PRIVATE

Code Assessment

of the CurveCryptoSwap2ETH Smart Contracts

April 1st, 2022

Produced for

Curve

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Michael,

Thank you for trusting us to help Curve with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of CurveCryptoSwap2ETH according to Scope to support you in forming an opinion on their security risks.

Curve implements a simplified pool has very similar functionality compared to Tricrypto, but is optimized for two tokens where one token might be ETH.

This report is an intermediate report.

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	2
• No Response	2
Medium-Severity Findings	2
• No Response	2
Low-Severity Findings	8
• No Response	8

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 CurveCryptoSwap2ETH 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	Feb 07 2022	e2a59ab163b5b715b38500585a5d1d9c0671eb34	Initial Version

For the vyper smart contracts, the compiler version `0.3.1` was chosen.

2.1.1 Excluded from scope

The LiquidityGauge was not part of this audit.

2.2 System Overview

This audit focuses on changes to the Tricrypto pool that supports tokens with different prices. The Tricrypto pool implementation has been optimized for two crypto assets, which allowed the reduction of gas costs. These new pools are deployed through a factory which checks for correctness of certain deployment parameters and allows for central control over some configuration parameters of those pools. The deployed pools use non-upgradable proxy contracts. In the following we will describe the overall system, mostly taken over from our previous Tricrypto audit.

Generally, Curve is a variant of a decentralized exchange (DEX) that relies on automated market making (AMM). Curve and similar AMM projects build upon the concept of liquidity pools and an invariant to determine the ratio/price to swap one coin vs another. A liquidity pool consists of multiple tokens. The tokens are added to the pool by so called liquidity providers. In return, liquidity providers receive a token that represents a share of the funds they own of the pool. Providing liquidity is incentivized by trading fees that the liquidity provider will receive when users trade (the fees are paid out indirectly by increasing the pool's value). By having a certain amount of tokens, trades can be executed immediately in one transaction. The execution can be done immediately because no counter-party is needed.

Curve modified their function compared to e.g. Uniswap in a way that the price is more robust by introducing a modified invariant. This is achieved by flattening the curve around the equilibrium and shifting the curve given certain conditions are met.

2.2.1 The Pool

A pool consists of two crypto assets with the same limitations as for the previous more general case. While most ERC20 tokens are supported, some tokens are not (see Supported Tokens).

2.2.2 The Curve

A pool always tracks the balances for both tokens. The price of the second token is denominated in the first token.

An invariant with the following parameters defines a curve which is used to determine the prices for trading. The parameters are `D` (invariant value), `A` (amplification factor), and `gamma` (which controls the size of the flat curve area). The invariant is fully defined in Curve's documentation.

2.2.3 Profit and Conditional Price Recalculations (unchanged)

There is a virtual price to track the development of liquidity shares. The virtual price is determined by the value of the pool in the equilibrium. Changes of this value are used as a profit/loss indicators. Changes to the virtual price determine whether a potential price change is accepted or not.

Many curve actions will trigger the check whether a price update should be performed. This check will evaluate whether the currently used prices differ significantly from the internal price oracle. Before accepting a price update, the resulting theoretical gain/loss is calculated by comparing the new updated prices and the resulting value of the pool with the accumulated profits (interest-bearing) the pool has made. The formula is defined in Curve's documentation in detail.

If a price update results in a loss for the pool (by the definition mentioned before) which exceeds half the accumulated profits, the transaction would not update the prices. As a result the curve would not be shifted but instead, a movement on the curve would happen. Hence, the exchange still works, but the flat area of the curve is not being utilized until the price update becomes possible or the prices shift back to the previous values.

2.2.4 The Fee Model (unchanged)

There are two kinds of fees, admin fees and dynamic fees. Admin fees occur only when the liquidity pool accumulates funds (measured as `xcp_profit`). Admin fees are paid by minting new liquidity provider token to an admin account.

Dynamic fees are paid when depositing, exchanging and withdrawing liquidity in one coin. The fee remains in the pool, hence, increasing the value of the liquidity tokens which is the incentive to provide liquidity. These fees depend on how close the current balances are to the equilibrium point of the curve.

