Security Assessment
Final Report

ReCLAMM

April 2025

Prepared for Balancer Labs

certora Balancer

Table of content

certora Balancer

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
ReCLAMM	reclamm	8207b33	EVM

Project Overview

This document describes the verification of **ReCLAMM** using manual code review. The work was undertaken from **15 April 2025** to **29 April 2025**.

The following contract list is included in our scope:

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

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	1	1	1
Low	2	2	1
Informational	5	5	5
Total	8	8	7

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
M-01	New price ratio update can silently overwrite an ongoing one and may potentially lead to outdated virtual balances	Medium	Fixed
L-01	setCenterednessMargin() can be griefed via a sandwich attack	Low	Acknowledged
L-02	The pool-within-range check does not match the specification	Low	Fixed
I-01	Intermediate floor division undermines intended rounding up	Informational	Fixed
I-02	Missing zero check for initialMaxPrice during pool initialization	Informational	Fixed
I-03	Inconsistent calculation of maxPrice across the protocol	Informational	Fixed
I-04	Superfluous lower bound check in _setCenterednessMargin()	Informational	Fixed
I-05	Hardcoded token indexes reduce clarity despite defined constants	Informational	Fixed

Medium Severity Issues

M-01 New price ratio update can silently overwrite an ongoing one and may potentially lead to outdated virtual balances

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: ReClammPool.sol#L611-L614	Status: Fixed	

Description: `ReClammPool.setPriceRatioState()` is used to schedule a gradual change in the pool's price ratio over time. However, this function allows an ongoing update to be silently overwritten and cancelled if a new update is scheduled before the previous one completes. If no user interactions occur during the original update window, the pool may continue operating with outdated virtual balances, leading to misaligned liquidity brackets and unintended pricing behavior.

When a new price ratio update is scheduled via `_setPriceRatioState()`, the function does not check whether a previous update is still ongoing. The `startFourthRootPriceRatio` is set based on the current interpolated value, computed by `_computeCurrentFourthRootPriceRatio()`. This allows a new update to be scheduled while the current one is still in progress, silently cancelling the ongoing update and using the current interpolated value as the new starting point.

Moreover, if no user interactions (e.g., swaps or liquidity changes) have occurred during the now-cancelled update window, the pool's virtual balances will remain outdated. This is because if a user interacts with the protocol before the new update window begins, `computeCurrentVirtualBalances()` will not trigger a recalculation, as the following condition will evaluate to `false`:

```
Unset
if (
    currentTimestamp > priceRatioState.priceRatioUpdateStartTime &&
    lastTimestamp < priceRatioState.priceRatioUpdateEndTime
)
```

As a result, the virtual balances are not updated based on the originally scheduled curve, leaving the pool operating on stale parameters.

Example Scenario:

1. A price ratio update is scheduled between timestamps `100` and `200`.
2. No swaps or liquidity actions are performed, so virtual balances are not updated.
3. At timestamp `150`, a new update is scheduled to run from `200` to `300`. The prior update is silently overwritten.
4. At timestamp `175`, a user interacts with the pool. However, the update condition is not satisfied, since the current `priceRatioUpdateStartTime` is now in the future.
5. As a result, virtual balances are never updated based on the originally intended price ratio curve between `100` and `200`, leading to incorrect pricing.

Recommendations: Introduce a check to ensure that no update is currently in progress before allowing a new one to be scheduled. For example:

```
Unset
require(
    block.timestamp >= _priceRatioState.priceRatioUpdateEndTime,
    "Price ratio update in progress"
);
```

Alternatively, apply the effects of the in-progress update (e.g., recomputing virtual balances) before overwriting the state.

Customer's response: Fixed in 23370ad.

Fix Review: Fixed confirmed.

Low Severity Issues

L-01 setCenterednessMargin() can be griefed via a sandwich attack		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: ReClammPool.sol#L559	Status: Acknowledged	

Description: The `setCenterednessMargin()` function uses the `onlyWithinTargetRange` modifier, which enforces that the pool remains within the targeted centeredness range between the margins both before and after the update. This is done via two calls to `_ensurePoolWithinTargetRange()` surrounding the main logic.

