Security Assessment & Formal Verification Final Report

Balancer V3

September 2024

Prepared for Balancer

certora Balancer

Table of content

certora Balancer

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Balancer V3	balancer-v3-monorepo	e4455af	EVM

Project Overview

This document describes the specification and verification of Balancer V3 using the Certora Prover and manual code review findings. The work was undertaken from August 6th 2024 to September 19th 2024.

All contracts in the balancer-v3-monorepo are considered in scope.

During the course of the audit and verification process, some significant changes and updates were made to the project code. It was agreed with Balancer to not limit the audit to a specific commit, but to use the most recent version.
Due to this, the audit was performed on several code versions between commit 8df7df5 from August 3, 2024 up until e4455af September 4, 2024. Some known issues that were present in earlier commits are not included in this report as they were not independently found by us.

The team performed a manual audit of all the Solidity contracts**.** In addition, the Certora Prover demonstrated that the implementation of the **Solidity** contracts above is correct with respect to the formal rules written by the Certora team. During the verification process and the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed in the following.

Protocol Overview

Balancer v3 is the successor of v2 and is a decentralized automated market maker (AMM) protocol built on Ethereum with a clear focus on fungible and yield-bearing liquidity.
Balancer v3's architecture focuses on simplicity, flexibility, and extensibility at its core. The v3 vault more formally defines the requirements of a custom pool, shifting core design patterns out of the pool and into the vault.

Balancer Pools are smart contracts that define how traders can swap between tokens on Balancer Protocol, and the architecture of Balancer Protocol empowers anyone to create custom pool types.

What makes Balancer Pools unique from those of other protocols is their unparalleled flexibility. With the introduction of Hooks and Dynamic Swap Fees, the degree of customization is boundless.

A major innovation in v3 is native Vault support for the ERC4626 standard. This allows boosted pools to have 100% capital efficiency. A boosted pool is simply a pool of any type containing all ERC4626 wrapped tokens. Vault buffers and convenience functions in the Router allow users to interact with boosted pools as if they were standard pools containing only underlying tokens.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	2	2	2
High	1	1	1
Medium	3	3	3
Low	4	4	3
Informational	13		
Total	23	10	9

Severity Matrix

Impact					
	Critical	Medium	Medium/High	High	Critical
	High	Low/Medium	Medium	Medium/High	High
	Medium	Low	Low/Medium	Medium	Medium/High
	Low	Informational	Low	Low/Medium	Medium
		Rare	Unlikely	Likely	Very Likely
			Likelihood		

Detailed Findings

ID	Title	Severity	Status
C-01	Unused token approval in_wrapWithBuffer can drain the vault	Critical	Fixed
C-02	The underlying token of ERC4626 can be changed to steal assets	Critical	Fixed
H-01	Any user can steal the yield of the wrapped tokens that are kept in the buffer	High	Fixed
M-01	A malicious user can cause a DoS in the erc4626BufferWrapOrUnwrap() external function by front-running the caller	Medium	Fixed
M-02	Swap fees are different depending on if EXACT_IN or EXACT_OUT are being used	Medium	Fixed
M-03	Rounding errors in weighted pool invariant allows to take out more tokens than added	Medium	Fixed
L-01	initializeBuffer can only be called when it mints double the minimal amount	Low	Fixed

L-02	Transposed digits in a constant value	Low	Fixed
L-03	It is possible to pay less fees by splitting the deposited amounts	Low	Known Issue
L-04	ReEntrancy Pool initialization possible	Low	Fixed
I-01	Fees not included in PoolBalanceChanged event for add and remove liquidity	Informational	
I-02	RouterCommon unnecessarily approves WETH to the vault	Informational	
I-03	Unnecessary self approval in Router.queryRemoveLiquidityHook	Informational	
I-04	In withdrawProtocolFees 1 bad token in pool can block fee collection of other tokens in the pool	Informational	
I-05	No support for ERC4626 buffers with Fee or slippage	Informational	
I-06	HooksConfigLib - Amounts ignored when enableHookAdjustedAmounts is false	Informational	
I-07	No custom error for tokens > 18 decimals	Informational	

I-08	Inconsistent function naming PoolConfigLib requireRemoveCustomLiquidityEnabled and requireAddCustomLiquidityEnabled	Informational	
I-09	Incorrect natSpec in VaultCommon.sol:_findTokenIndex	Informational	
I-10	Incorrect comments in FixedPoint.sol MulDivUp + mulUp	Informational	
I-11	Incorrect comment in VaultExtension getPoolConfig	Informational	
I-12	Lots of duplicate code between VaultAdmin initializeBuffer and addLiquidityToBuffer	Informational	
I-13	Suggestion for VaultExplorer	Informational	

Critical Severity Issues

C-01 Unused token approval in _wrapWithBuffer can drain the vault		
Severity: **Critical**	Impact: **Critical**	Likelihood: **Very Likely**
Files: Vault.sol	Status: Fixed in 2b0fa9d	Violated Property: None

Description: When calling _wrapWithBuffer() with SwapKind.EXACT_IN it gives token approval of underlyingToken to the ERC4626 vault before calling deposit.

