PUBLIC

Code Assessment

of the PegKeeperV2 Smart Contracts

Decemeber 12, 2023

Produced for

Curve

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Curve Team,

Thank you for trusting us to help Curve with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of PegKeeperV2 according to Scope to support you in forming an opinion on their security risks.

Curve implements PegKeeperV2 a more fine-grained version of PegKeeper. The goal of PegKeeperV2 is to maintain the peg of CRVUSD in its stablepools by adding or removing liquidity in the form of CRVUSD.

The most critical subjects covered in our audit are the correct implementation of the PegKeeperV2 and the PegRegulator, the handling of assets by the PegKeeper, and attack vectors based on the manipulation of the liquidity and price oracles. No major issues were uncovered during the review. All the issues have been addressed. Security regarding all the aforementioned subjects is high.

The general subjects covered are access control, gas efficiency, documentation, and specification and testing. The security regarding all the aforementioned subjects is high.

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

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	2
• Code Corrected	2

2 Assessment Overview

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

2.1 Scope

The assessment was performed on the source code files inside the PegKeeperV2 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	27 Oct 2023	41f44f25d8a303ef1d399e43b8a37400b92c16be	Initial Version
2	11 Dec 2023	5a46bb9c1f43b7d4062127b9919e3c2ed366ad34	Fixes

For the Vyper smart contracts, the compiler version `0.3.9` was chosen.

The following contracts are in scope of this review:

- `contracts/stabilizer/PegKeeperV2.vy`
- `contracts/stabilizer/PegKeeperRegulator.vy`

2.1.1 Excluded from scope

Excluded from the scope are the contracts not explicitly mentioned in scope. In particular, the stablecoin (CRVUSD) implementation, the pools and the price oracles with which the contracts in scope interact are considered to function correctly. We assume that read-only reentrancy attack vectors cannot manipulate the virtual price retrieved from `Pool.get_virtual_price()`. Moreover, there is an implicit assumption in the system that it will always recover its peg (see `calc_profit()`). We take this assumption for granted. The parameters of the system were chosen based on statistical analyses of historical data. In this report, these parameters were assumed to be correct. The contracts in scope implement a stricter version of `contracts/stabilizer/PegKeeper.vy`. This contract is assumed to function correctly. Behaviors of the PegKeeper replicated in PegKeeperV2 are assumed to be correct.

2.2 System Overview

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

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

Curve offers the PegKeeperV2 and the PegKeeperRegulator which improve on the mechanism implemented by PegKeeper. The PegKeeper is a component of the Curve stablecoin. Its goal is to maintain the peg of the CRVUSD in its stablepools e.g., the CRVUSD/USDC pool. To that end, the peg keeper holds an amount of CRVUSD that is added or removed from the stable pool. In particular, when the stable pool holds more USDC than CRVUSD, it adds CRVUSD to the system. When there's an excessive amount of CRVUSD in the pool, the peg keeper will remove some of it by burning part of the respective LPs it holds.

2.2.1 Issues with the PegKeeper

In this section, we present a few potential issues that have been previously identified regarding the original PegKeeper.

Spam Attack:

The PegKeeper can be prevented from properly updating its position in the pool, if it's sandwiched by transactions that manipulate the liquidity of the pool back and forth. Even though this attack is not profitable for the attacker, the issue is addressed to make the protocol more resilient. To that end, the price oracle of the stable swap is used to verify that the spot pool's price is close to the one reported by the exponential time-weighted price given by the oracle. The allowed deviation is 0.05%.

Depeg:

Let's assume a CRVUSD/USDC pool. In case USDC loses its peg, a large amount of USDC will flow into the CRVUSD/USDC pool. In this case, the PegKeeper will add more liquidity to keep the peg. Should the price of USDC not return to $1, the assets of the PegKeeper will remain locked in the pool. Hence, there's a need to limit the exposure of the PegKeeper's assets as much as possible. We specify a PegKeeper per pool.

