PUBLIC

Code Assessment

of the Dynamic Market Maker

Smart Contracts

April 23, 2021

Produced for

Kyber Network

On-chain Liquidity Protocol

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Loi, Dear Victor,

First and foremost we would like to thank Kyber.Network for giving us the opportunity to assess the current state of their Dynamic Market Maker system. This document outlines the findings, limitations, and methodology of our assessment.

The suite of contracts implement a Dynamic Market Maker (DMM) based on `UniswapV2`. The main changes are the use of an amplification model for the pools inventory function and fees based on the recently traded volume.

Our main concerns are around the implementation of the amplification model. The paper `Amplification Model` describes the model in detail, however, only covers the cases when trades and contribution of liquidity are done in a balanced manner in regard to the pools tokens. The actual implementation, however, allows unbalanced contributions. Three issues raised in the report are connected to unbalanced contributions.

One medium severity security issue has been identified during the assessment. Additionally one medium severity correctness issue and one medium severity aswell as several low severity design issues have been reported.

We hope that this assessment provides valuable findings as well as more insight into the current implementation. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	4
• Code Corrected	1
• Specification Changed	1
• Acknowledged	1
• No Response	1
Low-Severity Findings	6
• Code Corrected	6

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 Dynamic Market Maker 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	1 March 2021	ea642d04f531586afb14f06111a8c088294ece01	Initial Version
2	6 April 2021	fb470a33ed6a7e84fb06be39b3688d735e5f5a68	After first report
3	22 April 2021	f9e91ed05e41a733c27a28d81d630a5b5479f23e	After second report

For the solidity smart contracts, the compiler version `0.6.6` was chosen. After the intermediate report, the compiler version used was updated to `0.6.12`.

2.1.1 *Excluded from scope*

The solidity files in subfolders `examples` and `mock` are out of scope for this review. The `LiquidityMigrator` contract has been added in the final commit reviewed and is not part of this review.

2.2 System Overview

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

The suite of contracts implement a Dynamic Market Maker (DMM). The implementation is based on the `UniswapV2-core` and the `UniswapV2-periphery` smart contracts. It differs from UniswapV2 in two main ways:

1. The reserves use the **Amplification Model**. Instead of the inventory function `x*y = k` the function `x * y = k * a^2` is used which includes an amplification factor `a > 1`. This results in the reserve having a smaller spread and slippage rate than a reserve using the Uniswap model. For more details refer to the paper `Amplification Model`. Multiple pools for the same token pair may exist. There can only be one unamplified pool per token pair where a is equal to 1, but there can be multiple pools for the same token pair with an equal amplification factor > 1. Intuitively, the greater the amplification factor the more stable is the price between the two tokens. It's important to understand that the amplification factor itself is only used once during the first addition of liquidity to a pool in order to calculate the values for the `virtualReserves`. Afterwards, the `virtualReserves` are only updated proportionally to certain inputs. All invariants related to adding or removing liquidity are based on the values of the "traditional" reserves. All invariants related to swapping are based on the `virtualReserves`.
2. Dynamic fees: In contrast to UniswapV2, which enforces a 0.03% constant fee per exchange, the fees depend on the volume exchanged during a given time window. Fees are lower during periods of low activity which encourages trading while fees increase during high usage. For more details refer to the paper `XYZ model`.

The system consists of two components: the core `DMM` (`DMMPool.sol`, `DMMFactory.sol`) and the periphery (`DMMRouter.sol`, `DaoRegistry.sol`).

2.2.1 DMMPool

The DMMPool is based on the UniswapPair contract. It implements the core logic of the DMM (or AMM in the Uniswap case). This includes functions to add/remove liquidity into the pool, swap tokens, synchronise the reserves date and the functionality for the liquidity token.

These functions may be called directly although this practice is discouraged. Users are expected to interact with the contract through the DMMRouter02 contract. Inside the `DMMPool`, all functions assume that the tokens have already been transferred to the pool. This must only be done within the same transaction in a preceding function code. Otherwise the next transaction interacting with this pool will take advantage of these transferred tokens.

