

Produced for



by



Contents

1	Executive Summary	3
2	Assessment Overview	5
3	Limitations and use of report	14
4	Terminology	15
5	Open Findings	16
6	Resolved Findings	27
7	Informational	49
8	Notes	55

1 Executive Summary

Dear Curve team,

Thank you for trusting us to help you 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 stablecoin that is based on different mechanics to keep it stable and manage the loans. The stablecoin's logic is additionally re-used to implement a lending platform where markets can have the stablecoin either has the collateral or the borrowable token.

The most critical subjects covered in our audit are the solvency of the protocol, rounding and numerical precision, and oracles. Security regarding is good, although solvency issues remain because [Bad debt is not socialized](#) in the lending vaults. Security regarding rounding and numerical precisions is good. Some issues were uncovered, see [Low Decimals Tokens May Accumulate No Interest](#) and [Liquidation rounds debt toward 0](#) Few protocols implement fully on-chain oracles. This subject is therefore especially critical. Security of oracles is improvable, see [Oracle Manipulation on L2](#), [Intermediate currency value leakage](#), and [Vault pricePerShare can be manipulated downward](#)

There are still many low severity issues not fixed, 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 general, the unpermissioned nature of factory contracts allows anybody to create lending markets with arbitrary parameters, which could reveal dangers for lenders and borrowers. Curve should communicate this risk clearly to users.

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.

-Severity Findings	1
•	1
-Severity Findings	3
•	3
-Severity Findings	19
•	14
•	1
•	4
-Severity Findings	42
•	25
•	2
•	15

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	59171820b0b41510157778d49335dd3bbf06fcd	Version 2
3	17 February 2023	475ccb7572b93a2b826f10edc2678b4aff0bfc48	Version 3
4	19 April 2023	f7a514ae24f86fc4856401826f8bc6cc207451d1	Version 4
5	7 May 2023	7b1e773877c9e9055b41db320b131626fd98faf2	Version 5
6	1 July 2023	64dc13db563ec6067c75c662ee71a285442ef638	Version 6
7	12 August 2023	5c61cdf2cb2098595ad25cb5f6cc479b3201f4bd	Version 7
8	28 August 2023	b048fc782bd80a868d4ed882b3e6b371b40c1c03	Version 8
9	11 Dec 2023	5a46bb9c1f43b7d4062127b9919e3c2ed366ad34*	PegKeeperV2
10	23 February 2024	528c8d1987170baaa5f8fb51269cf99e6b226db5	Lending Version 1
11	13 March 2024	9e20913fb46db6d3774c56b13ba17d6911cb2caa	Lending Version 2
12	27 August 2024	7f192edba62856d48171991eadcc73a0bce52183	Integrability upgrade
13	27 November 2024	db6fcac9a341b3a612704ae0018a6593bbac04d5	Various fixes
14	22 January 2025	e742e1adfb22f837bf80dfa0fd5a4426f9d484c	Various fixes
15	15 February 2025	16b29c2dfcf725e27808bb0907bfa7c30568628	Various fixes

* This commit is no longer visible in the Curve repository as the branch it was on was force pushed to.

For the vyper smart contracts, depending on the commit reviewed, the compiler version 0.3.7, 0.3.9 and 0.3.10 were chosen.

The following files were in scope:

Version	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Controller.vy	X	X	X	X	X	X	X	X		X	X	X	X	X	X
ControllerFactory.vy	X	X	X	X	X	X	X	X				X	X	X	X
AMM.vy	X	X	X	X	X	X	X	X		X	X	X	X	X	X
Stablecoin.vy	X	X	X	X	X	X	X	X				X	X	X	X
mpolicies/AggMonetaryPolicy.vy	X	X	X	X	X	X	X	X				X	X	X	X
mpolicies/AggMonetaryPolicy2.vy						X	X	X				X	X	X	X
mpolicies/SemilogMonetaryPolicy.vy										X	X	X	X	X	X
price_oracles/AggregateStablePrice.vy	X	X	X	X	X	X	X	X				X	X	X	X
price_oracles/AggregateStablePrice2.vy						X	X	X				X	X	X	X
price_oracles/CryptoWithStablePriceAndChainlinkFraxeth.vy	X	X	X	X	X	X	X	X				X	X	X	X
price_oracles/CryptoFromPool.vy										X	X	X	X	X	X
price_oracles/CryptoFromPoolVault.vy										X	X	X	X	X	X
price_oracles/CryptoFromPoolVault_noncurve.vy										X	X	X	X	X	X
price_oracles/OracleVaultWrapper.vy										X	X	X	X	X	X
stabilizer/PegKeeper.vy	X	X	X	X	X	X	X	X							
stabilizer/PegKeeperV2.vy									X						
stabilizer/PegKeeperRegulator.vy									X						
lending/OneWayLendingFactory.vy										X	X	X	X	X	X
lending/OneWayLendingFactoryL2.vy												X	X	X	X

<code>lending/TwoWayLendingFactory.vy</code>										X	X	X	X	X	X
<code>lending/Vault.vy</code>										X	X	X	X	X	X
<code>BoostedLMCallback.vy</code>												X	X	X	X
<code>flashloan/FlashLender.vy</code>												X	X	X	X

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 () of the contracts as defined in the [Assessment Overview](#).

At the end of this report section we have added subsections for each of the changes accordingly to the versions.

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

The system is also adapted to offer lending markets where users can borrow and lend assets, as long as one of the asset is the stablecoin.

2.2.1 Common Components

2.2.1.1 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.

Borrowers interact with the Controller to borrow through functions `create_loan()`, `create_loan_extended()`, `borrow_more()`, and `borrow_more_extended()`. Borrowers can modify their collateralization ratio through functions `add_collateral()` and `remove_collateral()`. They also can repay their outstanding debt and recover their collateral through functions `repay()` and `repay_extended()`. Finally, arbitrageurs can liquidate positions whose collateral value minus `liquidation_discount` is less than the debt through the functions `liquidate()` and `liquidate_extended()`. The `_extended` version of each method implements a callback which is executed after the funds have been transferred to the caller, but before the caller has transferred funds to the controller.

The Controller has a set of privileged methods that can only be called by its admin. The admin of the Controller is queried from the `admin()` method of its deployer, in the case of the lending, the Vault. The privileged methods that the admin can use are:

- `set_amm_fee()`: sets the minimum fee applied to exchanges in the AMM
- `set_amm_admin_fee()`: sets the share of AMM fee that gets to the `fee_receiver`. Which should not be used in the case of the lending as the vault has no `fee_receiver`.

- `set_monetary_policy()`: sets the monetary policy used in the rate calculation
- `set_borrowing_discounts()`: sets the loan and liquidation discounts. Existing positions are not affected by changes in liquidation discounts.
- `set_callback()`: sets the liquidity mining callback that gets called by the AMM every time a borrower's collateral changes

The Controller keeps track of all individual loans, by saving the initial loan amount and the `rate_mul`, the value of an index at the time of loan creation. This index is global and compounds with the interest rate computed by the monetary policy. Given an initial loan amount and the `rate_mul` index at the time of loan creation, the current debt of a loan is computed as $\text{initial_loan} * \text{current_rate_mul} / \text{initial_rate_mul}$. The controller also keeps track of the total amount of debt of the system, which is accessible through the view method `total_debt()`. The total debt is used in the case of the lending by the Vault to estimate its total assets, and in general by the monetary policy to compute the utilization of the market.

2.2.1.2 AMM (LLAMMA)

The AMM is where the collateral of each position is deposited. As the price of the collateral decreases, it gradually becomes profitable for arbitrageurs to exchange the borrowable token for the collateral, which is sold at a discount w.r.t. to market price, as queried from the price oracle. The AMM internally represents the tokens in 18 decimals precision. Token amounts for tokens with fewer decimals are scaled to 18 decimals precision.

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 borrowable. This kind of soft liquidation ensures that, in the end, the collateral is fully liquidated before the debt gets underwater.

2.2.2 Stablecoin Specific

2.2.2.1 Stablecoin

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

2.2.2.2 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.

2.2.2.3 Oracle contracts

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

2.2.2.4 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.

2.2.2.5 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).

2.2.2.6 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.

2.2.2.7 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.

