PUBLIC

Code Assessment

of the tricrypto-ng Smart Contracts

June 23, 2023

Produced for

Curve

by

CHAINSECURITY

Contents

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 tricrypto-ng according to Scope to support you in forming an opinion on their security risks.

Curve implements an updated and optimized version of the existing Curve Curve Tricrypto Pool. It is an automatic market maker which allows exchanging of three tokens that do not need to be equivalent in value. The pools are rebalanced continuously to provide the most liquidity around the current price point.

The most critical subjects covered in our audit are overflow checks, the precision of arithmetic operations, and functional correctness. Some issues regarding overflows and precision losses were identified and subsequently fixed. Security regarding these subjects is high.

The general subjects covered are gas efficiency, access control, and trustworthiness. Security regarding all the aforementioned subjects is high. The efficiency of the current price calculation has been significantly improved.

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	1
• Code Corrected	1
Medium-Severity Findings	5
• Code Corrected	5
Low-Severity Findings	13
• Code Corrected	9
• Specification Changed	1
• Code Partially Corrected	1
• Risk Accepted	1
• 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 tricrypto-ng 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	16 February 2023	ec723dd9678eb046791280ff3a3769781f229bfd	Initial Version
2	6 March 2023	acaadf64fa0c209df896964e6716d3ee0384d76a	Second Version
3	2 May 2023	93569cc351c4e13dc2b49334d435212027d71fb8	Third Version
4	14 Jun 2023	5968bbeff84fb2b2cad125b9a2c6bfb7a92d5a72	Fourth Version

For the Vyper smart contracts, the compiler version `0.3.7` was chosen.

The following files were in scope:

- CurveCryptoMathOptimized3.vy
- CurveTricryptoFactory.vy
- CurveTricryptoOptimizedWETH.vy

Note that the assessment was performed with the assumption that only three coins are used for the pool, i.e. `N_COINS == 3` and `N**N == 27`.

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 (Version 1) of the contracts as defined in the Assessment Overview.

Curve offers tricrypto-ng, an updated and optimized version of the Curve Tricrypto Pool, an automatic market maker which continuously rebalances the pool to provide the most liquidity around the current price.

The AMM infrastructure revolves around the following parts:

1. **The factory**

   This allows the permissionless deployment of tricrypto pools, from a blueprint stored in the factory. Pools' information is also stored in the factory.

2. **The Tricrypto swap contract**

   AMM tricrypto pools deployed from the factory are an instance of this blueprint.

3. **The Math contract**

   This is a stateless library contract that implements the mathematical operations used by tricrypto pools.

4. **The Views contract**

   It is a stateless library contract that implements utility methods to query the state of a tricrypto pool.

5. **The Liquidity Gauge Blueprint contract**

   Liquidity Gauges deployed through the factory are instances of this contract.

2.2.1 The Factory

The factory introduces a permissionless way to deploy 3-coin cryptoswap (Tricrypto) pools. Its `deploy_pool()` method deploys a tricrypto pool, the `CurveTricryptoOptimizedWETH.vy` contract, and allows setting its parameters. It helps the user with operations such as packing the 3 fee parameters into a single `uint256`, packing the rebalancing parameters, and packing the initial prices. Moreover, it implements validation of the parameters. The pool is deployed through Vyper's built-in method `create_from_blueprint()`, which executes the `CREATE` opcode using code from a specific address as initialization bytecode. The address of the newly deployed pool is appended to the `pool_list`, and information about the pool (its coins, and their respective number of decimals) is stored in `pool_data[pool]`. Keys computed as the `xor` of each pair of pool coins are computed and used as keys in the `markets` hashmap to find the pool based on two of its coins.

A permissionless method `deploy_gauge()` allows deploying a liquidity gauge for the pool, through Vyper's `create_from_blueprint()`.

