Algebra Integral

Bug Hunting Report

September 19th, 2023

Prepared By:

Riley Holterhus - Independent Contractor

Table of Contents

Summary ---------- 2

Findings ---------- 3

- **[H-01] Incorrect Tick Crossing in Virtual Pools** ---------- 3
- **[H-02] Read-Only Reentrancy + Deactivation Issues** ---------- 5
- **[L-01] Edge Case Uses Incorrect windowStartIndex** ---------- 7

Disclaimer ---------- 8

Summary

From September 11th, 2023, through September 19th, 2023, Algebra Finance contracted Riley Holterhus to independently review their new Integral codebase. During this period, a manual analysis was undertaken to identify various security issues and logic flaws.

Based on the potential impact and likelihood of occurrence, each finding has been assigned a severity level of Critical (C), High (H), Medium (M), or Low (L). Particular attention was given to Critical and High severity vulnerabilities, facilitated by an arrangement that incentivized finding such issues. In total, 2 High severity issues were identified, as well as 1 Low severity issue. All findings have been addressed.

Besides addressing the immediate issues, the Algebra team has taken steps to enhance the overall safety of the code. For example, to address a reentrancy issue, the team introduced additional safety measures, such as enhanced helper functions and more detailed documentation. This proactive approach is particularly useful given the modular nature of the Integral codebase, which is designed to accommodate plugins that may be implemented in the future. Fuzz testing had been established for several of the more complex components before this review, and the team has added new test cases in response to the identified vulnerabilities. These steps have further improved the code quality and made the codebase more robust.

The integral codebase can be found on GitHub at https://github.com/cryptoalgebra/Algebra/tree/dev. The review's starting commit hash was 801e10ea9e7ab147bc4764f8e020f0ad24e6e920, and all changes up to (and including) commit hash 19175092a3b1542cda63cc828c7239df409aec33 were reviewed.

[H-01] Incorrect Tick Crossing in Virtual Pools

Description: The `EternalVirtualPool` contracts are used for rewards accounting in the Algebra farming system. These contracts will essentially mirror the underlying `AlgebraPool` contracts, specifically keeping track of the subset of positions being farmed. The virtual pool's `crossTo()` function maintains this mirrored relationship, and is called after every swap.

Now, whenever a tick is crossed, the `globalPrevInitializedTick` and `globalNextInitializedTick` storage variables need to be updated as well. This is handled by the following highlighted code:

```
if (zeroToOne) {
  while (_globalTick != TickMath.MIN_TICK) {
    if (targetTick >= previousTick) break;
    unchecked {
      int128 liquidityDelta;
      _globalTick = previousTick - 1; // safe since tick index range is narrower than the data type
      (liquidityDelta, previousTick, nextTick) = ticks.cross(previousTick, rewardGrowth0, rewardGrowth1);
      _currentLiquidity = LiquidityMath.addDelta(_currentLiquidity, -liquidityDelta);
    }
  }
} else {
  while (_globalTick != TickMath.MAX_TICK - 1) {
    if (targetTick < nextTick) break;
    int128 liquidityDelta;
    _globalTick = nextTick;
    (liquidityDelta, previousTick, nextTick) = ticks.cross(nextTick, rewardGrowth0, rewardGrowth1);
    _currentLiquidity = LiquidityMath.addDelta(_currentLiquidity, liquidityDelta);
  }
}
currentLiquidity = _currentLiquidity;
globalTick = targetTick;
globalPrevInitializedTick = previousTick;
globalNextInitializedTick = nextTick;
```

Notice that `previousTick` and `nextTick` are updated using the return values of `ticks.cross()`. Since these return values represent the previous/next tick of *the tick being crossed*, this is incorrect. In a correct implementation, the tick that is crossed should itself become either the `previousTick` or the `nextTick`.

This bug will often break the rewards accounting. In the worst case, variables will underflow/overflow and LP funds will be frozen until a protocol admin can intervene.

Recommendation: Change the tick traversal to instead use the following logic:

```
if (zeroToOne) {
  while (_globalTick != TickMath.MIN_TICK) {
    if (targetTick >= previousTick) break;
    unchecked {
      int128 liquidityDelta;
      _globalTick = previousTick - 1; // safe since tick index range is narrower than the data type
      nextTick = previousTick;
      (liquidityDelta, previousTick, ) = ticks.cross(previousTick, rewardGrowth0, rewardGrowth1);
      _currentLiquidity = LiquidityMath.addDelta(_currentLiquidity, -liquidityDelta);
    }
  }
} else {
  while (_globalTick != TickMath.MAX_TICK - 1) {
    if (targetTick < nextTick) break;
    int128 liquidityDelta;
    _globalTick = nextTick;
    previousTick = nextTick;
    (liquidityDelta, , nextTick) = ticks.cross(nextTick, rewardGrowth0, rewardGrowth1);
    _currentLiquidity = LiquidityMath.addDelta(_currentLiquidity, liquidityDelta);
  }
```

Status: Fixed in commit 065fdf7a20840943eaf2f3d09653f8f5c2ccf2f9.