2.2.3 Lending Specific

The logic of the Curve Stablecoin smart contract suite to implement lending markets. Stablecoin minting is indeed similar to borrowing an asset. In the case of Stablecoin minting, some collateral is given by minters in exchange for stablecoin, in the case of a lending platform, collateral is supplied in exchange for the borrowable asset. Extra logic is required by stablecoin minting to maintain the price peg with the reference asset, so interest rate policies and peg-keeping mechanisms are required. In particular, the following differences exist between the Curve Stablecoin system and Curve Lending: In the stablecoin system, the borrowable asset is fixed to crvUSD, which has 18 decimals. In Curve Lending the borrowable asset is an arbitrary ERC20 token, and the number of decimals is anything between 0 and 18. A further difference is that the price oracle for the stablecoin system prices the collateral in USD (the reference asset for the peg), while in lending, the collateral is priced in terms of the borrowable token. The supply of the Stablecoin is minted, while in Lending the supply comes from users who provide liquidity. The interest rate policy in the Stablecoin system aims to maintain the peg with the reference asset, while in lending the interest rate policy is designed to return market interest rates. Finally, the insolvency risk in the Stablecoin system is carried by the protocol, with the stablecoin potentially depegging in case of bad debt. With Lending, the insolvency risk is carried by lenders, who supply the liquidity of the borrowable tokens.

2.2.3.1 System Architecture

A Curve lending market allows lenders to deposit liquidity in the market (in the form of the borrowable asset). Other users (borrowers) can then receive the borrowable token, after depositing an amount of collateral token of value greater than the loan. The value of the borrowed amount has to be smaller than the value of the collateral, minus a loan discount. When the value of the borrowed amount becomes bigger than the value of the collateral amount minus the "liquidation discount", the loan can be liquidated, which means that it can be repaid by a third party, and the loan collateral (of higher value than the loan) is awarded to the liquidator. Liquidations, however, are a last resort in the system (hard liquidations). In normal operation, the solvency of positions is ensured by "soft liquidations": The collateral is deposited in an Automatic Market Maker (AMM), that gradually exchanges the collateral for the borrowed asset, as the price of the collateral decreases, by selling it at a discount over market price. This prevents the whole position from being liquidated entirely during short-lived price fluctuations.

2.2.3.2 Lending Markets

A lending market consists of a liquidity pool of a borrowable asset, of which amounts can be borrowed by depositing a collateral amount of higher value, that can be recovered after repaying the loan amount consisting of the initial amount borrowed plus the interest generated. Typically, a lending market is created by the *OneWayLendingFactory*, which is a singleton contract that acts as an unpermissioned deployer and a registry for lending markets. Every market is characterized by a single borrowable token and a single collateral token. A market is composed of a *Vault*, that enables users to deposit the borrowable token and become lenders, a *Controller* where borrowers can create, repay, and modify loans, and arbitrageurs can liquidate underwater positions, an *AMM*, where the collateral is deposited, and which gradually sells it at a discount as its price decreases. The price of the collateral, quoted in the borrowable asset, is provided by a *Price Oracle* contract. Finally, the interest rate paid by borrowers is calculated in a *monetary policy* contract.

A slightly more complex type of lending market is created through the *TwoWayLendingFactory*, which creates for tokens A and B a rehypothecating lending market, that is a lending market where the collateral is made available for borrowing and therefore earns interest for the borrower that owns it. Rehypothecating lending markets for tokens A and B are implemented as a pair of lending markets, one where the borrowable is A, and the other where the borrowable is B. The collaterals are respectively the shares of the other market's vault. So the collateral to borrow A is cvB (shares of the B vault), and the collateral to borrow B is cvA.

2.2.3.3 Vault

Vault is a new component of the system designed for the lending part, it implements the ERC-4626 tokenized vault standard for liquidity provision to a lending market. Lenders can deposit the borrowable asset through functions `deposit()` and `mint()`, and receive shares of the vault in exchange. The Vault in turn deposits the borrowable asset to the *Controller*, where it can be borrowed. Shareholders of the vault can `redeem()` or `withdraw()`, to exchange shares for the borrowable asset. The value of a share is computed as the balance in borrowable token of the controller, plus the amount of debt issued by the controller. This latter amount includes interest accumulated by outstanding loans. As the Vault only has access to the amount of borrowable tokens currently in the Controller, the shareholders (lenders) are not guaranteed to be able to withdraw their funds. However, in case of low liquidity, the interest rate will rise to encourage loan repayment (low borrowable balance on Controller, and therefore low withdrawable amount for shareholders). After every change of borrowable asset balance in the vault, the interest rate is for this reason updated in the Controller. The Vault implements the ERC20 standard for its own shares. To mitigate share price inflation attacks, the vault performs calculations by adding 1000 "virtual dead shares" to its total supply of shares. Shares have 18 decimals precision, while the underlying asset (borrowable token) has possibly less precision. The price per share is initialized as 1000 shares per 1 unit of underlying. Vaults have an `admin()` view method, however, they do not have privileged methods accessible by the admin. Their `admin()` method is queried by the Controller to perform access control on its own setters. The `admin()` method returns the admin of the Vault's factory, which is expected to be the DAO.

The Vault acts as the deployer for the Controller and the AMM. After it deploys the AMM, it sets its admin as the Controller.

2.2.3.4 Price Oracle

The price oracle is used by the AMM for price calculation, and by the Controller for loan creation and liquidation. As opposed to the Curve Stablecoin, it returns the price of the collateral in terms of the borrowable token. In the Stablecoin system, it returns the price of the collateral in terms of the peg reference asset (USD). Three implementations of the price oracle use Curve pools `price_oracle()` function as a source of information:

- `CryptoFromPool`
- `CryptoFromPoolVault`
- `CryptoFromPoolVault_noncurve`

For security reasons, only pools of type `TwoCrypto-ng`, `tricrypto-ng`, and `stableswap-ng` are expected to be used for their `price_oracle()`.

2.2.3.5 CryptoFromPool

The contract is an oracle to be used for the AMM of a Vault created by a `OneWayLendingFactory`. It is initialized with a Curve pool which should contain both the borrowable and the collateral token of the vault. Both `price()` and `price_w()` do not perform state changes and return the price of the collateral token in terms of the borrowed token. The price returned is subject to the time-weighted exponential average performed in the Curve pool as the `price_oracle` function is used.

2.2.3.6 CryptoFromPoolVault

`CryptoFromPoolVault` is very similar to `CryptoFromPool` except that it is supposed to be used for Vaults created by the `TwoWayLendingFactory`. The factory creates two vaults respectively for tokens A and B, emitting `cvA` and `cvB`. Given that the respective AMM of each vault contains `A/cvB` and `B/cvA`, instead of using a curve pool containing either `cvA` or `cvB`, `CryptoFromPoolVault` is designed such that a pool containing the two tokens A and B can be used for both AMMs.

This is done by multiplying the result of the price of one underlying token in terms of the other by the `pricePerShare()` of the vault containing the first token. For example, given that we want the price of `cvB` in A, we first get the price of B in terms of A from the pool, and then multiply it by `VaultB.pricePerShare()`.

2.2.3.7 CryptoFromPoolVault_noncurve

`CryptoFromPoolVault_noncurve` can be used to create a Vault where the collateral token is any ERC-4626 vault given that there exists a Curve pool containing both `crvUSD` (the to-be-created vault's borrowable token), and the underlying token of the third-party vault. The price of the third-party shares in terms of `crvUSD` is computed similarly to `CryptoFromPoolVault` by multiplying the price of the third-party vault's underlying token in terms of `crvUSD` by the result of the Vault's `convertToAsset(10**18)`.

2.2.3.8 Monetary Policy

The monetary policy defines the interest rate applied to loans. The interest rates are variable and change at every change of utilization. Utilization is defined as the ratio between the outstanding debt and the liquidity plus debt. The rate in the monetary policy implementation `SemilogMonetaryPolicy` depends on two parameters that can be set by the DAO which are `min_rate` and `max_rate`. A value between `min_rate` and `max_rate` is returned as the current rate, according to the following exponential curve:

$$r = \exp(\log(r_m) + U(\log(r_M) - \log(r_m))) = r_m \left(\frac{r_M}{r_m}\right)^U$$

2.2.3.9 Factories

For Lending Market creation, two types of singleton factory contracts are present in the system: `OneWayLendingFactory` and `TwoWayLendingFactory`. Both are unpermissioned deployers for lending markets and also act as a registry for existing lending markets. Lending Markets created by the factories require that either the collateral or borrowed token is `crvUSD`.

They expose the `create()` and `create_from_pool()` methods. `create_from_pool()` automatically deploys a price oracle based on the Curve pool supplied. Callers of `create()` and `create_from_pool()` have the freedom to choose a wide choice of parameters for newly created lending markets: The `A` parameter of the AMM, the AMM fee, loan and liquidation discounts, interest rates range. For the `create()` method, an arbitrary price oracle address is also supplied. For this reason, deployed lending markets are to be considered untrusted until their deployment parameters have been validated.

The factories have privileged methods that can be called by the admin:

- `set_implementations()`: sets the contract implementations used in newly deployed lending markets.
- `set_default_rates()`: sets the default interest rate parameters when none are specified during lending market deployment.
- `set_admin()`: sets the new admin of the factory.

The factories have `exchange` methods that facilitate interacting with the AMM of the deployed lending markets.

