PUBLIC

Code Assessment

of the Swaap Core V1

Smart Contracts

May 10, 2022

Produced for

Swaap

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Swaap.finance,

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 Swaap Core V1 according to Scope to support you in forming an opinion on their security risks.

Swaap Labs implements an automated market maker protocol, with the intention to eliminate the losses of the liquidity providers, while enabling them to collect the fees from trades. This is achieved by dynamic weighting of the underlying tokens and stochastic spread mechanism.

During the review, no critical issues were uncovered. All the uncovered issues have been mitigated or fixed.

The most critical subjects covered in our audit are resistance to assets siphon attacks, stochastic process simulation precision and integration with external systems. Security regarding all the aforementioned subjects is high.

The general subjects covered are trust model, functional correctness and specification quality. All the aforementioned subjects were of sufficient quality.

In summary, we find that the codebase provides a good 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	2
• Code Corrected	2
Medium-Severity Findings	4
• Code Corrected	4
Low-Severity Findings	7
• Code Corrected	6
• Specification Changed	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 following source code files inside the Swaap Core V1 repository: based on the documentation files.

- `interfaces/IAggregatorV3.sol`
- `interfaces/IERC20.sol`
- `structs/Struct.sol`
- `Const.sol`
- `Factory.sol`
- `GeometricBrownianMotionOracle.sol`
- `LogExpMath.sol` (comes from Balancer v2)
- `Math.sol`
- `Migrations.sol`
- `Num.sol` (only "bdivInt256" function)
- `Pool.sol`
- `PoolToken.sol`

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	February 7 2022	72bb2e1cae7710db7f29660ec4a0b20abd5c02c5	Initial Version
2	April 21 2022	270d192f1ad51baeaadf503d91f55f94a682af52	Version with fixes
3	April 27 2022	a19172410188513f588c48bff4055bf777ed11e2	Version with fixes
4	May 10 2022	ee3c5e8bb0efffeb14af38183a395cae3ba022fc	Version with fixes

For the solidity smart contracts, the compiler version `0.8.12` was chosen.

2.1.1 Excluded from scope

All functions from the `Num.sol` file are excluded from the scope, except the `bdivInt256` function.

3 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.

The Swaap Core V1 is an automated market maker, that functions as a self-balancing weighted portfolio, based on Balancer Pool implementation. The novelty of the Swaap Core V1 is a way it dynamically mitigates the impermanent loss of the liquidity providers. First, the weight of the tokens in the pool can be changed dynamically, based on the performance of the asset. Secondly, the geometric Brownian motion (GBM) method is used to simulate the asset prices at a certain time horizon during each trade, and the trade price might be adjusted to cover the potential losses of the liquidity providers. These 2 solutions are intended to provide low fee exchanges, while preventing the LPs losses.

3.1 Contracts

3.1.1 `Pool.sol`

This is the contract, where the main functionality of the system lives. It allows LPs to join and exit pools, by providing the assets for the pool. By utilizing the provided liquidity, users can perform swaps via `swapExactAmountInMMM` function. The Controller is a privileged role inside this contract. The holder of this role can:

- Set swap fees
- Bind and unbind tokens from the pool
- Set GBM horizon and lookback (rounds and seconds) parameters
- Make the Pool public
- Make the pool final
- Assign new Controller

Once pool is final, the fees and lookback params cannot be changed. Tokens cannot be added or removed. The LPs can join and exit pool only if pool is final.

The Exit fee of the pool is constant, and cannot be changed.

