moving average over the total supplies of the pools. Note that the *first* time the price is calculated in a block is then valid for the remainder of that block. This means that the price is still manipulable to some extent (e.g. using a flashloan), although due to the moving average the effect will be reduced. An solution such as using the last price from the previous block may be a more suitable alternative, however it would require moving the totalSupply EMA oracle from an external contract to the *StableSwap* contract.

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	3

- Checks-effects-interactions Pattern and Reentrancy Locks Code Corrected
- Incorrect Verification of Health Limit Code Corrected
- Oracle Price Updates Can Be Sandwiched Code Corrected

Medium-Severity Findings	6

- PegKeeper Can Be Drained if Redeemable Stablecoin Permanently Depegs Code Corrected
- Incorrect Max Band Code Corrected
- Interest Rate Does Not Compound Code Corrected
- Manipulation of Active Band Code Corrected
- Non-Tradable Funds Code Corrected
- Potential Denial of Service (DoS) Attack on Peg Keeper Code Corrected

Low-Severity Findings	20

- A User's Liquidation Discount Can Be Updated by Anyone at Any Time Code Corrected
- ApplyNewAdmin Event Emitted With Wrong Argument in PegKeeper Code Corrected
- Draining Funds Code Corrected
- Inaccurate _p_oracle_up(n) for High/Low Values of n Code Corrected
- Incorrect Array Length Specification Changed
- Incorrect Calculations in health_calculator Code Corrected
- Incorrect Comments Code Corrected
- Meaningful Revert Reasons Specification Changed
- Missing Sanity Checks Code Corrected
- Multiple Calls to the AMM Code Corrected
- No Events Code Corrected
- Non-Indexed Events Code Corrected
- Potential Optimization With Immutable PriceOracle Code Corrected
- Potentially Incorrect Admin Fees Code Corrected
- Simpler Calculations Possible Code Corrected
- Superfluous Check Code Corrected
- Superfluous Interface Definitions Code Corrected
- Superfluous Variable Assignment for Number of Bands Code Corrected
- Unnecessary Subtraction Code Corrected

- Unused Variables Code Corrected

6.1 Checks-effects-interactions Pattern and Reentrancy Locks

Design High Version 1 Code Corrected

CS-CRVUSD-015

Some external calls to the collateral token deviate from the checks-effects-interactions pattern. If no reentrancy lock is present, these calls might introduce reentrancy possibilities (especially read reentrancies) before the state is fully updated. We could not find a case where the non-updated state might be relevant information. Still, it might be worth considering fully adhering to the checks-effects-interactions pattern.

For example,

- in `AMM.exchange()`, the transfer is done before the bands are updated;
- in `AMM.withdraw()`, the old rate information would still be returned;
- in `Controller.create_loan()`, the intermediate stable coin balance is returned.

The Reentrancy locks appear to be set inconsistently. We at least cannot see the underlying logic of how they are added. Some admin setters have a `nonreentrant` decorator and some do not.

For important functions like `exchange` the decorator seems to be forgotten after a code change. For this reason, the issue was rated higher.

Code corrected

The missing reentrancy lock on `exchange()` has been added. Some missing reentrancy locks have been explained. All but one of the remaining external functions without locks seem to be safe even without a lock.

The Controller's `total_debt()` function will return outdated / inconsistent values compared to the AMM's state if called during the callback of `repay_extended` and `_liquidate`. More precisely, the AMM's state will already reflect the withdrawal / liquidation, whereas the Controller's state has not yet been updated. It should be carefully considered if this might pose problems for integrators or third-party contracts interacting with the Controller.

6.2 Incorrect Verification of Health Limit

Correctness High Version 1 Code Corrected

CS-CRVUSD-019

The `_liquidate` function checks whether the user's health is below a certain health limit. This health limit is passed as the user's liquidation discount by `liquidate` (and 0 by `self_liquidate`). But the health function already accounts for the user's liquidation discount and is supposed to return a value below 0 when the liquidation can start.

Code corrected

Curve fixed and identified this issue while the audit was ongoing.