2.2.3.10 TwoWayLendingFactory

`TwoWayLendingFactory` deploys two-way lending markets. These consist of a pair of lending markets, where the borrowable of one is the collateral of the other, and the collaterals are rehypothecating, that is they are borrowable and earn interest. This is achieved by deploying a pair of vaults where the collateral of one vault is the share of the other vault.

2.2.4 Changes in

Notable changes in are:

- `Vault.redeem()` gives the user the total asset of the vault if the shares to redeem are equal to the total supply of shares (excluding the dead shares) and `total_assets - self.convert_to_assets(shares, True, total_assets) < MIN_ASSETS`.
- All helper functions of the `OneWayLendingFactory` used to exchange tokens in one of the Vault's AMM have been removed.
- `CryptoPoolFromVault` and `OracleVaultWrapper` are no longer stateless, instead of calling some Vault's `pricePerShare()` method directly, they now cache it locally to be able to limit its growth.
- A new dynamic fee was introduced in the AMM.

2.2.5 Changes in

The merges both the stablecoin and lending systems and provide minor changes for better integrability.

In the the following two contracts were added and included to the scope:

- `FlashLoanLender`: The contract allows `crvUSD` flash loans. The flash loan amount is capped to the `crvUSD` balance in the contract. The contract is used like a controller with a debt ceiling to limit the amount available to flash loan. Hence, the associated factory has the

approval to mint/burn funds from the `FlashLoanLender` contract. Users can take a flash loan by calling `flashLoan()`.

- `BoostedLMCallback`: This contract integrates in Curve's reward system. It acts as a liquidity gauge to calculate the rewards to send to users who are liquidity provided to the associated AMMs (The stablecoin borrowers). The collateral a user has in an AMM and the locked voting escrow Curve tokens determine the users' reward share. If users are in a soft-liquidation (current trading band in a band a user deposited in), the gauge calculates rewards based on the remaining fractional collateral for the users.
- `OneWayLendingFactoryL2`: A layer 2 `OneWayLendingFactory` implementation with minor adaptations consisting in using existing gauge assumed to be already deployed by the `GaugeFactory` for the given vault, instead of deploying it for each vault.

The following notable changes were made to the one-way and two-way lending protocol:

- Minor changes to the `OneWayLendingFactory` and the Vault. A maximum supply cap was implemented for both contracts.
- Minor changes in the `TwoWayLendingFactory` to account for donated funds.

The remaining notable changes affected the AMM and Controller contracts:

- The AMM contract was adapted to work with the new `BoostedLMCallback` contract and a dynamic fee was introduced.
- Most changes were done in the Controller contract. Reminders of the ETH compatibility are removed. Callback bytes for the extended lending operations were introduced. An approval functionality that allows to perform operations on behalf of another account (if approved) were added and the AMM's admin fee related functionality removed.

2.2.6 Changes in `FlashLoanLender` and `FlashLoanBorrower`

Except for fixes for issues found in the previous versions, no notable changes were made in `FlashLoanLender` and `FlashLoanBorrower`.

2.2.7 Changes in `FlashLoanBorrower`

In `FlashLoanBorrower`, several fixes were implemented, and the following notable changes were made:

- Using the approval functionality, it is now possible for an approved account to perform self liquidations using `liquidate()` or `liquidate_extended()`, in such case, the liquidation is not subject to the liquidation discount.

2.3 Trust Model

- Tokens used as lending and borrowing tokens are expected to be ERC20-compliant tokens with no uncommon behavior like deflation, inflation, callbacks, fee-on-transfer or rebasing.
- All permissioned roles are trusted, in the worst case, the admin of the controller factory could mint an arbitrary amount of stablecoins to any address using `set_debt_ceiling()`.
- 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`.
- For the lending, the deployment of Vault is unpermissioned, anyone could deploy a Vault with malicious parameters (for example a price oracle that can be manipulated by the deployer). We

assume in this review that users ensure the Vaults they are interacting with were created with sensible parameters by honest actors.

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			
Medium			
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 Open 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:

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

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

-Severity Findings	0
--------------------	---

-Severity Findings	0
--------------------	---

-Severity Findings	5
--------------------	---

- [Bad Debt Is Not Socialized](#)
- [Low Decimals Tokens May Accumulate No Interest](#)
- [Oracle Manipulation on L2](#)
- [Vault pricePerShare Can Be Manipulated Downward](#)
- [Manipulable Price Calculation in AggregateStablePrice Method](#)

-Severity Findings	15
--------------------	----

- [Inconsistent max_p_base Implementations](#)
- [Pure Functions Read Immutables](#)
- [FlashLender Does Not Check the Return Value of onFlashLoan\(\)](#)
- [FlashLender Does Not Pull From the Lender](#)
- [Inflation of Vault Share Price Can Result in Unusable Price Oracle](#)
- [Underestimated Fee in get_amount_for_price](#)
- [previewRedeem\(\) Does Not Always Behave the Same as redeem\(\)](#)
- [Calling previewRedeem\(\) Succeed When redeem\(\) Would Revert](#)
- [Intermediate Currency Value Leakage](#)
- [Lending Market Could Reduce Oracle's Pool Liquidity](#)
- [Liquidation Feedback Loop for Illiquid Markets](#)
- [MAX_RATE Constant Too High in SemilogMonetaryPolicy](#)
- [Non-curve Vault Must Always Be the Collateral Token](#)
- [Vault Creation Lacks Sanity Checks](#)
- [CryptoFromPoolVault-noncurve Can Return Incorrect Price](#)

5.1 Bad Debt Is Not Socialized

In case a lending market incurs bad debt, that is a loan is worth less than its collateral, there is no way to incorporate the bad debt in the price of vault shares. The Vault share price will ignore the bad debt, so shares can still be redeemed at full value. However, the lending market will be insolvent toward the holders of the last shares that get redeemed.

Risk accepted:

Client acknowledged the issue and emitted the idea of diverting some protocol fees to serve as POL and as an insurance to liquidate any existing bad debts.

5.2 Low Decimals Tokens May Accumulate No Interest

CS-CRVUSD-038

Tokens with high value per wei (in general tokens with few decimals) might significantly underestimate the amount of interest earned. `Controller` accounts for the total debt in the borrowed token native precision, which might not be sufficient to account for the interest accumulated per block.

The total debt interest accrual is performed as:

```
loan.initial_debt * rate_mul / loan.rate_mul
```

In particular, the interest during one block is:

```
loan.initial_debt * (rate_mul - 10**18) / loan.rate_mul
```

If `loan.initial_debt` has low precision, the previous calculation will realistically round to zero.

Approximately, this happens when the total debt (in wei) is less than $10^{18} / (\text{rate} * B)$, where `rate` is the rate per second, and `B` is the rate update interval (at worst one block). For example, for values of 5% apr, and 1 rate update per block (so every 12 seconds), the maximum amount of debt for which the interest rounds to 0 is 52560000 wei. For high value-per-wei tokens such as WBTC this corresponds to a dollar amount of ~\$36k (for BTC at \$70k), or an unaccounted interest of \$1840 per year. For a token such as Gemini USD, which has 2 decimals, this corresponds to a debt amount of \$525k for which the interest is not accounted, which is \$26k per year.

When loans are repaid, the interest on the individual loans is subject to less precision loss, the unaccounted interest will suddenly appear in the `Controller` balance, this can cause jumps in price per share of the vault, which allows extraction of the profit from the vault by sandwiching the loan repayment.

5.3 Oracle Manipulation on L2

CS-CRVUSD-040

The latest price of Curve Stablesap and CryptoSwap pools has to persist over block boundaries to be incorporated in the EMA price (`price_oracle()`). The only safe way for an attacker to manipulate the price oracle on Ethereum without losing a big share of the manipulation capital to arbitrageurs is to control the blocks after the manipulating trade so that the attacker is guaranteed to be able to do the arbitrage themselves.

However, some L2 blockchains such as Arbitrum rely on a centralized sequencer which publishes transactions on a First-in First-out way. The centralized sequencer will publish transactions on a chain in the same order as they were received, possibly splitting them across block boundaries. Transactions do not enter a public mempool, so an attacker can publish consecutive transactions, doing and undoing a price manipulation, without the risk of losing to arbitrageurs. Since these transactions will sometimes split across block boundaries, and have different timestamps, this gives attackers a low-risk way to manipulate the Curve Pool price oracle.

Risk accepted:

Curve answered:

In principle, this precludes ANY way of creating a decentralized non-manipulatable oracle on L2. But in reality, no one stops a "blind counter-attack" - someone else spamming the network with a "counter-trade" to unwind one side of the sandwich and make a profit (but tx reverting if the pool was not manipulated). These blind attempts are cheap to do (L2s!) but can mean a huge loss for an attacker. So this "blind arbitrage" makes the strategy too risky for attackers if that vector ever becomes the thing.

