Pashov Audit Group

# Liminal Security Review

October 10th 2025 - October 19th 2025

# Contents

# 1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

# 2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

# 3. Risk Classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| Likelihood: High | Critical | High | Medium |
| Likelihood: Medium | High | Medium | Low |
| Likelihood: Low | Medium | Low | Low |

**Impact**

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

**Likelihood**

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

## 4. About Liminal

Liminal is a cross-chain vault system built on LayerZero that enables deposits and redemptions of assets across multiple blockchain networks through a hub-and-spoke architecture.

## 5. Executive Summary

A time-boxed security review of the **Lmnal/liminal-contracts** repository was done by Pashov Audit Group, during which **t.aksoy**, **Klaus**, **0xbepresent**, **pontifex**, **Rex** engaged to review **Liminal**. A total of **32** issues were uncovered.

| Protocol Summary | |
|---|---|
| **Project Name** | Liminal |
| **Protocol Type** | Cross-chain vaults |
| **Timeline** | October 10th 2025 - October 19th 2025 |

**Review commit hash:**

- 2627bc24f160c90db68debe9bf7a45fd5be4620a (Lmnal/liminal-contracts)

**Fixes review commit hash:**

- bfcc43eacf41ec766aa4a704e68018b5ab5a3873 (Lmnal/xtokens-pashov)

### Scope

`DepositForwarder.sol` `DepositPipe.sol` `FeeManager.sol` `NAVOracle.sol` `PythPriceOracle.sol` `RedemptionPipe.sol` `ShareManager.sol` `VaultComposerBase.sol` `VaultTimelockController.sol` `AssetOFT.sol` `OVaultComposerMulti.sol` `OVaultShareOFTAdapter.sol` `ShareOFT.sol` `interfaces/`

# 6. Findings

## Findings count

| Severity | Amount |
|---|---|
| High | 5 |
| Medium | 8 |
| Low | 19 |
| **Total findings** | **32** |

## Summary of findings

| ID | Title | Severity | Status |
|---|---|---|---|
| [H-01] | `_handleWithdrawAssets()` lacks dynamic receiver handling during `withdraw()` call | High | Resolved |
| [H-02] | Cannot bridge withdrawn assets in `_handleWithdrawAssets` | High | Resolved |
| [H-03] | Cross-chain withdraw parameter mismatch causes errors and stuck shares | High | Resolved |
| [H-04] | Users will lose some yield and protocol will mint too much performance fee | High | Resolved |
| [H-05] | Controller is set incorrectly in `_handleWithdrawAssets` | High | Resolved |
| [M-01] | Token approval missing before refund in `_refund()` | Medium | Resolved |
| [M-02] | Fee collection updates state even when no shares are minted | Medium | Resolved |
| [M-03] | `maxDeposit` and `maxMint` fail to return 0 when deposits are paused | Medium | Resolved |
| [M-04] | DOS due to token approvals in `OVaultComposerMulti` | Medium | Resolved |
| [M-05] | Pass the remaining fee recipient address as a parameter | Medium | Resolved |
| [M-06] | Price precision is lost in `_convertPythPriceTo18Decimals` | Medium | Resolved |

| ID | Title | Severity | Status |
|---|---|---|---|
| [M-07] | Retroactive fee updates leading to incorrect fee collection | Medium | Resolved |
| [M-08] | Performance fee wrongly calculated on NAV instead of PPS | Medium | Resolved |
| [L-01] | Bypass delay via unoverridden `TimelockController` | Low | Resolved |
| [L-02] | Improper removal of `OFT` asset in `removeDepositPipe` | Low | Resolved |
| [L-03] | Missing storage gap in `VaultComposerBase` | Low | Resolved |
| [L-04] | `convertAmount()` fails to revert if 0 and from and to are same | Low | Resolved |
| [L-05] | `VaultTimelockController` owner not initialized | Low | Resolved |
| [L-06] | Bypass `maxPercentageIncrease` in `setTotalAssets` | Low | Acknowledged |
| [L-07] | Set max age per price feed | Low | Acknowledged |
| [L-08] | `PYTH_PRICE_ORACLE_STORAGE_LOCATION` is set incorrectly | Low | Resolved |
| [L-09] | Unrevoked old OFT approvals after asset re-registration | Low | Resolved |
| [L-10] | Hardcoded `minAmountLD` may cause loss in cross-chain redemptions | Low | Resolved |
| [L-11] | Malicious fulfillment calls delaying asset recovery indefinitely | Low | Acknowledged |
| [L-12] | Asset config inconsistency between composer and redemption pipe | Low | Acknowledged |
| [L-13] | Lack of Pyth confidence interval usage | Low | Resolved |
| [L-14] | Missing validation of asset decimals in `setPriceId` and `setPriceIds` | Low | Resolved |
| [L-15] | `DepositPipe.convertToAssets()` understates deposit assets due to rounding | Low | Resolved |
| [L-16] | Rounding in `DepositPipe.mint()` favors depositors | Low | Resolved |

| ID | Title | Severity | Status |
|---|---|---|---|
| [L-17] | Missing liquidity and balance invariants in `withdraw()` | Low | **Resolved** |
| [L-18] | `previewDeposit` / `previewMint` improperly revert due to user or global limits | Low | Acknowledged |
| [L-19] | Processing later in `fulfillFastRedeems` provides an advantage | Low | Acknowledged |