3.1.1.1 *Detailed Pool overview*

Main roles in the pool contract are: controller, traders and liquidity providers (LPs). The role holders can perform certain actions on the pool:

- controller:
  - `setSwapFee`: update the `swapFee` within the `[0.0001%, 10%]` range. Its default value is `0.025%`. The pool must not be `finalized`.
  - `setController`: current `controller` can give the control to another address. The new `controller` cannot be `address(0)`.
  - `setPublicSwap`: allow/disallow token swaps before the pool is `finalized`. The pool must not be `finalized`.
  - `finalize`: set `_finalized` to true and allow token swaps. The pool must not be `finalized`.
  - `setDynamicCoverageFeesZ`: set the coverage value `z` for the computation of the spread factor within the `[0, 4]` range, which corresponds to `[0.5, > 0.99998)` in the cumulative range for a normally distributed random variable. Its default value is `0.6` (around `0.8` in the cumulative). The pool must not be `finalized`. User should be aware that `z` represents only the inverse of `erf(2p-1)` and not the inverse of `PHI(p)`, the `sqrt(2)` factor is already included in the implemented formula.
  - `setDynamicCoverageFeesHorizon`: set the time horizon for the price prediction within the `[1*BONE, 86400*BONE (24h)]` range. Its default value is `300*BONE (5min)`. The pool must not be `finalized`.

  - `setPriceStatisticsLookbackInRound`: set `lookbackInRound` parameter within the [1, 100] range. The pool must not be `finalized`.
  - `setPriceStatisticsLookbackInSec`: set `lookbackInSec` parameter within the [1, 86400 (24h)] range. The pool must not be `finalized`.
  - `rebindMMM`: update the denormalized weight and balance of an already bounded token. If the `balance` parameter is greater than the actual token balance, the difference is pulled from the `controller`. If the `balance` is lower than the actual balance, the difference is transferred to the `controller` The denormalized weight of the token must be within the `[BONE, BONE * 50]` range. The token must be bounded to the pool and the pool must not be `finalized`.
  - `bindMMM`: bind a new token to the pool, the `controller` must provide at least `10**6` tokens to the pool. The denormalized weight of the token must be within the `[BONE, BONE * 50]` range. The token must not be bounded to the pool and the pool must not be `finalized`.
  - `unbindMMM`: remove a bounded token from the pool. The amount of remaining tokens, is transferred to the `controller`. The token must be bounded to the pool and the pool must not be `finalized`.
- traders:
  - `swapExactAmountInMMM`: traders have to specify the input token and its input amount, the output token and its minimum output amount, as well as the maximum spot price (without shortage penalty) they are willing to pay. The system will retrieve the last price from a Chainlink price feed for both tokens to adjust the weight of each token, relative to their performance since the pool's inception. The formula for the weight update is: `w_0 * price_t / price_0`, where `w_0` and `price_0` are the weight and price of the token at pool's inception and `price_t` is the last queried price. The weight update is made to account for relative price changes, so each weight corresponds to its true share of value in the pool.

  The equilibrium quantity will be computed, it represents the amount of input tokens there should be in the pool to achieve the oracle price. If the current balance of input token is > (resp. <) equilibrium quantity it means that the input token is in abundance (resp. in shortage) and thus the output token is in shortage (resp. in abundance). The pool will then compute how many tokens it can swap, based on the equilibrium quantity, it will consider three cases:

    1. output token is already in shortage: penalty on the whole amount
    2. output token is in abundance, but will be in shortage after the trade: penalty on the amount taking the balance past the equilibrium quantity
    3. output token is in abundance before and after the trade: no penalty

  The penalty is applied to the output token's weight in order to increase its swap price to cope for impermanent loss. The penalty is the maximum between `1` and a pessimistic sample of a random variable following a lognormal distribution of parameters `((mu - s^2 / 2) * h, h * s^2)`, the drift (`mu`) and volatility (`s^2`) are provided by the `GeometricBrownianMotionOracle`. The penalty multiplies the updated output token's weight as to predict the price at time horizon h, starting from the current spot price. If not enough or no historical data at all is available, no penalty is given and there is a risk of impermanent loss.

  In order to mitigate the risk of oracle price update sandwich attack, two mechanisms work together:

    1. the increase of the ratio of price of the output token in the pool compared to oracle price is capped to limit the output token price increase
    2. the swap fee is computed such that the pool does not lose value in terms of output token after the swap

  This special swap fee is added under the conditions that the input token is in shortage and experienced a relatively increasing oracle price update within the current block.