After the deposit it uses the real transferred tokens to do the accounting.

Exploit Scenario: Doing a wrap of high amount, while not doing anything in deposit, will end with a large token approval to the erc4626 vault, allowing it to take those tokens.

Recommendations: Revoke the token approval after the deposit call

Customer's response: Fixed.

Fix Review: Token approval is revoked after the call to the vault.
After this this issue is not exploitable anymore.

C-02 The underlying token of ERC4626 can be changed to steal assets

Severity: **Critical**	Impact: **Critical**	Likelihood: **Very Likely**
Files: VaultAdmin.sol	Status: Fixed in 4d45660	Violated Property: underlyingAssetNotZeroAddress in P-01

Description: When adding liquidity to an ERC4626 buffer, a check is made to check that the underlying asset of the ERC4626 vault has not been changed.
This check can be bypassed to add liquidity and change the underlying token afterwards, allowing an attacker to steal all of the tokens in the vault.

```
Unset
    function addLiquidityToBuffer(...) … {
       address underlyingToken = wrappedToken.asset();

       // Amount of shares to issue is the total underlying token that the
       //user isd epositing.
       issuedShares = wrappedToken.convertToAssets(amountWrapped) + amountUnderlying;

       if (_bufferAssets[IERC20(address(wrappedToken))] == address(0)) {
```

The check if the asset is set or not is done by checking that the stored underlyingToken == 0.
By having wrappedToken.asset() return address(0) it is possible to add liquidity and later change the asset to another value.

Exploit Scenario: Create an ERC4626 contract that can return different values for asset().

1. call addLiquidity when the ERC4626 contract returns address(0) for asset().

```
Unset
  call addLiquidityToBuffer(
     wrappedToken, amountUnderlying. = 0, amountWrapped = 1e30, sharesOwner)
```

This will add shares, and set
_bufferAssets[IERC20(address(wrappedToken))] = address(0);

state: 1e30 bufferLpShares, bufferBalances 1e30/0

2. Change asset() to a dummy token to be able to balance the buffer. You need a working ERC20 token to be able to unwrap.

```
Unset
call  vault.erc4626BufferWrapOrUnwrap(
           BufferWrapOrUnwrapParams({
               kind: SwapKind.EXACT_IN,
               direction: WrappingDirection.UNWRAP,
               wrappedToken: ERC4626 contract,
               amountGivenRaw: 1e6,
               limitRaw: 0,
               userData: new bytes(0)
           })
       );
```

This will unwrap the given wrapped token to our fake token. bufferWrappedSurplus will rebalance the pool.

_bufferAssets[IERC20(address(wrappedToken))] == address(0) so the check for bufferAsset is skipped.

state: 1e30 bufferLpShares, bufferBalances 5e29/5e29

3. Change asset() to return the token you want to take (WETH, DAI, USDC, etc)

```
Unset
call addLiquidityToBuffer(
wrappedToken, amountUnderlying. = 0, amountWrapped = 1e30, sharesOwner)
```

Since _bufferAssets[IERC20(address(wrappedToken))] == address(0) still holds, it will process this and sets _bufferAssets[IERC20(address(wrappedToken))] = targetUnderlyingToken;

4. Now we simply calculate how many shares we need to remove all of the target token and call removeLiquidityFromBuffer()

Customer's response: Fixed.

Fix Review: A required InitializeBuffer step has been added for ERC4626 buffers. This ensures the underlying asset cannot be changed. The code now also checks that the returned asset is not address(0).

High Severity Issues

H-01 Any user can steal the yield of the wrapped tokens that are kept in the buffer

Severity: **High**	Impact: **High**	Likelihood: **Likely**
Files: VaultAdmin.sol	Status: Fixed in 3db8bd7	Violated Property: removeLiquidityAfterAddLiquidity in P-04

Description: It is possible to steal the yield of the wrapped tokens that are kept in the buffer. When one adds liquidity to a buffer, the amount of shares he receives is equal to the amount of underlying tokens deposited plus the worth of the wrapped tokens in terms of the underlying token

```
Unset
issuedShares = wrappedToken.convertToAssets(amountWrappedRaw) + amountUnderlyingRaw;.
```

When one removes liquidity, he receives the proportional part of the buffer balances:

```
Unset
removedUnderlyingBalanceRaw = (bufferBalances.getBalanceRaw() * sharesToRemove) /
totalShares;

removedWrappedBalanceRaw = (bufferBalances.getBalanceDerived() * sharesToRemove) /
totalShares;
```