# High findings

## [H-01] `_handleWithdrawAssets()` lacks dynamic receiver handling during `withdraw()` call

### Severity

**Impact:** Medium

**Likelihood:** High

### Description

```
function _handleWithdrawAssets(bytes32 _composeFrom, bytes memory _params, uint256 _assets)
        internal
    {
        (address receiver, SendParam memory sendParam, uint256 minMsgValue, uint256 maxShares) =
            abi.decode(_params, (address, SendParam, uint256, uint256));

        // Validate msg.value
        if (msg.value < minMsgValue) revert InsufficientMsgValue();

        OVaultComposerMultiStorage storage $ = _getOVaultComposerMultiStorage();

        // Calculate shares needed (this would need to be exposed by RedemptionPipe)
        // For now, we'll use a withdraw function that returns shares burned
        address withdrawer = _composeFrom.bytes32ToAddress();

        // @audit [M-02] Lacks dynamic receiver handling
        uint256 shares = IRedemptionPipe($.redemptionPipe).withdraw(_assets, receiver,
withdrawer);

        // Check slippage on shares
        _assertSlippage(maxShares, shares); // (100, 99)

        // If sendParam provided, send remaining assets cross-chain
        if (sendParam.dstEid != 0) {
            sendParam.amountLD = _assets;
            _send($.underlyingAssetOFT, sendParam, withdrawer);
        }
    }
```

When a user interacts with a shareOFT to withdraw assets by burning shares and then also specifying to bridge the received assets to another chain, the receiver in the `withdraw()` function call would need to be `address(this)` and not any other address since `address(this)` aka OVaultComposerMulti would need to receive the redeemed assets first and then bridge them to the intended recipient on the other chain the user specifies to withdraw to.

However, in the current implementation of `_handleWithdrawAssets`, the withdraw would fail as we do not change the receiver to be `address(this)` first before the call to `.withdraw()` since the withdraw function in RedemptionPipe expects the arguments as such below:

```
// @note seen
    function withdraw(
        uint256 assets,  // NET amount user wants to receive
        address receiver,
        address controller
    ) external whenNotPaused nonReentrant returns
```

For reference, this is the LayerZero OFT logic for send and who is debited the tokens being teleported:

```
function send(
        SendParam calldata _sendParam,
        MessagingFee calldata _fee,
        address _refundAddress
    ) external payable virtual returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
oftReceipt) {
        return _send(_sendParam, _fee, _refundAddress);
    }

    ...
    function _send(
        SendParam calldata _sendParam,
        MessagingFee calldata _fee,
        address _refundAddress
    ) internal virtual returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
oftReceipt) {
        // @dev Applies the token transfers regarding this send() operation.
        // - amountSentLD is the amount in local decimals that was ACTUALLY sent/debited from
the sender.
        // - amountReceivedLD is the amount in local decimals that will be received/credited to
the recipient on the remote OFT instance.
        (uint256 amountSentLD, uint256 amountReceivedLD) = _debit(
@>            msg.sender,
            _sendParam.amountLD,
            _sendParam.minAmountLD,
            _sendParam.dstEid
        );

        ...
    }
```

As can be seen, the `OVaultComposerMulti` contract would be debited the redeemed assets being bridged, and not the initial `receiver` from the message, which is why we need to dynamically change the `receiver` field first.

There is another issue with a wrong `controller` field passed into the `withdraw` function from the `_handleWithdrawAssets` function, but that issue is covered in a separate finding.

### Recommendations

```
+ address assetReceiver = sendParam.dstEid != 0 ? address(this) : receiver;

uint256 shares = IRedemptionPipe($.redemptionPipe).withdraw(_assets, assetReceiver, withdrawer);
```

This diff would be sufficient to fix this case since we later need to call `_send()`, which would teleport the assets to another chain, and the tokens need to be locked from `address(this)`

## [H-02] Cannot bridge withdrawn assets in `_handleWithdrawAssets`

### Severity

**Impact:** Medium

**Likelihood:** High

### Description

In `_handleWithdrawAssets`, when `dstEid` is not 0, the withdrawn asset tokens are bridged. To bridge, the asset tokens must be approved to underlyingAssetOFT, but this is not done. Therefore, `_send` always fails.

```
function _handleWithdrawAssets(bytes32 _composeFrom, bytes memory _params, uint256 _assets)
    internal
{
    ...

    // If sendParam provided, send remaining assets cross-chain
    if (sendParam.dstEid != 0) {
@>      sendParam.amountLD = _assets;
@>      _send($.underlyingAssetOFT, sendParam, withdrawer);
    }
}
```

### Recommendations

Call `approve`.

```
function _handleWithdrawAssets(bytes32 _composeFrom, bytes memory _params, uint256 _assets)
    internal
{
    ...

    // If sendParam provided, send remaining assets cross-chain
    if (sendParam.dstEid != 0) {
        sendParam.amountLD = _assets;
+       sendParam.minAmountLD = 0;
+       IERC20($.underlyingAsset).approve($.underlyingAssetOFT, _assets);
```

```
        _send($.underlyingAssetOFT, sendParam, withdrawer);
    }
}
```