5.4 Vault pricePerShare Can Be Manipulated Downward

CS-CRVUSD-041

Vault implements the following logic in `pricePerShare()`, to return the initial value when a Vault is not initialized:

```
supply: uint256 = self.totalSupply
if supply == 0:
    return 10**18 / DEAD_SHARES
```

The code above however can also be triggered after some shares were minted, and then burned, turning the total supply back to zero. In that case, the value returned by `pricePerShare()` could be changing from a higher value to $10^{18}/\text{DEAD_SHARES}$, the initial value.

In case the price per share was manipulated upward previously, the new price per share after the Vault has been emptied could be considerably lower. This opens up an attack vector against `TwoWayLendingFactory` that consists of the following:

1. Mint shares in Vault as the first depositor (the attacker needs to control the whole total supply so that it can later turn it back to zero)
2. Considerably raise the price per share, by donating to the Controller
3. Wait until the AMM's `price_oracle` catches up with the inflated price (the price per share used in collateral pricing is now high, so a share has a high value as collateral)
4. Empty the Vault: the price is reset to $10^{18}/\text{DEAD_SHARES}$
5. Mint shares at the low price, use them as collateral for borrowing (the `price_oracle` is delayed and will keep valuing them at the high price).

It is possible to mint shares at the reset price, in step 5, because the last withdrawer takes all the assets (step 4), even those owned by the dead shares:

```
if total_assets - assets_to_redeem < MIN_ASSETS:
    if shares == self.totalSupply:
        # This is the last withdrawal, so we can take everything
        assets_to_redeem = total_assets
    else:
        raise "Need more assets"
```

Performing this attack allows creating undercollateralized loans on freshly deployed two-way lending markets.

Risk accepted:

Client acknowledges this issue.

5.5 Manipulable Price Calculation in AggregateStablePrice Method

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.

Curve added:

Weighting is still manipulable to some extent, however given the frequency of calls it is not practical to manipulate it.

5.6 Inconsistent `max_p_base` Implementations

CS-CRVUSD-070

Except for `LeverageZaplinch`, the `Zap` contracts implement a function to calculate `max_p_base` as well as the `Controller` contract. The `Controller` has been updated to include additional checks for `n1` as well as switching from `log2` to `logn`. These changes have not been reflected on the zaps.

Risk accepted:

Curve acknowledges this issue.

5.7 Pure Functions Read Immutables

CS-CRVUSD-072

Vyper `pure` should not be able to read immutable variables, however, due to a bug in the Vyper compiler (see [issue 3894](#)), this is not enforced. The following functions are marked as `pure` but read immutables:

- `ControllerFactory.stablecoin()`.
- `factory()`, `amm()`, `collateral_token()`, `borrowed_token()` and `get_y_effective()` in `Controller`.
- `AMM.coins()`.
- `factory()`, `pegged()`, `pool()` and `aggregator()` for `PegKeeper`.
- `factory()`, `pegged()` and `pool()` for `PegKeeper2`.
- `ma_exp_time()` and `price_oracle_signature()` for `EmaPriceOracle`.

For `AggregatedStablePrice`, `AggregatedStablePrice2` and `AggregatedStablePrice3`:

- `sigma()`
- `stablecoin()`

For `CryptoWithStablePrice`, `CryptoWithStablePrieAndChainlink` and `CryptoWithStablePrieAndChainlinkFraxeth`:

- `tricrypto()`
- `stableswap_aggregator()`
- `stableswap()`
- `stablecoin()`
- `redeemable()`
- `ma_exp_time()`

Risk accepted:

Curve acknowledged the issue and will update the code when switching to a Vyper version that enforces this rule.

5.8 FlashLender Does Not Check the Return Value of `onFlashLoan()`

CS-CRVUSD-074

In FlashLender, `flashLoan()` calls `receiver.onFlashLoan()` without checking that the returned value is equal to `keccak256("ERC3156FlashBorrower.onFlashLoan")`. [EIP-3156](#) specifies however that:

The lender MUST verify that the `onFlashLoan` callback returns the keccak256 hash of `"ERC3156FlashBorrower.onFlashLoan"`.

Risk accepted:

Curve acknowledged the issue but decided to keep the current implementation as it is since they prefer a "push" architecture over a "pull" architecture.

5.9 FlashLender Does Not Pull From the Lender

CS-CRVUSD-075

In FlashLender, `flashLoan()` assumes that amount has been repayed by the lender during the callback, however [EIP-3156](#) specifies that:

After the callback, the `flashLoan` function MUST take the amount + fee token from the receiver, or revert if this is not successful.

[...]

The amount + fee are pulled from the receiver to allow the lender to implement other features that depend on using `transferFrom`, without having to lock them for the duration of a flash loan. An alternative implementation where the repayment is transferred to the lender is also possible, but would need all other features in the lender to be also based in using `transfer` instead of `transferFrom`. Given the lower complexity and prevalence of a "pull" architecture over a "push" architecture, "pull" was chosen.

Risk accepted:

Curve acknowledged the issue but decided to keep the current implementation as it is since they prefer a "push" architecture over a "pull" architecture.

5.10 Inflation of Vault Share Price Can Result in Unusable Price Oracle

CS-CRVUSD-043

The Vault share price of empty vaults can be inflated by minting shares and then donating to the Controller. The manipulation can make oracles unusable, such as `CryptoFromPoolVault` and `OracleVaultWrapper`, which rely on `Vault.pricePerShare()` but delay their update. For example, depositing `crvUSD 10**9` in an empty Vault will mint `1000*10**9` shares to the depositor. Donating `crvUSD 1000*10**18` (one thousand) to the Controller will inflate the `pricePerShare()` of the Vault by `10**12` (very slightly less because of 1000 wei of `DEAD_SHARES`). Oracles `CryptoFromPoolVault`

and `OracleVaultWrapper` limit the increase of `pricePerShare()` to 1% per minute. This means that the oracles will take about 46 hours to catch up, providing a seriously underestimated price until then.

Risk accepted:

Curve acknowledges the behavior and answered that it is by design:

Sudden change in `pricePerShare` is bad. So this smooth growth is much safer. It is good however that new markets are seeded because in the worst case market is not usable (as opposed to unsafe) for a few days after the manipulation.

5.11 Underestimated Fee in `get_amount_for_price`

CS-CRVUSD-044

In the `AMM.get_amount_for_price()` view function, `get_dynamic_fee()` is not used, while it is used in `calc_swap_in()` and `calc_swap_out()`. This can result in an underestimated fee amount on the part of the view function.

Risk accepted:

Curve answered:

That is in principle true, however that function is not expected to be very precise.

5.12 `previewRedeem()` Does Not Always Behave the Same as `redeem()`

CS-CRVUSD-045

According to EIP-4626, `previewRedeem()`:

MUST return as close to and no more than the exact amount of assets that would be withdrawn in a `redeem` call in the same transaction.

However, In the Vault, when `total_asset - assets_to_redeem < MIN_ASSETS` and `shares == self.totalSupply`, `previewRedeem(shares)` returns `assets_to_redeem` when `redeem(shares)` returns `total_asset`.

Risk accepted:

Curve acknowledges the behavior.

5.13 Calling `previewRedeem()` Succeed When `redeem()` Would Revert

CS-CRVUSD-046

According to EIP-4626, the `preview[...]()` functions allow an on-chain or off-chain user to simulate the effects of their action at the current block, given current on-chain conditions. However, none of the preview functions of the `Vault` take into consideration `MIN_ASSETS`. For example, given that the `vault_total_assets()` is 0, calling `previewDeposit(MIN_ASSETS-1)` will return some amount of shares while `deposit(MIN_ASSETS-1)` will revert.

Risk accepted:

Curve acknowledges the behavior.

5.14 Intermediate Currency Value Leakage

CS-CRVUSD-048

In the `CryptoFromPool`, `CryptoFromPoolVault` and `CryptoFromPoolVault_noncurve` price oracles, to estimate the collateral price in borrowable token, an independent intermediate currency could enter the calculation: for example if borrowable is coin 1, and collateral is coin 2 of the pool, the intermediate prices will be quoted in term of coin 0 in the $p_collateral * 10^{18} / p_borrowed$ calculation. The final result should not depend on the market movements of coin 0, however, since we are dealing with EMA values, which are arithmetic averages, dividing or multiplying doesn't cancel the intermediate terms, and the value of the intermediate currency can leak in the result.

Risk accepted:

Curve acknowledges the issue.

5.15 Lending Market Could Reduce Oracle's Pool Liquidity

CS-CRVUSD-049

The deployment of a lending market, with attractive interest rates, could cause the yield for lenders of the Lending Market to be higher than the yield for Liquidity Providers of the Pool which is used as a price oracle by the lending market. The consequence is that the lending market oracle would gradually become more volatile and manipulable as the TVL of the lending market increases, as well as liquidations causing more price impact.