2.2.5 Administration

The only role in the system is the admin. The admin of the factory is also the admin of each pool. The admin can transfer its role to another account by calling the function `commit_transfer_ownership`. The new admin can then call `accept_transfer_ownership`.

The curve parameters `A` and `gamma` can be changed by calling `ramp_A_gamma`. The change will take place gradually (over a defined period e.g. 24h) and not at once. The change process can be stopped by calling `stop_ramp_A_gamma`.

Fees and fee related parameters, the adjustment step, the moving average half-time parameter for the price oracle and the allowed extra profit can be adjusted by calling `commit_new_parameters` which allows to call `apply_new_parameters` after a fixed waiting period (3 days). The changes are immediate after `apply_new_parameters` has been called. Alternatively, if `revert_new_parameters` the proposed changes are reset.

There is no more a feature to stop (`kill`) the pool.

2.2.6 Liquidity (unchanged)

Initially, for the exchange to work, liquidity needs to be provided. A future liquidity provider can call `add_liquidity` to do so. The function pulls the funds into the exchange contract with a `transferFrom`. Based on the new balances and existing prices in the pool, the curve parameter `D` is calculated. `D` is needed to determine the amount of tokens representing the share the liquidity provider now owns from all deposited funds in the exchange pool (called liquidity tokens). A fee is deducted and the liquidity tokens are minted to the liquidity provider.

If `D > 0` (should be the case if it is not the first deposit), the function conditionally updates the price information and profit calculation. The condition for updating the price information and, hence, changing the curve is that no more than half of the accumulated historic exchange profit can be lost with price updates. The definition of profit and loss is provided in Curve's documentation. When liquidity has been added successfully, `add_liquidity` emits the event `AddLiquidity`.

To withdraw provided tokens, a liquidity provider can call `remove_liquidity` or `remove_liquidity_one_coin`. The functions burn the provided amount of pool liquidity tokens, calculate the corresponding amount of tokens and transfer the tokens to the function callee. `remove_liquidity` will transfer tokens from each coin in the pool's current ratio. `remove_liquidity_one_coin` will payout an equivalent amount in one token. Both functions will update `D`. Additionally, `remove_liquidity_one_coin` will deduct a fee and conditionally update the price information.

2.2.7 Trading

Users that want to exchange two tokens can call `exchange` (or `exchange_underlying` or `exchange_extended`). The user needs to provide the information about which tokens shall be exchanged, provide the amount to be exchanged and specify a minimum amount of tokens to be received. Usually, the exchange function pulls the funds to be exchanged into the exchange contracts via a `transferFrom`. Alternatively, a callback can be specified through which the funds will be provided. Then, it calculates how many tokens the user will receive, deducts the fees and transfers the resulting amount to the user.

As described above, the function conditionally updates the price information and profit calculation. When the trade was successful, the event `TokenExchange` is emitted.

2.2.8 Miscellaneous

The code has multiple checks for unsafe parameters. These unsafe parameters were obtained by fuzzing. They serve as sanity checks but don't ensure that all curves created with these parameters are safe.

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 our findings. 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	2

- Fees During Unbalanced Withdrawal
- Oracle Manipulation and the Consequences

Medium-Severity Findings	2

- Potential to Reinitialize Pool Storage
- Trade Fees Can Be Avoided

Low-Severity Findings	8

- Caching Fee Receiver to Save Gas
- Callback Branch Decides on Function Signature Instead of Presence of Callback Address
- Fallback Function Does Not Prevent User Errors
- LP Token burnFrom Variables Are Confusing
- Potential Inability to Trade or Add Liquidity
- Repeated Exponentiation
- State Changes When All Liquidity Is Burned
- Token Amount Calculation Reverts for Empty Pools

5.1 Fees During Unbalanced Withdrawal

Correctness High Version 1

First Finding:

In all other cases where fees are calculated the dynamic fees are calculated based on the resulting state of the pool. However, in case of an unbalanced withdrawal (`remove_liquidity_one_coin`) the fees are determined based on the initial state of the curve. This has the following consequences:

1. It is too cheap to bring the pool from a balanced to an unbalanced state
2. It is too expensive to bring the pool from a unbalanced to an balanced state
3. Removing one coin from a pool gives a higher output than performing a balanced removal and then trading for the one coin, even though those sequences should be roughly equivalent

Second Finding:

During the execution of `remove_liquidity_one_coin` the fee is charged on the change in the invariant `D`:

```
fee: uint256 = self._fee(xp)
dD: uint256 = token_amount * D / token_supply
D -= (dD - (fee * dD / (2 * 10**10) + 1))
```

However, there is a non-linear connection between the change in the invariant `D` and the change in the underlying token that is withdrawn. Hence, when measured in the underlying token the actually charged fee can be significantly below the minimum fee defined as `mid_fee`. As a consequence the pool does not receive those fees.

5.2 Oracle Manipulation and the Consequences

Security High Version 1

The pools use a time-weighted average price as an internal oracle. This oracle can be manipulated relatively easily for the following reasons:

1. It only takes into account the last trade within a block
2. For the last trade it remembers the average price of the trade and not its resulting point on the curve.
3. Due to points 1 and 2 trading to an imbalanced state and back to the start state of the pool, influences the price oracle. This is because only the second trade of that pair is considered and as its average price will significantly differ from the spot price at the start state of the pool.
4. As a consequence of step 3 price oracle manipulations are possible without the risk of being arbitraged and can make use of flash loans as they can be paid back.
5. Such a pair of trades can be injected at the of all blocks where the attacker observes activity for the victim pool.

Hence, overall a fairly cheap manipulation of the price oracle is possible. This is because the attacker will not be arbitraged (when compared to a Uniswap TWAP manipulation) and all regular pool activity can be blocked from influencing the oracle. Hence, a cheap manipulation over a longer time is possible which ultimately results in great control over the time-weighted average price. The only costs are the trading fees. However, as the price oracle does not factor in any trades that happened within the same block, it is impossible to manipulate and benefit within a single block.

Consequences

The `lp_price` function can be used by integrators, such as other DeFi protocols, to measure the value of an LP token, denominated in the first token. However, this value is susceptible to price manipulation. It works as follows:

```
def lp_price() -> uint256:
    return 2 * self.virtual_price * self.sqrt_int(self.internal_price_oracle()) / 10**18
```

Hence, it makes use of the internal price oracle, which is fundamentally a time-weighted average price. As explained above, the internal oracle can be manipulated. Then, the overall attack can take place as follows:

1. Attacker trades back and forth at the end of a block. This will record the exchange price of the second trade for later use inside the oracle. However, this price will not match the final state of the pool, but rather the slope that was used to reach this final state.
2. Attacker waits as time passes and hence the manipulated price rises in importance.
3. Attacker back-runs any other regular trades with fake trades as in step 1.
4. Eventually trigger the code that relies on `lp_price` and benefit from its incorrectness.

5.3 Potential to Reinitialize Pool Storage

Security **Medium** **Version 1**

The pool storage is initialized using the following function:

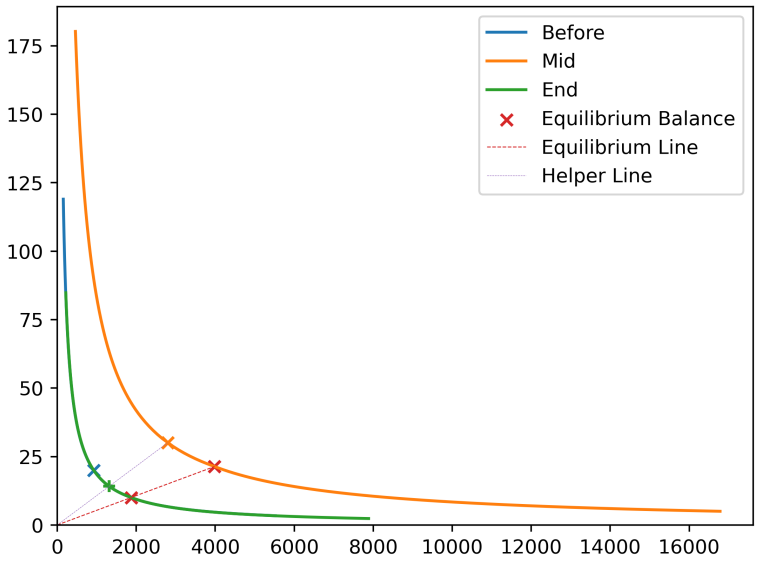
```
@external
def initialize(
    A: uint256,
    gamma: uint256,
    mid_fee: uint256,
    out_fee: uint256,
    ...
):
    assert self.mid_fee == 0  # dev: check that we call it from factory

    self.factory = msg.sender

    ...
```