## [H-03] Cross-chain withdraw parameter mismatch causes errors and stuck shares

### Severity

**Impact:** High

**Likelihood:** Medium

### Description

The `OVaultComposerMulti.handleCompose` function, when processing the `ACTION_WITHDRAW_ASSETS` action, incorrectly passes the `_amount` parameter (representing the number of shares transferred via `SHARE_OFT`) directly as the `_assets` parameter to `_handleWithdrawAssets`.

```
File: OVaultComposerMulti.sol
305:         } else if (action == ACTION_WITHDRAW_ASSETS) {
306:@>           _handleWithdrawAssets(_composeFrom, params, _amount);
```

The `_handleWithdrawAssets` function then calls `IRedemptionPipe.withdraw(_assets, receiver, withdrawer)`, which expects `_assets` to be the amount of underlying assets to withdraw, not shares.

```
File: OVaultComposerMulti.sol
420:@>   function _handleWithdrawAssets(bytes32 _composeFrom, bytes memory _params, uint256
_assets)
421:         internal
422:     {
...
434:@>       uint256 shares = IRedemptionPipe($.redemptionPipe).withdraw(_assets, receiver,
withdrawer);
```

This mismatch can lead to incorrect asset withdrawals, as the share amount is not converted to the equivalent asset amount based on the current NAV. Consider the following scenario:

1. Assume NAV = 2 (1 share = 2 assets). User wants to withdraw `150 assets` (expects to burn ~75 shares, allows max 80 shares for slippage).
2. User sends 80 shares cross-chain via `SHARE_OFT` for `ACTION_WITHDRAW_ASSETS`.
3. Vault receives 80 shares (minted to `OVaultComposerMulti`).
4. Code incorrectly calls `RedemptionPipe.withdraw(80, receiver, withdrawer)` (using shares as assets), burning ~40 shares (80 ÷ 2) and transferring 80 assets.

```
File: RedemptionPipe.sol
349:     function withdraw(
350:@>       uint256 assets,  // NET amount user wants to receive
351:         address receiver,
352:         address controller
353:     ) external whenNotPaused nonReentrant returns (uint256 shares) {
...
363:         // Calculate gross assets needed (same calculation as redeem() but in reverse)
364:@>       uint256 grossAssets = assets.mulDiv(
365:             BASIS_POINTS,
366:             BASIS_POINTS - $.fees.instantRedeemFeeBps,
367:             Math.Rounding.Floor
368:         );
369:
370:         // Calculate shares needed based on gross assets (before fee)
371:@>       shares = convertToShares(grossAssets);
```

1. Slippage check passes (40 shares ≤ 80 maxShares).

User gets 80 assets (not 150), 40 shares burned (40 remain unburned). This results in burning fewer shares than transferred, leaving excess shares stuck in the composer contract without automatic return mechanisms.

### Recommendations

Decode the desired asset amount from `_params` instead of using `_amount`. Calculate required shares via `convertToShares`, and verify the composer's share balance suffices. Handle excess shares explicitly to prevent being stuck.

## [H-04] Users will lose some yield and protocol will mint too much performance fee

### Severity

**Impact:** High

**Likelihood:** Medium

### Description

```
function collectPerformanceFee() external nonReentrant onlySafeManager returns (uint256
sharesMinted) {
        FeeManagerStorage storage $ = _getFeeManagerStorage();
        if ($.fees.performanceFeeBps == 0) return 0;

        uint256 currentNAV = $.navOracle.getNAV();
        uint256 currentSupply = $.shareManager.totalSupply();

        if (currentSupply == 0) return 0;

        // Always use PPS comparison to avoid charging fees on deposits
```

```
        // First collection: establish baseline with current PPS, no fees collected
        if ($.lastSupplyForPerformance == 0) {
            $.lastNAVForPerformance = currentNAV;
            $.lastSupplyForPerformance = currentSupply;
            return 0;
        }

        // Subsequent collections: compare PPS to detect real performance
        // Calculate Price Per Share (PPS) in 18 decimals for precision
        // lastPPS = lastNAV / lastSupply
        // currentPPS = currentNAV / currentSupply
        uint256 lastPPS = $.lastNAVForPerformance.mulDiv(1e18, $.lastSupplyForPerformance,
Math.Rounding.Floor);
        uint256 currentPPS = currentNAV.mulDiv(1e18, currentSupply, Math.Rounding.Floor);

        if (currentPPS <= lastPPS) return 0;

        // Real performance = (currentPPS - lastPPS) * currentSupply / 1e18
        // This represents the true value increase, excluding deposits/withdrawals
        uint256 realPerformance = (currentPPS - lastPPS).mulDiv(currentSupply, 1e18,
Math.Rounding.Floor);

        // Calculate performance fee on the real performance
        uint256 performanceFeeInAssets = realPerformance.mulDiv($.fees.performanceFeeBps,
10_000, Math.Rounding.Floor);

        if (performanceFeeInAssets > 0) {
            // Convert to shares: shares = fee * totalSupply / (NAV - fee)
            sharesMinted =
                performanceFeeInAssets.mulDiv(currentSupply, currentNAV -
performanceFeeInAssets, Math.Rounding.Floor);

            if (sharesMinted > 0) {
                $.shareManager.mintFeesShares($.feeReceiver, sharesMinted);
                emit PerformanceFeeTaken(sharesMinted, realPerformance, block.timestamp);
            }
        }

        $.lastNAVForPerformance = currentNAV;
        $.lastSupplyForPerformance = currentSupply;
        return sharesMinted;
    }
```

In the function above, we calculate the performance fees earned. Technically, it will be the yield gains from user deposited assets e.g USDC.