The factory has an admin who can set the admin fee receiver of every pool. The admin fee is 50% of all fees collected in the pool. The admin can set the pool implementation and the gauge implementation, which changes the blueprints that are used in later deployments of pools. The admin can set the Views contract in the factory. Users of Curve and external integrators should access the Views contract through the factory so that it can be updated by the admin should it be necessary. Finally, the admin has the power to change the parameters of every pool deployed through the factory and to set rewards on the liquidity gauges.

2.2.2 The Tricrypto swap

`CurveTricryptoOptimzedWETH.vy` implements the Curve 3-coin CryptoSwap. It features some changes and improvements over the *previous implementation <https://etherscan.io/address/0xd51a44d3fae010294c616388b506acda1bfaae46>* of the 3-coin CryptoSwap.

2.2.3 Tricrypto-ng improvements over Tricrypto V1

1. Handles native Ether
2. Optimizes gas usage through the use of unchecked mathematical operations
3. Improves the initial guesses used in Newton's method to compute the `D` value of the invariant
4. Improves swap performance by using a closed-form solution for the outgoing tokens, instead of Newton's method
5. Implements reentrancy locks for read operations for added security
6. Exposes getters for the oracle price which compute the real oracle price instead of returning the cached one
7. Introduces callback-based exchanges, improving the efficiency of Zap contracts
8. Includes the ERC20 functionality of the liquidity token in the contract itself

9. Implements the ERC-2612 permit extension, allowing signed approvals for the liquidity token
10. Removes the factory admin's power to kill the exchange
11. Uses the derivative of the invariant to update the price oracle, instead of using the price of the last exchange, which could be tampered with.

2.2.4 Tricrypto's Curve

The principle of operation of the Tricrypto swap is to keep the balances of the three coins satisfying a mathematical invariant of the form `F(x,y,z, D) = 0`, where `x, y, z` are the balances and `D` represents the total liquidity. The invariant defines a "curve" over which the balances of the coins are allowed to change. Depending on the nature of the operation being executed, different manipulations of this curve will happen.

Whenever a swap happens, the inbound coin balance (e.g. `x`) in the pool increases. To maintain the invariant, one of the other two coins' balance must change (e.g. `y`). Therefore, if `x'` is the new balance of the pool, the output amount will be the difference between `y - y'`, which is the value that satisfies the equation `F(x',y',z,D) == 0`. Executing a swap, therefore, consists of solving the invariant for the new value of the output coin. Solving for `y` is reduced to the solution of a cubic polynomial, and is performed in the math library (CurveCryptoMathOptimized3.vy) by the `get_y` function.

Changing the liquidity of the pool likewise reduces to computing the new parameter `D` after changing the value of the three coin balances, so solving `F(x',y',`z',D) == 0` for `D`. Solving for `D` is implemented as an iterative method in the math library in function `newton_D()`.

Every swap deduces a fee from the outgoing amount, which is retained in the pool, so the new `y` balance is higher than the solution of the invariant. This is equivalent to a change in liquidity, which is why `D` is recomputed after every swap.

The price of the coins in the pool can be computed as the derivative of one coin's balance with respect to another in the invariant, or equivalently the ratio between an infinitesimal input of a coin and the infinitesimal output received in the absence of fees. The coins' prices feed an internal price oracle, which keeps track of the price evolution in a smooth and delayed way. The price oracle is used to reshape the curve so that the flat area of the curve (where slippage is minimal for users) tracks the market price. The evolution of the shape of the curve is delayed and smoothened to avoid price manipulations that would allow profitable sandwiching attacks against the liquidity providers. The shape of the curve is also locked as long as the liquidity providers are not realizing a profit, according to the `xcp_profit` metric.