Exploit Scenario: In this scenario, the underlying token is USDC and the wrapped token is aUSDC. Let us assume that when the buffer is first initialized the index is 1, meaning 1 aUSDC is worth 1 USDC. The one who first deposits into the buffer deposits 1000 USDC and 1000 aUSDC. As a result, he receives 2000 shares (we ignore the small amount that goes to the 0 address). Now let's say that some time has passed and now 1 aUSDC is worth 2 USDC. An attacker can deposit 2000 USDC into the buffer, and receive 2000 shares. If he immediately calls removeLiquidityFromBuffer(), he'll receive half of the USDC in the buffer and half of the aUSDC in the buffer, meaning 1500 USDC and 500 aUSDC (which are worth 1000 USDC), netting 500 USDC, which is half of the yield of the wrapped tokens. By depositing an even larger amount of the underlying token, the attacker can steal an even larger portion of the yield (up to 100%).

Recommendations: Change the addLiquidityToBuffer() function such that the shares that are being issued are calculated based on the value of the deposit in proportion to the value of the buffer at the time of the deposit.

Customer's response: Fixed.

Fix Review: addLiquidityToBuffer() was changed to calculate the amount of issues shares to be proportional to the current value of the buffer.

```
Unset
        uint256 currentInvariant = bufferBalances.getBalanceRaw() +
            wrappedToken.convertToAssets(bufferBalances.getBalanceDerived());

        // The invariant delta is the amount we're adding (at the current rate) in terms of
underlying.
        uint256 bufferInvariantDelta = wrappedToken.convertToAssets(amountWrappedRaw) +
amountUnderlyingRaw;
        // The new share amount is the invariant ratio normalized by the total supply.
        // Rounds down, as the shares are "outgoing," in the sense that they can be redeemed
for tokens.
        issuedShares = (_bufferTotalShares[wrappedToken] * bufferInvariantDelta) /
currentInvariant;
```

Medium Severity Issues

M-01 A malicious user can cause a DoS in the erc4626BufferWrapOrUnwrap() external function by front-running the caller

Severity: **Medium**	Impact: **Medium**	Likelihood: **Unlikely**
Files: Vault.sol	Status: Fixed in c7b8bcf	Violated Property: None

Description: The _updateReservesAfterWrapping() internal function is being used both in the wrap and unwrap scenarios. This function returns two unsigned integers which represent the differences in vaultUnderlying and vaultWrapped before and after the transaction. In the wrap scenario, vaultUnderlyingBefore should be greater than vaultUnderlyingAfter while vaultWrappedAfter should be greater than vaultWrappedBefore. In the unwrap scenario, it's the opposite. The _updateReservesAfterWrapping() function "knows" which scenario it is by the sign of the difference between vaultUnderlyingAfter and vaultUnderlyingBefore, and assumes that the sign of the difference between vaultWrappedAfter and vaultWrappedBefore is the opposite. However, this is not necessarily the case as one can transfer tokens (either the underlying tokens or the wrapped tokens) to the vault before the transaction without updating the reserves in the storage (_reservesOf[]). If the transferred amount of tokens is larger than the amount that is being wrapped/unwrapped, a call to erc4626BufferWrapOrUnwrap() could revert as a result of an underflow in the _updateReservesAfterWrapping() function when calculating vaultWrappedDelta. The gifted tokens could later be reclaimed by the attacker by calling settle() and sendTo(), effectively executing the DoS attack with no significant loss of funds (outside of the gas cost)

Recommendations: Separate the _updateReservesAfterWrapping() into two functions, one for the wrap scenario and one for the unwrap scenario.

Customer's response: Fixed.

Fix Review: The _updateReservesAfterWrapping() function was separated into _settleWrap() and _settleUnwap().

M-02 Swap fees are different depending on if EXACT_IN or EXACT_OUT are being used		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Likely**
Files: Vault.sol	Status: Fixed in 814a9dd	Violated Property: None

Description: Fees are always paid in the token that its exact amount is not specified.
This causes the vaultSwapParams.kind == SwapKind.EXACT_IN and vaultSwapParams.kind == SwapKind.EXACT_OUT scenarios to be inherently different, as different amounts will be paid and received in each token. While it could have been fair if neither of the options would have been strictly better than the other in a non-negligible way, this is not the case.

Exploit Scenario: Let us consider a weighted pool with 3 tokens and weights [0.9, 0.09, 0.01]. The initial balance is [100000, 10000, 1000] and the swap fee is 2% of the swap. Assuming the swapper wants to swap about 50 tokens of token0 for about 43.1 tokens of token2, it would be better for him to call swap with vaultSwapParams.kind == SwapKind.EXACT_OUT:

Calling swap with vaultSwapParams.kind == SwapKind.EXACT_IN:

Pool balance after swap: [100050, 10000, 956.008]

Amount in: 50

Amount out: 43.1119

Calling swap with vaultSwapParams.kind == SwapKind.EXACT_OUT:

Pool balance after swap: [100048.97746, 10000, 956.888]

Amount in: 49.977

Amount out: 43.112

As calling swap with EXACT_OUT would cause the swapper to pay less to receive the same amount of tokens out.