Because centeredness depends on the current pool price, a malicious actor can grief the transaction and cause it to revert due to failed in-range check by manipulating the price just before it's mined. For example, if the current margin is `0.15` and governance or swap fee manager attempts to lower it to `0.1`, an attacker can sandwich the transaction and swap to shift the centeredness just below `0.15`, e.g. to `0.149`, causing the pre-check to fail and reverting the transaction. This allows a malicious actor to block margin changes under certain conditions.

This griefing vector is most effective when the current centeredness is already near the margin threshold, requiring only a small price movement. If not, the attacker would need to use large capital to shift the pool price, which may be economically impractical. This constraint reduces the likelihood and impact of the attack in most real-world scenarios.

Recommendations: Adopt a gradual update mechanism (similar to `GradualValueChange` used in `setPriceRatioState()`) where the margin is interpolated over time based on a start and end value. This removes hard state transitions and eliminates griefing opportunities based on sudden centeredness margin changes.

Customer's response: Acknowledged.

L-02 The pool-within-range check does not match the specification		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: ReClammMath.sol#L513	Status: Fixed	

Description: The function `isPoolWithinTargetRange()` currently uses a `>=` comparison to determine whether the pool is within range:

```
Unset
return centeredness >= centerednessMargin;
```

However, according to the specification:

> Margin is a number from 0 to 1, very similar to pool centeredness, and helps calculate whether the pool is `IN RANGE` (Pool Centeredness > Margin) or `OUT OF RANGE` (Pool Centeredness ≤ Margin). If the pool is `OUT OF RANGE`, the price interval will be recalculated.

This means that the pool should only be considered in range if `centeredness > centerednessMargin`. But under the current implementation, a pool with `centeredness == centerednessMargin` is treated as in range, which violates the spec. This mismatch may result in incorrectly skipping virtual balance recalculations, potentially leaving the pool in a suboptimal pricing range.

Recommendations: Update the implementation to use a strict inequality:

```
Unset
return centeredness > centerednessMargin;
```

Alternatively, if the current behavior is intentional, the specification and NatSpec should be updated to reflect this decision accurately.

Customer's response: Fixed in b20436c.

Fix Review: Fix confirmed.

Informational Severity Issues

I-01. Intermediate floor division undermines intended rounding up

Description: The following expression intends to round up the result of a multi-term division:

```
Unset
// Round up the centeredness, so the virtual balances are rounded down when the pool prices
are moving.
return
    ((balancesScaled18[indexTokenOvervalued] * virtualBalances[indexTokenUndervalued]) /

balancesScaled18[indexTokenUndervalued]).divUp(virtualBalances[indexTokenOvervalued]);
```

However, the initial division (`/ balancesScaled18[...]`) performs a floor division, which causes the intermediate result to round down before `divUp()` is applied. As a result, the final output may be lower than the mathematically correct ceiling of the full expression.

The simplified expression looks like this:

```
Unset
(A * B / C).divUp(D)
```

Due to the initial floor division, the final result may be less than the mathematically correct ceiling of:

```
Unset
(A * B).divUp(C * D)
```

Example:

- Let `A = 5`, `B = 5`, `C = 3`, and `D = 2`.
- Current implementation: `floor(5 * 5 / 3) = 8`, then `ceil(8/2) = 4`
- Expected implementation: `ceil(5 * 5 / 3 * 2) = 5`

Recommendations: To ensure the result is correctly rounded up, perform the full multiplication and apply a single division with rounding:

```
Unset
return
    (balancesScaled18[indexTokenOvervalued] * virtualBalances[indexTokenUndervalued])
        .divUp(balancesScaled18[indexTokenUndervalued] *
virtualBalances[indexTokenOvervalued]);
```

Customer's response: Fixed in b20436c.

Fix Review: Fix confirmed.