So, if a user deposits 100 USDC at timestamp 1,576,800,000 and 8 hours later 10 USDC funding fees have been earned and deposited, at timestamp 1,576,828,800, when we call the `collectPerformanceFee()` function, fees earned as stipulated by `realPerformance` will be 10 USDC.

Note: The protocol currently skips the very first periods yield, which is fine to set things up and running, but that doesn't have any impact or correlates to this bug, and I'm pointing this out so we don't get confused where that figure goes later on during our calculations.

Consider the sequence below to illustrate the bug:

1. At timestamp 1,576,800,000, user A deposits 100 USDC and gets minted 100e18 shares. The NAV at this point will be 100e18 (100 USDC scaled to 18 decimals). Total supply of shares will also be 100e18.
2. By timestamp 1,576,828,800 (8 hours later), funding fees of about 10 USDC have been earned. Thus, NAV will become 110e18 while total supply will stay 100e18.
3. When we call the `collectPerformanceFee()` function at this point, since `lastSupplyForPerformance` is 0, we set `lastNAVForPerformance` to 110e18 as well as `lastSupplyForPerformance` to 100e18. The function then stops executing and no fee is collected, which is fine (see my note point above since we collect no fee for first period yield)
4. Then at timestamp 1,576,857,600 (another 8 hours), an additional 10 USDC yield has been earned, thus NAV will grow to become 120e18 and total supply would still be 100e18
5. Then at timestamp 1,576,857,780 (3 minutes more), user B deposits 100e18 USDC and gets minted 83,333,333,333,333,333,333 (aka 83.33 shares by the logic of 100e18 * 100e18 / 120e18 - see `_convertToShares` in DepositPipe). Now total supply will become 100e18 + 83,333,333,333,333,333,333 aka 183,333,333,333,333,333,333 while NAV will become 220e18
6. Then, at timestamp 1,576,857,960 (3 minutes more), we call collectPerformanceFee. The sequence of fee will be calculated wrong as showcased below:

```
uint256 currentNAV = $.navOracle.getNAV(); // 220e18
uint256 currentSupply = $.shareManager.totalSupply(); // 183,333,333,333,333,333,333

uint256 lastPPS = $.lastNAVForPerformance.mulDiv(1e18, $.lastSupplyForPerformance,
Math.Rounding.Floor); // 110e18 * 1e18 / 100e18 = 1,100,000,000,000,000,000

uint256 currentPPS = currentNAV.mulDiv(1e18, currentSupply, Math.Rounding.Floor); // 220e18 *
1e18 / 183,333,333,333,333,333,333 = 1,200,000,000,000,000,000

uint256 realPerformance = (currentPPS - lastPPS).mulDiv(currentSupply, 1e18,
Math.Rounding.Floor); // (1,200,000,000,000,000,000 - 1,100,000,000,000,000,000) *
183,333,333,333,333,333,333 / 1e18 = 18,333,333,333,333,333,333

uint256 performanceFeeInAssets = realPerformance.mulDiv($.fees.performanceFeeBps, 10_000,
Math.Rounding.Floor); // 18,333,333,333,333,333,333 * 3000 / 10000 = 5,499,999,999,999,999,999
```

As we can see above, the yield calculated as earned yield is 18.3e18 (18.3 USDC) instead of 10 USDC or 10e18 in NAV (since we skipped period one' yield). This is also because during the calculation, we incorrectly assume part of the latest mint as yield when it isn't the case.

It is also important to note that this bug is different from the acknowledged frontrunning of performance bug since that bug allowed users to dilute the shares, ultimately reducing protocol earned, fees whereas this case outlined above with this report is the opposite and can happen naturally without any frontrunning of fees/yield.

## Recommendations

I believe the better calculation for this is:

```
- uint256 realPerformance = (currentPPS - lastPPS).mulDiv(currentSupply, 1e18,
Math.Rounding.Floor);
+ uint256 realPerformance = (currentPPS - lastPPS).mulDiv(lastSupplyForPerformance, 1e18,
Math.Rounding.Floor);
```

The diff above would yield the amount of 10e18 not 18.3e18 tokens.

I haven't yet tested this fix and simulated other newer shares diluting transactions with it, but will do so during fix review, since you might decide to fix it in another way.

Or we could add a new function that accrues pending performance fee accrual before every mint/deposit requests.

## [H-05] Controller is set incorrectly in `_handleWithdrawAssets`

### Severity

**Impact:** Medium

**Likelihood:** High

### Description

In `_handleWithdrawAssets`, when calling `RedemptionPipe.withdraw`, the controller is set to `withdrawer`. `withdrawer` is the user's address on the source chain.