Recommendations: Take fees in the same token for both scenarios (either always take fees in token_in or always take fees in token_out), so that the two options would be equivalent (with the possible exception of small rounding errors).

Customer's response: Fixed.

Fix Review: Fees are always taken in token_in.

M-03 Rounding errors in weighted pool invariant allows to take out more tokens than added

Severity: **Medium**	Impact: **Medium**	Likelihood: **Unlikely**
Files: WeightedMath.sol	Status: Fixed in 3cb2177	Violated Property: None

Description: When adding liquidity in an unbalanced way the pool calculates the invariant from the token balances. Depending on these balances the invariant can have different rounding errors from the pow function per invariant. Even though the invariantRatio which is used to calculate the BPTOut does a roundDown, the difference can be big enough to have an effect.
This allows a user to first add tokens in an unbalanced way and then remove liquidity in a balanced way and receive more tokens than he has added. The amount of tokens that an attacker can steal using this method appears to be insignificant.

Exploit Scenario: Example: a 20/80 pool

```
Unset
currentBalances 10000000000000000000000000 40000000000000000000000000
currentInvariant 30314331330207355316602532 9

Adding tokens via AddLiquidity unbalanced:
newBalances 20000000000000000000000000 80000000000000000000000000
newInvariant     60628662660414710671898907 0
invariantRatio   2000000000000000001
```

This means, the user adds 2.0x tokens and receives 2.000000000000000001x the BPT amount.
Doing an add liquidity followed by removeLiquidity BALANCED results in a net profit for the user.

in this example:

After settling all debts:

```
Unset
  RECEIVE:  255284147 token0. (LiveScaled18)
  RECEIVE:   63821036 token1. (LiveScaled18)
```

Customer's response: Fixed.

Fix Review: An extra parameter has been added to the function that calculates the invariant to specify the desired rounding direction. This makes sure all rounding is done in favor of the pool.

Low Severity Issues

L-01 initializeBuffer can only be called when it mints double the minimal amount		
Severity: **Low**	Impact: **Medium**	Likelihood: **Unlikely**
Files: VaultAdmin.sol	Status: Fixed in 10e7421	Violated Property: None

Description: In VaultAdmin.sol - initializeBuffer() the _MINIMUM_TOTAL_SUPPLY check is incorrect, making it only possible to initialize a buffer with 2 * _MINIMUM_TOTAL_SUPPLY totalSupply.

VaultAdmin.sol - initializeBuffer():

```
Unset
        issuedShares -= _MINIMUM_TOTAL_SUPPLY;
        _mintBufferShares(wrappedToken, sharesOwner, issuedShares);
        _mintMinimumBufferSupplyReserve(wrappedToken);
```

So the issuedShares parameter sent to _mintBufferShares() already has the _MINIMUM_TOTAL_SUPPLY subtracted.
In _mintBufferShares :

```
Unset
 function _mintBufferShares(IERC4626 wrappedToken, address to, uint256 amount) internal {
        ....
        uint256 newTotalSupply = _bufferTotalShares[wrappedToken] + amount;
        _ensureMinimumTotalSupply(newTotalSupply);
```

Here the _ensureMinimumTotalSupply check is actually checking the amount minus _MINIMUM_TOTAL_SUPPLY.

So in order to pass the test, the original IssuedShares needs to be 2*_MINIMUM_TOTAL_SUPPLY

Recommendations: Change the order of function calls.
_mintMinimumBufferSupplyReserve before _mintBufferShares

Customer's response: Fixed.

Fix Review: Order has changed and seems correct now.

L-02 Transposed digits in a constant value		
Severity: **Low**	Impact: **Medium**	Likelihood: **Unlikely**
Files: VaultStorage.sol	Status: Fixed in d5686f1	Violated Property: None

Description: The constant _MAX_PAUSE_WINDOW_DURATION was set to a value of 356 days * 4 while it was meant to be set to 365 days * 4 (4 years).

Recommendations: Set the value of _MAX_PAUSE_WINDOW_DURATION to 365 days * 4.

Customer's response: Fixed.

Fix Review: _MAX_PAUSE_WINDOW_DURATION was set to its intended value.

```
Unset
    uint256 internal constant _MAX_PAUSE_WINDOW_DURATION = 365 days * 4;
```

L-03 It is possible to pay less fees by splitting the deposited amounts		
Severity: **Low**	Impact: **Medium**	Likelihood: **Unlikely**
Files: BasePoolMath.sol	Status: Not fixed	Violated Property: None

Description: When adding liquidity, it might be beneficial for a user who wishes to deposit a certain amount of tokens into a pool to split that amount into several deposits. By splitting the deposited amount into two or more deposits, the user could pay less in fees and get more BPT out.

