

Code Assessment of the Curve ETH/sETH Smart Contracts

September 27, 2021

Produced for



by



Contents

1	Executive Summary	3
2	Assessment Overview	4
3	Limitations and use of report	7
4	Terminology	8
5	Findings	9
6	Resolved Findings	11
7	Notes	13



1 Executive Summary

Dear Curve Team,

First and foremost we would like to thank you for giving us the opportunity to assess the current state of your Curve ETH/sETH contract. This document outlines the findings, limitations, and methodology of our assessment.

We hope that this assessment helps to improve the project further. Our assessment uncovered one medium security issue concerning a reentrancy for the administrator account and some low severity findings which we describe in our findings section. We are happy that issues like the redundant use of `RATES` and `PRECISION`, helped to make the code more efficient and significantly reduced the code size. We performed multiple independent tests to uncover potential inefficiencies or discrepancies between specific operations like sandwiching exchanging coins vs. adding and removing liquidity in one coin - but we could not uncover major problems.

We warmly appreciate questions and feedback to improve our service.

Yours sincerely,

ChainSecurity

1.1 Overview of the Findings

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

Critical -Severity Findings	0
High -Severity Findings	0
Medium -Severity Findings	1
• Code Corrected	1
Low -Severity Findings	5
• Code Corrected	3
• Risk Accepted	2



2 Assessment Overview

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

2.1 Scope

The general scope of the assessment is set out in our engagement letter with Curve.Finance dated November 02, 2020. The assessment was performed on the source code files inside the Curve ETH/sETH 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	02 Nov 2020	8ad854eb6c1e95ffb4fc375d4359d4e140569d5a	Initial Version
2	15 Dec 2020	2428dd6c809a3e5bcf31285e3deb5c68eda4e55e	Updated Version

Initially, for the Vyper smart contracts, the compiler version 0.2.7 was chosen. The compiler was updated after the intermediate report to version 0.2.8.

2.1.1 Excluded from scope

The sETH token is treated as a regular ERC-20 token for the sake of the report. Furthermore, Curve's Liquidity Provider-Token is assumed to be a regular ERC-20 token with proper access control. Additionally, the security of the DAO that can perform admin operations on this pool is out of scope. Lastly, the underlying mathematical formulas, as described in the whitepaper, have already been checked separately.

2.2 System Overview

The reviewed project consists of one smart contract `StableSwapETH.vy` written in the Vyper programming language. It implements a liquidity pool based on an invariant called StableSwap and described in Curve's whitepaper.

2.3 Modifying Liquidity

Users can modify the liquidity of the pool in the following ways:

1. Add liquidity to the pool (`add_liquidity`).
2. Remove liquidity in a balanced manner (`remove_liquidity`). The users determine the number of `lp_tokens` they want to remove. The liquidity of each token is reduced proportionally to the number of the `lp_tokens`.
3. Remove liquidity in an imbalanced manner (`remove_liquidity_imbalance`). The users determines the amount of tokens they want to remove instead of the number of `lp_tokens`.
4. Remove liquidity of one coin (`remove_liquidity_one_coin`). The functionality is similar to `remove_liquidity_imbalance` with other coins set to zero, but it is more gas efficient and the user only pays fees in the removed coin.

2.4 Determining the invariant D

D is determined by the mathematical formula. It is calculated iteratively. The maximum iterations allowed are 255. D changes when:

1. Liquidity is added to the pool (`add_liquidity`)
2. Liquidity is removed from the pool (`remove_liquidity`, `remove_liquidity_imbalance`, `remove_liquidity_one_coin`)
3. When A changes due to `ramp_A` (see below)

It should not change after an `exchange`.

2.5 Setting and Updating the Amplification Coefficient A

The amplification coefficient A is set in the constructor and can be modified by the owner of the contract. The provided value should equal to $A * n^{(n-1)}$. The value of A can be changed by the owner using `ramp_A`. The new value of the coefficient cannot be more than ten times greater or smaller than its current value. The update takes place gradually. An update lasts `MIN_RAMP_TIME`. During this time window, the change in the value of A is proportional to the percentage `MIN_RAMP_TIME` which has passed. Note that the owner can stop the update by calling `stop_ramp_A`.

2.6 Fees

Two kind of fees are applied. One fee remains in the contract and goes towards the liquidity provider by increasing the amount they can withdraw with the same amount of lp_tokens. The other fee is an admin fee. Admins can withdraw their share or donate their share to the pool. Both fees are percentages. In the functions `add_liquidity`, `remove_liquidity_imbalance` and `remove_liquidity_one_coin` fees are applied as well while in `remove_liquidity` a fee is **not** applied. The fee, where applicable, comes as percentage of difference between a balanced change of the liquidity and the real change.