6.3 Oracle Price Updates Can Be Sandwiched

Security | High | Version 1 | Code Corrected

CS-CRVUSD-031

The AMM price range in a band (p_cd, p_cu) depends cubically on the oracle price p_o (p_cd = \frac{p_o^3}{p_\uparrow^2}, p_cu = \frac{p_o^3}{p_\downarrow^2}). Since trading can happen out of band, AMM price changes because of changes in p_o are greatly amplified for bands far from the current oracle price. The previous consideration makes it profitable for an attacker to leverage small oracle price increases by accessing the liquidity of low price bands. The attack scenario is like this:

1. Stablecoin is exchanged for collateral, in a large amount such that the active band is shifted toward lower prices bands
2. The oracle price is increased
3. part of the collateral obtained in step 1 is exchanged back at a higher price, recouping the stablecoin and allowing the attacker to keep part of the collateral.

Since after a price update the AMM price will move the most for bands which have a low price compared to the current oracle price (high collateral ratio), overcollateralized borrowers are most affected by this issue. Positions that should be the safest might suffer the most losses from sandwiching, more than supposedly "riskier" positions.

Code corrected

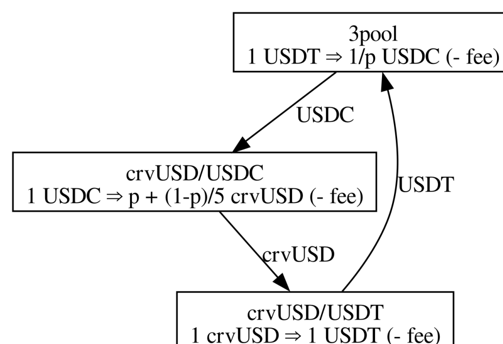
A new dynamic fee has been introduced, such that the fee scales in the same amount as the theoretical profit from sandwiching an oracle update.

6.4 PegKeeper Can Be Drained if Redeemable Stablecoin Permanently Depegs

Security | Medium | Version 5 | Code Corrected

CS-CRVUSD-001

If one of the reference stablecoins depegs, for example USDC falls to p = $0.95, the price in the corresponding StableSwap (crvUSD/USDC) will follow the external market price and also fall to $0.95. The PegKeeper will then try to raise the price, by supplying crvUSD to the StableSwap pool. Essentially the PegKeeper will try to keep USDC from depegging. This opens up the following arbitrage opportunity, where p is the current market price of USDC:



The arbitrage profit depends on the liquidity available in all the pools. If the following (for the purpose of a worst-case analysis) we assume no slippage for the arbitrageur. Assuming all pools have fee f, then the arbitrage becomes profitable if the price p of the depegged stablecoin is:

$$p < -\frac{(f-1)^3}{4f^3 - 12f^2 + 12f + 1}$$

Currently, f = 0.0001 meaning that the arbitrage would become profitable for:

$$p < 0.998502$$

Assuming that the market price of the depegged stablecoin **permanently** falls to p, this arbitrage would happen repeatedly until the PegKeeper has been drained. In this case the PegKeeper would suffer a loss trying to prop up the price of the depegging stablecoin.

Furthermore, the PegKeeper would try to keep crvUSD pegged to a depegging stablecoin, which would put the crvUSD price under pressure, but (assuming reasonably distributed liquidity) should not result in a depegging.

Lastly, please note that as part of the arbitrage crvUSD would accumulate in the crvUSD/USDT pool, but the PegKeeper of crvUSD/USDT pool would not be able to withdraw, due to the

```
assert p_agg <= 10**18
```

check, as `p_agg` would presumably be bigger than `10**18` due to the depegging stablecoin.

If the depegging is only **temporary**, meaning that the price recovers, then the PegKeeper was temporarily drained, but should have made a profit in the process.

Theoretically, this issue could also exist in the opposite direction, with a stablecoin gaining value. However, this seems less likely except for DAI in Maker endgame scenarios.

Code corrected:

In PegKeeperV2 at commit `5a46bb9c1f43b7d4062127b9919e3c2ed366ad34`, which is object of a separate ChainSecurity audit, the pegkeepers for different redeemable stablecoins interact and communicate to each other limits on how much crvUSD can be supplied to a pool. In the case of a single redeemable stablecoin depegging, the pegkeeping action on its pool will be limited.

