ATTIC LAB

Review Report

Produced by Attic Lab
for Impossible Finance

May 07, 2021

Reviewed code

The following review was conducted on the impossibleSwapCore repository. Source code reviewed was at the state of commit `3bc771b904d7c64f61a4afe88f4d6acdaa39bcd3` (`https://github.com/ImpossibleFinance/impossibleSwapCore/commit/3bc771b904d7c64f61a4afe88f4d6acdaa39bcd3`).

Tests

Tests are passing, however a check for governance in StableXPair contract has to be disabled manually.

Contracts

Since this repository is forked from the Uniswap V2 Core contracts we compared StableX contracts with the latest released Uniswap version (`https://github.com/Uniswap/uniswap-v2-core/commit/4dd59067c76dea4a0e8e4bfdda41877a6b16dedc`)

Reviewed contracts closely resemble Uniswap contracts. Here are the corresponding Uinswap V2 contracts for each of the StableX contracts: `StableXERC20 is UniswapV2ERC20 StableXFactory is UniswapV2Factory StableXPair is UniswapV2Pair`

From those 3 contracts only StableXPair has considerable differences from its Uniswap counterpart and was thoroughly reviewed. The rest were only checked to have the same logic as Uniswap V2.

The similar situation is in `interfaces/` and `libraries/` folders, however there all the code is exactly the same as in the Uniswap V2 repository.

Methodology

During the review process we had several meetings with the development team explaining their contract design. We've reviewed each contract individually as well as their interaction within the product architecture.
Contracts were also checked for the possibility of all known attack vectors, which could lead to lost user funds or contract stability issues.
All the issues we found were split into four groups:

- Recommendations: when the current solution in the code is fine, but we can recommend a way to improve it and make it better
- Low: issues which do not affect the stability of the contract, it can work fine with them present
- Medium: issues worth fixing, but not leading to the loss of funds or making contract impossible to use
- Critical: issues leading to the loss of funds or breaking contract functionality

Results

After review we identified no critical issues, 2 medium, 2 low and 4 recommendations.

Review

StableXPair

K calculated on each swap (Severity: Medium)

One of the important changes compared to the Uniswap contracts is the value of K calculated not only on mint and burn, but on each swap operation as well. It looks like this K value is used in mint fee calculation and cheap K calculation when ensuring proper balance ratio during the next swap call.
It seems that it is should be possible to avoid this by:

- Calculating expensive K during swap once (instead of the cheap version)
- Remove expensive K calculation during the balance update
- Add expensive K calculation in mint fee function

This way you can reduce the number of calculations during swap, make mint and burn slightly more expensive and remove extra variable, which is storing K when the last fee was calculated (`kLastFee`).

Leftover code from the cumulative prices calculation (Severity: Medium)

Cumulative price calculation is removed from this contract (if compared to the Uniswap V2 original), however some pieces of code are left over. Namely:

- `price0CumulativeLast` and `price1CumulativeLast` state variables
- `blockTimestampLast` variable and its update code

We understand a necessity to keep these variables to ensure compatibility with Uniswap Pair interface, but it can be achieved with creating pure functions returning `0` values for the cumulative price getters or simply returning the current blocktime in `getReserves` function (instead of saving last timestamp in the contract state).

Constant product is used for the initial liquidity calculation (Severity: Low)

Then initial liquidity is calculated `StableXPair` uses the same formula as the original Uniswap contract (square root of deposited amounts product). This does not make sense in the context of the custom curve and should be replaced by the custom K calculation or just a constant (since all other liquidity token amounts are calculated as a proportion of this initial value).

Variable can be constant (Severity: Low)

Variable `delay` does not change in the code and should be declared as a constant and not as a variable.

Hardcoded governance address (Recommendation)

Currently governance address is set to the constant value by the contract initialization code. This makes smart contract not really flexible which is indicated by the necessity to comment out governance checks when running tests.

It is recommended to move default governance address to the factory contract (make it one of the arguments in the constructor) and then provide it to the newly created pair contract as initialization parameter.

This makes factory and pair creation transactions use slightly more gas, but the contract overall is much more flexible. Also no need to comment the code when testing, simply provide one of the test addresses as a governance address on the factory creation in the test environment.

Using `getReserves` to get reserve values (Recommendation)

There is an inconsistency in the code when using `reserve0` and `reserve1` values from the contract state. Parts of the code taken from the original Uniswap V2 contract abstract getting those values via `getReserves` call while the new ones use the state variables directly.

It is recommended to keep consistency and use `getReserves` everywhere.

Time constants usage (Recommendation)

When `delay` variable is initialized `seconds` time constant is used and then multiplied by seconds in a minute, minutes in an hour and hours in a day. Instead `days` constant can be used.

Events naming scheme (Recommendation)

New events start with a lowercase letter and contain underscores in field names. It is recommended to follow the same guidelines as the rest of the code (start with capital letters and avoid underscores).