PUBLIC

Code Assessment

of the Curve LP & stETH oracle Smart Contracts

March 1, 2022

Produced for

by

Contents

1 Executive Summary

Dear Maker team,

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

The curve lp oracle contract implements a specialized oracle for the maker ecosystem that provides prices for lp tokens of a curve.finance pool. It determines the price based on the curve pools `get_virtual_price()` function. Its architecture is very similar to other pricefeeds such as e.g. the G-UNI LP Oracle. The stETH price feed implements a specialized oracle retrieving the price of stETH.

During the review no important issue was uncovered. The most critical subjects covered in our audit are functional correctness and access control. Security regarding all the aforementioned subjects is high. General subjects covered were code complexity and gas efficiency. All the aforementioned subjects were of high quality.

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	4
• Code Corrected	3
• Acknowledged	1

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 LP & stETH oracle 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	7 February 2022	e26498b29969aab343de253abf7afe73b24ee686	Initial Version
2	28 February 2022	79a5277f8f7daf0bea3f64fc0b2f66d29c92e05d	Second Version

For the solidity smart contracts, the compiler version `0.8.9` was chosen. For version two, `0.8.11` is used.

The file in scope for this review were: CurveLPOracle.sol and StETHPrice.sol. The internal workings of the Curve functions including `get_virtual_price()` and the internal workings of stETH functions including `getPooledEthByShares()` are not part of this review.

2.2 System Overview

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

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

2.2.1 CurveLPOracle

CurveLPOracle is a specialized oracle in the Maker ecosystem that provides prices for the lp (liquidity provider) tokens of a curve pool. It determines the price of the lp token based on the `get_virtual_price()` function of the curve pool and the lowest rate of one of underlying assets returned by one of the stored oracles. By multiplying the virtual price with the lowest underlying rate a lower bound evaluation of the LP token is created.

Curve pools are a special type of AMM pools that use liquidity more effectively by concentrating most of it at current prices.

Similarly to other oracles of Maker, CurveLPOracle operates with two `Feed` variables `cur` and `nxt` which store the current price and the queued price respectively. The prices propagate through the system with 1 hour delay, therefore allowing `wards` to take measures in case the queued price `nxt` is set to an incorrect value.

CurveLPOracle provides the following functionalities:

- `stop()`: can be called only by authorized wards to stop the oracle.
- `start()`: can be called only by authorized wards to remove the stop flag `stopped = 0`.
- `step()`: can be called only by authorized wards to update the `hop` value (default 1 hour).
- `link()`: can be called only by authorized wards to update the oracle address.

- `zzz()`: can be called by anyone and returns the timestamp of the last price update.
- `pass()`: can be called by anyone and returns `true` if enough time to compute the new price has passed since the last update.
- `poke()`: can be called by anyone and computes the new price of the lp token given that `pass()` returns `true`.
- `peek()`: can be called only by whitelisted addresses in the mapping `bud` and returns the current price and its validity.
- `peep()`: can be called only by whitelisted addresses in the mapping `bud` and returns the queued price (which will be set as current in the next call of `poke()`) and its validity.
- `read()`: can be called only by whitelisted addresses in the mapping `bud` and returns the current price as `bytes32`.
- `kiss()`: can be called only by authorized wards and sets a single (or an array of) address into the whitelist mapping `bud`.
- `diss()`: can be called only by authorized wards and removes a single (or an array of) address from the whitelist mapping `bud`.
- The standard authorization functions `rely()` and `deny()`.

CurveLPOracleFactory allows any user to deploy an CurveLPOracle by calling the function `build()` which takes as parameters:

- `address _owner`: the oracle calls `rely()` for this address.
- `address _pool`: The curve pool of this price feed
- `bytes32 _wat`: the label of the token whose price is tracked
- `address[] memory _orbs`: the addresses of the oracles for the tokens of the pool

2.2.2 StETH price

In contrast to the CurveLPOracle, StETHPrice does not implement a OSM but a medianizer. It is a specialized oracle for retrieving the price of stETH from the existing wstETH oracle and the stETH contract. The wstETH oracle, instead of the ETH oracle, is used since an ETH/stEth trading pair would not necessarily be required to trade at a one-to-one ratio. A medianizer returns the current price of an asset and has following functionality:

- `peek()`: can be called only by whitelisted addresses in the mapping `bud` and returns the current price and its validity.
- `read()`: can be called only by whitelisted addresses in the mapping `bud` and returns the current price as `uint256` only if it is a valid price.
- `kiss()`: can be called only by authorized wards and sets a single (or an array of) address into the whitelist mapping `bud`.
- `diss()`: can be called only by authorized wards and removes a single (or an array of) address from the whitelist mapping `bud`.
- The standard authorization functions `rely()` and `deny()`.