In a normal situation, the bridged share tokens deposited in the Composer should be burned and withdrawn. In the current implementation, the share tokens bridged by the withdrawal requester are locked in the Composer, and `RedemptionPipe.withdraw` burns the withdrawal requester's share tokens.

```
function _handleWithdrawAssets(bytes32 _composeFrom, bytes memory _params, uint256 _assets)
    internal
{
    ...

    // Calculate shares needed (this would need to be exposed by RedemptionPipe)
    // For now, we'll use a withdraw function that returns shares burned
@>  address withdrawer = _composeFrom.bytes32ToAddress();
@>  uint256 shares = IRedemptionPipe($.redemptionPipe).withdraw(_assets, receiver, withdrawer);
    ...
}
```

### Recommendations

Set the controller to `address(this)`.

```
function _handleWithdrawAssets(bytes32 _composeFrom, bytes memory _params, uint256 _assets)
    internal
{
    ...
```

```
+   IERC20(SHARE_ERC20).approve($.redemptionPipe, maxShares);

    // Calculate shares needed (this would need to be exposed by RedemptionPipe)
    // For now, we'll use a withdraw function that returns shares burned
    address withdrawer = _composeFrom.bytes32ToAddress();
-   uint256 shares = IRedemptionPipe($.redemptionPipe).withdraw(_assets, receiver, withdrawer);
+   uint256 shares = IRedemptionPipe($.redemptionPipe).withdraw(_assets, receiver,
address(this));

+   IERC20(SHARE_ERC20).approve($.redemptionPipe, 0);
    ...
}
```

# Medium findings

## [M-01] Token approval missing before refund in `_refund()`

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

The _refund() function attempts to send tokens back via the OFT interface after a compose failure. However, when the _oft address represents an OFT Adapter, the send() function internally calls innerToken.safeTransferFrom(), requiring prior token approval from the caller (the composer contract).

Currently, no approve() is performed before calling IOFT(_oft).send(). As a result, the refund transaction will revert.

```
function _refund(address _oft, bytes calldata _message, uint256 _amount, address _refundTo)
internal virtual {
        SendParam memory refundParam = SendParam({
            dstEid: _message.srcEid(),
            to: bytes32(uint256(uint160(_refundTo))),
            amountLD: _amount,
            minAmountLD: 0,
            extraOptions: "",
            composeMsg: "",
            oftCmd: ""
        });

        // Pass entire msg.value to send() which will handle excess and refund automatically
        IOFT(_oft).send{value: msg.value}(refundParam, MessagingFee(msg.value, 0), _refundTo);
    }
```

### Recommendations

Before calling IOFT(_oft).send(), ensure that the underlying token is approved for the refund amount.

## [M-02] Fee collection updates state even when no shares are minted

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

In both `collectPerformanceFee()` and `collectManagementFee()` functions, the contract **updates tracking variables** (like `$.lastNAVForPerformance` and `$.lastManagementFeeTimestamp`) even when **no shares are actually minted** (i.e., when `sharesMinted == 0`).

```
    function collectPerformanceFee() external nonReentrant onlySafeManager returns (uint256
sharesMinted) {
        ...
        if (performanceFeeInAssets > 0) {
            // Convert to shares: shares = fee * totalSupply / (NAV - fee)
            sharesMinted =
                performanceFeeInAssets.mulDiv(currentSupply, currentNAV -
performanceFeeInAssets, Math.Rounding.Floor);

            if (sharesMinted > 0) {
                $.shareManager.mintFeesShares($.feeReceiver, sharesMinted);
                emit PerformanceFeeTaken(sharesMinted, navIncrease, block.timestamp);
            }
        }
        $.lastNAVForPerformance = currentNAV;
        return sharesMinted;
```

This means that even when no actual fee is distributed, it prevents future valid fees from being collected if a small NAV or time increase later occurs. As a result, legitimate performance or management fees can be skipped.

### Recommendations

Only update tracking state variables **after** successful fee minting.

## [M-03] `maxDeposit` and `maxMint` fail to return 0 when deposits are paused

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

Per ERC-4626 specification, `maxDeposit(address)` and `maxMint(address)`:

> MUST factor in both global and user-specific limits, and if deposits are disabled (even temporarily), MUST return 0.

The current implementation calculates limits based only on user and supply caps. However there are **no checks** for:

- Global contract pause ( `whenNotPaused` ).
- ShareManager pause state.

As a result, `maxDeposit()` and `maxMint()` **can return non-zero values even when deposits are currently paused**, violating ERC-4626 compliance. A similar issue exist with redeem function maxWithdraw and maxRedeem. It is stated that these functions are ERC-4626 compliant.

```
    * @dev ERC4626 compliant - converts maxRedeem to assets after accounting for instant redemption fees
    */
   function maxWithdraw(address owner) external view returns (uint256) {
```

### Recommendations

Add explicit checks at the start of `maxDeposit` , `maxMint` , `maxWithdraw` , and `maxRedeem` :

```
if (paused() || $.shareManager.paused()) {
return 0;
}
```

## [M-04] DOS due to token approvals in `OVaultComposerMulti`

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

**USDT** implements a non-standard security feature that causes the `approve` call to **revert** if the existing allowance is non-zero, and the new amount is also non-zero.