As of _____, the vaults have a configurable `max_supply` that can cap the amount of liquidity that can be provided. This feature could be used to limit the effects described above.

Risk accepted:



Curve answered:

This is a correct observation, something to watch in general as crvUSD grows, and not related to lending markets.

5.16 Liquidation Feedback Loop for Illiquid Markets

CS-CRVUSD-050

Curve Stablecoin allows the creation lending markets based on Curve pools as the price source. However, some Curve Pools are the only source of liquidity for a given asset. In that case, liquidations happening in the lending market will push the price of the collateral in the pool down, without external markets to arbitrage it back up. This can cause a self-reinforcing liquidation spiral.

Risk accepted:

Curve acknowledged the issue and answered:

This already did happen indeed. So if makets on L2s are deployed - it's better to use oracles taking prices from more liquid places. For assets which are illiquid globally - better to use caps, and this was one of the main reasons why caps were introduced.

5.17 MAX_RATE Constant Too High in SemilogMonetaryPolicy

CS-CRVUSD-052

The `MAX_RATE` bound of `SemilogMonetaryPolicy` is set to $10^{19} / (365 * 86400)$, the comment says that this corresponds to 1000% interest per year. However, taking compounding into account, this corresponds to 2202643% per year. The same bound in the `Controller` is set more appropriately.

Risk accepted:

Curve acknowledges the issue and agrees that the `Controller` limits guard against this high rate.

5.18 Non-curve Vault Must Always Be the Collateral Token

CS-CRVUSD-053

Given `CryptoFromPoolVault_noncurve, _raw_price()`'s implementation:

```
p_collateral * VAULT.convertToAssets(10**18) / p_borrowed
```


The collateral of the to-be-created lending market must be the shares of the non-curve vault, if this is not the case and the shares of non-curve vault are the borrowable token of the lending market, the oracle will return incorrect prices. `CryptoFromPoolVault_noncurve` does not enforce this.

Risk accepted:

Curve answered:

The noncurve oracle is a prototype to use as a code example rather than something to directly use in prod directly.

5.19 Vault Creation Lacks Sanity Checks

CS-CRVUSD-054

In both the `OneWayLendingFactory` and the `TwoWayLendingFactory`, when creating vault(s) (using `create()` and `create_from_pool()`), the following sanity checks are missing:

- no sanity check is performed on the name.
- The price oracle given when using `create()` is not checked to be a valid price oracle for the given tokens.
- `create_from_pool()` requires that the pool used is `tricrypto-ng`, `twocrypto-ng`, or `stableswap-ng` but no validation is performed.

Since `price_oracle()`, when creating vault(s) using an existing Curve pool as a price oracle, the existence of a price oracle in the Curve pool is not enforced. It will simply be assumed that the pool `price_oracle()` function does not take any argument:

```
no_argument: bool = False
if N == 2:
    success: bool = False
    res: Bytes[32] = empty(Bytes[32])
    success, res = raw_call(
        pool.address,
        _abi_encode(empty(uint256), method_id("price_oracle(uint256)")),
        max_outsize=32, is_static_call=True, revert_on_failure=False)
    if not success:
        no_argument = True
NO_ARGUMENT = no_argument
```

Risk accepted:

Curve answered:

It is possible to do very shallow validation, but not extremely deep one. So it is anyway needed to check the markets before voting for them.

5.20 CryptoFromPoolVault-noncurve Can Return Incorrect Price

CS-CRVUSD-055

In `CryptoFromPoolVault_noncurve`, `_raw_price()` returns the following:

```
p_collateral * VAULT.convertToAssets(10**18) / p_borrowed
```

As `VAULT.convertToAssets(10**18)` is expected to return a price in 18 decimals, if the VAULT's underlying token and share token do not have the same number of decimals, the price will be incorrect as it would have a different amount of decimal.

That is because the price in 18 decimals of a share in assets is defined as:

```
VAULT.convertToAsset(10 ** VAULT.decimals()) * 10 ** (18 - VAULT.asset().decimals())
```

which, assuming that the following holds for VAULT:

```
VAULT.convertToAssets(x) * 10 ** y == VAULT.convertToAsset(x * 10 ** y)
```

can be reduced to:

```
VAULT.convertToAssets(10 ** (18 + VAULT.decimals() - VAULT.asset().decimals()))
```

Risk accepted:

Curve answered:

The noncurve oracle is a prototype to use as a code example rather than something to directly use in prod directly.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the [Open Findings](#) section.

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

-Severity Findings	1
--------------------	---

- [FlashLender Can Be Drained](#)

-Severity Findings	3
--------------------	---

- [Checks-effects-interactions Pattern and Reentrancy Locks](#)
- [Incorrect Verification of Health Limit](#)
- [Oracle Price Updates Can Be Sandwiched](#)

-Severity Findings	14
--------------------	----

- [BoostedLMCallback Can Be Reinitialized](#)
- [Unsafe Approvals](#)
- [Incorrect Conversion to Shares in exchange_dy\(\)](#)
- [Incorrect Receiver in exchange_dy\(\)](#)
- [Incorrect View Functions](#)
- [Monetary Policy Incorrectly Shared by the Vaults in TwoWayLendingFactory](#)
- [Surplus of dx Not Refunded When Using Factory Exchange Functions](#)
- [transfer_in\(\) Transfers in Incorrect Token](#)
- [PegKeeper Can Be Drained if Redeemable Stablecoin Permanently Depegs](#)
- [Incorrect Max Band](#)
- [Interest Rate Does Not Compound](#)
- [Manipulation of Active Band](#)
- [Non-Tradable Funds](#)
- [Potential Denial of Service \(DoS\) Attack on Peg Keeper](#)

-Severity Findings	27
--------------------	----

- [Inconsistent Access Control](#)
- [Inconsistent MIN_TICKS_UNIT Check](#)
- [Incorrect NatSpec](#)
- [BoostedLMCallback Is Not Compatible With Lending Factories](#)
- [Extra Wei Can Be Maliciously Credited to Borrower Every Block](#)
- [Liquidation Rounds Debt Toward 0](#)
- [Pool's Price Oracle Check Is Too Restrictive](#)
- [A User's Liquidation Discount Can Be Updated by Anyone at Any Time](#)
- [ApplyNewAdmin Event Emitted With Wrong Argument in PegKeeper](#)
- [Draining Funds](#)

- Inaccurate `_p_oracle_up(n)` for High/Low Values of `n`
- Incorrect Array Length
- Incorrect Calculations in `health_calculator`
- Incorrect Comments
- Meaningful Revert Reasons
- Missing Sanity Checks
- Multiple Calls to the AMM
- No Events
- Non-Indexed Events
- Potential Optimization With Immutable `PriceOracle`
- Potentially Incorrect Admin Fees
- Simpler Calculations Possible
- Superfluous Check
- Superfluous Interface Definitions
- Superfluous Variable Assignment for Number of Bands
- Unnecessary Subtraction
- Unused Variable in `Stableswap`

Informational Findings

1

- Condition for Fetching New Rate Is Always True

6.1 FlashLender Can Be Drained

CS-CRVUSD-065

To ensure that a loan has been repaid, the `FlashLender` checks after the callback that the new balance of the contract is equal to the balance before loaning the `crvUSD`. In case there is an active governance proposal to increase the debt ceiling of the `FlashLender`, this check can be abused by executing the proposal instead of repaying the loan.

In the following, we assume the current debt ceiling of the `FlashLender` is 1M `crvUSD` and that a governance proposal has been voted to increase the debt ceiling of the `FlashLender` to 2M `crvUSD`, but the proposal has not yet been executed on-chain.

1. A malicious actor calls `FlashLender.flashLoan()` to borrow 1M `crvUSD`.
2. As part of the callback `receiver.onFlashLoan()`, the receiver then calls the Voting Ownership's `executeVote()` to execute the action which will eventually call the `ControllerFactory.set_debt_ceiling()` function, that will mint an additional 1M `crvUSD` to the `FlashLender` contract.
3. The malicious actor then returns from the `onFlashLoan()` callback without repaying the 1M `crvUSD` borrowed, and the `FlashLender` contract will not revert as its current balance is 1M `crvUSD` due to the increase of the debt ceiling.

Code corrected:



The post-callback check has been updated to:

```
assert ERC20(CRVUSD).balanceOf(self) >= FACTORY.debt_ceiling_residual(self), "FlashLender: Repay failed"
```

Where `FACTORY.debt_ceiling_residual()` returns the debt of the `FlashLender` in `crvUSD`.