Exploit Scenario: We'll take a look at a scenario in which a user calls the computeAddLiquidityUnbalanced() function when depositing into a weighted pool. Let's say that the pool has 3 tokens with weights [0.6, 0.3, 0.1], an initial balance of [10000, 10000, 10000], and a 2% swap fee.

If a user will deposit 100 tokens of token0 in a single transaction, the pool balance would be:

[10099.197611133253, 10000, 10000]

And the tokens that were not deposited into the pool were taken as a fee.

However, if the same user would choose to split this deposit into two equal amounts (so 50 tokens each time), the pool balance would be:

[10099.19880573918, 10000, 10000]

Which is greater than it was in the normal scenario, meaning that by applying this method the user paid less fees and got more BPT out.

Recommendations: Rethink the math behind some of the pools and operations. Fixing this issue might require making some ground-breaking changes to the code and its design, so it might be best to keep the current behavior.

Customer's response: Known issue.

Fix Review: Current behavior was kept.

L-04 ReEntrancy Pool initialization possible		
Severity: **Low**	Impact: **Low**	Likelihood: **Unlikely**
Files:VaultExtension.sol	Status: Fixed in ecd42a5	Violated Property: None

Description: It is possible to re-enter pool initialization via the callBeforeInitializeHook.

Result: _initialize is called 2 times and PoolInitialized and PoolBalanceChanged event are fired 2 times for the same pool. The second time, _inititialize will overwrite the balances and in both calls, the _MINIMUM_TOTAL_SUPPLY is burned.

There is no profitable exploit.

Since balances cannot be negative, the only result that can be achieved is making own tokens disappear or being burned. At maximum impact, it can impact off-chain accounting given that in the reentered call a PoolBalanceChanged event is fired to inform that token balance has been added to the pool, while the added balances are later overwritten.

Recommendations: For safety/consistency I would recommend to move the isPoolInitialized check in finalize() to after the hook or add nonReentrant modifier to the initialize function.

Customer's response: Fixed.

Fix Review: A nonReentrant modifier was added to the function.

Informational Severity Issues

I-01. Fees not included in PoolBalanceChanged event for add and remove liquidity

Description: When adding or removing liquidity the vault emits a PoolBalanceChanged event with the pool token deltas.
For unbalanced operations, the add and remove liquidity functions can charge fees. These fees are not taken into account for the PoolBalanceChanged event, resulting in a difference between the actual pool token delta and the values in the event.

Example code snippet from _addLiquidity:

```
Unset
    ….
    for (uint256 i = 0; i < locals.numTokens; ++i) {
            …
            amountsInRaw[i] = amountInRaw;
            …
            poolData.updateRawAndLiveBalance(
                i,
                poolData.balancesRaw[i] + amountInRaw - locals.totalFeesRaw,
                Rounding.ROUND_UP
            );
    }


    emit PoolBalanceChanged(params.pool, params.to, amountsInRaw.unsafeCastToInt256(true));
```

The locals.totalFeesRaw is removed from the pool balance, but not taken into account in the event amounts.

Same goes for the removeLiquidity function.

Recommendation: Have the amounts in the event reflect the actual changed pool amounts.

I-02. RouterCommon unnecessarily approves WETH to the vault

Description: In the constructor of RouterCommon.sol the vault is given unlimited approval for WETH.

```
Unset
constructor(IVault vault, IWETH weth, IPermit2 permit2) VaultGuard(vault) {
   _weth = weth;
   _permit2 = permit2;
   weth.approve(address(vault), type(uint256).max);
}
```

The vault does not handle any token transfers in, so this approval is not needed.

Recommendation: Remove the unnecessary approval.

I-03. Unnecessary self approval in Router.queryRemoveLiquidityHook

Description: In queryRemoveLiquidityHook in Router.sol the router gives an unneeded self approval for the BPT token.

```
Unset
        // If router is the sender, it has to approve itself.
        IERC20(params.pool).approve(address(this), type(uint256).max);
```

In the removeLiquidity call the spender and owner of the BPT to burn are both set to the router. The ERC20MultiToken._allowance function returns unlimited allowance when owner==spender so it is not needed to self approve.

Recommendation: Remove the unneeded approval

I-04. In withdrawProtocolFees 1 bad token in pool can block fee collection of other tokens in the pool

Description: collectAggregateFees makes the vault call receiveAggregateFees in ProtocolFeeController.
This function loops over the tokens in the pool and transfers the fees from the vault to the ProtocolFeeController.

```
Unset
function _receiveAggregateFees(address pool, ProtocolFeeType feeType, uint256[] memory
feeAmounts) private {
        (IERC20[] memory poolTokens, uint256 numTokens) = _getPoolTokensAndCount(pool);
        for (uint256 i = 0; i < numTokens; ++i) {
            if (feeAmounts[i] > 0) {
                IERC20 token = poolTokens[i];

                token.safeTransferFrom(address(getVault()), address(this), feeAmounts[i]);
              ...
              }
       }
}
```

Then there is one bad token in the pool, this prevents the ProtocolFeeController from collecting fees from the other tokens in the pool.