- `mint`: mints the liquidity tokens after users have added liquidity to the pool.

  Assumption: The required amount of both pool tokens have already been transferred.

  The liquidity tokens minted correspond to the contribution of the user to the pool. For example if the user contributes 10% to the reserves (not the virtual ones) of the pool, they will hold 10% of the liquidity tokens. Usually users contribute with the same proportion in both reserves. In any other case the minimum contribution to a reserve is considered. The virtual reserves also increase by the same proportion. In case the amount in a reserve exceeds the corresponding one in the virtual reserve the new value of the virtual reserve is set to be the amount in the reserve.

  If no liquidity tokens have ever been issued the contract mints a number of tokens equal to the square root of the product of the contributed amounts. From these a `MINIMUM_LIQUIDITY` amount is sent to address `-1` and the rest to the user.

  During minting, `_mintFee()` is called which settles the fees accrued by the previously executed swaps through this pool. At the end of the minting the state of the pool is updated i.e., the size of the reserves and the virtual reserves.

- `burn`: this is the inverse procedure to `mint`. A user sends an amount of liquidity provider tokens they wish to burn and get back a corresponding proportion of both tokens.

  Assumptions: The liquidity tokens to be burned have already been transferred to the pool.

  Note, that `_mintFee` is also called in this case.

- `swap`: implements the logic of the exchange in the pool.

  The condition for a swap to conclude sucessfully is that the pools invariant `k`, based on the virtual reserves holds at the end of the swap after the fees have been deducted. The amplification factor of the pool is not directly involved in a swap.

  Tokens must be transferred to the pool either before or at the latest during the callback.

  The complexity of the function stems from two features it implements: 1. it allows for flash swaps 2. it allows an input token to be an output token. In other words, both tokens can be input and output tokens at the same time, while one would expect one token to be the input token and the other the output.

  During a swap the amount of output tokens are sent to the recipient. Then, an arbitrary call to the recipient is executed. As a next step, the balances of the reserves are calculated and the fees are implicitely accrued, and finally the state of the pool is updated.

2.2.2 DMMFactory

The factory allows the deployment of new pools. It is important to note that there can be only one unamplified pool for a specific pair, while there can be many pools even with the same amplification factor for the same pair. The main functionality of the contract is in function `createPool`. During its execution, a new pool contract is deployed and a registry in a from a map and an array is updated. The second functionality is `setFeeConfiguration` which allows the `feeSetter` to set the address that collects the fees on behalf of the protocol as well as the `governmentFeeBps` which determines the percentage of exchange fees kept by protocol.

2.2.3 The DaoRegistry

The `DMMFactory` can be used by everyone to deploy a pool. The `DaoRegistry` is used to register pools that are trusted by the Kyber Network. Only the administrator of the Registry is allowed to add or remove trusted pools.

2.2.4 The DMMRouter (Periphery)

The DMMRouter is the main proxy thourgh which users interact with the pools. It implements wrapper functions that check and calculate the input arguments for the low level calls implemented by the DMMPool.

It allows for high level calls similarly to Uniswap, namely:

- `addLiquidity` and variations `addLiquidityETH`, `addLiquidityNewPool`, `addLiquidityNewPoolETH`
- `removeLiquidity` and variations `removeLiquidityETH`, `removeLiquidityWithPermit`, `removeLiquidityETHWithPermit` `removeLiquidityETHSupportingFeeOnTransferTokens`, `removeLiquidityETHWithPermitSupportingFeeOnTransferTokens`
- `swapExactTokensForTokens`, `swapTokensForExactTokens`, `swapExactETHForTokens`, `swapExactTokensForETH`, `swapExactTokensForTokensSupportingFeeOnTransferTokens`, `swapExactTokensForETHSupportingFeeOnTransferTokens`. In these calls a full exchange path must be provided as well as the corresponding pools.