6.5 Incorrect Max Band

Correctness **Medium** **Version 1** **Code Corrected**

CS-CRVUSD-034

The `AMM` contract tracks the `max_band` variable. Bands above this band are empty. In the `withdraw` function the `max_band` is potentially updated:

```
if self.max_band <= ns[1]:
    self.max_band = max_band
```

If this withdrawal emptied all the touched bands, then this update would set the `max_band` to `0`. This might be incorrect, as other non-empty bands might still exist inbetween.

As the `max_band` variable is used in `calc_swap_out` exchanges on the AMM might work incorrectly because of this.

Code corrected

`max_band` is now set to the last known band with non-empty coins in the withdrawing loop.

6.6 Interest Rate Does Not Compound

Correctness Medium Version 1 Code Corrected

CS-CRVUSD-002

The AMM contract has a function _rate_mul to compute the rate multiplier. The function simply adds the rate multiplied by the time difference to the previous rate multiplier:

```
return self.rate_mul + self.rate * (block.timestamp - self.rate_time)
```

This approach, however, does not account for the compounding of interest over time. Linearly adding interest could lead to significant underestimation of the accrued interest over time.

The code should be modified to include interest compounding in line with common financial practice.

Code corrected:

The calculation was updated in order to compound each time the _rate_mul function is called (though the rate increases linearly over the time periods between these calls):

```
return unsafe_div(self.rate_mul * (10**18 + self.rate * (block.timestamp - self.rate_time)), 10**18)
```

6.7 Manipulation of Active Band

Security Medium Version 1 Code Corrected

CS-CRVUSD-020

It is possible to manipulate the active band. The lower the market liquidity the easier the manipulation is. Multiple other parameters are depending on the active band and, hence, are also manipulated. The manipulation is possible when liquidity which is in a band far above the current band, is reachable through trading. And is done by manipulated deposits, paybacks and trades.

The consequences of this manipulation might be manifold. E.g.:

- Deposits which should still be possible are impossible because they would be below the manipulated active band.
- The health ratio would be affected as it depends on the active band
- It could result in an active band that is more than `1024 + 50` away from the "true" active band. "True" if the external price oracle is assumed to be the truth.

Code corrected

It is now impossible to increase the distance between the active band and the oracle price further than 50 ticks. The active band is otherwise used as a reference point of the AMM, but its value does not affect where loans are created or the value of their health ratio.

6.8 Non-Tradable Funds

Design Medium Version 1 Code Corrected

CS-CRVUSD-026

In case of a very small trade in a band far away from the active band, the funds might be inaccessible through normal trading. It is caused by the new code `in_amount_done == 0` change which fixes the issue Draining funds but blocks the reversal trade now.

Code corrected:

Input amounts are now rounded up.

6.9 Potential Denial of Service (DoS) Attack on Peg Keeper

Security | Medium | Version 1 | Code Corrected

CS-CRVUSD-005

The `PegKeeper` contract contains a `update` function that imposes a delay of 15 minutes between actions.

```
if self.last_change + ACTION_DELAY > block.timestamp:
    return 0
```

This design makes it susceptible to a potential Denial of Service (DoS) attack. A malicious actor could effectively keep the `PegKeeper` occupied by directly rebalancing the stableswap pools, calling `update()`, and then unbalancing the pools again within a single transaction. The `PegKeeper` would be locked for the next 15 minutes, without having provided or withdrawn any amount of stablecoin. This strategy could be performed by an actor seeking to destabilize the peg.

Code corrected:

> PegKeeperV2, included at commit `5a46bb9c1f43b7d4062127b9919e3c2ed366ad34`, which is in the scope of a separate ChainSecurity audit, addresses this issue by preventing a pegkeeper update when the spot price of the underlying pool is in disagreement with the oracle price of the pool by more than 5 basis points.

6.10 A User's Liquidation Discount Can Be Updated by Anyone at Any Time

Security | Low | Version 1 | Code Corrected

CS-CRVUSD-006

The `repay` function allows anyone to repay any loan — even just partially. This function will also update the liquidation discount of the user who has taken the loan to the current liquidation discount. This implies that someone can repay a tiny amount for another user's loan just to change their liquidation discount. In the case where the liquidation discount has increased significantly since the loan was taken, this will be disadvantageous to the borrower. Conversely, borrowers can update their liquidation discounts to their advantage. The liquidation discount can also be updated by adding collateral.