2.7 Administrator actions

The administrator i.e., the `owner` of the contract can perform the following **exclusive** actions:

1. Change the value of the amplification coefficient A (`ramp_A`) and stop the ramping (`stop_ramp_A`).
2. Change the fees (`commit_new_fee`). This is just a commitment. The network is notified for the change, however, the change is not applied directly.
3. Apply the new fees (`apply_new_fee`).
4. Revert the change in the fees (`revert_new_parameters`).
5. Transfer ownership (`commit_transfer_ownership`). This is just a commitment. The network is notified for the change, however, the change is not applied directly.
6. Apply the ownership transfer (`apply_transfer_ownership`)
7. Withdraw admin fees (`withdraw_admin_fees`)
8. Donate admin fees (`donate_admin_fees`). The owner gives up on the surplus between the real balance of the tokens in the contract and the balance held by `self.balances`.
9. Freeze the contract (`kill_me`) and unfreeze the contract (`unkill_me`)

2.8 Killed (Frozen) Contract Functionalities

A contract can be killed until after 60 days from its deployment. When a contract is in a killed state **only** the following calls are **blocked**:

1. `add_liquidity`
2. `remove_liquidity`
3. `remove_liquidity_imbalance`
4. `remove_liquidity_one_coin`
5. `exchange`

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 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 associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. 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 the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than 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 severities. 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. All of the findings are split into these different categories:

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

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

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

- [Certain Inputs Unchecked in Constructor](#) **Risk Accepted**
- [Ramping Down Might Incentivize Delayed Liquidity](#) **Risk Accepted**

5.1 Certain Inputs Unchecked in Constructor

Security **Low** **Version 1** **Risk Accepted**

When new fees are committed through the `commit_new_fee` function they are checked against the respective maximum values to prevent mistakes. However, when the fees are initially set inside the constructor no such check is performed. Hence, initial fees might be outside the permitted value range.

Similarly, when the amplification factor is changed through ramping, its value range is checked. However, during the constructor this check for the amplification factor does not take place.

Risk accepted: As deployment is a rare event and as deployed contracts will be checked by the development team, there is no immediate need to add these checks. An incorrect contract can be "killed".

5.2 Ramping Down Might Incentivize Delayed Liquidity

Design **Low** **Version 1** **Risk Accepted**

While the amplification factor is ramping down in an imbalanced pool, liquidity providers have an incentive to wait before providing extra liquidity. This is because they will receive more liquidity tokens in the future for the same liquidity. In the extreme case of a maximally sharp ramp down and a very imbalanced pool, waiting for ten minutes provides roughly 0.14% additional liquidity tokens.

However, this only holds as long as no further fees are accumulated during this time and as long as no re-balancing takes place inside the pool and hence constitutes a fairly unlikely scenario.

Risk accepted: As mentioned above this only applies for very sharp ramps. As the DAO will control the parameter of these ramps, the DAO can also ensure that the sharpness is low enough to avoid any issues.



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	1
• Reentrancies Code Corrected	
Low -Severity Findings	3
• Redundant Use of RATES and PRECISION Code Corrected	
• _xp and _xp_mem Redundant Array Access Code Corrected	
• get_D Should Handle the Case of Non-convergence Code Corrected	

6.1 Reentrancies

Security **Medium** **Version 1** **Code Corrected**

1. During the execution of `remove_liquidity` and `remove_liquidity_imbalance` multiple asset transfers are made. One of these assets is ETH, while the others are ERC-20 tokens. The transfer of ETH can lead to the following reentrancy. Through the transfer of ETH, the execution might reenter the contract and call `donate_admin_fees`. Note that this requires `owner` privileges. Inside `donate_admin_fees`, the internal `balances` mapping for the ERC-20 tokens will be updated as follows:

```
self.balances[i] = ERC20(coin).balanceOf(self)
```

This assignment is incorrect in this context as the contract still holds the tokens that are about to be transferred due to the removed liquidity. Hence, after the transaction is complete: `self.balances[i] > ERC20(coin).balanceOf(self)`. This breaks an important invariant in the contract.