- liquidity providers:
  - `joinPool`: LP can specify the amount of LP tokens they want to get, as well as the maximum amount of each token they want to provide. The amount of LP tokens represents the shares in the pool. The LP tokens are minted and transferred to the LP The `maxAmountsIn` ordering must match the `_tokens` array ordering. The pool must be `finalized`. To mitigate just-in-time liquidity provision, once a LP has joined the pool, they must wait `BLOCK_WAITING_TIME` blocks to be able to move (exit or transfer) their LP tokens.
  - `joinPoolForTxOrigin`: like `joinPool` but the LP tokens will be sent to `tx.origin` instead of `msg.sender`.
  - `exitPool`: LP can specify the amount of LP tokens they want to give back in exchange for their share in bounded tokens, as well as the minimum amount of each token they want back. The LP tokens are transferred from the LP to the pool and burned. The `minAmountsOut` ordering must match the `_tokens` array ordering. The pool must be `finalized`.
- users:
  - `gulp`: for a given bound token, sync the pool's accounting with its token balance

Each pool can support up to 8 assets, minimum is 2. The tokens must not take fees upon transfer, otherwise the pool's accounting will be wrong.

3.1.2 `GeometricBrownianMotionOracle.sol`

The `GeometricBrownianMotion` library can provide historical prices statistics, mainly via its `getParametersEstimation`. This function will estimate the drift and volatility of Geometric Brownian Motion process, based on the lookback window data. Then the swap price will be adjusted, based on the estimated price at horizon. The historical prices and associated timestamps are provided by a Chainlink price aggregator. When tokens are bind to the pool, the controller need to provide an address of the Chainlink oracle, that will be used for price estimations. The estimations sequence will contain data coming from at most `lookbackInRound` Chainlink rounds and from at most the `lookbackInSec` last seconds, the most restrictive condition is applied.

3.1.3 `PoolToken.sol`

The `PoolToken` is an ERC20 token that will act as an LP share, that is minted and burned when assets are added/removed from the pool.

3.1.4 `Factory.sol`

The `Factory` contract allows user to deploy new `Pool` contracts. After the pool is deployed and registered, the `msg.sender` of `Factory.newPool` function gains the Controller role inside the pool. The `Factory` is also responsible for receiving the `exitFee` of the different pools.

3.2 Trust model

The Controller role holder is considered as trusted party. It is assumed that lookback parameters and fees setup is assumed to be correct and well tested. The oracle addresses and pool tokens are assumed to be not malicious.

3.3 Assumptions

The tokens that the pool will work with is assumed to be a regular ERC20 contract, for example, without missing return values, fees and balance changed without transfers.

The Chainlink provides the prices in different base currencies, for example USD, ETH. It is assumed that all tokens bound to the pool have properly set Chainlink price feeds addresses, with same base asset type.

Since the impermanent loss protection has a probabilistic nature, it is assumed that this can happen. Due to the impermanent loss protections of LPs, swap price and fees can vary in values. This behavior is assumed to be normal.

3.4 Version 2 changes

To mitigate Dynamic Weights Changing Problem issue, Swaap Labs introduced two solutions that limit the effectiveness of sandwich attacks.

1. The relative difference between pool price that swap operation can reach and oracle price is capped: `AfterSwapPoolPrice/OraclePrice <= 102% + fee`
2. If during the swap the user sells token that is in shortage, and the token price experienced increase in current block, extra fee is applied to compensate for a possible impermanent loss of the pool.

4 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.

5 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.