2.2.5 Fees

Fees accrue during exchanges. The general invariant enforced is:

```
(x1 – fee · xin / 10 ** 18) · (y1 – fee · yin / 10 ** 18) >= x0 · y0
```

where `x0`, `y0`, `x1`, `y1` are the corresponding amplified balance for tokens x and y before and after there exchange, `xin` and `yin`, the input values for the tokens and fee, the fee applied.

As mentioned, the fees are dynamic: The fee value depends on the RFactor and the amplification factor. `rFactor`: Expresses the ratio between the exchange volume in a short time window and a longer one. It is important to notice that both volumes are updated at the beginning of a new block. For optimization purposes both windows are calculated using EMA according to the recursive formula:

```
ema = (ema * (1 - alpha) + volume * alpha) * (1 - alpha) ^ (skip_block -1)
```

where

- skip_block: the number of blocks without an exchange
- alpha: a constant 2/N+1
- volume: the exchange volume

The fees are calculated using the following formula:

- if `r >= 1.477`:

  `6/1000`

- if `1.477 > r >= 1`:

  `(985/27 + 20000/27*(r - 120/100)^3 + 250/9*(r - 120/100)) /1000`

- else:

  ```
  [200 + sign(r - 836/1000) * 5 * (r - 836/1000)^2 /
      (2/1000 + (r - 836/1000)^2) ]/1000
  ```

Depending on the amplification factor a, the fee may be reduced:

- if `amp <= 2`: `fee`
- if `2 < amp <= 5`: `2/3 * fee`
- if `5 < amp <= 20`: `1/3 * fee`
- else: `4/30 * fee`

Intuitively, the higher the correlation between two tokens the less fee someone would pay.

2.3 Trust Model

In general pools are not trusted since anyone can deploy a pool with arbitrary tokens. Pools considered as trusted are registered in the `DaoRegistry`.

Users are expected to interact with the system via the Router. The router features some slippage protection which protects users from changing liquidity conditions before the transaction is mined.

The owner of the registry is a trusted entity.

The `feeToSetter` role of the `DMMFactory` is trusted since it is responsible for setting the address which collects the protocol fees `feeTo` as well as the `governmentFeeBps` parameter related to the fees. We understand that the fees will be collected and distributed by the KyberDAO.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. We draw attention to the fact that due to inherent limitations in any software development process and software product an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third party infrastructure which adds further inherent risks as we rely on the correct execution of the included third party technology stack itself. Report readers should also take into account the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severities. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved, have been moved to the Resolved Findings section. All of the findings are split into these different categories:

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

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

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	2

- Obsolete Storage Writes During Pool Deployment
- Actual Amplification Reduces After Unblanced Contribution Acknowledged

Low-Severity Findings	0

5.1 Obsolete Storage Writes During Pool Deployment

Design Medium Version 2

After the intermediate report, the following functions have been added to DMMPool:

```
function name() public override view returns (string memory) {
    IERC20Metadata _token0 = IERC20Metadata(address(token0));
    IERC20Metadata _token1 = IERC20Metadata(address(token1));
    return string(abi.encodePacked("KyberDMM LP ", _token0.symbol(), "-", _token1.symbol()));
}

function symbol() public override view returns (string memory) {
    IERC20Metadata _token0 = IERC20Metadata(address(token0));
    IERC20Metadata _token1 = IERC20Metadata(address(token1));
    return string(abi.encodePacked("DMM-LP ", _token0.symbol(), "-", _token1.symbol()));
}
```

The pool storage still contains the old name and symbol variables which are set during execution of the constructor. Due to the new functions, the new name and symbol will be returned while the storage variables are now obsolete.

```
constructor() public ERC20Permit("KyberDMM LP", "DMM-LP", "1") VolumeTrendRecorder(0) {
```

These unnessesary storage writes makes the deployment of new pools more expensive than necessary. In particular 100,000 gas (roughly 20 USD at the time of writing) could be saved during each pool deployment.

5.2 Actual Amplification Reduces After Unblanced Contribution

Design Medium Version 1 Acknowledged

Users may add liquidity to a pool by directly invoking `DmmPool.mint()`.