2.2.5 User interactions with the Tricrypto swap

Liquidity providers, exchangers, and the admin are expected to interact with the pools. Liquidity providers can supply liquidity as any amount of the three coins. In exchange, they receive liquidity tokens, which are implemented as an ERC20 token in the same contract. To increase the liquidity of the pool, the method `add_liquidity` is to be used. A fee is deducted from the minted liquidity tokens according to how unbalanced the liquidity is supplied. If the liquidity is supplied proportionally to the internal price of the tokens, a minimal fee is deducted. The fee increases as the supplied token values get more unbalanced. Liquidity providers can withdraw liquidity through the methods `remove_liquidity` and `remove_liquidity_one_coin`. `remove_liquidity()` withdraws the liquidity in a balanced way, by burning the liquidity tokens provided, and transferring an amount of each of the three coins proportional to the burned liquidity w.r.t. the total supply. No fee is applied, and the `D` parameter of the pool is simply reduced proportionally to the withdrawn liquidity. `remove_liquidity_one_coin()` behaves similarly to removing the liquidity in a balanced way, and then trading the two other coins for the third. A fee is applied according to how unbalanced the pool is. Liquidity providers retain a fraction of every fee that is collected through the pool's methods. Half of the total fees belong to the pool admin, and the other half to the liquidity providers. Fees are automatically reinvested as additional liquidity in the pool. Exchangers interact with the pool through `exchange()`, `exchange_underlying()`, or `exchange_extended()`. An amount of one of the three coins is transferred in, and an equal value of another coin, minus a fee, is transferred out. For values in Ether, either native ETH or WETH can be used. `exchange_extended()` allows transferring the input value during a callback to the exchange initiator. Slippage protection should be specified to avoid frontrunning. After the exchange, the pool recomputes a new `D` value, since the fee is deducted from the output amount, and therefore reinvested as liquidity in the pool. Another permissionless entrypoint worth of note is `claim_admin_fees()`, which updates the coin balances with any amounts of coins held by the contract but not accounted for in `self.balances`, distributes part of the collected fees to the admin, in the form of freshly minted liquidity tokens, and recomputes `D` to account for the new balances.

Finally, the pool's admin (who is also the factory's admin), can change the pool parameters. Changes to `A` and `gamma` are gradually applied, through the ramp process. The `A` and `gamma` parameters are linearly interpolated in time from their initial value, to reach their target value at the target time. This restricts the Curve from changing quickly, preventing profitable sandwiching strategies that would hurt liquidity providers. `commit_new_parameters()` allows setting new values for the fee parameters, and the Oracle update parameters. The new values must be committed first, then applied by the admin after a minimum wait of 3 days.

2.2.6 The Math contract

To reduce the size of the swap contract the mathematical operations have been factored in the `CurveCryptoMathOptimized3.vy` contract. The external functions that it exposes that are necessary for pool operation are:

1. `get_y()`

   computes the amount of token `i`, given the balances of the other tokens and the value of `D`. This has been upgraded from the previous version to save 75% gas through the use of a closed-form solution, reducing the calculation to finding a root of a cubic polynomial. A fallback using Newton's method is present.

2. `newton_D()`

   The new value of `D` is computed given the pool's parameters and its balances. An initial value can be specified for the iterative Newton-Halley method.

3. `get_p()`

   computes the prices of each token with respect to the first one. This is implemented as taking the partial derivative of the implicit function defined by the invariant.

4. `cbrt()`

implements the cube root as a fixed number of iterations of Newton's algorithm. An initial value is computed using one-third of the logarithm.

5. `geometric_mean()`

   computes the geometric mean of three 1e18 precision numbers.

6. `reduction_coefficient()`

   computes the coefficient that interpolates the fee from the `mid_fee` value for balanced pools to the `out_fee` value for unbalanced pools.

7. `wad_exp()`

   implements natural base exponentiation in 1e18 precision. Used in the exponential moving average for the internal price Oracle

The mathematical operations have been reworked to make use of unchecked math, which saves a considerable amount of gas.