In OVaultComposerMulti contract, the `redeemAndSend` and `_handleRedeemShares` functions approve the `underlyingAssetOFT` for the `assets` amount, and `depositAssetAndSend` approves the `SHARE_OFT` for the `shares` amount.

If the composer contract attempts to call `approve` for the USDT token with a non-zero allowance, the **transaction will revert**, leading to a **Denial of Service (DoS)** for the user attempting the cross-chain operation.

Also, USDT has a different function signature for `approve()` . This will also cause a revert because of a different function signature for the approve() function.

`IERC20(targetAsset).approve(pipe, _amount);`

### Recommendations

Implement safe approve function from OpenZeppelin instead of the normal approve.

## [M-05] Pass the remaining fee recipient address as a parameter

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

When requesting deposits and withdrawals with LayerZero, if there are remaining fees, they are returned to the user who requested cross-chain message. The account for receiving the refund is the address on the current chain.

The `_send` function of VaultComposerBase returns the remaining fee to `_refundAddress` after calling `OFT.send`. `_refundAddress` is the address to receive native tokens on the same chain where VaultComposerBase is deployed.

```
function _send(address _oft, SendParam memory _sendParam, address _refundAddress) internal
virtual {
    ...

    IOFT(_oft).send{value: fee.nativeFee}(_sendParam, fee, _refundAddress);

    if (msg.value > fee.nativeFee) {
@>      payable(_refundAddress).transfer(msg.value - fee.nativeFee);
    }
}
```

The `_handleDepositAsset`, `_handleRedeemShares`, and `_handleWithdrawAssets` functions bridge share or asset tokens using the `_send` function when `sendParam.dstEid` is not 0. `_message.composeFrom()` is passed as the `_refundAddress` parameter of the `_send` function, which means the requester's address on the source chain. If a contract wallet was used on the source chain, the same address on the destination chain may not be their account. Therefore, tokens should not be returned to `composeFrom`.

```
function lzCompose(address _composeSender, bytes32 _guid, bytes calldata _message, address,
bytes calldata)
    public
    payable
    virtual
    override
{
    ...
```

```
@>  bytes32 composeFrom = _message.composeFrom();
    uint256 amount = _message.amountLD();
    bytes memory composeMsg = _message.composeMsg();

@>  try this.handleCompose{value: msg.value}(_composeSender, composeFrom, _guid, composeMsg,
amount) {
        emit Sent(_guid);
    } catch (bytes memory _err) {
        ...
    }
}

function handleCompose(
    address _oftIn,
@>  bytes32 _composeFrom,
    bytes32 _guid,
    bytes memory _composeMsg,
    uint256 _amount
) public payable override nonReentrant {
    ...

    if (action == ACTION_DEPOSIT_ASSET) {
@>      _handleDepositAsset(_oftIn, _composeFrom, params, _amount);
    } else if (action == ACTION_REDEEM_SHARES) {
@>    _handleRedeemShares(_composeFrom, params, _amount);
    } else if (action == ACTION_WITHDRAW_ASSETS) {
@>      _handleWithdrawAssets(_composeFrom, params, _amount);
    } else {
        revert InvalidAction(action);
    }
}

function _handleDepositAsset(
    address _oftIn,
    bytes32 _composeFrom,
    bytes memory _params,
    uint256 _amount
) internal {
    ...

    // Deposit through pipe
@>  address depositor = _composeFrom.bytes32ToAddress();

    ...

    // If sendParam is provided, send shares cross-chain
    if (sendParam.dstEid != 0) {
        sendParam.amountLD = shares;

        // Approve exact amount for cross-chain send
        IERC20(SHARE_ERC20).approve(SHARE_OFT, shares);

@>       _send(SHARE_OFT, sendParam, depositor);
    }
    // If no cross-chain send, shares are already at receiver address
}
```

The same applies to `_refund` function. `OFT.send` returns the remaining fees to `_refundTo` received as a parameter. For cases other than ACTION_DEPOSIT_ASSET, it is returned to `composeFrom`. The same problem as above occurs. In the case of ACTION_DEPOSIT_ASSET, the deposited asset tokens are bridged back to `extractedRecipient` at source chain. However, the remaining fees are also sent to `extractedRecipient` on the current chain. The address to receive returned assets on the source chain and the address to receive refunded fees on this chain should be distinguished.

```
function lzCompose(address _composeSender, bytes32 _guid, bytes calldata _message, address,
bytes calldata)
    public
    payable
    virtual
    override
{
    ...

@>  bytes32 composeFrom = _message.composeFrom();
    uint256 amount = _message.amountLD();
    bytes memory composeMsg = _message.composeMsg();

    try this.handleCompose{value: msg.value}(_composeSender, composeFrom, _guid, composeMsg,
amount) {
        emit Sent(_guid);
    } catch (bytes memory _err) {
        if (bytes4(_err) == InsufficientMsgValue.selector) {
            assembly {
                revert(add(32, _err), mload(_err))
            }
        }
@>      address refundRecipient = composeFrom.bytes32ToAddress();
        (uint8 action, bytes memory params) = abi.decode(composeMsg, (uint8, bytes));
        // For ACTION_DEPOSIT_ASSET (1), extract refund recipient
        if (action == 1) {
@>        bytes32 extractedRecipient = this._decodeRefundRecipientExternal(params);
          // Use extracted recipient if not zero, otherwise fallback to composeFrom
          if (extractedRecipient != bytes32(0)) {
@>          refundRecipient = extractedRecipient.bytes32ToAddress();
          }
        }
@>      _refund(_composeSender, _message, amount, refundRecipient);
        emit Refunded(_guid);
    }
}

function _refund(address _oft, bytes calldata _message, uint256 _amount, address _refundTo)
internal virtual {
    SendParam memory refundParam = SendParam({
        dstEid: _message.srcEid(),
@>      to: bytes32(uint256(uint160(_refundTo))),
        amountLD: _amount,
        minAmountLD: 0,
        extraOptions: "",
        composeMsg: "",
        oftCmd: ""
```

```
    });

    // Pass entire msg.value to send() which will handle excess and refund automatically
@>  IOFT(_oft).send{value: msg.value}(refundParam, MessagingFee(msg.value, 0), _refundTo);
}
```