Normally, liquidity is added in balanced amounts of `token0` and `token1` according to the pool's inventory as the amount of liquidity tokens minted in return is based on the lower contribution. The surplus amount of the other token is kept by the pool.

After minting, the values of the virtual reserves are updated as follows:

```
liquidity = Math.min(
    amount0.mul(_totalSupply) / data.reserve0,
    amount1.mul(_totalSupply) / data.reserve1
);
uint256 b = liquidity.add(_totalSupply);
_data.vReserve0 = Math.max(data.vReserve0.mul(b) / _totalSupply, _data.reserve0);
_data.vReserve1 = Math.max(data.vReserve1.mul(b) / _totalSupply, _data.reserve1);
```

Unbalanced contributions reduce the factor between the value of the actual `reserve` and the `virtualReserve`, hence the pool "looses amplification" figuratively speaking. In an extreme scenario of an unbalance contribution, which is rather costly for an attacker and has no clear benefit, the following scenario may arise:

Assume a pool has following state: `reserve0 = 1000`, `reserve1 = 1000`, `vReserve0 = 2000` and `vReserve1 = 2000`.

1. A user adds `2000 token0` and `1 token1` to the pool. The values for `vReserve0` and `vReserve1` should now be `2002`. However, as the pool received an additional amount of `token0` the value of `reserve0` (`3000`) is now higher than the result of the calculation for the new `vReserve0` amount, hence the value for `vReserve0` is set to `_data.reserve0`.
2. This step may be repeated for the other token: A user adds `3` tokens to `reserve0` and `2998 tokens` to `reserve1`. Then again the `vReserve1` will get the value of `reserve1`.
3. Now it holds that `reserve0` = `vReserve0` and `reserve1` = `vReserve1`.

After such a scenario an amplified pool is no longer amplified.

Note that a similar attack vector can be implemented using `burn` for tokens that accrue rewards on transfer.

The documentation provided does not describe the expected behavior when liquidity is added in case of an unbalanced contribution. In the paper `Amplification Model`, in section `Adding liquidity in Ampfliciation model` on page 7 the only case described is when the contributions match the expected ratio.

Acknowledged:

Kyber is aware of this scenario and states:

> Note that liquidity providers get benefits if this scenario happens and the attacker has no economic incentives to do this.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

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

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

- Conflicting Statements About Contribution Ratio Specification Changed
- Sandwich Attack on New Liquidity Providers Code Corrected

Low-Severity Findings	6

- Outdated Compiler Version Code Corrected
- Redundant Modulo Operation Code Corrected
- Unused Library Code Corrected
- Unused blockTimestampLast Code Corrected
- Wrong Inequality Code Corrected
- vReserve Wrong Naming Code Corrected

6.1 Conflicting Statements About Contribution Ratio

Correctness Medium Version 1 Specification Changed

The document `Dynamic AMM model design and rationals` in section `2.3.3 Add Liquidity` reads:

> When users add liquidity to existing pools, they must add liquidity with the current ratio of token in the pool. The amount of mint token will be the min increase proportion of 2 tokens, the virtual balances will scale out with the mint token to assure that the price range of the pools is only bigger. Special case: the pool has reserve0=1000 and reserve1=1000 and vReserve0=2000 and vReserve1=2000. An user adds 2000 token0 and 1 token1 to the pool. The vReserve0 and vReserve1 should be 2002. But the reserve0 (3000) is higher than vReserve0. Therefore, we must vReserve0 = max(reserve0, vReserve0) to assure the assumption that vReserve0 >= reserve0

The first statement clearly states:

> must add liquidity with the current ratio of token in the pool

while the next statement handles a special case where this does not hold - hence the two statements are contradicting.

The actual implementation does not enforce that adding liquidity must be done with the current ratio of the tokens in the pool.

Finally the rational behind setting `vReserve` to `reserve` in case the new value for `vReserve` is less than `reserve` is not clear. It's understood that `vReserve` cannot be smaller than `reserve` as the amplification factor must be `>= 1`, however it's questionable and not documented why setting the value equal to `reserve` is the correct action in this case.

