Balancer Security Review

Auditors

Desmond Ho, Lead Security Researcher

Jonah Wu, Lead Security Researcher

Mario poneder, Security Researcher

Jeiwan, Security Researcher

Phaze, Security Researcher

Report prepared by: Lucas Goiriz

November 27, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Balancer is a decentralized automated market maker (AMM) protocol built on Ethereum that represents a flexible building block for programmable liquidity.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Balancer according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 28 days in total, Balancer engaged with Balancer engaged with Spearbit to review the balancer-monorepo protocol. In this period of time a total of **23** issues were found.

Summary

Project Name	Balancer
Repository	balancer-monorepo
Commit	ed437cf6
Type of Project	DeFi, Vaults
Audit Timeline	Sep 2 - Sep 30
Two week fix period	Oct 1 - Oct 4

The following PRs were also part of the scope:

1. PR 967: Remove `_CONVERT_FACTOR` from buffers
2. PR 975: Support negative imbalance in ERC4626 buffers
3. PR 982: Restrict `addLiquidityToBuffer` to proportional
4. PR 983: Swap fees equivalence (token in)
5. PR 1010: Roundtrip fee on proportional remove
6. PR 1020: Adjust stable math live balances rounding
7. PR 1032: Fix rounding scale down fees
8. PR 1033: Optimize scaling
9. PR 1035: Fix `computeBalance` in stable pool math
10. PR 1040: Optimization: Use `mcopy` to copy array in add liquidity unbalanced

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	0	0	0
Low Risk	9	8	1
Gas Optimizations	2	2	0
Informational	11	8	3
Total	**23**	**7**	**4**

5 Findings

5.1 High Risk

5.1.1 Rounding direction in token rates leads to inconsistent price between addLiquidity and removeLiquidity

Severity: High Risk

Context: Vault.sol#L784

Description: The Vault rounds balance up in addLiquidity rounds balance down in removeLiquidity. This in most cases favors the protocol. However, the minted amount depends not only on the price of the bpt but also on the price of each token. The token price in the pool would slightly change after rounding leading to inconsistent pricing between addLiquidity and removeLiquidity.

Assume an extremely unbalanced pool where dai:sDai = 1e18:1, with sDai rate is 1.1. sDai = 2 when rounding up and sDai = 1 when rounding down. In this extreme case, dai is more expensive when adding liquidity and cheaper when removing liquidity. This creates a potentially profitable path.

When dealing with tokens with rates, the token balance is calculated within the vault so that the pool. VaultCommon.sol#L316-L330. To make sure users could not extract value from round-trip interactions, the rounding directions are crafted carefully. However, the change of token amounts usually comes with second effects in AMM pools. In the addLiquidity and removeLiquidity cases, the minted amount depends not only on the price of the bpt but also on the price of each token. The token price in the pool would slightly change after rounding leading to inconsistent pricing between addLiquidity and removeLiquidity.

As mentioned in the comment, the vault Round balances up when adding liquidity, and Round down when removing liquidity, favors the protocol in most cases. The rounding directions change not only the price of the bpt, but the price of the token. For unbalanced liquidity, the price of the provided token has a larger impact on the minted amount.

- Vault.sol#L531:

```
function computeAddLiquidityUnbalanced(
    // ...
) {
    // ...

    // Loop through each token, updating the balance with the added amount.
    for (uint256 i = 0; i < numTokens; ++i) {
        newBalances[i] = currentBalances[i] + exactAmounts[i];
    }

    // Calculate the new invariant ratio by dividing the new invariant by the old invariant.
    // Rounding current invariant up reduces BPT amount out at the end (see comments below).
    uint256 currentInvariant = pool.computeInvariant(currentBalances, Rounding.ROUND_UP);
    // Round down to make `taxableAmount` larger below.
    uint256 invariantRatio = pool.computeInvariant(newBalances,
↪  Rounding.ROUND_DOWN).divDown(currentInvariant);

    // ...
}
```

In the above case, currentInvariant is larger when it rounds up, and lower when it rounds down. However, the rounding direction on the token balances contributes somehow equally to currentInvariant and the new invariant.

Impact: As mentioned above, assume an extreme case where sDai : dai = 1 : 1e18 and it rounds to sDai : dai = 2: 1e18 when adding liquidity and sDai : dai = 1: 1e18 when removing liquidity. In this extreme case, the user can get profit by imbalanced adding dai into the pool and imbalanced removingdai from the pool.

The impact depends on the implementation of the pool. For pool like stable poo when the price grows exponentially when pool's unbalanced, the pool can be drained in edge case.

```
function testMockPoolBalanceWithRate() public {
    uint tokenAmount = 1e6;
    uint dustAmount = 1;
    IERC20[] memory tokens = new IERC20[](2);
    tokens[0] = IERC20(usdc);
    tokens[1] = IERC20(wsteth);
    vault.manualSetPoolTokensAndBalances(
        pool,
        tokens,
        [tokenAmount, dustAmount].toMemoryArray(),
        [tokenAmount, dustAmount].toMemoryArray()
    );

    rateProviderWstEth.mockRate(1.001e18);
    vm.startPrank(lp);

        uint256[] memory exactAmountsIn = [tokenAmount, uint(0)].toMemoryArray();
        uint mintLp = router.addLiquidityUnbalanced(pool, exactAmountsIn, 0, false, "");
        uint lpBurned = router.removeLiquiditySingleTokenExactOut(pool, 1e50, usdc, tokenAmount, false,
↪   "");
        vm.assertGt(mintLp, lpBurned);

}
```

Likelihood: This depends on whether the pool implementation allows such an edge case where 1 wei of token amount has a huge impact on the token price. It depends on the future development of the protocol. At this point, we can evaluate the `weightedPool` and the `stablePool`.

For stable pools, the exploit would be difficult if not impossible. Invariant would be hard to converge when the token amount is low. The invariant would be hard to converge in edge cases. However, it's important to note that, there's no strong guarantee of this property.

```
StableMath.computeInvariant(
    2_000 * StableMath.AMP_PRECISION,
    [BigAmount, dust].toMemoryArray()
);

StableMath.computeInvariant(
    2_000 * StableMath.AMP_PRECISION,
    [BigAmount, dust + 1].toMemoryArray()
);
```

There's no specific range where the invariant would revert. For the above two computations, one may revert while the other one converges.

