certora

Security Assessment

Final Report

Balancer

ReCLAMM Fix Review

June 2025

Prepared for Balancer Labs

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
ReCLAMM	reclamm	06b7d84	EVM

Project Overview

This document describes the verification of **ReCLAMM** using manual code review. The work was undertaken from **2 June 2025** to **9 June 2025**.

The following contract list is included in our scope:

```
contracts/ReClammPool.sol
contracts/ReClammPoolFactory.sol
contracts/lib/ReClammMath.sol
contracts/lib/ReClammPoolFactoryLib.sol
```

During the manual audit, the Certora team discovered issues in the Solidity contracts code, as listed on the following page.

Protocol Overview

The ReCLAMM is a liquidity pool design based on the constant product model, enhanced with virtual balances to concentrate liquidity within a dynamic price range. Unlike traditional concentrated liquidity pools, where liquidity providers must actively manage their positions as prices move, ReCLAMM automatically readjusts the active price range. This enables passive liquidity provision while maintaining capital efficiency and targeted exposure near the market price. The pool is configured through two parameters: the price range ratio, which defines the width of the concentrated liquidity bracket, and the daily price shift rate, which controls how quickly the bracket moves with market price changes.

This fix review addressed two key areas of improvement. First, it focused on reviewing the simplified mathematical operations made to enhance code clarity and reduce computational complexity. Second, it ensured that new price ratio updates do not silently overwrite ongoing ones, preventing potential issues with outdated virtual balances that could compromise the pool's accuracy and functionality.

Security Considerations

The main security consideration was that the system kept 1:1 functionality with its original version after the introduction of simplified mathematical operations and a fix to how new price ratio updates are processed.

Audit Goals

1. Verifying the simplification of math operations did not introduce any accounting mistakes and the code still behaves exactly the same as the more complex previous version.

2. Verifying the fixes from the last audit have been implemented correctly, specially ensuring new price ratio updates are processed correctly.

3. Ensuring the pricing mechanism works as intended under edge cases and there are no scenarios where the various moving parts (price shift updates, price ratio updates, etc.) could interact to produce a bad state such as :
    a. Pools with low liquidity.
    b. High value transactions and flashloans.
    c. Tokens with different decimals.

4. Ensure nobody can change or manipulate the price ratio besides the owner of the pool.

Coverage and Conclusions

1. We have concluded the simplification of math operations did not introduce any accounting mistakes and the code still behaves exactly the same as the more complex previous version.

2. The fixes from the last audit have been implemented correctly, and new price ratio updates are now processed correctly.

3. The pricing mechanism works as intended under edge and there are no scenarios where the various moving parts (price shift updates, price ratio updates, etc.) could interact to produce unwanted pool states.

4. It is not possible for anybody besides the pool owner to change or manipulate the price ratio besides.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	-	-	-
Informational	7	7	6
Total	7	7	6

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
I-01	Missing, outdated or misleading NatSpec	Informational	Fixed
I-02	Missing explicit validation for initialMinPrice < initialMaxPrice	Informational	Fixed
I-03	Unused function ReClammMath.pow4()	Informational	Fixed
I-04	Make onBeforeRemoveLiquidity more symmetrical to its mirror function onBeforeAddLiquidity	Informational	Fixed
I-05	Pool price should not be used as oracle	Informational	Fixed
I-06	Inconsistent direction of initialization price checks	Informational	Acknowledged
I-07	Zero startFourthRootPriceRatio on initial _startPriceRatioUpdate	Informational	Fixed

Informational Severity Issues

I-01 Missing, outdated or misleading NatSpec

Description: The audited codebase contains several NatSpec annotations and inline comments that are missing, outdated, or inaccurate, which may reduce overall documentation quality and hinder clear understanding of contract behaviors:

- As per the latest changes, the pool is defined as `IN RANGE` if the centeredness is greater than or equal to the centeredness margin. However, there are still parts of the codebase in which the comments or NatSpec are outdated.
  - **ReClammPool:**

```
Unset
    // Used to define the target price range of the pool (i.e., where the pool centeredness >
centeredness margin).
    uint64 internal _centerednessMargin;
```

  - **IReClammPool:**

```
Unset
    /**
     * @notice Compute whether the pool is within the target price range.
     * @dev The pool is considered to be in the target range when the centeredness is greater
than the centeredness
     * margin (i.e., the price is within the subset of the total price range defined by the
centeredness margin.)
     * ...
     */
```

```
Unset
    /**
     * ...
     * @return isWithinTargetRange True if pool centeredness is greater than the centeredness
margin
     */
```

- The NatSpec of `ReClammMath.computeTheoreticalPriceRatioAndBalances()` is outdated. It still mentions the `fourthRootPriceRatio` being calculated by the function whereas the function actually calculates the raw `priceRatio`, i.e. the ratio of the max price to the min price.

```
Unset
  /**
     * ...
     * @dev This function calculates three key components needed to initialize a ReClamm
pool:
     * ...
     * 3. Fourth root price ratio - A key parameter that helps define the pool's price
boundaries
     * ...
     */
```

- The `ReClammPriceParams` struct lists `initialMinPrice`, `initialMaxPrice`, and `initialTargetPrice` but does not specify they must be 18-decimal fixed-point values. For example, passing `3000` instead of `3000e18` will break the pool's internal math and revert during initialization.

Recommendation: Fix all missing, outdated or misleading NatSpec and comments.

Customer's response: Fixed in f5a4cea.

Fix review: Fix confirmed.