We define the debt ratio as follows:

$$r_i = \frac{debt}{limit}$$

Where $debt$ is the amount of liquidity given by the PegKeeper to Pool i and $limit$ is the maximum amount that could be given. As a result, $r_i \leq 1$. Two more constraints are defined:

- One PegKeeper can use up to $r_i \leq \frac{1}{4}$
- Three PegKeepers can use in limit whole i.e., $r_i \leq 1$

Using the constraints above we derive the following formula for the maximum debt ratio allowed for PegKeeper i.

$$max_i = (\alpha + \beta \cdot \sum_{j \neq i} \sqrt{r_j})^2$$

where $\alpha = \frac{1}{2}, \beta = \frac{1}{4}$ constants, r_j is the debt ratio of the other PegKeepers. Note for a single Pegkeepr the first constraint is met i.e., $max_i = (\alpha + \beta \cdot 0)^2 = \frac{1}{4}$. For 3 or more peg keepers which use the maximum debt ratio, $max_i = (\alpha + 3 \cdot \beta \cdot 1)^2 = 1$

2.2.2 Implementation Details

The PegKeeperV2 implements the following interface:

- `update`: it can be called by any user and executed successfully as long as the peg keeper can make a profit and gets an allowance from the regulator to add or remove liquidity. At the end of the transaction parts of the profit of the Keeper goes to the caller. The rest of the profits can be redeemed by anyone on behalf of a receiver.
- `withdraw_profit`: it can be called by anyone and transfers the assets to a receiver specified by the contract.

Admin interface:

The following methods can only be called by the admin of the contract:

- `set_new_caller_share`: sets the part of profits sent to the caller of `update`. It sanitizes the value.
- `set_new_regulator`: sets a new regulator.

- `commit_new_admin`: sets a new admin who can claim the administration of the contract within a deadline.
- `apply_new_admin`: can be called by the new admin within the time window defined by the contract.
- `set_new_receiver`: sets a new receiver of the profits.

The PegKeeperRegulator implements the following interface:

- `provide_allowed`: it determines the allowed amount that can be provided by the peg keeper. It returns 0 if:
    1. CRVUSD price is less than or equal to 1 or,
    2. the price reported by the oracle deviates from the price of the pool or,
    3. the price of CRVUSD against the peg asset is greater than the other stable pools.
- `withdraw_allowed`: it determines the allowed amount that can be withdrawn by the peg keeper. It returns 0 if:
    1. CRVUSD price is greater than or equal to 1 or,
    2. the price reported by the oracle deviates from the price of the pool

Admin interface:

The following methods can only be called by the admin of the contract:

- `[add|remove]_peg_keepers`: adds or removes pegkeepers.
- `set_[price_deviation | debt_parameters]`: sets the relevant parameters of the system.
- `set_killed`: (un)pauses the deposits and/or the withdrawals from the system.
- `set_[emergency_]admin`: it sets the (emergency) admin in one step.

2.2.3 Roles and Trust Model

We derive the following roles for the contracts in scope:

- End users: they can call `PegKeeper.update()` as long as the system is not paused.
- The PegKeeperV2 admin: For the actions they can perform please refer to the Implementation Details section. The admin is set during the construction of the contract and can change after an admin's action. The role is assumed to be controlled by the Curve DAO and considered fully trusted.
- The PegKeeperRegulator admin: For the actions they can perform please refer to the Implementation Details section. The admin is set during the construction of the contract and can change after an admin's action. The role is assumed to be controlled by the Curve DAO and considered fully trusted.
- The PegKeeperRegulator emergency admin: They can pause deposits or withdrawals. The role is assumed to be controlled by a trusted entity within the Curve DAO and considered fully trusted.

No user's funds should be held and handled by the contracts.