If we play around with the `stablePool` math, we can easily find a counter-example with ~100 runs.

```
function testFuzzEdgeTokenAmount(uint tokenAmount) public {
    tokenAmount = bound(tokenAmount, 1e6 ether, 1e12 ether);
    uint dustAmount = 2;
    uint[] memory currentBalances = [tokenAmount, dustAmount].toMemoryArray();
    bool first = false;
    bool second = false;
    try StablePool(pool).computeInvariant(currentBalances, Rounding.ROUND_UP) returns (uint256
↪   invariant) {
        first = true;
    } catch {
    }
    if(first) {
        currentBalances = [tokenAmount, dustAmount + 1].toMemoryArray();
        try StablePool(pool).computeInvariant(currentBalances, Rounding.ROUND_DOWN) returns (uint256
↪   invariant) {
            second = true;
        } catch {
        }
    }
    vm.assertTrue(!second);
    vm.assertTrue(!first);
}
```

Given the Balancer design encourages customized pool development and composability, this edge case is "unlikely" but not impossible.

Recommendation: *(optional)*

1. Recommend documenting this risk for future upgrades. Pool developers should fully understand this risk before enabling unbalanced liquidity.
2. Recommend imposing an optional minimum token amount besides the minimum trade amount. Token balances in a pool should not be lower than the minimum token amount.

Balancer: Addressed in PR 1020.

Cantina Managed: Fixed by adding and implicit tax to mitigate the edge case scenario. The implicit tax would be significant if the exploiter pushes the pool to edge cases.

5.2 Low Risk

5.2.1 ERC4626 buffer fails to rebalance for token deficits

Severity: Low Risk

Context: Vault.sol#L1185-L1206

Description: The current ERC4626 buffer logic is only able to rebalance itself when there is a surplus in the underlying tokens when a user wraps tokens using the buffer. It fails to rebalance itself when there is a deficit. In a similar manner, the buffer logic fails to rebalance itself when there is a deficit in wrapped tokens when a user requests to unwrap tokens using the buffer.

Using the liquidity present in the ERC4626 token buffer, wrapped and underlying tokens can be exchanged at the current rate without requiring external calls to the ERC4626 contract. Without knowing the next user's swap direction beforehand, the buffer is in an optimal state when the underlying and wrapped assets have a 1:1 ratio measured in underlying value. Therefore, for optimization purposes, any call to the ERC4626 token will include additional tokens from the buffer in order to rebalance it's asset rate after the call.

When the buffer is used to wrap tokens and not enough liquidity is present, the underlying token surplus is only measured if it is positive in `getBufferUnderlyingSurplus`.

```
uint256 surplus = 0;
if (underlyingBalance > wrappedBalanceAsUnderlying) {
    unchecked {
        surplus = (underlyingBalance - wrappedBalanceAsUnderlying) / 2;
    }
}
```

The buffer does not not rebalance itself in the case that there is a deficit leading to an unbalanced buffer after an external call.

Impact: Low: This issue does not lead to any loss of funds for the user or protocol. However, the buffer's main functionality of reducing external calls by using the buffer's balanced liquidity is impacted.

Likelihood: Medium: This issue occurs when the buffer does not have enough liquidity to perform the wrap/unwrap operation and when there is a deficit for underlying tokens when wrapping or a deficit for wrapped tokens when unwrapping.

Proof of Concept: The buffer remains unbalanced after the vault calls `deposit` on the wrapper contract.

```
function test_unbalancedBuffer() public {
    // Initializes the buffer with an amount that's enough to fulfill the deposit operation without
↪   interacting
    // with the ERC4626 protocol.
    _wrapAmount = 100e18;

    vm.prank(lp);
    router.initializeBuffer(IERC4626(address(waDAI)), _wrapAmount / 2, _wrapAmount / 4);

    BufferAndLPBalances memory beforeBalances = _measureBuffer();
    console.log("=== Buffer Balance ===");
    console.log("  DAI %18e", beforeBalances.buffer.dai);
    console.log("waDAI %18e", beforeBalances.buffer.waDai);
    console.log("");

    IBatchRouter.SwapPathExactAmountIn[] memory paths =
    // _exactInWrapUnwrapPath(_wrapAmount, 0, dai, IERC20(address(waDAI)), IERC20(address(waDAI)));
        _exactInWrapUnwrapPath(_wrapAmount, 0, IERC20(address(waDAI)), dai, IERC20(address(waDAI)));

    (,, IERC20[] memory tokens) = _getTokenArrayAndIndexesOfWaDaiBuffer();
    BaseVaultTest.Balances memory balancesBefore = getBalances(lp, tokens);

    vm.prank(lp);
    (uint256[] memory pathAmountsOut,,) = batchRouter.swapExactIn(paths, MAX_UINT256, false, bytes(""));

    BufferAndLPBalances memory afterBalances = _measureBuffer();
    console.log("=== Buffer Balance ===");
    console.log("  DAI %18e", afterBalances.buffer.dai);
    console.log("waDAI %18e", afterBalances.buffer.waDai);
}
```

```
[PASS] test_unbalancedBuffer() (gas: 504993)
Logs:
  === Buffer Balance ===
    DAI 50
  waDAI 25

  === Buffer Balance ===
    DAI 50
  waDAI 25
```

Recommendation: Refactor the buffer logic such that it is able to rebalance itself when there is a positive (surplus) and negative (deficit) delta in underlying/wrapped tokens when wrapping/unwrapping via the buffer.

Balancer: Fixed in PR 975.

Cantina Managed: Fixed.

5.2.2 Swap fees are not equivalent for `EXACT_IN` and `EXACT_OUT` cases

Severity: Low Risk

Context: Vault.sol#L382-L428

Description: The swap fees are applied in a way such that the effective fee is not equivalent for both swap kinds `EXACT_IN` and `EXACT_OUT`.