6 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	0

7 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	2

- Chainlink Query May Revert Code Corrected
- Dynamic Weights Changing Problem Code Corrected

Medium-Severity Findings	4

- Geometric Brownian Motion Parameter Estimation Code Corrected
- Inverted Token Performance Code Corrected
- View Functions Reentrancy Code Corrected
- Zero Exit Fee Allows Just-In-Time Liquidity Provision Code Corrected

Low-Severity Findings	7

- Num.abs Function Name Code Corrected
- getRoundData Function Duplication Code Corrected
- Compiler Version Not Fixed and Outdated Code Corrected
- Gas Inefficiency and Duplicated Checks Code Corrected
- Num Library Function Visibility Code Corrected
- Specification Mismatch Specification Changed
- Time Window of 1 Will Revert Code Corrected

7.1 Chainlink Query May Revert

Design High Version 1 Code Corrected

The Pool contract relies on ChainLink assumptions that do not hold. Chainlink's round IDs do not always increase monotonically. Therefore, the `getRoundData` queries can revert. Relying on `_roundId--` in `GeometricBrownianMotionOracle.getHistoricalPrice` is not correct, since querying an invalid ID will make the swap revert.

Code corrected:

The call to the price feed's `getRoundData` function has been moved in a `try/catch` block and the function returns `(0, 0)` if the oracle call reverts.

7.2 Dynamic Weights Changing Problem

Security High Version 1 Code Corrected

Assume the pool has 10 X and 10 Y tokens that both have weight of 1. Initial invariant:

$$10_X * 10_Y = 100$$

Now assume attacker sees an update in oracle price, that will change the weight of X tokens to 2. Attacker performs a trade: in 990 Y, out 9.9 X. New constant product:

$$0.1_X * 1000_Y = 100$$

After ChainLink price update, the X tokens weight become 2. New invariant:

$$(0.1_X)^2 * 1000_Y = 10$$

In 0.9 X, out 990 Y. The invariant holds:

New constant product:

$$(1_X)^2 * 10_Y = 10$$

With 2 these trades that surround the price update, attacker profited by 9 X tokens.

The sandwiching can be performed using the Flashbots service. This issue is similar to the one that was discovered in Curve.

Code corrected:

Swaap Labs introduced 2 solutions:

1. The relative difference between oracle price and pool price is capped: `AfterSwapPoolPrice/OraclePrice <= 102% + fee`
2. If the user sells token that is in shortage, and the token price experienced increase in the current block, extra fee is applied to compensate for a possible impermanent loss of the pool.

Together these 2 solutions help with the weight change sandwich attack.

7.3 Geometric Brownian Motion Parameter Estimation

Design | Medium | Version 1 | Code Corrected

For a returns `R` over time window `T`, the code estimates Geometric Brownian Motion parameters using these formulas:

$$\mu = \frac{\sum_{i=0}^{N} R_i}{T}$$

$$\sigma^2 = \frac{\sum_{i=0}^{N} (R_i - \mu)^2 + (T - N) * \mu^2}{T - 1}$$

According to specification, the second term in σ computation is responsible for times, when the sample is missing and thus the return at that point is assumed to be 0.

Assuming `dt` is a regular sampling period, the `N = T/dt` - number of samples. In that case, a common way to estimate the GBM parameters using successive observations method is given by:

$$\hat{\mu} = \frac{\sum_{i=0}^{N} R_i}{N}$$

$$\hat{\sigma}^2 = \frac{\sum_{i=0}^{N} (R_i - \hat{\mu})^2}{T}$$

$$\sigma^2 = \frac{\hat{\sigma}^2}{dt} = \frac{\sum_{i=0}^{N} (R_i - \hat{\mu})^2}{T * dt}$$

$$\mu = \frac{\hat{\mu}}{dt} + \frac{\sigma^2}{2} = \frac{\sum_{i=0}^{N} R_i}{T} + \frac{\sigma^2}{2}$$

Comparing these estimations to code estimations, we can see following discrepancies:

1. Code μ estimate lacks a `0.5 * σ^2` term, and thus will be underestimated.
2. Code σ estimate lacks a `dt` scaling factor, and thus will be overestimated.
3. Code σ estimate has a `(T - N)/T * μ^2` term, that also doesn't help with precision of the estimate.

To summarize, the outputs of `GeometricBrownianMotionOracle.getStatistics` can have big errors, that might lead to impermanent losses of LPs as well as to overpriced swaps.