None of the smart contracts is upgradeable. The system relies heavily on oracles to make decisions so oracles are considered fully trusted and up-to-date. Moreover, CRVUSD implementation as well as the

LLAMMA mechanism are considered to work correctly. Any issue related to them could completely break the contracts under scope.

2.2.4 Changes in Version 2

The following changes are implemented:

- `LAST_PRICE_THRESHOLD` constant was replaced by the `worst_price_threshold` settable by the admin.
- `peg_keeper_i` is used to access to current pegkeeper info in constant time.

3 Limitations and use of report

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

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

4 Terminology

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

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

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

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

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

5 Findings

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

- Design: Architectural shortcomings and design inefficiencies

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

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

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

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

6.1 Missing Events

Design Low Version 1 Code Corrected

CS-CRVPKV2-003

The following methods of the `PegKeeperRegulator` do not emit any event.

1. `set_price_deviation()`
2. `set_debt_parameters()`
3. `set_killed()`
4. `set_emergency_admin()`

Moreover, setting a new `admin`, `price_deviation`, `debt_parameters` or `emergency_admin` in the constructor of `PegKeeperRegulator`, does not emit an event.

Code corrected:

The following events have been added:

1. `set_price_deviation` emits a `PriceDeviation` event.
2. `set_debt_parameters` emits a `DebtParameters` event.
3. `set_killed` emits a `SetKilled` event.
4. `set_emergency_admin` emits a `SetEmergencyAdmin` event.

Moreover, the constructor emits now all the respective events.

6.2 Missing Sanity Checks

Design Low Version 1 Code Corrected

CS-CRVPKV2-004

The following sanity checks could be applied:

1. `PegKeeperV2.commit_new_admin()` does not check that `_new_admin` is non-zero.
2. `PegKeeperV2.set_new_regulator()` does not check that `_new_regulator` is non-zero. Moreover, it is not guaranteed that the new regulator set will implement the interface required.
3. `PegKeeperRegulator.add_peg_keeper()` does not check if the new keepers to be added are not already part of the peg_keepers. Moreover, the `_peg_keepers` array could have duplicates.

Code corrected:

1. `_new_admin` is checked to be non-zero.
2. `_new_regulator` is checked to be non-zero.
3. `add_peg_keeper()` checks if the peg keeper has already been added.

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 Gas Optimizations

Informational **Version 1**

CS-CRVPKV2-001

The following gas optimizations could be applied:

1. `PegKeeperV2.apply_new_admin()`: `self.new_admin_deadline` is read twice from storage but it could be cached.
2. `PegKeeperV2._withdraw()`: the function reads `self.debt` twice from storage but it could be cached.
3. `PegKeeperRegulator.withdraw_allowed()`: This function is expected to be called often. Therefore, it could make sense to be able to store the peg keepers in a map so that their info can be retrieved in :math:`O(1)` instead of :math:`O(n)`.

Code corrected:

The optimizations are implemented as follows:

1. `self.new_admin_deadline` is cached in `new_admin_deadline`.
2. `self.debt` is cached in `debt`.
3. `peg_keeper_i` is introduced to allow access to peg keeper info in constant time.

7.2 Redundant Events

Informational **Version 1**

CS-CRVPKV2-002

The following methods of the `PegKeeperV2` emit a redundant event should the same parameters are set.

1. `set_new_caller_share()`
2. `set_new_regulator()`
3. `commit_new_admin()`
4. `set_new_receiver()`

The method `set_admin()` of the `PegKeeperRegulator` emits a redundant event should the same admin be set.

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 TVL Manipulation

Note Version 1

While the new system prevents price manipulation with the use of oracles through the regulator, it might still be possible to manipulate the TVL of a pool to trick the PegKeeper into providing more crvUSD than it should. Such manipulation could look like the following:

1. Provide a large amount of both assets to the pool in a balanced way.
2. Call the PegKeeper to provide crvUSD, the result of `(balance_peg - balance_pegged) / 5` will be inflated.
3. Remove the previously provided assets from the pool.