2.2.7 Trust model

There are a few users and contracts that are trusted to behave honestly and fairly. They are:

- **Factory Admin:**

  They control both the factory and the settings for all pools deployed by the factory. They are trusted to set favorable parameters for the users of the various pools. They are also trusted not to list malicious pool-, gauge-, and view implementations in the factory contract. Lastly, they are trusted not to transfer ownership of the factory to a malicious actor.

- **Pool Deployer:**

  The deployer of a pool generally has free reign over the parameters set in that pool, although they must be within certain ranges. However, note that it is possible for a user to deploy a pool that interacts with a malicious token. Given that any user can deploy a pool, caution should be taken when interacting with pools deployed by untrusted users. For this assessment, we have assumed that the pool deployers can be trusted not to deploy pools that interact with malicious tokens.

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:

- 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	3

5.1 Inefficient Defaulting to _newton_y

Design Low Version 1 Acknowledged

CS-TRICRYPTO-NG-001

The analytical solution implemented in `get_y` defaults back to the iterative `_newton_y` in the following situation:

```
if sqrt_arg > 0:
    sqrt_val = convert(isqrt(convert(sqrt_arg, uint256)), int256)
else:
    return [self._newton_y(_ANN, _gamma, x, _D, i), 0]
```

However, this means that the `_newton_y` starts over from scratch and has to recalculate everything from the initial values. Instead, a new method could be written that uses the existing values for `a`, `b`, `c`, and `d` which calculates `K0` using Newton's method to solve the equation:

$$aK_0^3 + bK_0^2 + cK_0 + d = 0$$

Then, the value for `y` could be determined from this result. This way, the `get_y` function can return a useful value for `K0` instead of just defaulting to `0`. This value can then be used as an initial guess for the next call to `newton_D`, saving further gas in the future.

Acknowledged:

Defaulting to `_newton_y()` is rare when running the new code on historic tricrypto data, so Curve accepts the risk of incurring more gas costs in rare edge cases.

5.2 Typo in Event, Unused Variables

Design | Low | Version 1 | Code Partially Corrected

CS-TRICRYPTO-NG-002

Event `UpdatePoolImplementation` in `CurveTricryptoFactory` has first argument called `_implementtion_id`. Field `token` in struct `PoolArray` of `CurveTricryptoFactory` is unused. Argument `calc_price` of `_calc_withdraw_one_coin()` is unused.

Code partially corrected:

The `token` field of the `PoolArray` struct was removed. The `calc_price` argument was removed from the `_calc_withdraw_one_coin()` function.

The first argument of the `UpdatePoolImplementation` event was changed to `_implemention_id`, which is still spelled incorrectly.

5.3 CREATE in Pool Deployment Could Reuse Addresses on Different Chains

Security | Low | Version 1 | Risk Accepted

CS-TRICRYPTO-NG-003

If the address of the pool factory is the same on two blockchains, then the deployment addresses of pools will match on different chains, even if the pool parameters are different (different coins). This can result in user mistakes or scam attempts.

Risk accepted:

Curve accepts the risk of pool contracts on different chains having the same address.

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	1

- Loss of Precision in get_p() for Some Values of A Code Corrected

Medium-Severity Findings	5

- First Depositor Can Manipulate the Share Value to Steal Future Deposits Code Corrected
- Safety Parameters Differ Between Factory, Swap, and Math Contract Code Corrected
- Simpler Price Calculations Code Corrected
- Unsafe Operations Code Corrected
- _log2() Returns Incorrect Results Code Corrected

Low-Severity Findings	10

- Admin Can Set Unsafe Parameters Through commit_new_parameters() Specification Changed
- Fee on remove_liquidity_one_coin() Is Computed on Initial Balance Code Corrected
- Incomplete Validation of Coins in Factory Code Corrected
- Initial Value K0_prev Recalculated Needlessly Code Corrected
- Magic Number 10000 Used Instead of Constant A_MULTIPLIER Code Corrected
- Math Implementation Cannot Be Upgraded in the Factory Code Corrected
- No Getter for Length of Markets List in Factory Code Corrected
- Pool Registered Twice in the Markets List for Each Key Code Corrected
- Possible Precision Loss in get_y Code Corrected
- Redundant Asserts in Call to _newton_y() Code Corrected