Specification changed:

The specification has been updated and now describes the scenario of an unbalanced contribution more detailed.

6.2 Sandwich Attack on New Liquidity Providers

Security | Medium | Version 1 | Code Corrected

This attack works against new liquidity providers when they are adding liquidity. The overall idea of the attack is that the virtual reserve values are out of sync with the reserve values. Hence, the slippage protection of `addLiquidity()` can be circumvented. The reserve values are brought out of sync by adding unbalanced liquidity. Adding unbalanced liquidity by itself is good for liquidity providers, but in this combination it can be used for an attack.

Prerequisites:

- A pool with little liquidity, e.g. new pool
- The pool is amplified
- The attacker has the ability to perform a sandwich attack

Setup:

- The pool has two token `T0`, `T1`
- `T0` is worth 100 USD
- `T1` is worth 1 USD
- The pool is balanced, e.g. 1 `T0` and 100 `T1`

Attacks Steps:

1. Attacker adds liquidity regularly through the router. Hence, the pool is still correctly balanced. In particular the reserves and virtual reserves have the ratio 1:100.
2. The victim looks at the pool and decides to add liquidity
    - The victim uses the router and allows for no slippage or a tiny amount of slippage (hence, following best practices)
    - The victim sets up amountADesired and amountBDesired in 1:100 ratio, also amountAMin and amountBMin have 1:100 ratio

4. The attacker detect the victim transaction in the mempool and starts a sandwich attack
5. First attacker transaction:
    - The attacker swaps all of `T0` out of the pool
    - The attacker adds unbalanced liquidity (as described in our report)
    - These two steps can be repeated
    - As a result the reserves are in a 1:100 ratio but the virtual reserves are in a different ratio, e.g. 1:210 in our example
6. The victim transaction is executed, all checks pass, the transaction is successfully completed
7. Second attacker transaction:
    - Attacker removes all its liquidity from the pool, now only the victim's liquidity is in the pool

- Attacker uses the incorrect ratio of the virtual reserves to execute a swap that is bad for the victim

Effect and Analysis:

- The "gifted" liquidity through unbalanced minting here goes back to the attacker as they are the only/primary liquidity provider
- In our example with an amplification factor of 100, the attacker can steal 12.69% of the victim's funds. Hence, the more the victim deposits, the more can be stolen.
- The attacker's funds can be smaller than the victim's funds. The percentage of stolen funds remains the same.
- This is independent of the price ratios between T0 and T1 (1:100 in this example). Different ratios lead to the same outcome.
- Other amplification factors lead to different results, but there are probably ways to make this attack more effective

Example Numbers:

Pool after liquidity has been added:

```
[++] T0: 1.0
[++] T1: 100.0
[++] Value: 200.0 USD
[+] Value of 1 LP Share: 20.00 USD
[+] Virtual Reserves: 10.00, 1000.00
```

At this point all seems fine and the victim decides to add liqudity.

Pool after pre-manipulation:

```
[++] T0: 5.5249
[++] T1: 552.49
[++] Value: 1104.97 USD
[+] Value of 1 LP Share: 110.50 USD
[+] Virtual Reserves: 6.89, 1452.49
```

At this point the reserves are still in a 1:100 ratio, but the virtual reserves are not. There ratio is 1:210.

Pool before final swap to exploit incorrect ratios:

```
[++] T0: 100.0
[++] T1: 10000.0
[++] Value: 20000.0 USD
[+] Value of 1 LP Share: 110.50 USD
[+] Virtual Reserves: 124.68, 26290.01
```

At this point only the victim's liquidity is left. The ratio of the virtual reserves is still 1:210.

Code corrected:

The router now features a slippage protection on the ratio of the virtual reserves. The function takes two new arguments where users can specify the lower and upper bound for the ratio between the virtual reserves. This mitigates the attack described above as the attacker can no longer arbitrarily unbalance the virtual reserves. Note that the protection is in the Router, hence, users interacting with the pool contract directly are not protected.