Hence, `mid_fee` is used to check whether the storage has been initialized already or not. However, `mid_fee` can be set to zero again through `commit_new_parameters` and `apply_new_parameters`. Once that happens, anyone can re-initialize the storage and thereby (among other things) control the admin and the admin fees.

5.4 Trade Fees Can Be Avoided

Correctness **Medium** **Version 1**

If a pool is currently not in the equilibrium state (as defined by `price_scale`), then the trade fees can be avoided when trading towards the equilibrium. This is as follows possible:



We assume that the blue X represents the initial pool state. The attacker can move the pool to the green cross, and hence perform a trade, without paying the trading fees. The steps are:

1. The attacker adds liquidity in the balance as defined by the `price_scale`. Hence the pool state moves to the orange X. During this step only the much smaller `NOISE_FEE` is paid as a fee as it is a balanced addition.
2. The attacker removes liquidity using `remove_liquidity`. Hence, no fee is paid. However, the liquidity is removed according to the current pool balances which are different from the ratio of `price_scale`. Hence, the resulting state is the green cross.

Hence, overall a trade occurred, but the attacker only paid the small `NOISE_FEE` and not the trade fee. As a consequence the pool misses out on those fees. However, it only works if the attacker has access to the tokens needed for the additional liquidity.

5.5 Caching Fee Receiver to Save Gas

Design Low Version 1

Trades, liquidity additions and unbalanced liquidity removals can trigger the execution of `_claim_admin_fees`. As part of that, the fee receiver's address is queried:

```
receiver: address = Factory(self.factory).fee_receiver()
```

This query is fairly gas-expensive, as it involves the penalty of calling a previously unused contract. This penalty could be avoided if the variable was cached inside the pool contract along with a trustless function to update it by querying the factory.

5.6 Callback Branch Decides on Function Signature Instead of Presence of Callback Address

Design Low Version 1

Inside the `_exchange` function of the `CurveCryptoSwap2ETH` contract there is the following code:

```
if callback_sig == EMPTY_BYTES32:
    ...
else:
    b: uint256 = ERC20(in_coin).balanceOf(self)
    raw_call(
        callbacker,
        concat(slice(callback_sig, 0, 4), _abi_encode(sender, receiver, in_coin, dx, dy))
    )
```

The goal is to only execute a callback if the user intended to do so. A user who wants to call a function with the *0x00000000* signature, perhaps to save gas, has to append non-zero bytes after the inital 4 bytes to pass this check, which is counter-intuitive.

5.7 Fallback Function Does Not Prevent User Errors

Design Low Version 1

The default fallback function for the crypto pools accepts any input, including value transfers with ETH. Here, it doesn't differentiate whether the pool has ETH as one of its tokens. Furthermore, it also accepts ETH if calldata is present even though this might indicate an incorrect call, which e.g. was supposed to be a trade, but where the interface was specified with a small mistake.

5.8 LP Token burnFrom Variables Are Confusing

Design Low Version 1

The Curve LP token implementation contains a function:

```
def burnFrom(_to: address, _value: uint256)
```

in which the *_to* parameter specifies the address from which tokens are burned from. This naming is confusing and renaming the parameter to *_from* might make the purpose clearer.

5.9 Potential Inability to Trade or Add Liquidity

Security Low Version 1

In case the following situation ever takes place, the pool becomes "deadlocked":

1. A Ramping is going on
2. All or almost all liquidity is burned

From now on, no more liquidity can be added back and no trades can take place as the computation of `newton_D` and `newton_y` will revert. The only way to release this deadlock is to gift funds and claim admin fees which only works if there is a small amount of LP tokens left.