- Compute Out Given Exact In (`computeOutGivenExactIn`):

  Currently, when the exact token input amounts a_1 are specified (`SwapKind.EXACT_IN`), the aggregate swap fee percentage γ_s is applied to the token output amount a_0 in order to arrive at the net amount a_{0net}.

```
// amountOut. Round up to avoid losses during precision loss.
locals.swapFeeAmountScaled18 = amountCalculatedScaled18.mulUp(swapState.swapFeePercentage);

// Need to update `amountCalculatedScaled18` for the onAfterSwap hook.
amountCalculatedScaled18 -= locals.swapFeeAmountScaled18;
```

  This translates to the following formula.

$$\begin{aligned} a_{0net} &= \text{computeOutGivenExactIn}(a_1) \cdot (1 - \gamma_s) \\ &= a_0 \cdot \bar{\gamma}_s \ , \end{aligned}$$

where $\bar{\gamma}_s = 1 - \gamma_s$.

- Compute In Given Exact Out (`computeInGivenExactOut`):

  In case the exact token output amount a_0 is specified (`SwapKind.EXACT_OUT`) the fees are applied as follows.

```
locals.swapFeeAmountScaled18 = amountCalculatedScaled18.mulDivUp(
    swapState.swapFeePercentage,
    swapState.swapFeePercentage.complement()
);

amountCalculatedScaled18 += locals.swapFeeAmountScaled18;
```

$$a_{1net} = \text{computeInGivenExactOut}(a_0) \cdot (1 + \frac{\gamma_s}{1 - \gamma_s})$$

$$a_{1net} = \text{computeInGivenExactOut}(a_0) \cdot (\frac{1}{1 - \gamma_s})$$

$$a_{1net} = \text{computeInGivenExactOut}(a_0) \cdot (\frac{1}{\bar{\gamma}_s})$$

However, when starting out with the formula $a_{0net} = a_0 \cdot \bar{\gamma}_s$, it is not possible to derive a closed form solution which applies a constant factor to the output of the computations to both cases, e.g. $a_{1net} = a_1 \cdot \hat{\gamma}_s$, as $\hat{\gamma}_s$ will depend on the current balances, the output amount and the invariant formula.

If we start by assuming that we want to apply the swap fees to the output when given the exact input amount $a_{0net} = \text{computeOutGivenExactIn}(a_1) \cdot \bar{\gamma}_s$, we can derive the following formulas (exemplified using the input/output formulas given the weighted pool's swap invariant).

$$a_0 = b_0 \left[1 - \left(\frac{b_1}{b_1 + a_1} \right)^{\frac{w_1}{w_0}} \right]$$

$$a_1 = b_1 \left[\left(\frac{b_0}{b_0 - a_0} \right)^{\frac{w_0}{w_1}} - 1 \right]$$

In order to apply the fee to the output amount, we can make the following substitution for the formulas: $a_0 = \frac{a_0'}{\bar{\gamma}_s}$.

$$\begin{aligned} \implies \quad a_0' &= \bar{\gamma}_s b_0 \left[1 - \left(\frac{b_1}{b_1 + a_1} \right)^{\frac{w_1}{w_0}} \right] \\ &= \text{computeOutGivenExactIn}(a_1) \cdot \bar{\gamma}_s \\ \implies \quad a_1 &= b_1 \left[\left(\frac{b_0}{b_0 - \frac{a_0'}{\bar{\gamma}_s}} \right)^{\frac{w_0}{w_1}} - 1 \right] \\ &= \text{computeInGivenExactOut}(\frac{a_0'}{\bar{\gamma}_s}) \end{aligned}$$

In these new formulas, a_0' can be seen as the net output amount a_{0net}. In order to apply the fees equivalently, the fees need to scale the output amount first, before applying the formula which computes the output `computeInGivenExactOut`.

In general, it is not possible to derive a formula applying a constant factor to the result for both cases.

Impact: Low, the actual impact and difference in fees is quite low. This does not affect protocol solvency. In certain cases, the actual fees charged might differ when specifying `EXACT_IN` versus `EXACT_OUT`. One might lead to less or more fees applied (dependent on the pool's liquidity and input amount).

Likelihood: High, this case is reached whenever the exact token output amount is specified.

Recommendation: Adjust the formulas such that the fee is applied correctly by scaling the given output value by $\frac{1}{\bar{\gamma}_s}$ before computing the required token input amount.

$$a_{0net} = \text{computeInGivenExactOut}(\frac{a_0'}{\bar{\gamma}_s})$$

Balancer: Fixed in PR 983.

Cantina Managed: Fixed.

5.2.3 The `_CONVERT_FACTOR` is inconsistently applied to shares and assets

Severity: Low Risk

Context: Vault.sol#L1152, Vault.sol#L1164, Vault.sol#L1296, Vault.sol#L1310

Description: In the `Vault` contract, the constant `_CONVERT_FACTOR` is applied to share as well as asset amounts in 4 instances, and therefore has a different impact in terms of magnitude depending on the underlying token's decimals.

```
wrappedToken.convertToShares(amountGiven) - _CONVERT_FACTOR  // applied to shares
// vs.
wrappedToken.convertToAssets(amountGiven) + _CONVERT_FACTOR  // applied to assets
```

Recommendation: We suggest to consistently apply the `_CONVERT_FACTOR` to shares only.

Balancer: Fixed in PR 967.

Cantina Managed: Fixed.

5.2.4 `setGlobalProtocolSwap/YieldFeePercentage()` allow specifying higher precision than allowed

Severity: Low Risk

Context: ProtocolFeeController.sol#L398-L413

Description: The precision of the new values `newProtocolSwapFeePercentage` and `newProtocolYieldFeePercentage` aren't checked, it's possible to set higher precision percentages that will force creator fees to be set to only 0% or 100%.