I-05. No support for ERC4626 buffers with Fee or slippage

Description: ERC4626 vaults can apply slippage or fees for wrapping and unwrapping underlying tokens.
The previewMint/previewDeposit functions take these fees into account in the calculation, while the convertToAssets/convertToShares do not.
Balancer uses the convertTo functions. As a result wrapping tokens for an ERC4626 buffer with fees with SwapKind.EXACT_OUT will revert as the approved amount is based on the convertToShares results

Recommendations: Use previewMint/previewDeposit for all ERC4626 calls or clearly document that ERC4626 vault with fees/slippage are not supported.

I-06. HooksConfigLib - Amounts ignored when enableHookAdjustedAmounts is false

Description: When the enableHookAdjustedAmounts config is false, all call….Hook functions return the original amount without checking if the hook has actually done anything with the amounts

```
Unset
// If hook adjusted amounts is not enabled, ignore amounts returned by the hook
if (config.enableHookAdjustedAmounts() == false) {
     return amountsOutRaw;
}
```

Recommendation: As commented in the code, any received funds will be absorbed by the vault. When a Hook takes tokens while enableHookAdjustedAmounts==false this is ignored and it is assumed that the difference will cause an error when locking the vault.
It would be better to revert with a 'HookNotAllowedToAdjustBalance' if the tokendeltas have been changed after the hook call when enableHookAdjustedAmounts is false.

I-07. No custom error for tokens > 18 decimals

Description: The vault does not allow tokens with more than 18 decimals in any pool.
When registering a pool with a token with more than 18 decimals it will result in a revert caused by arithmetic underflow.

Recommendation: It would be preferable to check this and return an InvalidToken error.

I-08. Inconsistent function naming PoolConfigLib requireRemoveCustomLiquidityEnabled and requireAddCustomLiquidityEnabled

Description: In all locations the naming "LiquidityCustom" is used except for requireRemoveCustomLiquidityEnabled and requireAddCustomLiquidityEnabled where the order of words is "CustomLiquidity"

Recommendation: To be consistent with the rest of the code, the function should be renamed to requireRemoveLiquidityCustomEnabled and requireAddLiquidityCustomEnabled

I-09. Incorrect natSpec in VaultCommon.sol:_findTokenIndex

Description: For _findTokenIndex the natSpec reads

```
/// @dev Find the index of a token in a token array. Returns -1 if not found.
```

While code reverts when not found

Recommendation: Change comment to indicate the function reverts when not found

I-10. Incorrect comments in FixedPoint.sol mulDivUp + mulUp

Description: Some of the comments in FixedPoint.sol are incorrect

Unset

```
    function mulDivUp:
        // This check is required because Yul's `div` doesn't revert on b==0
        if (c == 0) {
            revert ZeroDivision();
        }
```

Unset

```
    function mulUp(uint256 a, uint256 b) internal pure returns (uint256 result) {
        // Multiplication overflow protection is provided by Solidity 0.8.x
```

```
        uint256 product = a * b;

        // The traditional divUp formula is:
        // divUp(x, y) := (x + y - 1) / y
        // To avoid intermediate overflow in the addition, we distribute the division andget:
        // divUp(x, y) := (x - 1) / y + 1
        // Note that this requires x != 0, if x == 0 then the result is zero

        // Equivalent to:
        // result = product == 0 ? 0 : ((product - 1) / FixedPoint.ONE) + 1;
        assembly {
            result := mul(iszero(iszero(product)), add(div(sub(product, 1), ONE), 1))
        }
    }
```

Recommendation:

`mulDivUp: b==0  should be c==0`
`mulUp: remove the divUp part in the comments`

I-11 Incorrect comment in VaultExtension getPoolConfig

Description: The comment about supportUnbalancedLiquidity in VaultExtension getPoolConfig seems incorrect or can be rephrased.

```
Unset
 // NOTE: supportUnbalancedLiquidity is inverted because false means it is supported.
disableUnbalancedLiquidity: !config.supportsUnbalancedLiquidity(),
```

The comment seems to suggest that supportUnbalancedLiquidity==false means Unbalanced Liquidity is supported.

Recommendation: Rephrase to “supportUnbalancedLiquidity false means it is disabled".

I-12 Lots of duplicate code between VaultAdmin initializeBuffer and addLiquidityToBuffer

Description: There is a huge overlap between the code in initializeBuffer and addLiquidityToBuffer.

Recommendation:
It seems a better option to have initializeBuffer only do the initialization (save _bufferAssets, _mintMinimumBufferSupplyReserve) and then call addLiquidityToBuffer

I-13 Suggestion for VaultExplorer

Description: The VaultExplorer contract seems to only exist to give users an easy way to query view functions of the vault from blockchain explorers. Users would still need to know the existence of the VaultExplorer contract and know the address where it is deployed.
An alternative approach could be to use the EIP1967 functionality of explorers to achieve the same.