### Recommendations

Add a refund address to receive remaining fees in the `composeMsg` of all actions. Use this address when refunding fees.

## [M-06] Price precision is lost in `_convertPythPriceTo18Decimals`

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

In `_convertPythPriceTo18Decimals`, the price decimals are adjusted to 8, and then multiplied by 1e10 to adjust decimals to 18. Therefore, if the price uses decimals greater than 8, precision is lost.

```
function _convertPythPriceTo18Decimals(PythStructs.Price memory price) internal pure returns
(uint256) {
    require(price.price > 0, "PythOracle: invalid price");

    // First convert to 8 decimals using existing logic
@>  uint256 price8Decimals = _convertPythPrice(price);

    // Then convert from 8 decimals to 18 decimals
@>  return price8Decimals * 1e10; // 18 - 8 = 10
}

function _convertPythPrice(PythStructs.Price memory price) internal pure returns (uint256) {
    ...
    int32 targetExpo = -8;

    if (price.expo == targetExpo) {
        ...
    } else if (price.expo > targetExpo) {
        ...
    } else {
        // expo is more negative (more decimals), need to remove decimals
        // expo=-10, target=-8: divide by 10^2
@>      uint256 scaleFactor = 10 ** uint256(int256(targetExpo - price.expo));
@>      normalizedPrice = price.price / int256(scaleFactor);
    }
```

```
    require(normalizedPrice > 0, "PythOracle: price conversion failed");
    return uint256(normalizedPrice);
}
```

For example, suppose `price.expo` is -18. In `_convertPythPrice`, to adjust the price to 8 decimals, `price.price` is divided by 1e10. Then multiplying by 1e10 again fills the lower 10 digits with zeros. This results in using a lower price than the actual price.

### Recommendations

In `_convertPythPriceTo18Decimals`, adjust directly to 18 decimals without going through the step of adjusting to 8 decimals.

## [M-07] Retroactive fee updates leading to incorrect fee collection

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

The `setFees` function in the `FeeManager` contract allows updating the `managementFeeBps` and `performanceFeeBps` without first collecting accrued fees based on the previous rates.

```
File: FeeManager.sol
216:     function setFees(uint256 _managementFeeBps, uint256 _performanceFeeBps) external
onlyTimelock {
217:         require(_managementFeeBps <= 500, "FeeManager: management fee too high");
218:         require(_performanceFeeBps <= 3000, "FeeManager: performance fee too high");
219:
220:         FeeManagerStorage storage $ = _getFeeManagerStorage();
221:         $.fees.managementFeeBps = _managementFeeBps;
222:         $.fees.performanceFeeBps = _performanceFeeBps;
223:
224:         emit FeeConfigUpdated(_managementFeeBps, _performanceFeeBps);
225:     }
```

This results in retroactive application of new fees to past periods:

- For management fees, calculated in `collectManagementFee` using `timeElapsed` since `lastManagementFeeTimestamp`, a fee change without prior collection applies the new rate to the entire elapsed period, including time under the old rate.
- For performance fees, calculated in `collectPerformanceFee` based on NAV increase since `lastNAVForPerformance`, a similar change applies the new rate to past gains.

This breaks the intended time-based or performance-based accrual, leading to under/over-collection and value leakage.

### Recommendations

Modify `setFees` to automatically call `collectManagementFee` and `collectPerformanceFee` before updating fees, ensuring accrued fees are collected at the old rate.

## [M-08] Performance fee wrongly calculated on NAV instead of PPS

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

The performance fee logic in `FeeManager.collectPerformanceFee()` currently calculates fee accrual based on **absolute NAV increase** ( `currentNAV - lastNAVForPerformance` ) rather than **profit per share (PPS) increase**.

As a result, if the vault supply changes significantly (due to new deposits or redemptions), the NAV delta may not accurately reflect real profits attributable to existing LPs. This can lead to **miscalculated performance fees** - either overcharging or undercharging fee shares.

The team acknowledged this issue and noted that a fix (switching the comparison basis from NAV difference to PPS/profit difference) was already prepared in a previous PR but not merged at the time of this review.

### Recommendations

Use **per-share profit (PPS)** rather than total NAV increase as the basis for performance fee calculation. Specifically:

- Track `lastPPS` instead of `lastNAVForPerformance` .
- Calculate performance fee only on positive `PPS` growth since the last fee collection.
- Ensure the calculation normalizes for share supply changes to avoid dilution artifacts.

