hexens

ALGEBRA FINANCE SMART CONTRACT AUDIT REPORT

11.08.2022

CONTENTS

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	0
MEDIUM	3
LOW	3
INFORMATIONAL	0

TOTAL: 6

SCOPE

The analyzed contracts are located in the following repository folder:
https://github.com/cryptoalgebra/Algebra/commit/f62e15a3381a2f40e85f020fe57773ef95c39b4b

WEAKNESSES

This section contains the list of discovered weaknesses.

1. ULTRA-CONCENTRATED LIQUIDITY IN FARMING

SEVERITY: Medium

RESOLUTION: consider adding a minimal tick width to incentive programs and check for narrow range liquidity before entering the farming

STATUS: fixed

DESCRIPTION:

Limit and eternal farming incentive programs do not check for narrow range liquidity positions (e.g. tick spacing sized), this gives an opportunity to amplify the farmings; a user can mint tick-spacing sized liquidity positions, enter a farming and monitor tick for crossing the range, in case of crossing the range exit the farming and mint a new position.

2. DISCREPANCY IN AVERAGE CUMULATIVE VOLATILITY CALCULATION

SEVERITY: Medium

RESOLUTION: volatilityCumulative should be divided by the seconds passed starting from the oldest timepoint instead of being diluted throughout the whole window as it is done in _getAverageTick()

STATUS: fixed

DESCRIPTION:

In case where the oldest timepoint is still in the WINDOW, the **volatilityCumulative** is being diluted as it is divided by the whole time window (**WINDOW** > time passed), although at the same time **volumePerLiquidityCumulative** represents the metrics for only the time that has passed from the oldest timepoint. This can result in an unfair fee calculation during the first **WINDOW** of pool creation, e.g. volatility being high, with the fee staying low. Below presented is the code snippet from line 348 of **DataStorage.sol**.

```
  if (lteConsideringOverflow(oldest.blockTimestamp, time - WINDOW,
time)) {
    Timepoint memory startOfWindow = getSingleTimepoint(self, time,
WINDOW, tick, index, oldestIndex, liquidity);
    return (
     (endOfWindow.volatilityCumulative -
startOfWindow.volatilityCumulative) / WINDOW,
      uint256((endOfWindow.volumePerLiquidityCumulative -
startOfWindow.volumePerLiquidityCumulative)) >> 57
    );
  } else {
    return ((endOfWindow.volatilityCumulative) / WINDOW,
uint256((endOfWindow.volumePerLiquidityCumulative)) >> 57);
  }
 }
```

3. INCORRECT INTRA-WINDOW AVERAGES CALCULATION

SEVERITY: Medium

RESOLUTION: subtract the oldest timepoint's volatilityCumulative and volumePerLiquidityCumulative before returning the averages

STATUS: fixed

DESCRIPTION:

The case of the oldest timepoint being within the **WINDOW** is managed with an assumption that such situation is only possible at the first time window. As the timepoint array is **uint16.max** size (65536) and the default **WINDOW** is 86400 seconds, and also considering that only one timepoint can be written for a block, the assumption is true for the current setup on Polygon (~2 seconds block creation time), although in a case where the time window may be bigger in the future or the block creation time lowers (e.g. Polygon getting faster blocks, or the project is deployed on faster chains like Fantom or Avalanche with 0.5-1.5 seconds block creation time), in that case an illicit liquidity provider can spam the blocks with very low volume swaps to overflow the timepoint array index and bypass the check: **lteConsideringOverflow(oldest.blockTimestamp, time - WINDOW, time),** thus returning current **volatilityCumulative** without subtracting it with the windows start timepoint's volatility, which effectively will be equal to the whole accumulated volatility starting from the pool creation. This will result in an unfairly inflated fee.

```
  if (lteConsideringOverflow(oldest.blockTimestamp, time - WINDOW,
time)) {
    Timepoint memory startOfWindow = getSingleTimepoint(self, time,
WINDOW, tick, index, oldestIndex, liquidity);
    return (
     (endOfWindow.volatilityCumulative -
startOfWindow.volatilityCumulative) / WINDOW,
      uint256((endOfWindow.volumePerLiquidityCumulative -
startOfWindow.volumePerLiquidityCumulative)) >> 57
    );
  } else {
    return ((endOfWindow.volatilityCumulative) / WINDOW,
uint256((endOfWindow.volumePerLiquidityCumulative)) >> 57);
  }
 }
```

4. REDUNDANT CHECK

SEVERITY: Low

RESOLUTION: as the getFee() function is called throughout all the liquidity and swap operations, it is recommended to remove the redundant check

STATUS: fixed

DESCRIPTION:

The check in the function **getFee()** on line 39 in the file **AdaptiveFee.sol** is redundant since the fee is calculated via addition of **baseFee** and a sigmoid with maximum value of **alpha1** + **alpha2**. Also, the check is already implemented in **AlgebraFactory.sol** (line 109).

```
if (sumOfSigmoids > type(uint16).max) {
  // should be impossible
  sumOfSigmoids = type(uint16).max;
}

uint256 result = config.baseFee + sigmoid(volumePerLiquidity,
config.volumeGamma, uint16(sumOfSigmoids), config.volumeBeta);
if (result > type(uint16).max) {
  // should be impossible
  fee = type(uint16).max;
} else {
  fee = uint16(result);
}
```

5. MISLEADING ARGUMENT NAME

SEVERITY: Low

RESOLUTION: rename the argument

STATUS: fixed

DESCRIPTION:

The first argument of the **verifyCallback()** function is called "**factory**", but Algebra uses **poolDeployer** instead of the **factory** contract.

```
function verifyCallback(
    address factory,
    address tokenA,
    address tokenB
) internal view returns (IAlgebraPool pool) {
    return verifyCallback(factory, PoolAddress.getPoolKey(tokenA, tokenB));
}

/// @notice Returns the address of a valid Algebra Pool
/// @param factory The contract address of the Algebra factory
/// @param poolKey The identifying key of the V3 pool
/// @return pool The V3 pool contract address
function verifyCallback(address factory, PoolAddress.PoolKey memory poolKey)
    internal
    view
    returns (IAlgebraPool pool)
{
    pool = IAlgebraPool(PoolAddress.computeAddress(factory, poolKey));
    require(msg.sender == address(pool));
}
```

6. INCONSISTENCY OF DYNAMIC FEE AND VOLATILITY FORMULAS IN THE TECHNICAL DOCUMENTATION

SEVERITY: Low

RESOLUTION: update the technical documentation

STATUS: fixed

DESCRIPTION:

The functions **getFee()** and **_volatilityOnRange()** in the files **AdaptiveFee.sol** and **DataStorage.sol** respectively implement formulas that differ from the ones described in the technical paper:
fee formula uses a different sigmoid formula and also the **baseFee** is not included in the description. The volatility formula is described to be calculated with square root.