This means that if the borrower did not repay the entire loan and:

- Called `set_debt_ceiling()` in the callback to increase the debt ceiling by `diff`, the check will fail as the balance of the `FlashLender` will be increased by `diff`, but so would be the `debt_ceiling_residual`.
- Called `set_debt_ceiling()` or `rug_debt_ceiling()` in the callback to decrease the debt ceiling, the check will fail as:
 - Either the `FlashLender` had some `crvUSD` at the moment of the call and the call burned `diff` tokens, which is reflected in `debt_ceiling_residual`.
 - Or the `FlashLender` had no `crvUSD` at the moment of the call and the call do not perform any action.

6.2 Checks-effects-interactions Pattern and Reentrancy Locks

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`.

6.3 Incorrect Verification of Health Limit

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.

6.4 Oracle Price Updates Can Be Sandwiched

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.

6.5 BoostedLMCallback Can Be Reinitialized

CS-CRVUSD-066

In `BoostedLMCallback`, the `initialize` function can be called to set the initial values of the `amm`, `inflation_rate` and `future_epoch_time` variables. Given that the function is permissionless and implements no logic to prevent reinitialization, it is possible for an attacker to call the function again later.

This could lead to unexpected behavior when the last call to `_checkpoint_collateral_shares` was made in a previous epoch `X` compared to the current block timestamp which belongs to epoch `X+1`.

- By setting `future_epoch_time` to `CRV.future_epoch_time_write()`, the next call to `_checkpoint_collateral_shares()` will not enter the following branch as it should have to account for the old rate for the remaining of the epoch `X` that was not yet accounted for.

```
if prev_future_epoch >= prev_week_time and prev_future_epoch < week_time:
    # If we went across one or multiple epochs, apply the rate
    # of the first epoch until it ends, and then the rate of
    # the last epoch.
    # If more than one epoch is crossed - the gauge gets less,
    # but that'd mean it wasn't called for more than 1 year
    delta_rpc += rate * w * (prev_future_epoch - prev_week_time) / working_supply
    rate = new_rate
    delta_rpc += rate * w * (week_time - prev_future_epoch) / working_supply
```

- By setting `inflation_rate` to `CRV.rate()`, even if the first behavior described above was not an issue, the rate to use for the remaining of epoch `X` (the "old" rate) would be the current rate instead of the rate at epoch `X`.

Code corrected:

The contract was refactored to remove the `initialize` function and performs all the necessary setup in the constructor.

6.6 Unsafe Approvals

CS-CRVUSD-067

The system often call `approve()` on ERC20 tokens by using the interface `approve(_spender: address, _value: uint256) -> bool: nonpayable` without using `default_return_value`. In such cases, the execution will revert for token contracts that do not respect the ERC20 standard such as USDT as there will not be enough return data to decode given that it does not return a boolean.

Bellow is a list of such occurrences when the token is not known to be ERC20 compliant:

- `Controller.__init__()`:

```
_borrowed_token.approve(msg.sender, max_value(uint256))
```

- `TwoWayLendingFactory._create()`:

```
ERC20(borrowed_token).approve(amm, max_value(uint256))
ERC20(collateral_token).approve(vault_short.address, max_value(uint256))

...

ERC20(borrowed_token).approve(vault_long.address, max_value(uint256))
ERC20(collateral_token).approve(amm, max_value(uint256))
```

- `LeverageZaplinch._approve()`:

```
ERC20(coin).approve(spender, max_value(uint256))
```

Code corrected:

The code was corrected by using the `default_return_value` kwarg for all mentioned `approve()` calls.

6.7 Incorrect Conversion to Shares in `exchange_dy()`

CS-CRVUSD-036

In the `TwoWayLendingFactory`, `exchange_dy()` converts the given amount to shares when `i == 1` using `other_vault.convertToShares(amount)`, but `amount` represents the output token, which is `j == 0`, so the borrowed token. The conversion should happen when `i==0` instead, when `amount` represents an amount of the other vault's underlying token.

6.8 Incorrect Receiver in `exchange_dy()`

CS-CRVUSD-037

In `TwoWayLendingFactory.exchange_dy()`, when `j==1`, `_receiver` is set to `msg.sender`. This means that the shares of the other vault obtained when exchanging the borrowed token will be directly sent to the message sender, while they should instead first be redeemed before being sent to the receiver, which is not necessarily the message sender.

6.9 Incorrect View Functions

CS-CRVUSD-063

The following view functions of the `Vault` are incorrectly implemented:

- `maxDeposit()` which returns `self.balanceOf[receiver]` when the function should return some amount of asset and not shares.
- `maxMint()` which passes `self.balanceOf[receiver]`, an amount of shares to `self._convert_to_shares()` which takes an amount of asset as argument.

Code corrected:

Both functions now return `max_value(uint256)`.

6.10 Monetary Policy Incorrectly Shared by the Vaults in TwoWayLendingFactory

CS-CRVUSD-039

In TwoWayLendingFactory, a single monetary policy is deployed in `_create()`, and used to initialize both `vault_long` and `vault_short`:

```
monetary_policy: address = create_from_blueprint(  
    self.monetary_policy_impl, borrowed_token, min_rate, max_rate, code_offset=3)
```

Since `SemilogMonetaryPolicy` takes `borrowed_token` as argument to calculate the interest rate, it will perform correctly for `vault_long`, which lends the `borrowed_token`, but not for `vault_short`, which lends `collateral_token`.

Code corrected:

Two monetary policies are now deployed in `_create()`, one for each vault.

6.11 Surplus of dx Not Refunded When Using Factory Exchange Functions

CS-CRVUSD-056

When calling `OneWayLendingFactory.exchange()`, or `exchange()` and `exchange_dy()` of the `TwoWayLendingFactory` with `i==0` and `j==1`, if the full amount dx passed to the AMM is not used, the surplus is not refunded to the user. `amm.exchange()` can use less than the amount supplied as argument if not enough liquidity is available.

Code corrected:

1. The function `exchange()` was removed from the `OneWayLendingFactory`.
2. The function `exchange()` and `exchange_dy` in the `TwoWayLendingFactory` were updated to refund the surplus of dx to the user.

6.12 `transfer_in()` Transfers in Incorrect Token

CS-CRVUSD-042

`TwoWayLendingFactory` defines the function `transfer_in` that is used by `exchange()` and `exchange_dy()` to transfer the tokens to be exchanged from the caller to the contract. In the case that `i==1`, the token being transferred in is `vault.collateral_token()` (the share of the given vault) instead of the underlying token of the other vault. This implies that the call to

`other_vault.deposit(amount)` will most likely revert as the factory does not have any underlying token to be deposited into the other vault.

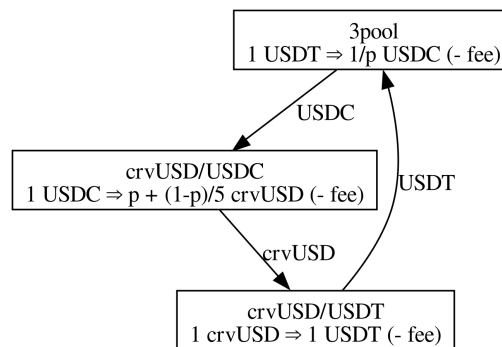
Code corrected:

The correct token is now transferred in when `i == 1`.

6.13 PegKeeper Can Be Drained if Redeemable Stablecoin Permanently Depegs

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.

6.14 Incorrect Max Band

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.

6.15 Interest Rate Does Not Compound

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)
```

6.16 Manipulation of Active Band

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.

6.17 Non-Tradable Funds

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.

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

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.

6.19 Inconsistent Access Control

CS-CRVUSD-068

The functions `user_checkpoint` and `claimable_tokens` in `BoostedLMCallback` do almost the same action except for the return value and the access restrictions. `claimable_tokens` is unrestricted whereas `user_checkpoint` can only be called by the provided address or by the minter. this behavior is inconsistent given they perform the same state transition.

Furthermore, it should be noted that `claimable_tokens` can be used to "kick" a user by updating the user's boost and checkpoint them.

Code corrected:

Checks in `user_checkpoint` were removed, and the function was made unrestricted.

6.20 Inconsistent `MIN_TICKS_UINT` Check

CS-CRVUSD-069

In the Controller contract, `max_borrowable()` checks that $N \geq \text{MIN_TICKS_UINT}$ and $N \leq \text{MAX_TICKS_UINT}$ but `min_collateral()` only checks $N \leq \text{MAX_TICKS_UINT}$. As a lower N usually implies less collateral, checking the `MIN_TICKS_UINT` seems important in that context.

Code corrected:

The check for `MIN_TICKS_UINT` has been added to the `min_collateral()` function.

6.21 Incorrect NatSpec

Multiple NatSpec comments across the codebase are incorrect, missing or duplicated.

Additionally, the following NatSpec issues lead to an inconstant behavior of the compiler depending of the compilation mode used (see Vyper [Issue 3911](#)), which could lead to issues for example when trying to verify the contract on different platforms as they could use a different compilation mode than the one used by Curve.

- `Controller.repay():_for` is documented twice.
- `BoostedLMCallback.__init__():` all parameter are appended with `:`.
- `BoostedLMCallback._user_amounts():` user and `@return` are appended with `:`.
- `BoostedLMCallback._checkpoint_user_shares():` user is appended with `:`.
- `BoostedLMCallback.user_collateral():` user and `@return` are appended with `:`.
- `BoostedLMCallback.working_collateral():` user and `@return` are appended with `:`.
- `BoostedLMCallback.callback_user_shares():` user is appended with `:`.

Code corrected:

While several NatSpec comments are still missing, all NatSpec issues leading to an inconsistent behavior of the compiler have been fixed.

6.22 BoostedLMCallback Is Not Compatible With Lending Factories

CS-CRVUSD-073

The `BoostedLMCallback` is documented as being compatible with `LlamaLend`, however, in the `initialize()` function, the contract calls `FACTORY.get_amm()`, which does not exist in the lending factories as opposed to the `ControllerFactory`.

Code corrected:

The `BoostedLMCallback` now takes the AMM as a parameter in the constructor, and the `initialize()` function has been removed.

6.23 Extra Wei Can Be Maliciously Credited to Borrower Every Block

CS-CRVUSD-047

The `_debt()` internal function will round up the debt amount of a borrower if the debt has not been updated that same timestamp.