6.1 Loss of Precision in `get_p()` for Some Values of A

Correctness High Version 1 Code Corrected

CS-TRICRYPTO-NG-014

Line `851` of `CurveCryptoMathOptimized3.vy` performs a division of `ANN` by `10000`:

```
unsafe_div(ANN, 10000)
```

Value `ANN` ranges from `2700` to `270000000`. The division can incur a substantial loss of precision that affects the return value of `get_p()`. With `ANN = 1707629`, the current USDT/WBTC/WETH A value, a price error of close to 1% is returned by `get_p()`

Code corrected:

The order of operation has been modified so that the division by `10000` is performed when the denominator has sufficient precision. The relative loss of precision on the `c` coefficient is now at most of `1e-5`.

6.2 First Depositor Can Manipulate the Share Value to Steal Future Deposits

Security Medium Version 1 Code Corrected

CS-TRICRYPTO-NG-004

A malicious user can mint a single wei of shares before any deposit exists, then increase the price of the single share through a direct transfer to the pool followed by calling `claim_admin_fees()`, which sweeps unaccounted tokens and recomputes `D`. The next depositors will suffer severe rounding errors on the number of shares they receive.

The shares distributed for the next deposits are calculated according to

```
d_token = token_supply * D / old_D - token_supply
...
d_token -= 1
```

Since `token_supply` will be `1`, if `D` is between `2*old_D` and `3*old_D`, the tokens received by the victim will round down to zero, but their deposit will still be transferred to the pool. `old_D` is under complete control of the attacker, who can steal legitimate deposits by investing half of the deposit value.

Code corrected:

The share value manipulation was enabled by being able to call `claim_admin_fees()` to increase significantly the value of single shares, when the total supply is low. `claim_admin_fees()` now will not gulp tokens when the total supply is below `10**18`. This makes the attack unfeasible, while not affecting general operation, since the total supply in normal conditions will be in the order of magnitude of the `D` parameter, which is between 10**17 (generally more) and 10**33.

6.3 Safety Parameters Differ Between Factory, Swap, and Math Contract

Design Medium Version 1 Code Corrected

CS-TRICRYPTO-NG-016

Safety bounds on pools parameters are different in the factory and the math contract.

Some are more restrictive in the factory:

1. `MAX_GAMMA` is 2*10**16 in the factory and swap, and 5*10**16 in MATH
2. `MIN_A` is 27000 in the factory and 2700 in MATH and swap

Some are less restrictive in the factory, which may lead to the deployment of invalid pools:

1. `MAX_A` is 27*10**9 in the factory, but 27*10**7 in MATH and swap

Code corrected:

1. `MAX_GAMMA` is `5*10**16` across all contracts.
2. `MIN_A` is 2700 across all contracts.
3. `MAX_A` is `27 * 10**7` across all contracts.

6.4 Simpler Price Calculations

Design | Medium | Version 1 | Code Corrected

CS-TRICRYPTO-NG-017

The derivation of the price calculations leads to more expensive calculations than necessary. The gas costs of the `get_p` can be greatly reduced by simplifying the formula for the price. For example, by defining the value `G` as follows:

$$G \cdot K_0 = 2{K_0}^3 - {K_0}^2(2\gamma + 3) + (\gamma + 1)^2$$

The formula for the price of y with respect to x becomes:

$$p_y = \frac{x}{y} \cdot \frac{G \cdot K_0 + N^N A \gamma^2 K_0 \frac{y}{D}}{G \cdot K_0 + N^N A \gamma^2 K_0 \frac{x}{D}}$$

An efficient implementation of this formula can reduce the costs of the price calculation by around 66%.