I-02 Missing explicit validation for initialMinPrice < initialMaxPrice

Description: The `ReClammPool` constructor allows `initialMinPrice` and `initialMaxPrice` to be equal. If both values are identical, the subsequent call to `computeTheoreticalPriceRatioAndBalances()` will revert due to a division by zero because `sqrtPriceRatio - FixedPoint.ONE` will be equal to `0`.

Although such a configuration is considered invalid, the lack of an explicit check in the constructor means that gas is wasted before the low-level revert occurs.

Recommendation:

Insert a validation in the `ReClammPool` constructor to ensure `initialMinPrice < initialMaxPrice`. For example:

```
Unset
if (params.initialMinPrice >= params.initialMaxPrice) { revert MinEqualsMaxPrice(); }
```

This early guard prevents wasted gas and yields a clear error when the price range is zero or inverted.

Customer's response: Fixed in f5a4cea.

Fix review: Fix confirmed.

I-03 Unused function ReClammMath.pow4()

Description: The function `ReClammMath.pow4()` is only used in tests and is not part of the ReCLAMM operating system.

Recommendation: Consider moving them as helper functions in test-related contracts.

Customer's response: Fixed in f5a4cea.

Fix review: Fix confirmed.

I-04 Make onBeforeRemoveLiquidity() more symmetrical to its mirror function onBeforeAddLiquidity()

Description: The functions `onBeforeAddLiquidity()` and `onBeforeRemoveLiquidity()` are mirror functions of each other. To improve code readability we recommend to maintain consistency between the in how their equivalent variables are named and used.

Recommendation: Update the implementation of `onBeforeRemoveLiquidity()` as follows:

```
Unset
- uint256 bptDelta = poolTotalSupply - maxBptAmountIn;
+ uint256 proportion = (poolTotalSupply - maxBptAmountIn).divDown(poolTotalSupply);

// When adding/removing liquidity, round down the virtual balances. This favors the vault in
swap operations.
// The virtual balances are not used in proportional add/remove calculations.
- currentVirtualBalanceA = currentVirtualBalanceA.mulDown(bptDelta).divDown(poolTotalSupply);
- currentVirtualBalanceB = currentVirtualBalanceB.mulDown(bptDelta).divDown(poolTotalSupply);
+ currentVirtualBalanceA = currentVirtualBalanceA.mulDown(proportion);
+ currentVirtualBalanceB = currentVirtualBalanceB.mulDown(proportion);
```

Customer's response: Fixed in f5a4cea.

Fix review: Fix confirmed.

I-05. Pool price should not be used as oracle

Description: It is important to highlight in the documentation that it is extremely important not to use ReCLAMM pools as price oracles. This will help foster security in the general ecosystem by preventing third party integrators from making unintentional mistakes.

Recommendation: Update the project's documentation and in-code NatSpec comments to clearly warn integrators not to use any pool as a price oracle.

Customer's response: Fixed in f5a4cea.

Fix review: Fix confirmed.

I-06. Inconsistent direction of initialization price checks

Description: The `_checkInitializationPrices()` function validates that the spot price and price range derived from the provided balances fall within a tolerance band of the initialization parameters (`minPrice`, `maxPrice`, and `targetPrice`). However, this logic assumes the initialization parameters are the base values and checks whether the *derived price* is within tolerance of them.

This approach is conceptually inconsistent with `_checkInitializationBalanceRatio()`, which treats the theoretical values as the ground truth and validates the user-provided input against them. While both approaches are mathematically valid, maintaining consistency in validation improves code clarity and auditability.

Recommendation: Consider aligning the direction of the comparison in `_checkInitializationPrices()` with `_checkInitializationBalanceRatio()`, treating the theoretical price as the reference and validating the provided price against it.

Additionally, update comments or variable names to clarify which value is considered authoritative.

Customer's response: Acknowledged.

I-07 Zero startFourthRootPriceRatio on initial _startPriceRatioUpdate()

Description: On the very first call to `_startPriceRatioUpdate(locals.priceRatio, block.timestamp, block.timestamp)` inside `onBeforeInitialize()`, the code takes the "not yet initialized" branch:

```
Unset
if (_vault.isPoolInitialized(address(this))) {
    // … (never taken on first init)
} else {
    startFourthRootPriceRatio = priceRatioState.endFourthRootPriceRatio; // == 0
    startPriceRatio = endPriceRatio;
}
endFourthRootPriceRatio = fourthRoot(endPriceRatio);
```

Because `priceRatioState.endFourthRootPriceRatio` is still its default of `0`, the stored `startFourthRootPriceRatio` becomes `0`, even though the intended `start` and `end` should both equal `fourthRoot(locals.priceRatio)`. Functionally this does not change execution—since `priceRatioUpdateStartTime == priceRatioUpdateEndTime`, any subsequent price ratio read (even in the same block as the created pool) returns `endFourthRootPriceRatio`. However, it means:

- The emitted `PriceRatioStateUpdated` event will show `startFourthRootPriceRatio = 0` and `endFourthRootPriceRatio = fourthRoot(locals.priceRatio)`, which is misleading.
- The on-chain state temporarily records a start of `0`, even though the true ratio was never zero.

Recommendation: In the "pool not initialized" branch of `_startPriceRatioUpdate()`, initialize both `startFourthRootPriceRatio()` and `endFourthRootPriceRatio` to `fourthRoot(endPriceRatio)` instead of inheriting the default zero.

Customer's response: Fixed in f5a4cea.

Fix review: Fix confirmed.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the rarest & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.