However, as this scenario only occurs when there are very little funds left, the impact is very small.

5.10 Repeated Exponentiation

Design Low Version 1

The token precisions are calculated upon access through exponentiation:

```
def _get_precisions() -> uint256[2]:
    p0: uint256 = self.PRECISIONS
    p1: uint256 = 10 ** shift(p0, -8)
    p0 = 10 ** bitwise_and(p0, 255)
    return [p0, p1]
```

As only two precisions are stored and as the maximum precision value is *10**18*, the exponentiations can be performed once as part of the initialization. Hence, a slight gas cost saving is possible.

5.11 State Changes When All Liquidity Is Burned

Correctness Low Version 1

In case all liquidity is removed and thereby, all liquidity tokens are burned, then the next `add_liquidity` operation will again trigger the following snippet:

```
self.D = D
self.virtual_price = 10**18
self.xcp_profit = 10**18
```

1. The `xcp_profit_a` variable is not reset. Hence, admin fee are essentially disabled until the previous `xcp_profit` is reached again.
2. The `not_adjusted` variable is not reset. Hence, some gas might be wasted inside the `tweak_price` function.

5.12 Token Amount Calculation Reverts for Empty Pools

Correctness Low Version 1

The function `calc_token_amount` of the swap contract should predict how many liquidity tokens will be received when adding specific amounts of liquidity. However, it is incorrect if queried when the `totalSupply` of the associated token is `0`. In that case, which is true for the first liquidity addition, the function `calc_token_amount` will revert while adding liquidity will work fine.

6 Open Questions

Here, we list open questions that came up during the assessment and that we would like to clarify to ensure that no important information is missing.

6.1 Factory Events

Open Question **Version 1**

1. Do you foresee any situations where the factory's events should be indexed? As an example: Filtering for the event that created a particular token to find initial deployment values?
2. Are you aware that the pool address is not part of the `CryptoPoolDeployed` event?

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Possible Price Manipulations and Inaccuracies

Note Version 1

As stated in a previous report, we see the following price manipulations as possible:

1. Pushing `price_scale` towards `price_oracle`: In case a user wants to perform a larger exchange and the price inside the `price_oracle` is significantly better for that exchange than the price inside `price_scale`, then the user can push the `price_scale` towards `price_oracle` using small trades. This works as the update for `price_scale` only depends on its distance to `price_oracle` and not on previous actions within the same block.
2. The `price_oracle` is only affected by the last price seen in each block. Hence, previous exchanges can be "hidden" from the `price_oracle` if they are followed by other exchanges with a different rate. Note that these trailing exchanges can be way smaller. Such trailing exchanges, if reliably inserted, allow full control over the `price_oracle` and thereby (as mentioned in the previous comment) also over `price_scale`.
3. In low-volume pools the oracle price might be outdated as trades only incur irregularly. Hence, all dependent values might also be outdated.

7.2 Splitting Into Multiple Operations

Note Version 1

Due to the dynamic fee structure, it is beneficial in terms of curve fees to split up an operation into smaller operations in the following cases:

- Trading from one side of the equilibrium to the other side
- Trading away from the equilibrium
- Adding unbalanced liquidity moving the pool further away from the equilibrium
- Adding unbalanced liquidity moving the pool across the equilibrium

In those cases, assuming no changes in `price_scale`, curve fees can be saved by splitting one operation into multiple. This would lead to a loss of fees for the pool. However, in many of those cases the higher transaction costs will outweigh the saved curve fees.

7.3 Supported Tokens

Note Version 1

There is are variety of different token implementations on the Ethereum blockchain. Using tokens with unusual behavior will lead to unexpected changes of the pool or put the smart contracts into a bad state. In particular, the following token types will **not** work:

- rebasing tokens, where balances can change without transfers. These tokens in particular include deflationary tokens and will lead to incorrect accounting
- tokens with transfer fees. These tokens will lead to incorrect accounting
- tokens with incorrect ERC20 implementations
- tokens with more than 18 decimals
- tokens with extremely high total supply. These tokens can lead to arithmetic overflows.