Code corrected:

Only the debt owner can repay in such a way that their position becomes or stays unhealthy. Moreover, the `liquidation_discount` of a position is only updated if the debt owner is the message sender.

6.11 ApplyNewAdmin Event Emitted With Wrong Argument in PegKeeper

Correctness Low Version 1 Code Corrected

CS-CRVUSD-007

The `__init__` constructor function of a `PegKeeper` emits a `ApplyNewAdmin(msg.sender)` event. However `msg.sender` is not necessarily the contract admin, which is specified as the `_admin` constructor argument.

Code corrected:

The `_admin` constructor argument is now emitted in the event.

6.12 Draining Funds

Security Low Version 1 Code Corrected

CS-CRVUSD-016

It is possible to drain 1 WEI per trade from the exchange when a loan is present. Simply by trading back and forth with a very small amount. On Ethereum, the transaction cost should always outweigh the drained WEI.

Code corrected

in amounts are now rounded up.

6.13 Inaccurate `_p_oracle_up(n)` for High/Low Values of `n`

Correctness Low Version 1 Code Corrected

CS-CRVUSD-008

The `AMM` contract implements the `_p_oracle_up` function, which performs numerical computations to determine its return value. The maximum and minimum values of the `power` variable (which is derived from the parameter `n`) are constrained by assert statements.

However, these bounds are too permissive, allowing extreme values of `n` to pass through, leading to potential issues. When the value of `n` is excessively high or low, the output of the `_p_oracle_up` function can result in collisions (identical results for different `n`) or return a value of `0`.

For example, when `n = 4124`, the function returns `0`. It does not revert until `n = 4193`.

The `AMM` expects non-zero prices, and non-overlapping bands. The bounds on the possible input values for `_p_oracle_up` should therefore be narrowed.

Code corrected:

The result of the exponential is asserted to be more than `1000`, corresponding to a maximum value of `n = 3436`.

6.14 Incorrect Array Length

Correctness | Low | Version 1 | Specification Changed

CS-CRVUSD-017

The `Stableswap` contract needs to be initialized with a `_coins` array of length 4. However, only two values are needed (and can be used), as the maximum number of coins is two.

Specification changed

Curve explained this is intentional to keep compatibility with the factory.

6.15 Incorrect Calculations in `health_calculator`

Design | Low | Version 1 | Code Corrected

CS-CRVUSD-018

When calculating the health factor in `Controller.health_calculator`, the collateral value for a non-converted deposit is calculated as follows:

```
collateral = convert(xy[1], int256) + d_collateral
n1 = self._calculate_debt_n1(xy[1], convert(debt, uint256), N)
```

As the function wants to predict the health ratio after the collateral change, `n1` should be calculated with `d_collateral` included and not on the present value `xy[1]`. Later, `p0` is calculated to convert the collateral into stablecoins. But this is only needed if `ns[0] > active_band`. The following code block might be written into the first condition checking `ns[0] > active_band`:

Code corrected

The calculation of `n1` has been corrected.

6.16 Incorrect Comments

Correctness | Low | Version 1 | Code Corrected

CS-CRVUSD-009

The following comments contain inaccuracies:

- The NatSpec of function `withdraw` says: `Withdraw all liquidity for the user.` However, partial withdrawals are also possible.
- The NatSpec of function `_get_dxdy` says that parameter `amount` is an amount of input coin. In fact, `amount` could specify either an input or an output amount, depending on the function parameter `is_in`.

Fixed:

The NatSpec have been edited to reflect the actual behavior of the functions.

6.17 Meaningful Revert Reasons

Design | Low | Version 1 | Specification Changed

CS-CRVUSD-021

Multiple asserts do not throw a revert reason, making it hard to determine where the code failed while debugging. Additionally, many revert messages are quite short. E.g., `assert xy[0] >= min_x, "Sandwich"`. It might be clear to developers but might cause some confusion for anyone else reading the message (e.g., just "Sandwich") as a revert reason. Technically, the error is also not necessarily caused by a sandwich attack.