```
# Use ceil div
debt: uint256 = loan.initial_debt * rate_mul
```

```
if debt % loan.rate_mul > 0: # if only one loan -> don't have to do it
    if self.n_loans > 1:
        debt += loan.rate_mul
    debt /= loan.rate_mul
```

An attacker can potentially increment the borrower debt by 1 wei every block, by calling `add_collateral()` with a 1 wei collateral amount. This can be a problem if:

- 1 wei of collateral is much less valuable than 1 wei of borrowable.
- The network fees are low (such as on L2s).
- The block time is frequent.
- 1 wei of borrowable token has a high value.

For example, \$22k ($365 * 86400 * 1e-8 * \$70e3$) of extra debt per year can be credited to a borrower of WBTC on Arbitrum (at BTC price \$70K).

Code corrected:

The issue was addressed with the following assertion:

```
assert d_collateral * AMM.price_oracle() > 2 * 10**18 * BORROWED_PRECISION / COLLATERAL_PRECISION
```

which ensures that the worth of the added collateral is at least twice that of the wei of extra debt caused by rounding.

6.24 Liquidation Rounds Debt Toward 0

CS-CRVUSD-051

In the function `_liquidate` of the Controller, the debt of the loan is updated as follow:

```
final_debt: uint256 = debt
debt = unsafe_div(debt * frac, 10**18)
assert debt > 0
final_debt = unsafe_sub(final_debt, debt)
```

The `unsafe_div` will always round toward 0.

On the other side, the amounts of borrowed and collateral tokens to be withdrawn from the AMM contract are obtained as follows:

```
xy: uint256[2] = AMM.withdraw(user, self._get_f_remove(frac, health_limit))
```

In the case the computation of `debt` rounds down while the computation of `xy` does not round down as much, the liquidator might be able to withdraw some amount of collateral that will leave the loan with bad debt as the debt being repaid is insufficient.

This situation can be made more profitable for the liquidator in the case the borrowable token has a high value-per-wei as the 1 wei that is not repaid is worth a lot.

Code corrected:



The division is now rounded up.

6.25 Pool's Price Oracle Check Is Too Restrictive

CS-CRVUSD-064

In both the `OneWayLendingFactory` and the `TwoWayLendingFactory`, when creating lending markets using an existing oracle Curve pool as a price oracle, the following check is performed:

```
if N == 2:
    assert Pool(pool).price_oracle() > 0, "Pool has no oracle"
else:
    assert Pool(pool).price_oracle(0) > 0, "Pool has no oracle"
```

This check is used to ensure that the given pool has a price oracle, however, it assumes that pools with 2 tokens always have a price oracle function with the signature `price_oracle()`. This is not always the case as stableswap-ng pools have the signature `price_oracle(uint256)` for the price oracle, even when the pool has 2 tokens.

Code corrected:

The check has been removed in .

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

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.

6.27 ApplyNewAdmin Event Emitted With Wrong Argument in PegKeeper

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.

6.28 Draining Funds

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.

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

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`.

6.30 Incorrect Array Length

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.

6.31 Incorrect Calculations in `health_calculator`

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.

6.32 Incorrect Comments

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.

6.33 Meaningful Revert Reasons



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.

6.34 Missing Sanity Checks

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.

6.35 Multiple Calls to the AMM

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.

6.36 No Events

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.

6.37 Non-Indexed Events

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`

As of `0.10.0`, the `UpdateLiquidityLimit` event in `BoostedLMCallback` does not index any field.

Code corrected:

Curve indexed all relevant events.

6.38 Potential Optimization With Immutable PriceOracle

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.

6.39 Potentially Incorrect Admin Fees

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.

6.40 Simpler Calculations Possible

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:

$$p_{mid} = \frac{p_o^3}{p}$$

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:

$$y_o = Ay_o(1 - \frac{p}{p_o})$$

$$x_o = Ay_o p_o(1 - \frac{p_o}{p})$$

Code corrected:

Both suggestions have been implemented.

6.41 Superfluous Check

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.

6.42 Superfluous Interface Definitions

CS-CRVUSD-028

- In `Stableswap Factory.convert_fees`

The following interface definitions are not needed and could be 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`
- In `BoostedLMCallback` the `GaugeController` interface's defines `period_write`, `period_timestamp` and `voting_escrow` which are not used.
- In `BoostedLMCallback`, `Minter.token()` and `Minter.controller()` are unused.
- In Controller, `LLAMA.admin_fees_x()`, `LLAMA.admin_fees_y()`, `LLAMA.reset_admin_fees()` and `LLAMA.set_admin_fee()` are unused.

Code corrected:

Curve removed all unused definitions except for the ones reported as of .

6.43 Superfluous Variable Assignment for Number of Bands

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.

6.44 Unnecessary Subtraction

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.

6.45 Unused Variable in Stableswap

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.

6.46 Condition for Fetching New Rate Is Always True

CS-CRVUSD-087

The function `_checkpoint_collateral_shares()` fetches the new future epoch and current rate from the CRV token contract when the following condition is met: `prev_future_epoch >= I_rpc.t`.

This condition, under normal circumstances (the gauge is called at least once a year), will always be true. `I_rpc.t` correspond to the last time `_checkpoint_collateral_shares()` was called, and `prev_future_epoch` to the future epoch timestamp, at the time of the last call: `I_rpc.t`. Since by definition of the function `future_epoch_time_write()` the future epoch timestamp is always greater than the current time, the `prev_future_epoch` cached in the inflation params at time `I_rpc.t` will always be greater than or equal to `I_rpc.t`.

Code corrected:

The condition was changed to `block.timestamp >= prev_future_epoch`.

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 Receiver Can Receive Dx When Exchanging in TwoWayLendingFactory

CS-CRVUSD-086

In the `TwoWayLendingFactory`, when calling `exchange()` or `exchange_dy()`, it could be that the entire amount of input token provided by the caller is not consumed. In this case, the leftover tokens are sent to the receiver and not back to the caller. If the receiver and the caller are different, this could be an issue as the caller might expect to receive the leftover tokens, or the receiver could not be able to handle them.

7.2 Events Lack Caller Information

CS-CRVUSD-076

Although it is now possible to interact with the controller for someone else's loan, the different events do only log the touched loan's owner, but not the address that made the call, this includes:

- `Repay()`
- `Borrow()`
- `RemoveCollateral()`

7.3 Inconsistent Bypass of Controller Approval

CS-CRVUSD-077

The `Controller` allow loan owner to give approval to another address to interact with their loan. Additionally, only in the case of `create_loan()` or `create_loan_extended()`, an address without approval may interact with the loan only if `tx.origin` is the loan owner. Other functionalities require the caller to have approval.

7.4 Misleading `set_implementations()`

CS-CRVUSD-078

In `OneWayLendingFactoryL2`, the function `set_implementations()` is documented as allowing to set new implementations (blueprints) for various contracts. However, it can also set the `gauge_factory`, which is not a blueprint.

7.5 Reused Callback Signature

CS-CRVUSD-079

The different `_extended()` functions of the controller can call a callback function in another contract. The signature is specific to the functionality of the controller being used (for example `CALLBACK_REPAY` for `repay_extended()`). This is not the case for `borrow_more_extended()` which reuses `CALLBACK_DEPOSIT` and `CALLBACK_DEPOSIT_WITH_BYTES` already used by `create_loan_extended()`.

7.6 Trust Assumption of the Controller Approval Mechanism

CS-CRVUSD-080

The Controller allows for loan holder to give approval to addresses to allow them to interact with their loan. It should be noted that such approved address can extract value from the loan holder, and should be hence fully trusted.

For example, a malicious approved address could use `borrow_more_extended()` to max up the loan if it was not yet the case, without providing more collateral, and get the additional borrowed tokens as they control the `callback` address.

7.7 Unreachable Functions in AMM

CS-CRVUSD-081

In , the logic to set and collect the AMM's admin fee has been removed from the Controller. This means that `AMM.set_admin_fee()` and `AMM.reset_admin_fees()` are no longer reachable and could be removed as the contract's admin cannot call them.

In the case the `admin_fee` will always be set to 0 at deployment of the AMM, the entire admin fee logic could be removed from the contract. If `admin_fee` is not set to 0 at deployment, the admin fee will be locked in the contract and cannot be retrieved.

7.8 Vyper Loops

CS-CRVUSD-082

As of , the codebase uses Vyper 0.3.10, this version of the language provides loops in the form of:

```
for i: uint256 in range(stop, bound=N):
    ...

for i: uint256 in range(start, end, bound=N):
    ...
```

which improves readability compared to the following pattern used across the codebase:

```
for i in range(N):
    if i == stop:
        break
    ...
```

7.9 BoostedLMCallback Cannot Be Killed

CS-CRVUSD-083

As opposed to regular curve gauges, the BoostedLMCallback contract cannot be killed.

7.10 gauge_for_vault() Does Not Check That the Vault Is From the Factory

CS-CRVUSD-084

In OneWayLendingFactoryL2, the function `gauge_for_vault()` return the gauge for a vault if it exists, but there is no check that the provided vault has been deployed by the factory, and it could return a gauge for a vault that was not deployed by that factory.

7.11 Admin Fee Can Be Set but Not Recovered

CS-CRVUSD-057

In the Controller for a lending market, the admin (the DAO) can potentially call `set_amm_admin_fee()`. The admin fee accumulated by the AMM will however not be recoverable, as `Controller.collect_fees()` will revert since `FACTORY.fee_receiver()` reverts for lending markets, because `FACTORY` is a Vault that doesn't implement `fee_receiver()`.

7.12 Gas Savings

CS-CRVUSD-058

- In `TwoWayLendingFactory.transfer_out()`, `other_vault.redeem()` is called even if the shares to redeem are 0. The external call could be avoided in such cases.
- In both `TwoWayLendingFactory._create()` and `OneWayLendingFactory._create()`, `min_default_borrow_rate` and `max_default_borrow_rate` are loaded from storage even in the case they are overridden by `min_borrow_rate` and `max_borrow_rate`.
- In `Controller._debt()`, `self.n_loans` is loaded from storage, costing 2300 gas on every call. `n_loans` will be bigger than 1 in almost every active market. Performing the rounding-up regardless of `n_loans` can bring considerable gas savings.

As of :

- In `BoostedLMCallback._update_liquidity_limit()`, `self.user_boost[user]` is read twice from storage.
- In `BoostedLMCallback._checkpoint_collateral_shares()`, `prev_week_time` is rounded down to the start of the week before being passed as argument to `GAUGE_CONTROLLER.gauge_relative_weight()`. This is done again in `gauge_relative_weight()`.

7.13 Mismatch Between Documentation and Implementation, Typos

CS-CRVUSD-059

- The documentation of `Controller._log_2()` states the following although the function does not allow the user to select the rounding direction:

An `internal` helper function that returns the log in base 2 of `x`, following the selected rounding direction

- The documentation of `Controller.wad_ln()` states the following although the function reverts if given 0:

Note that it returns 0 if given 0.

- the documentation of `Controller._debt()` mentions updating the `rate_mul` counter, but it is a view function:

Get the value of debt and rate_mul and update the rate_mul counter

- In `TwoWayLendingFactory`, the `create()` and `create_from_pool()` functions accept `borrowed_token` and `collateral_token` arguments. The naming of these arguments is inaccurate as the two tokens act both as borrowed and collateral.

The following typos were found:

- In `OneWayLendingFactory` and `TwoWayLendingFactory` `set_default_rates()` natspec: "maxumum".
- In `Controller.add_collateral` natspec: "avoid bad liquidations".
- In `OneWayLendingFactory` and `TwoWayLendingFactory` `set_implementations()` natspec: "policy".
- In `Vault.mint()` natspec: "sharess".

The following additional mismatch and typos are present as of :

- In `Vault.maxDeposit()` and `Vault.maxMint()`, the notice states that the function returns `inf` although it can now return different values based on `maxSupply`.

7.14 Missing View Function in the Factory

CS-CRVUSD-060

Both `TwoWayLendingFactory` and `OneWayLendingFactory` defines `get_dx()`, `get_dy()` and `get_dydx()` that wrap the respective homonymous functions of the AMM. However, `get_dxdy()` is missing.

7.15 Total Debt Is Potentially Lower Than the Sum of User Debts

CS-CRVUSD-061

Even when operating in normal conditions, the total debt tracked by a `Controller` can be less than the sum of the debts of all the borrowers. This can be explained by the `total_debt` update rounding down, and being performed frequently, while the user debt updates round up and are performed rarely (more time between updates implies more precision in accrued interest multiplier).

7.16 Unused Variables

CS-CRVUSD-062

- In `TwoWayLendingFactory.transfer_out()`, the variable `token` is assigned `ERC20(vault.collateral_token())` in the case `i!=0` but the variable is not used in this branch.
- In `Controller`, the `use_eth` argument of several methods is unused.
- In `Controller`, constant `MAX_ETH_GAS` is unused.
- `FlashLender` defines `fee` but never uses it except via its public getter. This getter is not needed according to EIP-3156.

Code partially corrected:

The `use_eth` arguments and `MAX_ETH_GAS` constant were removed from the `Controller`.

7.17 Magic Numbers and Constants

CS-CRVUSD-085

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.

Code partially corrected:

`MAX_RATE` comment was changed to 300% to match the value.

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 Dirty Wipe

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.

8.2 Exchange Does Not Revert if It Did Not Succeed

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

8.3 Liquidate Callback Passes Address of the Liquidated User

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.

8.4 Lost Dust Balance on Exchange

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.

8.5 Max Band Over-Estimates the Actual Maximal Band

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.

8.6 Min Band Update

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.

8.7 Peg Keeper Assumptions

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.

8.8 Read Only Reentrancy Protection for Integrating Systems

Integrators using the `rate()` view function of `SemilogMonetaryPolicy` should be aware that the method is susceptible to read-only reentrancy, in case it is queried while the Controller is performing a callback. Before querying `rate()`, `Controller.check_lock()` should be used.

8.9 Rebasing Tokens and Tokens That Transfer Less Are Not Supported as Borrowable or Collateral

Curve Stablecoin does not support rebasing tokens as collateral or borrowable tokens. Indeed the AMM assumes that the balances in each band do not change. For rebasing tokens, the balance accumulated would be locked in the AMM. Furthermore, tokens that transfer less than the specified amount (such as cUSDCv3, which transfers the balance of the caller when the transfer amount specified is `max(uint256)`), are not supported, as the contracts assume that the whole amount specified in the `transfer()` and `transferFrom()` calls are transferred if the calls succeed.

8.10 Rounding Decimals

With allowing the pools to use tokens with decimals other than 18, rounding errors will happen. The AMM's internal accounting artificially uses 18 decimals. This could be compared to assuming a discrete series being continuous. But the controller and the subsequent transfers need to be done in the native decimals. Thus, even though the AMM calculates with the highest precision possible, some values can never be reached in reality. This might have manifold consequences. E.g., arbitrageurs will act with a delay in reality.

8.11 Sandwiching Peg Keeper Actions

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.

8.12 Use of LLAMMA Price

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.

8.13 BoostedLMCallback Freezing Boost Can Be Gamed

The `BoostedLMCallback` contract prevents the boost of an user to be updated in the case they are soft liquidated. This behavior can be gamed to abuse the boost without the possibility to be kicked.

1. Given a pool with low liquidity, a user open a loan over N bands with a small amount of collateral such that they are the only user in the leftmost band of their loan.

2. The user trade the AMM to move the price to the leftmost band of their loan.
3. The user repeatedly trade within the band to increase the value of their shares (and hence the collateral per share of the band).
4. The user trade the AMM back to its original state.

After those actions, the value of the user's loan is concentrated to the leftmost band, in case the rightmost bands of their position are soft liquidated, the user boost will be frozen, while their amount of collateral will not decrease greatly.

8.14 `deploy_gauge()` Is Unpermissioned

In both the `OneWayLendingFactory` and `TwoWayLendingFactory`, `deploy_gauge()` is unpermissioned, this means that if a Vault is created by an honest actor, a malicious actor could deploy a gauge for this vault and have himself as this manager of the gauge. This would prevent an honest actor from creating a gauge through the factory as only one can be created per Vault.