Code corrected:

The suggested formula was implemented in `get_p`. The `_snekmate_mul_div` function was removed as it was no longer used.

6.5 Unsafe Operations

Correctness | Medium | Version 1 | Code Corrected

CS-TRICRYPTO-NG-019

Some multiplications in the `get_y` function are performed using `unsafe_mul`. However, several of these can potentially overflow:

1. The following multiplication in the calculation of b can overflow:

```
unsafe_mul(unsafe_mul(unsafe_div(D**2, x_j), gamma**2), ANN)
```

   For example with the following values:

   `D=10**33`, `x_j=10**31`, `gamma=5*10**16`, `ANN=2.7*10**8`

   In this case, the result is greater than `2**255` and hence overflows the `int256` type.

   Code corrected:

   The outermost `unsafe_mul`, where the second factor is `ANN`, which could cause an overflow, has been replaced with a safe multiplication.

2. This multiplication occurs when calculating `delta1`:

```
unsafe_mul(9, a * c)
```

It can overflow when `2**255 / 9 < a*c < 2**255 / 3`. Previously, only the multiplication of `3a * c` is done using overflow checks.

Code corrected:

The expression is now evaluated as `3 * (unsafe_mul(3, a) * c)`, which is safe.

3. Again in the calculation of `delta1`:

```
unsafe_mul(27, a**2)
```

This can overflow when `a**2` is close to `2^255`, but not greater. For example, this can occur when b is very close to zero.

Code corrected:

The expression has been replaced with `27 * a**2`, which is safe.

4. Lastly, the following multiplication in the calculation of `sqrt_arg` could potentially overflow when `delta0**2` is close to `2^255`:

```
unsafe_mul(4, delta0**2)
```

Code corrected:

The expression has been replaced with `4 * delta0**2`, which is safe.

6.6 `_log2()` Returns Incorrect Results

Correctness | Medium | Version 1 | Code Corrected

CS-TRICRYPTO-NG-009

Results of function `_log2()` in `CurveCryptoMathOptimized3` are off by one.

Example:

```
In [2]: math.log2_(2**1)
Out[2]: 0

In [3]: math.log2_(2**2)
Out[3]: 1

In [4]: math.log2_(2**130)
Out[4]: 129

In [5]: math.log2_(2**255)
Out[5]: 254

In [6]: math.log2_(2**256-1)
Out[6]: 254
```

The only values for which a correct result is produced are `x = 0`, and `x in [2**128, 2**129-1]`

```
In [7]: math.log2_(2**0)
Out[7]: 0

In [8]: math.log2_(2**128)
```

```
Out[8]: 128

In [9]: math.log2_(2**129-1)
Out[9]: 128
```

Code corrected:

The custom `_log2()` implementation has been replaced with Snekmate `log_2()`. The new implementation is correct, except for the value of `log2(0)`, which evaluates to `0` but which ought to be undefined. In the context where `_snekmate_log_2()` is used, which is evaluation of the cube root, returning `0` for `log2(0)` leads to the correct result.

6.7 Admin Can Set Unsafe Parameters Through `commit_new_parameters()`

Design | Low | Version 1 | Specification Changed

CS-TRICRYPTO-NG-005

The same bounds are not applied when setting parameters at initialization or with `commit_new_parameters()`.

`mid_fee` can be set down to 0 through `commit_new_parameters()`, but must be at least `MIN_FEE` in `deploy_pool()`.

`allowed_extra_profit` can be set to values between `10**16` and `10**18` through `commit_new_parameters()`, but it can be at most `10**16` with `deploy_pool()`.