Specification changed

The "Sandwich" revert message was renamed to "Slippage". Curve explained that the contract is close to the bytecode limit. The chosen revert messages are the trade-off between bytecode limit and meaningful reverts.

6.18 Missing Sanity Checks

Design | Low | Version 1 | Code Corrected

CS-CRVUSD-022

The following functions set important parameters but have no sanity checks for the arguments. Even though some are permissioned and called by a trusted account, sanity checks might prevent accidents. E.g.:

In `ControllerFactory`:

- `__init__`
- `add_market` performs no checks for `debt_ceiling`.
- `set_admin`
- `set_debt_ceiling`

In `AggMonetaryPolicy`:

- `__init__` .. corrected
- `setRate` .. corrected
- `setAdmin`
- `ConstantMonetaryPolicy` has no checks in the setters.

In `CryptoWithStablePrice`

- `__init__` for `ma_exp_time` .. corrected

In `PegKeeper`:

- `__init__` the `_receiver` and `_caller_share`

In the `AMM` contract:

- `__init__`
- `set_rate` might be a problem when no check in the policy was done.
- `create_loan` might fail earlier for no amounts.

In `Stableswap`

- `exchange` could perform checks to fail early.

Some sanity checks might be a trade-off between security and performance.

Code corrected

Some of the missing sanity checks were fixed by Curve independently while the audit was ongoing. We assume the issue raised awareness and the sanity checks were added as intended by Curve.

6.19 Multiple Calls to the AMM

Design | Low | Version 1 | Code Corrected

CS-CRVUSD-023

In `Controller` `repay` and `health_calculator`, there is the following loop:

```
for i in range(MAX_SKIP_TICKS):
    if AMM.bands_x(active_band) != 0:
        break
    active_band -= 1
```

This loop might be executed inside of the AMM contract to avoid an external call in each iteration.

Code corrected

The loop execution was moved from the Controller to the AMM.

6.20 No Events

Design | Low | Version 1 | Code Corrected

CS-CRVUSD-024

The following functions perform an important state change but don't emit an event.

- In `ControllerFactory`: `set_admin`, `set_implementations`, `set_debt_ceiling`, `rug_debt_ceiling`
- `ConstantMonetaryPolicy`: Does not emit any events at all.
- `PegKeeper`: Functions that apply and commit admin, commit and apply new receiver and `set_new_caller_share` .. corrected

---

Code corrected

While the audit was ongoing some events have been added. Without specification, it is unclear which events are intended. We assume the issue raised awareness and all events have been added as intended.

6.21 Non-Indexed Events

Design Low Version 1 Code Corrected

CS-CRVUSD-025

Multiple events allow no filtering for a specific address as they miss indexing. This includes the following examples:

- All events in `ControllerFactory`
- `SetPriceOracle` in `AMM`
- Multiple events in the `AggMonetaryPolicy`
- `AddPricePair` in `AggregateStablePrice`
- Multiple events in `PegKeeper`
- `SetMonetaryPolicy` in `Controller`

Code corrected

Curve indexed multiple events. We assume that after reviewing the events, the current event indexing is the intended indexing as no specification is provided.

6.22 Potential Optimization With Immutable PriceOracle

Design Low Version 1 Code Corrected

CS-CRVUSD-013

The `AMM` contract declares `price_oracle_contract` as a public storage variable. This address is accessed frequently and cannot be replaced in the current implementation.

However, this public declaration results in a storage access each time the `price_oracle_contract` is accessed. Since this is a frequent operation and `price_oracle_contract` cannot be overwritten, typing it as an immutable variable could have significant effects on overall gas usage.

Code corrected:

The `price_oracle_contract` variable is now declared immutable.

6.23 Potentially Incorrect Admin Fees

Design Low Version 1 Code Corrected

CS-CRVUSD-027

In `AMM.exchange` the following check is done before the in and out amounts are transferred:

```
if out_amount_done == 0:
    return 0
```

If the trade does not return any tokens, the function returns 0 but does not revert. Before that point, the state variables `admin_fees_x` and `admin_fees_y` are incremented.

When testing, we could not get the system into the desired state. Therefore, we list this as a more theoretical low-severity issue.

Code corrected

The admin fees are updated after the potential zero return.