2. During the call to `withdraw_admin_fees` an ETH transfer takes place. The transfer of ETH can lead to the following reentrancy. Through the transfer of ETH, the execution might reenter the contract and call `donate_admin_fees`. Note that this requires `owner` privileges, but these were already needed for `withdraw_admin_fees`. As a result, the admin fees for some of the coins will be donated while the admin fees for other coins will be withdrawn, leading to a state that is only reachable through a reentrancy.
3. Certain admin functions have no reentrancy protection. Hence, they can be called in a reentrancy from any of the functions that transfers ETH. However, for those reentrancies the only effects are incorrectly ordered events. As an example, a `NewFee` event could be emitted in between multiple events belonging to a `remove_liquidity` call.

Code corrected: Additional Reentrancy Guards were added. These now also cover the functions `donate_admin_fees` and `apply_new_fee` among others.



6.2 Redundant Use of RATES and PRECISION

Design **Low** **Version 1** **Code Corrected**

RATES is a constant vector containing in all cells the value 10^{18} . PRECISION is a constant of value 10^{18} .

There are cases, such as in `exchange`, where the value of a cell of RATES is divided by PRECISION. This division is redundant.

```
rates: uint256[N_COINS] = RATES
# Both multiplication with rates[i] and division with PRECISION can be avoided
x: uint256 = xp[i] + dx * rates[i] / PRECISION
```

Code corrected: The code was changed accordingly to remove the redundancies and to save gas.

6.3 `_xp` and `_xp_mem` Redundant Array Access

Design **Low** **Version 1** **Code Corrected**

In both `_xp` and `_xp_mem` the array `results` is initialized with the array RATES. However, `results` later ends up equal to `self.balance`. This is because of the multiplication (with `result[i]`) and a redundant division (with `LENDING_PRECISION`). Note, that RATES equals to `LENDING_PRECISION` for all `i`. In the general case this code is useful, however for this token pair, it provides no additional value. RATES and `LENDING_PRECISION` are constants, the gas overhead is fairly low.

```
result: uint256[N_COINS] = RATES
for i in range(N_COINS):
    result[i] = result[i] * self.balances[i] / LENDING_PRECISION
return result
```

Code corrected: The code was changed accordingly to remove the redundancy and to save gas.

6.4 `get_D` Should Handle the Case of Non-convergence

Correctness **Low** **Version 1** **Code Corrected**

The calculation of the invariant D is limited to 255 steps. If there is no convergence then a wrong invariant is returned. The invariant is used to mint liquidity provider tokens. Thus, incorrect number of tokens can be minted. For the case of non-convergence, a verification step of the computed solution could be added.

Code corrected: The new implementation reverts in case of non-convergence. This ensures that no faulty results are used for further computation.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues.

7.1 Content of Events

Note Version 2

The events `RemoveLiquidityImbalance` and `AddLiquidity` contain the value `D1` which represents the intermediate calculation of the invariant. Including `D2` might be more helpful.

The event `RemoveLiquidityOne` does not include the information which coin was removed from liquidity. That might be relevant information.

7.2 Fee Avoidance

Note Version 1

It is theoretically possible to avoid fee payments completely by repeatedly exchanging, adding or removing such small amounts that fees are zero due to arithmetic errors. This results in a loss of fees for both liquidity providers and admins. However, in almost all cases the saved fees will be overcompensated by the additional gas costs. Hence, such a scenario would only be realistic in the context of Zero-Gasprice Transactions.

7.3 Incentive to Remove Liquidity

Note Version 1

There might be an incentive for liquidity providers to remove liquidity while the amplification factor is ramped down. In case of a really imbalanced pool and a very rapid ramping down of the amplification factor, the following sequence might leave the liquidity provider with more liquidity tokens that they started with:

1. Remove liquidity by withdrawing only the non-scarce asset
2. Wait for the ramping to continue
3. Re-add the removed asset to regain liquidity tokens

In case of a very imbalanced pool and a sharp ramp, the liquidity provider could end up with 0.14% more liquidity tokens than they started with by waiting just ten minutes in step 2. This, however, only works if no other transactions take place inside the pool.

7.4 Inefficiencies When Removing Single Coin

Note Version 1

When removing just a single coin from the pool liquidity, the `remove_liquidity_one_coin` function can be used. However, this function in certain cases is less efficient than using the `remove_liquidity_imbalance` function and just setting all values except for the desired one to zero. In our limited experiments, the biggest difference occurred when `remove_liquidity_imbalance` provided 0.00008% additionally withdrawn assets.

Hence, the difference is very small and mostly negligible. Furthermore, the `remove_liquidity_one_coin` function is generally expected to have lower gas costs. Finally, it is important to note that the fee structure is different for the two functions as `remove_liquidity_one_coin` will only pay fees in the withdrawn coin, while `remove_liquidity_one_coin` will pay a roughly equivalent amount of fees in all coins.