Proof of Concept:

```
function testSetMaliciousGlobalFeePercentages(uint256 creatorFee) public {
   // same as _registerPoolWithMaxProtocolFees(); but higher precision bits for global percentages
   authorizer.grantRole(
      feeControllerAuth.getActionId(IProtocolFeeController.setGlobalProtocolSwapFeePercentage.selector),
      admin
   );
   authorizer.grantRole(

↪   feeControllerAuth.getActionId(IProtocolFeeController.setGlobalProtocolYieldFeePercentage.selector),
      admin
   );

   vm.startPrank(admin);
   feeController.setGlobalProtocolSwapFeePercentage(CUSTOM_PROTOCOL_SWAP_FEE_PCT + 1234567890123);
   feeController.setGlobalProtocolYieldFeePercentage(CUSTOM_PROTOCOL_SWAP_FEE_PCT + 123456890123);
   vm.stopPrank();

   pool = createPool();
   vm.startPrank(lp);
   // cannot set any creator fee that isn't 100% (perhaps there are some specific values that could be,
↪   but would be few)
   creatorFee = bound(creatorFee, 1, FixedPoint.ONE - 1);
   vm.expectRevert(IVaultErrors.FeePrecisionTooHigh.selector);
   feeController.setPoolCreatorSwapFeePercentage(pool, creatorFee);
}
```

Recommendation: Check the precision of the new fee values.

Balancer: Fixed in PR 951.

Cantina Managed: Fixed.

5.2.5 Inability to batch router operations involving native funds

Severity: Low Risk

Context: RouterCommon.sol#L157, Router.sol#L107-L108, Router.sol#L307-L308, Router.sol#L611-L614

Description: Multiple native operations cannot be batched because excess ETH is forcibly refunded at the end of each operation. In addition, a minor inconvenience exists where one has to use `permitBatchAndCall()` instead of `multicall` because the latter lacks the `payable` modifier.

One won't be able to batch multiple operations involving native funds. For instance, adding liquidity to 2 separate pools WETH-USDC & WETH-DAI using ETH will revert, because the 1st operation would refund the ETH meant for the 2nd.

Proof of Concept:

```
function testPermitBatchCallWithMultipleNativeOps() public {
    // empty permit signatures, do "normal" approvals
    IRouterCommon.PermitApproval[] memory permitBatch;
    bytes[] memory permitSignatures;
    IAllowanceTransfer.PermitBatch memory permit2Batch;
    bytes memory permit2Signature;

    // do a normal initialize + add liquidity with native funds
    bytes[] memory multicallData = new bytes[](2);
    // action 1: init pool
    multicallData[0] = abi.encodeWithSelector(
        router.initialize.selector,
        address(wethPoolNoInit),
        wethDaiTokens,
        wethDaiAmountsIn,
        initBpt,
        true,
        bytes("")
    );
    // action 2: add liquidity into existing WETH-DAI pool
    multicallData[1] = abi.encodeWithSelector(
        router.addLiquidityCustom.selector,
        address(wethPool),
        wethDaiAmountsIn,
        bptAmountOut,
        true,
        bytes("")
    );

    // do the call, expect revert because ETH for the 2nd call was refunded
    vm.startPrank(alice);
    vm.expectRevert(RouterCommon.InsufficientEth.selector);
    bytes[] memory results = router.permitBatchAndCall{value: 2 * ethAmountIn}(
        permitBatch,
        permitSignatures,
        permit2Batch,
        permit2Signature,
        multicallData
    );
}
```

Recommendation:

- Allow a flag to be passed so that `returnEth()` can be optionally called. The downside is that developers have to be trusted in ensuring that any excess ETH will be refunded and not left in the `Router`.
- Make `multicall` `payable`.

Balancer: Fixed in PR 988.

Cantina Managed: Fixed.

5.2.6 Protocol fees are not counted in `PoolBalanceChanged` events

Severity: Low Risk

Context: Vault.sol#L756

Description: When adding and removing liquidity from pools, the `PoolBalanceChanged` event is emitted:

1. _addLiquidity():

```
function _addLiquidity() {
    // ...
    // A pool's token balance increases by amounts in after adding liquidity, minus fees.
    poolData.updateRawAndLiveBalance(
        i,
        poolData.balancesRaw[i] + amountInRaw - locals.totalFeesRaw,
        Rounding.ROUND_UP
    );

    // ...

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

2. _removeLiquidity():

```
function _removeLiquidity() {
    // ...
    // A Pool's token balance always decreases after an exit
    // (potentially by 0). Also adjust by protocol and pool creator fees.
    poolData.updateRawAndLiveBalance(
        i,
        poolData.balancesRaw[i] - (amountOutRaw + locals.totalFeesRaw),
        Rounding.ROUND_DOWN
    );

    // ...