6.24 Simpler Calculations Possible

Design | Low | Version 1 | Code Corrected

CS-CRVUSD-014

In the AMM's `get_xy_up` function, some calculations can be simplified to save gas:

1. The calculation for `p_current_mid`:

   ```
   p_current_mid: uint256 = unsafe_div(unsafe_div(p_o**2 / p_o_down * p_o, p_o_down) * Aminus1, A)
   ```

   This is equivalent to the simpler formula:

   $p_{mid} = \frac{{p_o}^3}{p_\downarrow p_\uparrow}$

2. The calculations for `y_o` and `x_o` in the general case:

   ```
   y_o = unsafe_sub(max(self.sqrt_int(unsafe_div(Inv * 10**18, p_o)), g), g)
   x_o = unsafe_sub(max(Inv / (g + y_o), f), f)
   ```

   These equations can be simplified to the following expressions:

   $y_o = Ay_0(1 - \frac{p_\downarrow}{p_o})$

   $x_o = Ay_0 p_o(1 - \frac{p_o}{p_\uparrow})$

Code corrected:

Both suggestions have been implemented.

6.25 Superfluous Check

Design | Low | Version 1 | Code Corrected

CS-CRVUSD-029

Under the assumption that the AMM is always called by the controller, the following checks in `AMM.deposit_range` are not needed because the controller will pass them in ascending order:

```
band: int256 = max(n1, n2)
lower: int256 = min(n1, n2)
```

Code corrected

The checks have been removed as the controller passes sorted values to the AMM.

6.26 Superfluous Interface Definitions

Design | Low | Version 1 | Code Corrected

CS-CRVUSD-028

- In `Stableswap` `Factory.convert_fees`

The following interface definitions were not needed and removed:

- In `Controller` `LLAMMA.get_y_up` is unused.
- In `Stablecoin` the `Controller.admin` interface is unused.
- In `AMM` the ERC20's `balanceOf` function is unused.
- The `AggMonetaryPolicy` and `AggregateStablePrice` contracts implement the `ERC20` interface but do not use it.
- `AggregateStablePrice` does not use the `balances` definition of `Stableswap`
- `PegKeeper` does not use `StableAggregator.stablecoin` and `CurvePool.lp_token` an `ERC20.balanceOf`
- In `Controller` `LLAMMA.get_base_price` and `ERC20.totaSupply`
- In `ControllerFactory` `ERC20.transferFrom`

Code corrected

Curve removed most of the unused definitions.

6.27 Superfluous Variable Assignment for Number of Bands

Design | Low | Version 1 | Code Corrected

CS-CRVUSD-030

In `AMM.deposit_range()` the variable `n_bands` is defined as:

```
i: uint256 = convert(unsafe_sub(band, lower), uint256)
n_bands: uint256 = unsafe_add(i, 1)
```

The variable `dist` is defined as

```
dist: uint256 = convert(unsafe_sub(upper, lower), uint256) + 1
```

and `upper: int256 = band`.

Code corrected

The redundant calculation was removed.

6.28 Unnecessary Subtraction

Design | Low | Version 1 | Code Corrected

CS-CRVUSD-032

In `Controller.__init__`, the variable `Aminus1` is set to `_A - 1`. Later in the code `Aminus1` is not used but recalculated as `_A - 1`.

Code corrected

The calculation is now done once in `__init__` and the variable `Aminus1` is reused in the code.

6.29 Unused Variables

Design | Low | Version 1 | Code Corrected

CS-CRVUSD-033

In `Stableswap` we found `EXP_PRECISION` which is not used in the contract anymore.

Code correct

The unused variable `EXP_PRECISION` has been removed.

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 Dirty Wipe

Note Version 1

The state variable `AMM.user_shares` is never completely cleared. Only the first values are emptied to indicate the user has no shares anymore. The other values are not accessible but remain in storage until they are overwritten. This is more gas efficient if the user wants to deposit again, and we could not find a way to access the outdated values. Still, this might be worth keeping in mind as future code changes might make the values accessible.

7.2 Exchange Does Not Revert if It Did Not Succeed

Note Version 1

When exchanging on an empty LLAMMA or the desired token has no balance, there is no error message for the exchange transaction.