6.3 Outdated Compiler Version

Design | Low | Version 1 | Code Corrected

The project uses an outdated version of the Solidity compiler.

```
pragma solidity 0.6.6;
```

Known bugs in version 0.6.6 are: https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json#L1378

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 `0.6.12`. For version `0.6.x` the most recent release is `0.6.12` which contains some bugfixes but no breaking changes.

Code corrected:

After the intermediate report the compiler version has been updated to `0.6.12`.

6.4 Redundant Modulo Operation

Design | Low | Version 1 | Code Corrected

In `DMMPool._update` there is a redundant use of modulo operation `uint32 blockTimestamp = uint32(block.timestamp % 2**32);`. With optimizations enabled, the solidity compiler version 0.6.6 generates almost identical bytecode for `uint32(uint256(block.timestamp))` and `uint32(uint256(block.timestamp)%2**32);`.

This code no longer exists in the updated implementation.

6.5 Unused Library

Design | Low | Version 1 | Code Corrected

Library `UQ112x112` is present in the repository but never used.

Code corrected:

The unused library has been removed.

6.6 Unused `blockTimestampLast`

Design | Low | Version 1 | Code Corrected

Variable `blockTimestampLast` in `DMMPool` is regularly updated but never used. The purpose of the variable is not documented.

Code corrected:

The unused variable has been removed.

6.7 Wrong Inequality

Design Low Version 1 Code Corrected

`DMMLibrary.getAmount` ensures `reserveOut >= amountOut`. However, if the equality holds the transaction will later fail since a later call in swap requires `amount0Out < data.reserve0 && amount1Out < data.reserve1`.

Code corrected:

The equality check has been removed.

6.8 vReserve Wrong Naming

Design Low Version 1 Code Corrected

In `DMMLibrary.getReserves()`, the return values of `DMMPool.getReserves` are assigned to `vReserve` variables while the values returned by the function correspond to the unamplified reserves.

```
(uint256 vReserve0, uint256 vReserve1, ) = IDMMPool(pool).getReserves();
```

`IDMMPool(pool).getReserves()`:

```
function getReserves()
        external
        override
        view
        returns (
        uint112 _reserve0,
        uint112 _reserve1,
        uint32 _blockTimestampLast
        )
{
        _reserve0 = reserve0;
        _reserve1 = reserve1;
        _blockTimestampLast = blockTimestampLast;
}
```

Code corrected:

The naming of the variables in the code has been corrected.

7 Notes

We leverage this section to highlight potential pitfalls which are fairly common when working Distributed Ledger Technologies. As such technologies are still rather novel not all developers might yet be aware of these pitfalls. Hence, the mentioned topics serve to clarify or support the report, but do not require a modification inside the project. Instead, they should raise awareness in order to improve the overall understanding for users and developers.

7.1 Amplification Increases Risk for Liquidity Providers

Note Version 1

A higher amplification coefficient increases the risk for the liquidity providers. Due to a large amplification factor, larger trade volumes are required in order for the current price to be reached. Moreover, the smaller spread may be exploited by arbitrage bots balancing liquidity accross markets.

7.2 Tokens With Multiple Entrypoints

Note Version 1

This is more a theoretical issue but has applied to tokens in the past. Nowadays this is a less common issue. Some (very few) tokens have multiple addresses as entry points, e.g. a proxy not using delegatecall and the actual implementation contract. `TrueUSD` is such an example.

In the `DMM` system, this may has following consequences.

- The check in `DMMFactory.create()` to prevent the creation of a pool where `tokenA` and `tokenB` are equal can be bypassed.
- A second unamplified pool may exist for the same token pair.

7.3 Volume Increase

Note Version 1

By swapping large amounts of funds of a pool with the receiver being the pool itself anyone may execute a trade with a large volume. The requirement is that some additional tokens are transferred to the pool during the callback in order to cover the fees so that the transaction can succeed.

As any swap, such a trade gets recorded in the `VolumeTrendRecorder`. The volume observed by the `VolumeTrendRecorder` may be increased by anyone willing to spend the fee in order to do so.