In addition, for Chainlink price oracles the sampling periods are not consistent and affected by Deviation and Heartbeat Thresholds. Thus the code computed estimations in most cases will fail to accurately estimate the price evolution process, even if it has the GBM nature.

Code modified:

The parameters estimation method has been modified to use the price ratios between two successive period instead of the return. The new implementation uses the following formulas:

$$S_i = \mathtt{price_i}$$

$$\Delta_i = \mathtt{timestamp_i} - \mathtt{timestamp_{i-1}}$$

$$\mu = \frac{1}{T}\log(\frac{S_n}{S_0})$$

$$\sigma^2 = \frac{1}{N-1}[-\frac{1}{T}\log(\frac{S_n}{S_0})^2 + \sum_{i=1}^{N}\frac{\log(\frac{S_i}{S_{i-1}})^2}{\Delta_i}]$$

These formulas come from the maximum likelihood estimation (MLE) for the GBM parameters. However to be the true MLE, mu should have a correction factor of `+ 0.5 * sigma^2`. This correction factor is not needed here because Swaap Labs computes the z-percentile of the lognormal distribution, which only needs `mu + 0.5 * sigma^2 - 0.5 * sigma^2 = mu`. Thus, the computed `mu` and `sigma` are consistent with their future usage.

7.4 Inverted Token Performance

Correctness | Medium | Version 1 | Code Corrected

The signature of the function is:

`_getTokenPerformance(uint256 initialPrice, uint256 latestPrice)`

and computes the performance ratio as `latestPrice / initialprice`. However, the function is always called with the arguments in the following order `(latestPrice_param, initialPrice_param)`, the result of the call will yield the inverted performance ratio `initialPrice_param / latestPrice_param`.

Code corrected:

Natspec and `_getTokenPerformance` call input order was fixed.

7.5 View Functions Reentrancy

Security | Medium | Version 1 | Code Corrected

Some view functions don't use the `_viewlock_` modifier. In case of reentrancy due to ERC20 token calls (e.g. ERC777), these getters can return unreliable data. This may break the integration with other contracts and systems that rely on these getters. Such getter functions are:

- `getAmountOutGivenInMMM`

Please note, this list might be incomplete. Any function of a contract that does external call need to be `lock` or `viewlock` protected, if other external contract might rely on the data from this contract, such as spot prices, weights, etc.

Code corrected:

View locks have been added to all view functions in the `Pool.sol` contract.

7.6 Zero Exit Fee Allows Just-In-Time Liquidity Provision

Design | Medium | Version 1 | Code Corrected

Since the system is not totally impermanent loss resistant, liquidity providers are still exposed to a risk. To cope with the risk, malicious liquidity providers can sandwich large swaps transactions and collect most of the swap fee without the risk of an impermanent loss.

Code corrected:

JIT liquidity provision is mitigated by the use of a cooldown timer of 2 blocks. A LP that provided liquidity to a pool cannot exit the pool or transfer LP tokens (by either `transfer` or approval and `transferFrom`) for a period of 2 blocks after the liquidity provision.

However, this may block proxy contracts to manage funds for users. To cope with this issue, Swaap Labs added the `joinPoolForTxOrigin`, a function that pulls funds from `msg.sender`, but deposits them to the `tx.origin`. Since it is not the authorization by the `tx.origin`, this does not raise problems like https://swcregistry.io/docs/SWC-115.

7.7 `Num.abs` Function Name

Correctness | Low | Version 2 | Code Corrected

The name of `Num.abs` function does not match its functionality.

Code corrected:

The `Num.abs` function has been renamed `Num.positivePart`.

7.8 `getRoundData` Function Duplication

Design | Low | Version 2 | Code Corrected

The function `getRoundData` and its functionality is duplicated. It is implemented in `GeometricBrownianMotion` and in `ChainlinkUtils`. Functionality duplication should be avoided as it increases the amount of code to deploy and deteriorates code maintainability.