I-02. Missing zero check for initialMaxPrice during pool initialization

Description: The `ReClammPool` constructor checks that `initialMinPrice` and `initialTargetPrice` are non-zero. However, it fails to apply the same validation to `initialMaxPrice`, which is also used to initialize an immutable variable and is equally important to the contract's configuration:

```
Unset
if (params.initialMinPrice == 0 || params.initialTargetPrice == 0) {
    // If either of these prices were 0, pool initialization would fail with division by
zero.
    revert InvalidInitialPrice();
}
```

Recommendation: Add a zero check for `params.initialMaxPrice` alongside the existing validation for `initialMinPrice` and `initialTargetPrice` to ensure safe input and maintain consistent input validation.

Customer's response: Fixed in b20436c.

Fix Review: Fix confirmed.

I-03. Inconsistent calculation of maxPrice across the protocol

Description: The protocol calculates `maxPrice` using two mathematically equivalent expressions in different parts of the codebase, but with different rounding behavior due to fixed-point arithmetic:

In `_checkInitializationPrices()`:

```
Unset
uint256 currentMaxPrice = currentInvariant.divDown(virtualBalanceA).divDown(virtualBalanceA);
```

In `computeCurrentPriceRange()`:

```
Unset
maxPrice = currentInvariant.divDown(virtualBalanceA.mulDown(virtualBalanceA));
```

Although both approaches intend to compute $\frac{currentInvariant}{virtualBalanceA^2}$, the use of chained `divDown()` calls in the first case leads to double truncation. This can produce slight inconsistencies due to how `divDown()` performs fixed-point division:

```
Unset
function divDown(uint256 x, uint256 y) internal pure returns (uint256) {
    return x * 1e18 / y;
}
```

For example, with:

- `currentInvariant = 100e18`
- `virtualBalanceA = 0.3e18`

The results are:

- `111111111111111111111` using the single-step expression
- `111111111111111111110` using the chained expression

This 1 wei difference stems from intermediate rounding, which may lead to inconsistencies.

Recommendation: Unify the calculation of `maxPrice` across the protocol using the single-step expression to minimize rounding errors and ensure consistency:

```
Unset
maxPrice = currentInvariant.divDown(virtualBalanceA.mulDown(virtualBalanceA));
```

Customer's response: Fixed in b20436c.

Fix Review: Fix confirmed.

I-04. Superfluous lower bound check in _setCenterednessMargin()

Description: The `_setCenterednessMargin()` function includes a lower-bound check that is unnecessary:

```
Unset
if (centerednessMargin < _MIN_CENTEREDNESS_MARGIN || centerednessMargin >
_MAX_CENTEREDNESS_MARGIN) {
    revert InvalidCenterednessMargin();
}
```

Here, `_MIN_CENTEREDNESS_MARGIN` is defined as `0`, but `centerednessMargin` is a `uint256` and can never be negative. Therefore, `centerednessMargin < _MIN_CENTEREDNESS_MARGIN` is always `false` and serves no practical purpose.

Recommendation: Simplify the check by removing the unnecessary condition:

```
Unset
if (centerednessMargin > _MAX_CENTEREDNESS_MARGIN) {
    revert InvalidCenterednessMargin();
}
```

Customer's response: Fixed in b20436c.

Fix Review: Fix confirmed.

I-05. Hardcoded token indexes reduce clarity despite defined constants

Description: Although constants `a = 0` and `b = 1` are defined to improve readability, some functions still use hardcoded values `0` and `1` when assigning `indexTokenUndervalued` and `indexTokenOvervalued`:

```
Unset
(uint256 indexTokenUndervalued, uint256 indexTokenOvervalued) = isPoolAboveCenter ? (0, 1) :
(1, 0);
```

This reduces consistency and makes the code slightly harder to follow..

Recommendation: Use `a` and `b` in place of hardcoded indexes to improve clarity and maintain consistency in the following functions:

- `calculateVirtualBalancesUpdatingPriceRatio()`
- `calculateVirtualBalancesUpdatingPriceRange()`
- `computeCenteredness()`

Customer's response: Fixed in ae0d88e.

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 most rare & 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.