Specification changed:

The `MIN_FEE` check has been removed from the factory. Max value for parameter `allowed_extra_profit` has been increased from `10**16` to `10**18`

6.8 Fee on `remove_liquidity_one_coin()` Is Computed on Initial Balance

Correctness | Low | Version 1 | Code Corrected

CS-TRICRYPTO-NG-015

The fee for `remove_liquidity_one_coin()` is computed in `_calc_withdraw_one_coin()` at line `1349` as

```
fee: uint256 = self._fee(xp)
```

At this point, `xp` is still the unchanged balance of the pool. Removing liquidity with one coin from a perfectly balanced pool, and making it unbalanced, will ask for `mid_fee`. Making an unbalanced pool balanced by removing liquidity will ask for `out_fee`. This is the opposite of what should happen.

Code corrected:

A rough but gas inexpensive calculation of the resulting balance is performed, for the purpose of calculating the fee. The fee calculation is not exact but more accurate than in the previous version.

6.9 Incomplete Validation of Coins in Factory

Security Low Version 1 Code Corrected

CS-TRICRYPTO-NG-008

Coins in a pool shouldn't be duplicated, the following line in `CurveTricryptoFactory.vy` asserts it:

```
assert _coins[0] != _coins[1] and _coins[1] != _coins[2], "Duplicate coins"
```

However, the case where `coins[0] == coins[2]` is not covered. Therefore, a pool could be deployed with the same coin listed twice.

Code corrected:

The missing check has been included.

6.10 Initial Value `K0_prev` Recalculated Needlessly

Design Low Version 1 Code Corrected

CS-TRICRYPTO-NG-018

The value `K0_prev` is used to compute an initial value for `newton_D()`. In `_exchange()`, `K0_prev` is first computed during the call to `MATH.get_y()`, but is discarded and the same value is recomputed a few lines later in `MATH.get_K0_prev()`. This is unnecessary since the same value is returned during both calls.

The method `get_K0_prev()` of `CurveCryptoMathOptimized3` is redundant.

Code corrected:

The `K0_prev` value obtained from `MATH.get_y()` is now used. The `get_K0_prev` function was removed.

6.11 Magic Number 10000 Used Instead of Constant `A_MULTIPLIER`

Design Low Version 1 Code Corrected

CS-TRICRYPTO-NG-010

Despite the constant `A_MULTIPLIER` being defined, code in `CurveCryptoMathOptimized3.vy` at lines `737`, `766`, `835`, `851` uses the magic number `10000` directly.

Code corrected:

The magic numbers have been replaced with the constant.

6.12 Math Implementation Cannot Be Upgraded in the Factory

Design Low Version 1 Code Corrected

CS-TRICRYPTO-NG-011

New pool implementations can be deployed in the factory, but the math implementation can't be changed. The event `UpdatePoolImplementation` is unused. A new pool implementation using another math contract could still be added to the factory, by changing the hardcoded value of the math contract in the pool implementation's constructor, instead of receiving it from the factory.

Code corrected:

Function `set_math_implementation` has been introduced in the factory so that the admin can change the math implementations of newly deployed pools.

6.13 No Getter for Length of Markets List in Factory

Design Low Version 1 Code Corrected

CS-TRICRYPTO-NG-012

Private variable `self.market_counts` does not have a getter. The only way to know how many pools have been deployed for a coin pair is to iterate `find_pool_for_coins()` until a zero value is returned.

Code corrected:

Public function `get_market_counts` has been introduced to return the market count for a token couple.

6.14 Pool Registered Twice in the Markets List for Each Key

Correctness Low Version 1 Code Corrected

CS-TRICRYPTO-NG-013

The following logic includes pools in the `self.markets[key]` list of the factory:

```
for coin_a in _coins:
    for coin_b in _coins:

        if coin_a == coin_b:
            continue

        key: uint256 = (
            convert(coin_a, uint256) ^ convert(coin_b, uint256)
```

```
        length = self.market_counts[key]
        self.markets[key][length] = pool
        self.market_counts[key] = length + 1
```

Each coin pair is iterated twice, first as `(A,B)` and then as `(B,A)`. The keys for the two pairs are the same. As a consequence, each pool is included twice for a certain key.