Code corrected:

The getRoundData function in GeometricBrownianMotion has been removed and its use has been replaced by the getRoundData function from ChainlinkUtils.

7.9 Compiler Version Not Fixed and Outdated

Design Low Version 1 Code Corrected

The solidity compiler is not fixed in the contracts. The version, however, is defined in the truffle-config.js to be 0.8.0.

In the Factory contract the following pragma directive is used:

```
pragma solidity ^0.8.0;
```

Known bugs in version 0.8.0 are:

https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json#L1531

More information about these bugs can be found here:

https://docs.soliditylang.org/en/latest/bugs.html

At the time of writing the most recent Solidity release is version 0.8.12 which contains some bugfixes.

Code corrected:

The compiler was fixed to version 0.8.12.

7.10 Gas Inefficiency and Duplicated Checks

Design Low Version 1 Code Corrected

1. In the GeometricBrownianMotionOracle.getHistoricalPrices function, idx is set to hpParameters.lookbackInRound + 1 and then directly to 1. The second first assignation has no effect.
2. Math.getLogSpreadFactor checks horizon and variance for >= 0, this check is useless since both values are uint256.
3. Math.getLogSpreadFactor does division by two with 5 * Const.BONE / 10, simply dividing by 2 would save gas.
4. In Math.getInAmountAtprice it is possible to pack computations to save calls to LogExpMath.pow.
5. Some state variables can fit in smaller types (e.g., with its current bounds, dynamicCoverageFeesZ could fit in a uint64). Saving storage slots might save gas.
6. TokenBase's _burn and _move functions check that there is enough balance, the check for underflow is by default since compiler version 0.8.0.
7. PoolToken.transferFrom check that there is enough allowance, the check for underflow is by default since compiler version 0.8.0.
8. PoolToken's _name, _symbol and _decimal can be constant and their respective getter functions can be external. This will save gas.

9. The overflow checks in numerous `Num` functions are not necessary anymore since compiler version 0.8.0, which features automatic overflow check. The division by zero checks are not necessary, solidity division will revert on a division by 0.
10. Elements of the `Pool.Record` struct can have a smaller type (e.g. `index`, `denorm`) and be reordered to save storage.
11. The `Pool`'s state variable `_factory` can be immutable.
12. `Pool.joinPool`, `Pool.getAmountOutGivenInMMM`, `Pool.finalize` can be `external`.
13. The check for `_controller` address and finalization in `Pool.bindMMM` are redundant with the ones in `Pool.rebindMMM`.
14. When resetting storage slots on mappings, e.g. in `unbindMMM`, the use of `delete` is recommended for lower gas usage.
15. The second require of `_getAmountOutGivenInMMMWithTimestamp` is a less strict version of the first requirement.

Version 2:

1. `Const.MAX_IN_RATIO` and `Const.MAX_OUT_RATIO` are never used in the code, they should be removed.
2. `getMMMWeight` is always called with `shortage = true`, removing the argument and code related to `shortage = false` will save gas.

Code corrected:

1. `idx` is set to `1` at variable declaration.
2. Both checks for `>= 0` have been removed.
3. The multiplication by `5 / 10` has been replaced by a division by `2`.
4. The terms under `w_o / (w_o + w_i)` have been grouped together.
5. Acknowledged. Some state variables have been changed to use a smaller type.
6. The checks for sufficient balance have been removed.
7. The check for sufficient allowance has been removed.
8. `PoolToken`'s `_name`, `_symbol` and `_decimal` have been changed to `constant` and their respective getter function have `external` visibility.
9. Unnecessary overflow checks in `Num` library have been removed.
10. `index` and `denorm` types have been reduced to `uint8` and `uint80` resp.
11. `_factory` state variable has been changed to immutable.
12. `Pool.joinPool`, `Pool.getAmountOutGivenInMMM`, `Pool.finalize` visibility has been changed to `external`.
13. The checks have been moved to the common `_rebindMMM` function.
14. `delete` is now used to reset the storage fields in the mappings.
15. Both `require` have been removed. The check has been replaced by the oracle update sandwich protection.