Recommendation: Instead of a separate VaultExplorer contract that calls the vault, one could deploy a VaultInterfaces contract with only the IVaultAdmin and IVaultExtension interfaces without any implementation.
When the Vault constructor saves the address of this contract in the EIP1967 _IMPLEMENTATION_SLOT most common blockchain explorers can detect this via the “Is this a proxy” option.
After that, all functions of VaultAdmin and VaultExtension can be called via the Read/Write as proxy from the Vault contract page in the blockchain explorer.

Formal Verification

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

General Assumptions and Simplifications

1) **Hooks:** Hooks are a new feature by Balancer V3, however no concrete hook contracts have been provided so far. Due to the far reaching possibilities of changes on vault balances, we assume that hooks do not interfere with the vault's storage but only return nondeterministic values, that is we underapproximate the hooks' behaviors. Without concrete implementations, it is otherwise not possible to give formal guarantees as any sound overapproximation would result in random contract storage changes whenever a hook is called. We, thus, strongly recommend another audit that solely focuses on the correctness of hook contracts once they have been sufficiently implemented.
2) **Pools:** In order to simplify the verification of the Vault's functionality, we make an underapproximation using `PoolMock.sol` whenever a pool is used in the vault.
3) **Code Munges:** We simplified parts of the code, specifically we introduced loops that initialize newly generated complex data structures such as arrays of structs explicitly in the underlying code to circumvent static analysis issues. These changes affect `PoolDataLib.sol` as well as `BasePoolMath.sol`.
4) Further simplifications and underapproximations are listed below according to the rules adhering to them.

Formal Verification Properties

Vault Invariants

Module General Assumptions

The Vault is made up of three contracts: the main contract Vault.sol, as well as VaultExtension.sol and VaultAdmin.sol. To prove invariants on all external functions of these three contracts, we extended the main vault contract with both the VaultExtension.sol's and VaultAdmin.sol's external functions in order to formally prove the correctness of all adequate methods.
The following properties are thus proven correct on:

- Vault.sol
- VaultAdmin.sol
- VaultExtension.sol

Module Properties

P-01. Vault Buffer Invariants

Status: Verified

Rule Name	Status	Description	Link to rule report
underlyingAssetNotZeroAddress	Verified after fix	*This rule verifies that when bufferTotalShares[wrappedToken] > 0, the wrappedToken was initialized in the buffer, i.e. underlying asset address has been stored in bufferAssets[wrappedToken].*	*Report*
underlyingAssetNotInitialized	Verified	*This rule verifies that when bufferTotalShares[wrappedToken] < MINIMUM_TOTAL_SUPPLY, the wrappedToken was not initialized in the buffer, i.e. bufferAssets[wrappedToken] is address(0).*	*Report*

sumOfBufferLpSharesIsBufferTotalShares	Verified	*This rule verifies that the sum of bufferLPShares[wrappedToken][*] + MINIMUM_TOTAL_SUPPLY equals bufferTotalShares[wrappedToken].*	Report
bufferAssetsNotRegisteredWhenTotalSupplyIsZero	Verified	*This rule verifies that bufferAssets should not be initialized for a wrappedToken whose totalShares have not been set.*	Report
bufferTotalSharesHasMinSharesOrUnderlyingIsZeroAddress	Verified	*This rule verifies that an initialized wrappedToken has at least minimum total supply or has not been initialized (totalShares == 0).*	Report
noTwoBalancesExceedTotalBufferSupply	Verified	*This rule verifies that no sum of balances is bigger than the sum of lp shares.*	Report

P-02. Vault Accounting Invariants

Status: Verified	Assumptions: No donations to vault's balances.

Rule Name	Status	Description	Link to rule report
reservesOfStoresCorrectBalances	Verified	*The internal vault reserves per token cover the actual balances of the vault in the respective token. This rule assumes no external donations to the vault's balance in the token.*	Report
lockedVaultHasNoNonZeroDelt	Verified	*This rule verifies that when the vault is locked, there are no outstanding non-zero deltas.*	Report

as			

P-03. Pool Supply

Status: Verified

initializedPoolHasMinTotalSupply	*Verified*	*This rule verifies that when a pool has been initialized, its total supply is >= MINIMUM_TOTAL_SUPPLY.*	*Report*
uninitializedPoolHasNoTotalSupply	Verified	*This rule verifies that an uninitialized pool has no total supply.*	*Report*

Vault Rules

Module General Assumptions

The following rules are verified on an underapproximation of the Vault.sol contract. Due to the complexity of the underlying code, we verify the most important properties on a severe underapproximation to ensure that no funds can be lost under the most critical conditions.
We assume the following state space:

- We use a modified version of the PoolMock as `pool` containing two ERC20 tokens.
  - computeInvariant() returns the sum of poolToken balances.
  - onSwap() is based on a simple ratio between both tokens, e.g. `2*token0 == token1`
- The pool contains exactly two ERC20 tokens.
- We assume 0 swapFees.
- We assume a tokenRate of 1 in decimals.
- We assume a decimalScalingFactor of 1 for each token, i.e. no decimal difference between both toolTokens.