[H-02] Read-Only Reentrancy + Deactivation Issues

Description: When a user updates a farming position, the pool's current tick is read from the `AlgebraPool` global state. This current tick is later utilized in the virtual pool's `applyLiquidityDeltaToPosition()` function as follows:

```
function applyLiquidityDeltaToPosition(
    int24 bottomTick,
    int24 topTick,
    int128 liquidityDelta,
    int24 currentTick
  ) external override onlyFromFarming {
    uint128 _currentLiquidity = currentLiquidity;
    uint32 _prevTimestamp = prevTimestamp;
    bool _deactivated = deactivated;
    int24 _lastKnownTick = globalTick;
    int24 _nextActiveTick = globalNextInitializedTick;
    int24 _prevActiveTick = globalPrevInitializedTick;

    if (
       (currentTick < _nextActiveTick != _lastKnownTick < _nextActiveTick) ||
       (currentTick >= _prevActiveTick != _lastKnownTick >= _prevActiveTick)
    ) {
      _deactivated = true;
      deactivated = true;
    }

    if (!_deactivated) {
      globalTick = currentTick;
    }

    if (uint32(block.timestamp) > _prevTimestamp) {
      _distributeRewards(_prevTimestamp, _currentLiquidity);
    }

    if (liquidityDelta != 0) {
      bool flippedBottom = _updateTick(bottomTick, currentTick, liquidityDelta, false);
      bool flippedTop = _updateTick(topTick, currentTick, liquidityDelta, true);
      if (currentTick >= bottomTick && currentTick < topTick) {
        currentLiquidity = LiquidityMath.addDelta(_currentLiquidity, liquidityDelta);
      }
      if (flippedBottom || flippedTop) {
        _addOrRemoveTicks(bottomTick, topTick, flippedBottom, flippedTop, currentTick, liquidityDelta<0);
      }
    }
  }
```

Based on this control flow, there are two different issues which can combine to create a larger problem.

Firstly, note that *reading* the current tick does not invoke any reentrancy guards on the `AlgebraPool`. Also, note that the farming `crossTo()` function is called *after* each swap, meaning virtual pools are not synced until the underlying swap fully concludes. As a result, if a user modifies a farming position during the `algebraSwapCallback()`, no reentrant guards will be invoked and the unsynced variables would trigger virtual pool deactivation.

Secondly, notice that a deactivated virtual pool doesn't update its `globalTick` variable, implying the pool is meant to "freeze" at the time of deactivation. On the other hand, the `currentLiquidity` variable *does* change when deactivated. This is incorrect, as `currentLiquidity` might be decreased by values that never initially contributed to the sum.

To see how these issues can be exploited together, consider the following scenario:

1. The `currentTick` is 5
2. An attacker LPs and farms 1000 liquidity in tick range [0, 10)
3. The attacker LPs and farms 999 liquidity in tick range [20, 30)
4. The attacker swaps to move the tick to 25, and exits the 999 farming in the swap callback
5. Pool is deactivated, but 25 is in range of [20, 30) so `currentLiquidity` decreases from 1000 to 1
6. Swap finishes, `currentLiquidity` is incorrect and rewards-per-unit-liquidity generate 1000x faster
7. Attacker claims from their first position at the increased rate, stealing all pending rewards

Recommendation: To address the read-only reentrancy issue, check the pool's reentrancy guard when reading its global state. To address the deactivation issue, consider adding an early return when a deactivation scenario is detected.

Status: Read-only reentrancy was fixed in commit 14278e28f52e25c9431d529a213f07293f503c54 and PR 96. These changes fixed the immediate farming issue and added some helper functions/documentation to warn against future read-only reentrancy problems. Deactivation was fixed in PR 97. This PR added an early return if a deactivation is detected in the `applyLiquidityDeltaToPosition()` function.

[L-01] Edge Case Uses Incorrect windowStartIndex

Description: In the volatility oracle, each timepoint records its `windowStartIndex`. This value represents the most recent timepoint that was less than or equal to the timepoint's timestamp minus `WINDOW` seconds (or if no such point exists, `windowStartIndex` is simply the oldest timestamp).

In one specific case within `_getTickCumulativeAt()`, the target timestamp is strictly greater than the `beforeOrAt` variable's timestamp, but exactly equal to the `atOrAfter` variable's timestamp. However, in this scenario the index of the `beforeOrAt` value is returned as the `windowStartIndex`, which is incorrect:

```
uint32 target = time - secondsAgo;
(Timepoint storage beforeOrAt, Timepoint storage atOrAfter, bool samePoint, uint256 _indexBeforeOrAt) =
_getTimepointsAt(
  self,
  time,
  target,
  lastIndex,
  oldestIndex
);

(uint32 timestampBefore, int56 tickCumulativeBefore) = (beforeOrAt.blockTimestamp,
beforeOrAt.tickCumulative);
if (target == timestampBefore) return (tickCumulativeBefore, _indexBeforeOrAt);
if (samePoint) return ((tickCumulativeBefore + int56(tick) * int56(uint56(target - timestampBefore))),
_indexBeforeOrAt);

(uint32 timestampAfter, int56 tickCumulativeAfter) = (atOrAfter.blockTimestamp,
atOrAfter.tickCumulative);
if (target == timestampAfter) return (tickCumulativeAfter, _indexBeforeOrAt);
```

In reality, the *next* index is the index which satisfies the intended `windowStartIndex` behavior. Since the resulting window will start earlier than it should, it appears that the `getAverageVolatility()` function may have slightly incorrect calculations.

Recommendation: In this scenario, return the *next* index. This is `uint16(_indexBeforeOrAt+1)`.

Status: Fixed in commit 19175092a3b1542cda63cc828c7239df409aec33.

Disclaimer

This report is intended to detail the identified vulnerabilities of the reviewed smart contracts and should not be construed as an endorsement or recommendation of any project, individual or entity. While the author has made reasonable efforts to detect potential issues, the absence of any undetected vulnerabilities or issues cannot be guaranteed. Additionally, the security of the smart contracts may be affected by future changes or updates. By using the information in this report, you acknowledge that you are doing so at your own risk and that you should exercise your own judgment when implementing any recommendations or making decisions based on the findings. This report has been provided on an "as-is" basis and **DOES NOT CONSTITUTE A GUARANTEE OR WARRANTY OF ANY FORM**.