    // 8) Off-chain events
    emit PoolBalanceChanged(
        params.pool,
        params.from,
        // We can unsafely cast to int256 because balances are stored as uint128 (see
↪   PackedTokenBalance).
        amountsOutRaw.unsafeCastToInt256(false)
    );
}
```

The amounts emitted in the events don't account for the protocol fee (which is not counted in pool balances):

1. when adding liquidity, the pool's balance is updated by the input amount minus the protocol fee, but the event reports the full input amount;
2. when removing liquidity, the pool's balance is updated by the output amount plus the protocol fee, but the event reports the full output amount.

As a result, off-chain monitoring and analysis tools will be impacted, and the pool balances calculated by monitoring the events won't match the actual balances.

Recommendation: In the above two cases, consider counting the protocol fees in the balance change amounts reported in the `PoolBalanceChanged` event.

Balancer: This was addressed in PR 983. We now also emit the fees in this event (for swaps, adds and removes, vs. swaps only).

Cantina Managed: As far as we can see, this wasn't fixed:

1. In `_addLiquidity()`, `PoolBalanceChanged` reports `amountsInRaw.unsafeCastToInt256(true)`, which doesn't account for `locals.aggregateSwapFeeAmountRaw`: Vault.sol#L739-L753.
2. In `_removeLiquidity()`, it also reports `amountsOutRaw.unsafeCastToInt256(false)` and ignores `locals.aggregateSwapFeeAmountRaw`: Vault.sol#L983-L1009.

5.2.7 Global `userData` definition inconveniences parsing and decoding

Severity: Low Risk

Context: IBatchRouter.sol#L47

Description: `userData` is defined as a global parameter, but it could be necessary to specify different `userData` for each `SwapPathStep`, e.g: data that's parsed by pool hooks.

Recommendation: Consider shifting `userData` into `SwapPathStep`.

Balancer: Redundant `userData` for `bufferWrapOrUnwrap` struct has been removed in PR 1022.

Cantina Managed: Fixed.

5.2.8 Tokens reverting on zero value may block operation flows

Severity: Low Risk

Context: Vault.sol#L144

Description: Some obscure tokens may revert on zero value. There are cases where `sendTo()` is called with zero value, eg. in the `BatchRouter` when wrapping / unwrapping ERC4626 & `limits[i] == underlyingAmounts[i]`.

Recommendation: Consider wrapping this line with an `if (amount > 0)` conditional check.

Balancer: Fixed in PR 1014.

Cantina Managed: Fixed.

5.2.9 Missing `payable` modifier of router methods

Severity: Low Risk

Context: Router.sol#L411-L417, Router.sol#L439-L442

Description: The router contract's `permitBatchAndCall` method is `payable` to enable natively funded batch operations. However, the underlying multicall relies on `delegatecall` and therefore requires all methods, which can be part of a batch operation, to be `payable` as well.

The following methods of the `Router` contract are missing the `payable` modifier and are therefore incompatible with natively funded batch operations:

- `removeLiquidityCustom`.
- `removeLiquidityRecovery`.

Recommendation: It is suggested to add the `payable` modifier to the aforementioned methods.

Balancer: Fixed in PR 1012.

Cantina Managed: Fixed.

5.3 Gas Optimization

5.3.1 Buffer's underlying surplus is recomputed unnecessarily

Severity: Gas Optimization

Context: Vault.sol#L1234

Description: When wrapping underlying tokens while specifying the swap kind as `EXACT_IN`, in the buffer's underlying surplus value is recomputed inside the `_wrapWithBuffer` function.

```
vaultUnderlyingDeltaHint = amountInUnderlying + bufferUnderlyingSurplus;
// ...
bufferUnderlyingSurplus = vaultUnderlyingDeltaHint - amountInUnderlying;
```

The buffer's underlying surplus amount has not changed, however, and this line can be omitted.

The same applies to the `EXACT_IN` swap kind path for unwrapping tokens in the `_unwrapWithBuffer` function. Here, the buffer's wrapped surplus (`bufferWrappedSurplus`) is recomputed unnecessarily.

Recommendation: Remove the lines recomputing the buffer's surplus values for the swap kind path `EXACT_IN` in the `_wrapWithBuffer` and `_unwrapWithBuffer` functions, as these are effectively no-ops.

Balancer: Fixed in PR 967.

Cantina Managed: Fixed.

5.3.2 Redundancies

Severity: Gas Optimization

Context: RouterCommon.sol#L106, Router.sol#L1012-L1013, Router.sol#L850-L852, Router.sol#L1111-L1113

Description: Self approval of BPT in `Router` is redundant because `_spendAllowance()` skips the allowance check for self-spending.

```
function _spendAllowance(address pool, address owner, address spender, uint256 amount) internal {
    uint256 currentAllowance = _allowance(pool, owner, spender);
    if (currentAllowance != type(uint256).max) {/*...*/}
}