2.2.3 Trust Model & Roles

Wards: Fully trusted to behave honestly and correctly at all times. They can set the oracles, the parameters `hop`, can stop the oracle by calling `stop()` or resume it with `start()` and add/remove whitelisted addresses to/from the mapping `bud`. We assume the `wards` monitor the price feed continuously and take measures in case `nxt` holds an incorrect price value before it propagates into the system.

Curve pool: Fully trusted to work correctly. This price feed only works for Curve Pools version 2. In particular version 1 pools and tricrypto-style pools are not supported by this price feed. Furthermore, the use of the virtual price implies the assumption that the pools are balanced, which they will not be at all points in time.

In case a Curve-Lending Pools are used, the protocol receiving the actual funds are also trusted to work correctly. Incorrectly set-up pools, e.g. certain trustlessly factory-produced pools, will not work correctly.

stETH contract: Fully trusted to work correctly.

External users: Untrusted. Can call the functions of CurveLPOracleFactory or CurveLPOracle with arbitrary parameters.

Oracles: Part of the Maker Ecosystem, fully trusted. Used to query the prices of the underlying tokens. These oracles return the **current** live rate without any delay. The overall returned price is only correct if different oracles exist for the different tokens that should theoretically be of equal value, e.g. USDC and USDT.

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
- Correctness: Mismatches between specification and implementation

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Mismatches With Documentation and Lack Thereof Acknowledged

5.1 Mismatches With Documentation and Lack Thereof

Correctness Low Version 1 Acknowledged

Documentation plays a crucial part for understanding a codebase and integrating it into a live system. However, the code lacks project specific documentation and is only described in a generic way in the MakerDAO Oracle documentation.

Moreover, the interfaces of the CurveLPOracle mismatch the interface specified:

- The documentation specifies `step()` to take a `uint16` as an input parameter while the code defines `step(uint256)` which checks that a provided argument does not exceed the maximum `uint16`.
- Documented function `change()` is missing. However, `link()` is undocumented but implements the specified functionality of `change()`.
- According to documentation, `stop()` should only set the `stopped` flag while `void()` should set the flag but also reset `nxt`, `cur` and `zph`. In contrast, `void()` is missing while `stop()` implements the semantics of `void()`.

Similarly StETHPrice deviates from the MakerDAO medianizer documentation.

Acknowledged:

MakerDAO acknowledges this.

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	3

- Curve Registry Code Corrected
- Outdated Compiler Version Code Corrected
- Potential Inconsistency Code Corrected

6.1 Curve Registry

Correctness Low Version 1 Code Corrected

According to the Curve Documentation of their registry contracts, the central source of truth in the Curve system is the address provider. That contract allows changing the registry through `set_address()` when the `id` parameter is set to zero. Currently, the oracle stores the registry as an immutable. Hence, in case the registry changes, the oracle will utilize a wrong registry.

Code corrected:

The registry is now queried from the address provider.

6.2 Outdated Compiler Version

Design Low Version 1 Code Corrected

The solc version is fixed to version `0.8.9`. The introduced changes in versions `0.8.10` and `0.8.11` could reduce gas consumption of the inline-assembly code of `poke()`.

Code corrected:

Compiler version `0.8.11` was chosen.

6.3 Potential Inconsistency

Design Low Version 1 Code Corrected

`step()` sets a new value to hop which specifies the minimum time between calls to `poke()`. Function `zzz()` should return the time of the last `poke()`.

Consider now the following scenario where `hop` is 1 hour and the `zph` is set to the current time + 1 hour. Assume that a call to `step()` sets `hop` to 10 minutes. Now, `zzz()` returns current time + 50 minutes which is in the future. Moreover, the next poke requires waiting for one hour instead of only 10 minutes.

Ultimately, that could lead to temporary inconsistencies.

Code corrected:

`zph` (Time of last price update plus `hop`, the minimum time between price updates) is now updated on `step()` using the new value of `hop`. The update of `zph` is skipped when it hasn't been set yet but `hop` is updated.