Version 2:

1. Unused constants have been removed.
2. The `shortage` parameter of function `getMMMWeight` has been removed.

7.11 Num Library Function Visibility

Design | Low | Version 1 | Code Corrected

The functions of `Num` library have `public` visibility. This way, any contract that will need to deploy this library, will use it as an external contract. It means that any call to the library functions will result in quite expensive `CALL` opcode. If the visibility of those functions were `internal`, the function code would be then inlined at the point of use. This way bytecode size of Pool will be smaller and gas cost for each library call operation will be smaller as well. For more info see: https://docs.soliditylang.org/en/latest/contracts.html#libraries

---

Code corrected:

The visibility of the functions in the `Num` library has been changed to `internal`.

7.12 Specification Mismatch

Design | Low | Version 1 | Specification Changed

1. The formula provided in the documentation for `getInAmountAtPrice` multiplies the desired price by `w_out / w_in` which is wrong, however the implementation correctly multiplies by `w_in / w_out`.
2. In the whitepaper, when the stochastic buy-sell spread is computed, the p-percentile of the random variable is divided by the latest oracle price, this is not the case in the implementation.
3. The `@dev` natspec of `getNextSample` is incomplete
4. The `@dev` natspec of `getRoundData` makes a wrong assumption, the function will revert if no data can be found as specified in https://docs.chain.link/docs/faq/#can-the-data-feed-read-revert.
5. The specification of some public and external functions, e.g. `joinPool`, `finalize`, `calcSpotPrice`, is missing.
6. The `@notice` natspec of `rebindMMM` is incomplete.
7. The natspec of `Pool._getTokenPerformance` defines twice the first parameter and not the second one

Version 2:

1. The `@dev` natspec of `GeometricBrownianMotionOracle.getHistoricalPrices` does not reflect the implementation. If no historical data was found, the latest data and `startIndex == 0` will be returned. If round data is 0, the round will simply be skipped, the algorithm will not stop filling prices/timestamps.
2. The `_getParametersEstimation` doesn't describe all `@param`.

Specification partially corrected:

1. The formula in the documentation has been corrected.
2. Specification changed.
3. The `@dev` natspec for `getNextSample` has been completed.
4. The implementation of `getRoundData` now matches the natspec.

5. Comments or natspec have been added for some `public` and `external` functions.
6. The natspec for `rebindMMM` has been completed.
7. The second parameter is now described in the natspec.

Version 2:

1. The `@dev` natspec has been updated to reflect the implementation.
2. The missing parameters natspec has been added.

7.13 Time Window of 1 Will Revert

Design Low Version 1 Code Corrected

A time window of `1` second will make `getStatistics` revert due to a division by zero. The `Const.MIN_LOOKBACK_IN_SEC` limit enforced on `_priceStatisticsLookbackInSec` storage variable in `setPriceStatisticsLookbackInSec` function does not prevent this case from happening.

Code corrected:

If time window = 1, the variance and mean are considered to be 0.

8 Notes

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

8.1 Compatibility Issues Due to `tx.origin`

Note Version 1

The use of `tx.origin` limits certain functionality of the contract. Such contracts can be not deployable on chains that don't support `ORIGIN` opcode, e.g. Optimism. In addition, usage of this contract by wallet contracts like Gnosis wallet can also be limited.

8.2 ERC20 Compatibility

Note Version 1

The `_pull/_pushUnderlying` functions of the `Pool` expect `transferFrom` and `transfer` to always return a boolean. However, some tokens, for example USDT, do not follow this pattern and are thus incompatible with the system. OpenZeppelin has a SafeERC20 library, which helps with such tokens.

In addition, the usage of ERC20 tokens with fees, rebalancing tokens, or tokens with reentrancies can be problematic to integrate. Swaap Labs needs to carefully consider what tokens can be supported by the `Pool`.

The `_pull/_pushUnderlying` functions have been modified to use the SafeERC20 library for token transfer.