Security Assessment

Final Report

Nested Pool Router functions

September 2025

Prepared for Balancer

Table of content

certora Balancer

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Balancer	balancer-v3	64ed31e	EVM

Project Overview

This document describes the verification of **Nested Pool add and remove liquidity operations in the CompositeLiquidityRouter** using manual code review.
The work was undertaken from **September 4** to **September 8, 2025**.

The scope is limited to the following functions in vault/contracts/CompositeLiquidityRouter.sol:

```
CompositeLiquidityRouter::addLiquidityUnbalancedNestedPool()
CompositeLiquidityRouter::removeLiquidityProportionalNestedPool()
```

Protocol Overview

Nested pool operations in the Balancer V3 protocol allow users to add or remove liquidity from pools that themselves contain other pools or ERC4626 vaults as tokens (known as "nested pools"). This enables complex pool structures, such as meta-pools or hierarchical pools, where a parent pool can hold BPTs (Balancer Pool Tokens) of child pools, ERC20 tokens, and ERC4626 vault tokens.

When adding liquidity, the router coordinates deposits into child pools first (if needed), wraps tokens (ERC4626), and then settles the parent pool's liquidity.

When removing liquidity, the router can recursively withdraw from child pools, unwrap tokens, and return the correct assets to the user.

Transient storage and sets are used to track which tokens and amounts are being processed, ensuring correct accounting and preventing double-counting.

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	1	1	1
Informational	2	2	2
Total			

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
M-01	Incorrect minAmountsOut check when removing from pool in Recovery mode	Medium	Fixed in commit 4034469
L-01	Inconsistent behaviour when parent pool and child pool contain the same token	Low	Fixed in commit 4034469
I-01	Redundant check for currentSwapTokensOut().contains(tokenOut)	Informational	Fixed in commit 4034469
I-02	Unreachable InvalidTokenType Error Branches	Informational	Fixed in commit 4034469

Medium Severity Issues

M-01 Incorrect minAmountsOut check when removing from pool in Recovery mode		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files:	Status: Not Fixed	

Description:

In the `removeLiquidityProportionalNestedPoolHook` function at line 472, there is a parameter mismatch when the parent pool is in recovery mode. The function passes `params.minAmountsOut` directly to `_vault.removeLiquidityRecovery()`, but these arrays represent different token sets:

`params.minAmountsOut` corresponds to `tokensOut`, which represents the final output tokens that users expect to receive (potentially including child pool tokens, unwrapped ERC4626 tokens, etc.)

`_vault.removeLiquidityRecovery()` expects minimum amounts corresponding to `parentPoolTokens`, which are the actual tokens held by the parent pool

Impact of the incorrect check:

- Incorrect slippage protection: Users' minimum amount expectations are applied to wrong tokens
- Potential transaction failures: Recovery mode operations may revert due to mismatched array lengths or incorrect minimum amount checks
- Security risk: Users may receive less than expected amounts without proper slippage protection on the correct tokens

Proof of Concept: Consider a nested pool where:

```
None
Parent pool has tokens: [BPT_A, USDC]
Child pool BPT_A has tokens [DAI, USDT]

User expects output tokens: [DAI, USDT, USDC] (from unwrapping BPT_A)

minAmountsOut = [100, 200, 300] for [DAI, USDT, USDC]

removeLiquidityRecovery applies [100, 200] as minimums for [BPT_A, USDC]
```

Recommendations:
The recovery mode path should use zero minimums for the parent pool operation (similar to the non-recovery path on line 477) and apply the actual minimum amount checks later in the function where `params.minAmountsOut[i]` is properly validated against the final output amounts at line 583:

Customer's response:
Fixed in commit 4034469

Fix Review:
The call to `_vault.removeLiquidityRecovery()` now uses `0` for min amounts out by supplying `new uint256[](parentPoolTokens.length)` instead of `params.minAmountsOut` for the `minAmountsOut` parameter.
This correctly fixes the issue.

Low Severity Issues

L-01 Inconsistent behaviour when parent pool and child pool contain the same token		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files:	Status: Not Fixed	

Description:
`addLiquidityUnbalancedNestedPool` does not correctly handle nested pools where the parent pool and child pool(s) contain overlapping tokens.
The `_settledTokenAmounts()` mapping is used to prevent double-processing of tokens during nested pool operations. However, when the same token exists in both parent and child pools, the accounting logic fails:

When the child pool (BPT) contains TokenA, Depending on the order of tokens and token types, addLiquidityUnbalancedNestedPool will:

- Ignore amountsIn for child pool when parent pool has [TokenA, BPT]
- Fail when parent pool has [BPT, TokenA]

Recommendations:
Given that configurations where the parent pool and child pool share the same token are rare but technically possible, the simplest and most effective solution is to explicitly prohibit such setups. This can be achieved by adding validation logic to detect shared tokens and reverting the transaction with a clear error message. This approach ensures that users are immediately informed of the unsupported configuration, preventing unexpected behavior.

If the decision is made not to support shared token configurations, it is recommended to also document this limitation clearly.

Customer's response:

A comment was added in commit 4034469 to state that "overlapping" tokens between parent and child pool are not supported. The gas cost to explicitly detect this rare edge case would be too high to enforce this on chain.

Informational Severity Issues

I-01. Redundant check for _currentSwapTokensOut().contains(tokenOut)

Description:

`removeLiquidityProportionalNestedPoolHook` contains the following code:

```
Rust
uint256 tokenIndex = _currentSwapTokensOut().indexOf(tokenOut);

if (_currentSwapTokensOut().contains(tokenOut) == false || checkedTokenIndexes[tokenIndex]) {
      // If tokenOut is not in transient tokens out array or repeated, the tokensOut array is wrong.
     revert ICompositeLiquidityRouterErrors.WrongTokensOut(_currentSwapTokensOut().values(), tokensOut);
}
```

In the case where `_currentSwapTokensOut()` does not contain `tokenOut`, the call to `_currentSwapTokensOut().indexOf(tokenOut)` will revert.
This means that `_currentSwapTokensOut().contains(tokenOut) == false` can never be true

Recommendations:

Remove the redundant `_currentSwapTokensOut().contains(tokenOut) == false` check

Customer's response:

Fixed in commit 4034469. The redundant check is removed and a comment is added that indexOf will revert if tokenOut is not in`_currentSwapTokensOut.

I-02. Unreachable InvalidTokenType Error Branches.

Description:
Two revert branches using `InvalidTokenType` in **CompositeLiquidityRouterHooks.sol** are logically unreachable.
In `_addLiquidityToParentPool`, after handling `BPT` and `ERC4626`, the only remaining enum value is `ERC20`, making the subsequent else `if (parentPoolTokenType != ERC20)` impossible. In `_addLiquidityToChildPool`, after handling `ERC4626`, the conditional that reverts when the type is neither `ERC20` nor `BPT` can never trigger because `_computeEffectiveCompositeTokenType` only returns `ERC20`, `BPT`, or `ERC4626`.
This dead code adds noise, slightly increases bytecode size, and may confuse reviewers about nonexistent edge cases.

Recommendations:
Remove both unreachable `InvalidTokenType` branches and add comments to reflect that all enum cases are exhaustively handled.

Customer's response:
Fixed in commit 4034469. The error path is there to not silently proceed when a new token type is added in the future. For clarity, a comment "*Future-proofing against later addition of token types.*" was added in the code.

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.