function _allowance(address pool, address owner, address spender) internal view returns (uint256) {
    // Owner can spend anything without approval
    if (owner == spender) {
        return type(uint256).max;
    }
}
```

- WETH approval to the `Vault` is redundant because of the settle mechanism; there doesn't seem to be a circumstance where the vault will pull funds in.
- Query hooks don't need to be `payable` because the vault always calls back with zero `msg.value`.

Recommendation: Remove the approvals and `payable` keyword.

Balancer: Addressed in PR 976; redundant approvals addressed in PR 1016.

Cantina Managed: Fixed.

5.4 Informational

5.4.1 Weighted pool math derivations

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Invariant Formula. The core of the weighted pool math is the invariant formula:

$$I = \prod_i b_i^{w_i}$$

Where:

- I is the invariant.
- b_i is the balance of token i.
- w_i is the normalized weight of token i.

The invariant after modifying balances is:

$$I = \prod_i (b_i \pm a_i)^{w_i}$$

Where:

- $a_1 > 0$ is the input and $a_0 > 0$ is the output.
- inputs are added and outputs are subtracted.

Givens and Assumptions

Givens:

- `uint256[] currentBalances` b_i
- `totalSupply` bpt_{total}
- `swapFee` γs

Assumptions:

- Pools use two tokens.
- Token 0 is the output.
- Token 1 is the input.

1. Add Liquidity Exact Tokens In: `computeAddLiquidityUnbalanced`:

- Input: `uint256[] exactAmountsIn` $a_1 > 0$
- Output: `bptAmountOut` bpt_{out}

Derivations:

1. New balance: $b_{1new} = b_1 + a_1$
2. Invariant ratio: $I_{ratio} = \frac{I_{new}}{I_{curr}}$
3. Proportional amount: $bp_1 = b_1 \cdot I_{ratio}$
4. Fees on taxable amount: $b_{1fee} = b_{1\ taxable} \cdot \gamma_s = (b_{1new} - bp_1)\gamma_s$

5. Net balances:

$$\begin{aligned} b_{1net} &= b_1 - (b_{1new} - bp_1)\gamma_s \\ &= b_1 - (b_1 + a_1 - bp_1)\gamma_s \\ &= b_1(1 - (1 + \frac{a_1}{b_1} - I_{ratio})\gamma_s \\ b_{0net} &= b_0 \end{aligned}$$

6. Invariant with taxes applied:

$$\begin{aligned} I_{net} &= b_{1net}^{w_1} \cdot b_{0net}^{w_0} \\ &= b_1^{w_1}(1 - (1 + \frac{a_1}{b_1} - I_{ratio})\gamma_s b_0^{w_0} \\ &= (1 - (1 + \frac{a_1}{b_1} - I_{ratio})\gamma_s I_{curr} \end{aligned}$$

7. Balancer pool tokens out:

$$\begin{aligned} bpt_{out} &= bpt_{total}\frac{I_{net} - I_{curr}}{I_{curr}} \\ &= bpt_{total}\left((1 + (I_{ratio} - 1)\gamma_s)^{w_1} - 1\right) \end{aligned}$$

The invariant ratio i_{ratio} represents the increase/decrease in the pools underlying value. The proportional amounts bp_i would be the new balances if the invariant increased and balances were added in proportional amounts. Adding and removing liquidity in disproportionate amounts can equal a swap operation. That's why the swap fee γ_s is levied on the taxable amount. The taxable amount is measured as the difference of the new balances to the proportional balances $b_{i\ taxable} = (b_{i\ new} - bp_i)$.

2. Add Liquidity Single Token In Exact BPT Out: `computeAddLiquiditySingleTokenExactOut`

- Input: `exactBptOut` bpt_{out}
- Output: `amountIn` a_1

Derivations:

1. Invariant ratio: $I_{ratio} = \frac{bpt_{total} + bpt_{out}}{bpt_{total}}$
2. New balances: $b_{1new} = b_1 \cdot I_{ratio}^{\frac{1}{w_1}}$
3. New invariant:

$$\begin{aligned} I_{new} &= b_{1new}^{w_1} b_{0new}^{w_0} \\ &= b_1^{w_1} \cdot I_{ratio} \cdot b_0^{w_0} \\ &= I_{curr} \cdot I_{ratio} \end{aligned}$$

4. Fees on taxable amount: $b_{1fee} = (b_{1new} - bp_1)\bar{\gamma}s$, where $\bar{\gamma}s = \frac{\gamma s}{1 - \gamma s}$
5. Amount in:

$$a_1 = b_{1new} - b_1 + b_{1fee}$$

Since the exact bpt token amounts out bpt_{out} are specified by the user, the new total bpt supply can be computed. The invariant ratio i_{ratio} represents the increase/decrease in underlying value. Since the bpt tokens represent the underlying value, the invariant ratio is given indirectly by the increase in the bpt total supply.

Given a new invariant ratio, the pool can compute what balance increase in a single token is required to increase its final invariant ratio computed with the new balances b_{1new}.

The fee is computed on the taxable amount, the difference of the new balance compared to its proportional balance.

3. Remove Liquidity Single Token Exact Out: `computeRemoveLiquiditySingleTokenExactOut`

- Input: `exactAmountOut` a_0
- Output: `bptAmountIn` bpt_{in}

Derivations:

1. New balance and invariant:

$$
\begin{aligned}
b_{0new} &= b_0 - a_0 \\
I_{new} &= b_1^{w_1} \cdot (b_0 - a_0)^{w_0} \\
I_{ratio} &= I_{new} / I_{curr}
\end{aligned}
$$

2. Fees on taxable amount:

$$
\begin{aligned}
b_{0fee} &= (bp_0 - b_{0new})\bar{\gamma}s \\
&= (b_0 \cdot I_{ratio} - b_0 + a_0)\bar{\gamma}s
\end{aligned}
$$

where $\bar{\gamma}s = \frac{\gamma s}{1-\gamma s}$

3. Net balance:

$$b_{0net} = b_{0new} - b_{0fee}$$

4. Net invariant:

$$I_{net} = b_1^{w_1} \cdot b_{0net}^{w_0}$$

5. BPT amount in:

$$bpt_{in} = bpt_{total} \cdot \frac{I_{curr} - I_{net}}{I_{curr}}$$

Since the exact token amounts out a_j are specified by the user, the new total pool balances b_{0new} and invariant i_{new} can be computed. As before, the swap fee is levied on the amount which was not provided in balanced/proportional amounts. The balancer pool tokens to be burned bpt_{in} equate to the decrease in the invariant.

4. Remove Liquidity Single Token Exact BPT In: `computeRemoveLiquiditySingleTokenExactIn`

- Input: `exactBptAmountIn` bpt_{in}
- Output: `amountOut` a_0

Derivations:

1. Invariant ratio: $I_{ratio} = \frac{bpt_{total} - bpt_{in}}{bpt_{total}}$
2. New balance: $b_{0new} = b_0 \cdot I_{ratio}^{\frac{1}{w_0}}$
3. Fees on taxable amount:

$$b_{0fee} = (bp_0 - b_{0new})\gamma s$$

4. Net amount out:

$$a_0 = b_0 - b_{0new} - b_{0fee}$$

Weighted Pools Swap Derivations

1. Compute Out Given Exact In (`computeOutGivenExactIn`)

- Input: a_1: Amount of tokens being swapped in
- Output: a_0: Amount of tokens to be swapped out

Derivation:

Starting with the invariant:

$$\begin{aligned}
& I_{\text{curr}} = I_{\text{new}} \\
\implies\quad & b_1^{w_1} b_0^{w_0} = (b_1 + a_1)^{w_1} (b_0 - a_0)^{w_0} \\
\implies\quad & \left(\frac{b_0 - a_0}{b_0}\right)^{w_0} = \left(\frac{b_1}{b_1 + a_1}\right)^{w_1} \\
\implies\quad & \frac{b_0 - a_0}{b_0} = \left(\frac{b_1}{b_1 + a_1}\right)^{\frac{w_1}{w_0}} \\
\implies\quad & 1 - \frac{a_0}{b_0} = \left(\frac{b_1}{b_1 + a_1}\right)^{\frac{w_1}{w_0}} \\
\implies\quad & \frac{a_0}{b_0} = 1 - \left(\frac{b_1}{b_1 + a_1}\right)^{\frac{w_1}{w_0}} \\
\implies\quad & a_0 = b_0 \left[1 - \left(\frac{b_1}{b_1 + a_1}\right)^{\frac{w_1}{w_0}}\right]
\end{aligned}$$

Derivation with Fees on Output:

The invariant uses the gross amount before fees $a_0 = \frac{a_{0net}}{\bar{\gamma}_s}$:

$$\begin{aligned}
& I_{\text{curr}} = I_{\text{net}} \\
\implies\quad & b_1^{w_1} b_0^{w_0} = (b_1 + a_1)^{w_1} (b_0 - \frac{a_0}{\bar{\gamma}_s})^{w_0} \\
\implies\quad & a_0 = \bar{\gamma}_s b_0 \left[1 - \left(\frac{b_1}{b_1 + a_1}\right)^{\frac{w_1}{w_0}}\right]
\end{aligned}$$

where:

- γ_s is the swap fee (e.g., 5%).
- $\bar{\gamma}_s = 1 - \gamma_s$ is the complement (e.g., 95%).

2. Compute In Given Exact Out (`computeInGivenExactOut`)

- Input: a_0: Amount of tokens being swapped out
- Output: a_1: Amount of tokens that must be swapped in

Derivation:

Starting with the invariant:

$$\begin{aligned}
& I_{\text{curr}} = I_{\text{new}} \\
\implies\quad & b_1^{w_1} b_0^{w_0} = (b_1 + a_1)^{w_1} (b_0 - a_0)^{w_0} \\
\implies\quad & \left(\frac{b_1 + a_1}{b_1}\right)^{w_1} = \left(\frac{b_0}{b_0 - a_0}\right)^{w_0} \\
\implies\quad & \frac{b_1 + a_1}{b_1} = \left(\frac{b_0}{b_0 - a_0}\right)^{\frac{w_0}{w_1}} \\
\implies\quad & a_1 = b_1 \left[\left(\frac{b_0}{b_0 - a_0}\right)^{\frac{w_0}{w_1}} - 1\right]
\end{aligned}$$

Derivation with Fees on Input:

Incorporate the net amount after fees:

$$
\begin{aligned}
& I_{\text{curr}} = I_{\text{net}} \\
\implies \quad & b_1^{w_1} b_0^{w_0} = (b_1 + a_1)^{w_1} (b_0 - \frac{a_0}{\bar{\gamma}_s})^{w_0} \\
\implies \quad & a_1 = b_1 \left[\left(\frac{b_0}{b_0 - \frac{a_0}{\bar{\gamma}_s}} \right)^{\frac{w_0}{w_1}} - 1 \right]
\end{aligned}
$$

Balancer: Fee equivalence was addressed in PR 972.

Cantina Managed: Fixed.

5.4.2 `unchecked` blocks do not extend to internal functions

Severity: Informational

Context: VaultCommon.sol#L107-L118

Description: When incrementing or decrementing the non-zero delta count variable `_nonZeroDeltaCount()`, the internal library functions `.tDecrement()` and `.tIncrement()` are being used inside an `unchecked` arithmetic block. The idea is to save gas by omitting safemath operations, as they should not be required in this context.

The `unchecked` block, however, does not extend to code contained in other function bodies. This means that the `unchecked` effectively does nothing.

Recommendation: Remove the outer `unchecked` block. Consider writing `tIncrementUnchecked` and `tDecrementUnchecked` functions that use unchecked arithmetic if needed.

Balancer: Fixed in PR 958.

Cantina Managed: Fixed.

5.4.3 Returned amounts from buffer wrap operations differ in query context

Severity: Informational

Context: Vault.sol#L1141-L1153

Description: The amounts returned by buffer wrap/unwrap operations subtract a `_CONVERT_FACTOR` from the amounts returned. This behavior differs compared to when in a query context.

In a query context the wrapped token output amount is determined by the `previewDeposit` function.