7.3 Liquidate Callback Passes Address of the Liquidated User

Note Version 1

CS-CRVUSD-003

In the Controller's `_liquidate` function, the `execute_callback` function is called with the user set as the address being liquidated, not the liquidator (*msg.sender*). Special care has to be taken by callback contracts to know the initiator of the liquidation.

7.4 Lost Dust Balance on Exchange

Note Version 1

CS-CRVUSD-011

In the presence of dust balances in an AMM band, the `_get_y0()` calculation can return 0. The consequence is that the band content is not traded because `f == 0` if *dumping* and `g == 0` if *pumping*, which causes the exchange code for the band in `calc_swap_in` and `calc_swap_out` to be skipped even if some balance is present. When the `DetailedTrade` struct is inspected in `_exchange()`, it is assumed no amount of out token is left in the bands between the trade start and the last band. This means that the dust balance that was in the bands where `_get_y0() == 0` is forgotten, and its value becomes untransferable.

An example state for `_get_y0()` to equal `0` is `_get_y0(1, 1, int(1000e18), int(1000e18*1.01**1)) == 0`.

As the client states, this doesn't prevent from trading over that band. The small amount of dust lost does not affect the operation otherwise.

7.5 Magic Numbers and Constants

Note **Version 1**

Some "magic numbers" are used in the code. For example, in `ControllerFactory.vy`, the collaterals index is updated as follows:

```
for i in range(1000):
    if self.collaterals_index[token][i] == 0:
        self.collaterals_index[token][i] = 2**128 + N
        break
```

We recommend defining all such numbers as constants with clear names.

Ideally, how these constants are picked should also be described. For example, it was not clear how a `MAX_RATE` of 43959106799 corresponds to 400% APY (as commented), or why `MAX_TICKS = 50` and `MAX_SKIP_TICKS = 1024` are appropriate values.

Also, the number of decimals (`10**18`) is often hardcoded.

7.6 Max Band Over-Estimates the Actual Maximal Band

Note **Version 1**

CS-CRVUSD-012

The `max_band` variable which tracks the maximum band of the `AMM` might not be decreased to the actual maximum band with liquidity when liquidity is withdrawn. `max_band` only provides an upper bound on the bands which could currently hold liquidity, but could overstate it. This has no visible effect except making swaps that exhaust all the available liquidity more gas expensive.

7.7 Min Band Update

Note **Version 1**

The min and max band indicate in which range liquidity is provided. Everything above and below should be empty. In `withdraw` the min and max bands are updated. In case a user who has liquidity in the lowest ticks withdraws their liquidity, the min band is set to the former max band `n[1]` of this user. Hence, min band guarantees that there is no liquidity below it but it's not the lowest band with liquidity.

Similarly, the max band will not be decreased if a single user owns all the liquidity in all of their bands, and `max_band == n[1]`. In this case, `max_band` will not be changed when they withdraw their funds.

7.8 Peg Keeper Assumptions

Note **Version 1**

The Peg Keeper actions will always balance a pool. This implies a constant 1:1 target ratio, assuming that no token loses its peg. Events have shown, however, that stablecoins can lose their peg and even

become quite volatile. It might be beneficial to have additional security mechanisms in place to monitor and pause the actions of a peg keeper.

7.9 Sandwiching Peg Keeper Actions

Note **Version 1**

The Peg Keeper acts on the simple condition of an unbalance pool combined with some sanity checks on the post-price changes. As the pool balances can easily be manipulated with flash loans and the Peg Keeper acts in a deterministic way without slippage protection, this action is prone to be sandwiched in an attack. Yet, we could not think of a scenario that would directly hurt the audited system itself. In all scenarios, the Peg Keeper will balance the pool in the "correct" direction (balancing the pool). This is usually beneficial and not harmful to the system.

Even though the actions of the Peg Keeper should be monitored closely, it might be beneficial to add security mechanisms to pause the Peg Keeper's actions and absolute investment limits instead of relative ones.

7.10 Use of LLAMMA Price

Note **Version 1**

The LLAMMA price is easy to manipulate. The price and the functions `AMM.get_p()` and `Controller.amm_price` (that return the price) should not — or very carefully — be used in any critical operation. Especially, in third party contracts querying this information.