Module Properties

P-04. Vault Underapproximation	
Status:	

swappingBackAndForth	*Verified*	*This rule verifies that swapping some amount from one token to another and back again does not result in a gain of funds.*	*Verified on unreleased prover versions*
removeLiquidityAfterAddLiquidity	Verified	*This rule verifies that adding liquidity proportionally and removing the same amount with removeLiquidity proportionally cannot result in an increase in both tokens.*	

Vault Buffer Rules

Module General Assumptions

The following rules about buffer correctness are verified on the VaultAdmin.sol contract

Module Properties

P-05. Buffer Correctness	
Status: Verified	

Rule Name	Status	Description	Link to rule report
addLiquidityToBufferIntegrity	Verified	*This rule verifies that adding liquidity to the buffer correctly updates bufferBalances, bufferLPShares, bufferTotalShares, and tokenDeltas.*	*Summarized Report*
removeLiquidityFromBufferIntegrity	Verified	*This rule verifies removing liquidity from the buffer correctly updates bufferBalances, bufferLPShares, bufferTotalShares, and tokenDeltas.*	
removeLiquidityAfterAddLiquidity	Manually reviewed after fix	*This rule verifies that removing liquidity, that is the amount of shares that was just added, after adding liquidity does not result in more tokens than were originally supplied. This is violated due to shares/removed liquidity calculation only depending on the number of shares but not on the value of total assets as explained in Issue H-01. This rule does not apply anymore after the fix for this violation provided in H-01.*	
splitAddLiquidity	Verified	*This rule verifies that splitting amounts of wrapped and underlying tokens into two portions when supplying liquidity is not advantageous for the user.*	

initializeBuffer OnlyOnce	Verified	*This rule verifies that a buffer can only be initialized once, such that further calls to initialize always revert.*	

Vault Extension

Module General Assumptions

The following rules about the correctness of the Balancer Pool Token and the underlying ERC20 MultiToken implementation are verified on the BalancerPoolToken.sol contract.

Module Properties

P-06. Pool Initialization	
Status: Verified	Note: This rule is solely verified on VaultExtension.sol

Rule Name	Status	Description	Link to rule report
initializePoolOnlyOnce	*Verified*	*Repeated calls to initialize will always revert.*	*Report*

Balancer Pool Tokens and ERC20 MultiToken

Module General Assumptions

The following rules about the correctness of the Balancer Pool Token and the underlying ERC20 MultiToken implementation are verified on the BalancerPoolToken.sol contract.

Module Properties

P-07. Balancer Pool Token/ERC20 MultiToken Correctness

Status: Verified	Assumptions:

Rule Name	Status	Description	Link to rule report
approveIntegrity	*Verified*	*Calling approve changes the allowance correctly.*	*Summarized Report*
transferIntegrity	*Verified*	*Calling transfer increases and decreases the according balances correctly.*	
transferIsOneWayAdditive	*Verified*	*Splitting an amount into multiple transfers works the same as transferring one big amount.*	
transferDoesNotAffectThirdParty	*Verified*	*Calls to transfer do not change any third party balances.*	
transferFromIntegrity	*Verified*	*Calling transferFrom increases and decreases the according balances correctly.*	
transferIsOneWayAdditive	*Verified*	*Splitting an amount into multiple transfers works the same as transferring one big amount.*	
transferDoesNotAffectThirdParty	*Verified*	*Calls to transferFrom do not change any third party balances.*	

permitIntegrity	*Verified*	*Calls to permit correctly change allowances, increment the holder's nonce and respect the expiration deadline.*	
permitDoesNotAffectThirdParty	*Verified*	*Permit does not affect third party allowances.*	
permitRevertWhenDeadlineExpires	*Verified*	*Calls to permit reverts when the call lies outside the current deadline.*	
totalSupplyIsSumOfBalances	*Verified*	*totalSupply is the sum of all balances.*	
totalSupplyNeverOverflow	*Verified*	*totalSupply can never overflow.*	
noMethodChangesMoreThanTwoBalances	*Verified*	*No method can change more than two balances at once.*	
noSumOfBalancesLargerThanTotalSupply	*Verified*	*No sum of balances is larger than the totalSupply.*	
onlyAllowedMethodsMayChangeAllowance	*Verified*	*Only calls to approve, permit and transferFrom can decrease allowance, only approve and permit can increase allowance.*	
onlyAllowedMethodsMayChangeBalance	*Verified*	*Only calls to transfer and transferFrom can change user balance.*	

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.