```
amountOutWrapped = wrappedToken.previewDeposit(amountGiven)
```

In a normal context, however, the token output amount is given by the `convertToShares` function and by subtracting a fixed constant from the result.

```
amountOutWrapped = wrappedToken.convertToShares(amountGiven) - _CONVERT_FACTOR
```

The actual difference should be negligible for most practical purposes. Yet there should not be any reason to have the calculations diverge in a query context. The same applies to the `SwapKind.EXACT_OUT` path.

Recommendation: Remove the alternative code path which only applies to the query context such that the calculated amounts are equal in both cases.

Balancer: Fixed in PR 967.

Cantina Managed: Fixed.

5.4.4 Pool creator fee is levied on the complement of protocol fee

Severity: Informational

Context: ProtocolFeeController.sol#L338-L340

Description: The pool creator's fee is charged on the complement of the protocol fee. This might lead to pool creators receiving less fees than anticipated. The natspec of the `setPoolCreatorSwapFeePercentage` does not mention this detail.

```
/**
 * @notice Assigns a new pool creator swap fee percentage to the specified pool.
 * @param pool The address of the pool for which the pool creator fee will be changed
 * @param poolCreatorSwapFeePercentage The new pool creator swap fee percentage to apply to the pool
 */
function setPoolCreatorSwapFeePercentage(address pool, uint256 poolCreatorSwapFeePercentage) external;
```

As an example, assume the current `protocolFeePercentage = 10%` and Alice wants sets her pool swap fee to 5%. She therefore inputs `0.05e18` into `setPoolCreatorSwapFeePercentage`. The effective pool swap fee for the creator is however, `5% * 90% = 4.5%`.

Recommendation: Document in the function's natspec that the protocol creator's fees are always levied on the complement of the protocol fee amount and therefore the actual values might be lower. Alternatively consider computing the `aggregateFeePercentage` as the sum of the `protocolFeePercentage` and the `poolCreatorFeePercentage`.

Balancer: We do intend for the pool creator fee to be "*net*" the protocol fee, as calculated. We can update the docs for the function (we are sure that's documented somewhere; will make sure it's everywhere it needs to be).

See PR 986.

Cantina Managed: Fixed.

5.4.5 Unused return value accumulation in `ethAmountIn` variable

Severity: Informational

Context: BatchRouter.sol#L1018-L1035

Description: The `_settlePaths` method of the `BatchRouter` contract iteratively accumulates the return values of the calls to `_takeTokenIn` in a local variable named `ethAmountIn`. However, this variable is not used anywhere in the subsequent context.

- `BatchRouter.sol#L1031`:

```
  ethAmountIn += _takeTokenIn(sender, IERC20(tokenIn), _currentSwapTokenInAmounts().tGet(tokenIn),
↪  wethIsEth);
```

Recommendation: We suggest to implement the intended use case of the variable or remove it from the method.

Balancer: Fixed in PR 950.

Cantina Managed: Fixed.

5.4.6 Silent truncation of `uint256` amounts to `uint160` and `uint64`

Severity: Informational

Context: BatchRouter.sol#L229, BatchRouter.sol#L487, ProtocolFeeController.sol#L388, ProtocolFeeController.sol#L392, ProtocolFeeController.sol#L515, ProtocolFeeController.sol#L533, RouterCommon.sol#L237, Router.sol#L102, Router.sol#L302

Description:

1. In a total of 5 instances in the `BatchRouter`, `Router` and `RouterCommon` contracts, amounts of type `uint256` are silently cast/truncated to `uint160`.
2. In a total of 4 instances in the `ProtocolFeeController` contract, amounts of type `uint256` are silently cast/truncated to `uint64`.

In case of amounts exceeding `type(uint160).max/type(uint64).max`, this can lead to unexpected behavior or subsequent unexpected/unrelated errors further down the execution path.

Recommendation: We suggest to rely on `SafeCast` instead of plain downcasting.

Balancer: Fixed in PR 955.

Cantina Managed: Fixed.

5.4.7 Superfluous `payable` modifier in router query hooks

Severity: Informational

Context: Router.sol#L850-L852, Router.sol#L1111-L1113

Description: Both, the `queryAddLiquidityHook` and `querySwapHook` methods of the `Router` contract have the `payable` modifier. However, these hooks are only invoked in a strictly non-payable context via the `Vault` contract's `query` method. Therefore, the `payable` modifier is unnecessary.

Recommendation: We suggest to remove the `payable` modifier form these hook methods.

Balancer: Fixed in PR 976.

Cantina Managed: Fixed.

5.4.8 Comment Improvements

Severity: Informational

Context: IRouterCommon.sol#L8, BasePoolFactory.sol#L20, Router.sol#L101, VaultExtension.sol#L860, Vault.sol#L289-L290, Vault.sol#L205, Router.sol#L1107

Description: The referenced lines have typos or incorrect / unclear comments that can be modified for better clarity.

Recommendation:

```
- funtions
+ functions

- easiliy
+ easily

- // Rransfer tokens from the user to the Vault.
+ // Transfer tokens from the user to the Vault.

- * This function actually returns whatever the VaultExtension does when handling the request.
+ * This function actually returns whatever the VaultAdmin does when handling the request.

// redundant as a result of refactoring, moved to `_computeAmountGivenScaled18`
- // If the amountGiven is entering the pool math (ExactIn), round down, since a lower apparent
↪   amountIn leads
- // to a lower calculated amountOut, favoring the pool.

// incorrect comment as some state params are modified subsequently
- // State is fully populated here, and shall not be modified at a lower level.
+ // State is fully populated here

- * @dev Can only be called by the Vault. Also handles native ETH.
+ * @dev Can only be called by the Vault.
```

Balancer: Fixed in PR 999.

Cantina Managed: Fixed.

5.4.9 Clearer revert reason for `EXACT_OUT` if swapFeePercentage is 100%

Severity: Informational

Context: Vault.sol#L410-L413

Description: The `swapFeePercentage` could be set to 100% for pools with dynamic fee hooks. In such a scenario, `EXACT_OUT` swaps will revert from `ZeroDivision()` as the `complement()` would be 0.

Recommendation: Consider providing a clearer revert reason for this case.

```
if (swapState.swapFeePercentage == FixedPoint.ONE) revert UnsupportedFeeForExactOut();
```

Balancer: See issue 887 we'd created for this (originally suggesting avoiding this by setting a lower maximum).

Cantina Managed: Acknowledged.

5.4.10 Different rounding directions could lead to an extra charge on protocol yield fees for tokens with a rate below 1

Severity: Informational

Context: PoolDataLib.sol#L134

Description: Different rounding directions could lead to an extra charge on protocol yield fees for tokens with a rate below 1.

PoolDataLib.sol#L185-L214

The protocol charges yield fees based on the difference between current live balances and the last recorded live balances. Since different rounding directions can lead to a 1 wei difference in live balances, pool tokens with a rate below 1 may be double charged.

A 1 wei difference in live balances would result in a raw balance greater than 1 after scaling for tokens with a rate below 1. Consequently, an extra protocol fee would be charged for every operation

```
aggregateYieldFeeAmountRaw = aggregateYieldFeeAmountScaled18.toRawUndoRateRoundDown(
    poolData.decimalScalingFactors[tokenIndex],
    poolData.tokenRates[tokenIndex]
);
```

Recommendation: The protocol does not support tokens with a rate below 1. This issue is only for documentation purposes.

Balancer: Acknowledged.

Cantina Managed: Acknowledged.

5.4.11 `StableMath.ComputeBalance` tends to return a higher value when pools are unbalanced

Severity: Informational

Context: StableMath.sol#L181-L231

Description: `StableMath.computeBalance` is intended to calculate the correct token amount for a given invariant. However, computeBalance may converge to a higher value when the pool is unbalanced due to precision issues. This results in an implicit tax charged to users. Since the protocol cannot charge this implicit tax from the pool's liquidity providers (LPs), it would lead to slightly lower protocol revenue.

Proof of Concept: The following script demonstrates that computeBalance converges to the wrong result.

```
function testComputeBalanceCase1() public {
    uint amp = 2000;
    uint256[] memory balances = new uint256[](2);
    balances[0] = 1e10 ether + 1586633207475;
    balances[1] = 1335700565212;
    uint invariant = StableMath.computeInvariant(amp, balances);
    uint balance = StableMath.computeBalance(amp, balances, invariant, 0);
    if(balance > balances[0]) {
        console2.log("diff:", balance - balances[0]);
    }
}
```

Recommendation: In edge cases, the implicit tax generally protects the protocol. I do not recommend changing the implementation. This issue is documented only for informational purposes, as computeBalance implies it would revert if an answer isn't found; however, it actually converges to a higher value.

Balancer: Acknowledged.

Cantina Managed: Acknowledged.