Code corrected:

The code has been refactored so that the three token couples are now individually added.

6.15 Possible Precision Loss in `get_y`

Design | Low | Version 1 | Code Corrected

CS-TRICRYPTO-NG-007

In the `get_y` function, additional precision is added conditionally:

```
d0: int256 = abs(unsafe_mul(3, a) * c / b - b)  # <------------ a is smol.

divider: int256 = 0
if d0 > 10**48:
    divider = 10**30
elif d0 > 10**44:
    divider = 10**26
elif d0 > 10**40:
    divider = 10**22
elif d0 > 10**36:
    divider = 10**18
elif d0 > 10**32:
    divider = 10**14
elif d0 > 10**28:
    divider = 10**10
elif d0 > 10**24:
    divider = 10**6
elif d0 > 10**20:
    divider = 10**2
else:
    divider = 1

additional_prec: int256 = 0
if abs(a) > abs(b):
    additional_prec = abs(unsafe_div(a, b))
    a = unsafe_div(unsafe_mul(a, additional_prec), divider)
    b = unsafe_div(b * additional_prec, divider)
    c = unsafe_div(c * additional_prec, divider)
    d = unsafe_div(d * additional_prec, divider)
else:
    additional_prec = abs(unsafe_div(b, a))
    a = unsafe_div(unsafe_mul(a, additional_prec), divider)
    b = unsafe_div(b * additional_prec, divider)
    c = unsafe_div(c * additional_prec, divider)
    d = unsafe_div(d * additional_prec, divider)
```

However, there are some cases where `divider > additional_prec` and a precision loss occurs instead. For example, when `b ≈ a`, `divider` can still be as large as `10**18`, but `additional_prec` will be `1`. Therefore, up to 18 decimals are removed from `a`, `b`, `c` and `d`, resulting in a precision loss.

It should be considered whether it is necessary to adjust the decimals in the case where `divider > additional_prec`.

Code corrected:

The additional precision calculations were incorrect in the original version. The else branch has been updated to the following:

```
else:
    additional_prec = abs(unsafe_div(b, a))
    a = unsafe_div(a / additional_prec, divider)
    b = unsafe_div(unsafe_div(b, additional_prec), divider)
    c = unsafe_div(unsafe_div(c, additional_prec), divider)
    d = unsafe_div(unsafe_div(d, additional_prec), divider)
```

Curve also provided an explanation for the precision adjustment:

> The idea behind this is that a is always high-precision constant `10**36 / 27` while b, c, and d may have excessive or insufficient precision, so we compare b to a and add or remove precision via `additional_prec`. But we should also take into account not only difference between a and other coefficients, but their value by themselves (`10**36` precision will lead to overflow if coin values are high), so we use `divider` to reduce precision and avoid overflow. The `divider > additional_prec` case is fine unless it produces vulnerability.

6.16 Redundant Asserts in Call to `_newton_y()`

Design | Low | Version 1 | Code Corrected

CS-TRICRYPTO-NG-006

The arguments of `get_y()` are checked to be in a reasonable range through the following asserts:

```
# Safety checks
assert _ANN > MIN_A - 1 and _ANN < MAX_A + 1, "dev: unsafe values A"
assert _gamma > MIN_GAMMA - 1 and _gamma < MAX_GAMMA + 1, "dev: unsafe values gamma"
assert _D > 10**17 - 1 and _D < 10**15 * 10**18 + 1, "dev: unsafe values D"
```

The same checks are duplicated when entering the internal function `_newton_y()`, which is only called in the body of `get_y()`

Code corrected:

The redundant asserts were removed from `_newton_y()`.

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 Funds Could Be Transferred Before Callback

Note Version 1

When using `exchange_extended()`, a callback to the caller is executed to transfer the inbound exchange amount. The callback is executed before the outgoing tokens are received by the user. Executing the callback after the outgoing tokens have been received would allow more flexible use cases, by acting as a flashloan.