# Low findings

## [L-01] Bypass delay via unoverridden `TimelockController`

VaultTimelockController introduces per-function delay logic by redefining schedule() and scheduleBatch() to automatically determine delay values. However, the inherited TimelockControllerUpgradeable functions with signatures including an explicit uint256 delay argument remain callable. Since they are not overridden, proposers can bypass the enhanced delay enforcement by calling the inherited functions directly.

```
function schedule(
    address target,
    uint256 value,
    bytes calldata data,
    bytes32 predecessor,
    bytes32 salt,
    uint256 delay
) public virtual onlyRole(PROPOSER_ROLE) {
    bytes32 id = hashOperation(target, value, data, predecessor, salt);
    _schedule(id, delay);
    emit CallScheduled(id, 0, target, value, data, predecessor, delay);
    if (salt != bytes32(0)) {
        emit CallSalt(id, salt);
    }
}
```

Explicitly override and disable the base schedule() and scheduleBatch() functions that include the delay parameter.

## [L-02] Improper removal of `OFT` asset in `removeDepositPipe`

In the removeDepositPipe() function, when a deposit pipe is removed, the corresponding asset's OFT approval is revoked, and the mapping assetOFTs[asset] is deleted. However, if the removed asset corresponds to the underlying asset, this operation will also remove the underlying asset OFT (underlyingAssetOFT) from the approved list.

Since the underlying asset OFT is required for redemption and cross-chain operations, removing it will break redemption flows and make further cross-chain asset redemptions fail.

Add a condition to prevent removing or unapproving the underlying asset OFT.

## [L-03] Missing storage gap in `VaultComposerBase`

The VaultComposerBase contract is designed to be upgradeable, but it does not include a reserved storage gap (__gap) at the end of its storage layout. While the derived contract OVaultComposerMulti stores its state in a custom storage slot, future versions of VaultComposerBase may introduce new variables. Without a reserved gap, such additions could lead to storage layout conflicts in future upgrades.

Add a reserved storage gap to VaultComposerBase `uint256[50] private __gap;`

## [L-04] `convertAmount()` fails to revert if 0 and from and to are same

```
In `convertAmount`, when `fromAsset == toAsset`, the transaction is not reverted even if
`amount` is 0. This can be fixed by calling `require(amount > 0, "PythOracle: zero amount");`
first.

```solidity
function convertAmount(address fromAsset, address toAsset, uint256 amount)
    external
    view
    override
    returns (uint256)
{
    PythPriceOracleStorage storage $ = _getPythPriceOracleStorage();

@>  if (fromAsset == toAsset) {
        // Same asset conversion - always return the same amount
@>      return amount;
    }
    ...

@>  require(amount > 0, "PythOracle: zero amount");
    ...
}
```

## [L-05] `VaultTimelockController` owner not initialized

VaultTimelockController inherits from Ownable2StepUpgradeable, but the owner is not set because `__Ownable_init` is not called. Add `__Ownable_init` in `initialize`.

## [L-06] Bypass `maxPercentageIncrease` in `setTotalAssets`

`setTotalAssets` limits the amount that can be increased via `maxPercentageIncrease`, but this can be bypassed by calling it multiple times in small increments. To prevent the attack, the time when `setTotalAssets` was called should be stored, and a delay should be introduced before calling `setTotalAssets`.

## [L-07] Set max age per price feed

Pyth oracle can have different heartbeats for each feed. (reference) Therefore, it is recommended to allow max age to be set per feed.
```

## [L-08] `PYTH_PRICE_ORACLE_STORAGE_LOCATION` is set incorrectly

The `PYTH_PRICE_ORACLE_STORAGE_LOCATION` should be `keccak256(abi.encode(uint256(keccak256("liminal.storage.pythPriceOracle.v1")) - 1)) & ~bytes32(uint256(0xff))`, which is `0x6f85a36bf60a82bac3899b2e1a52e858797e8ed23189778ff18c2c3bcd836600`.

The `PYTH_PRICE_ORACLE_STORAGE_LOCATION` is currently set to an incorrect value.

```
    // keccak256(abi.encode(uint256(keccak256("liminal.storage.pythPriceOracle.v1")) - 1)) &
~bytes32(uint256(0xff))
@>  bytes32 private constant PYTH_PRICE_ORACLE_STORAGE_LOCATION =
        0x79f8fe64cf697304b8736b5ceebe50109f667154b58cb0fe6be0d930c76b5e00;
```

## [L-09] Unrevoked old OFT approvals after asset re-registration

In the `OVaultComposerMulti` contract, the `registerDepositPipe` function allows re-registering an existing asset to a new deposit pipe and OFT without automatically revoking the approval for the previous OFT.

```
File: OVaultComposerMulti.sol
200:     function registerDepositPipe(address asset, address pipe, address assetOFT)
201:         external
202:         onlyTimelock
203:     {
204:         require(asset != address(0), "OVaultComposerMulti: zero asset");
205:         require(pipe != address(0), "OVaultComposerMulti: zero pipe");
206:         require(assetOFT != address(0), "OVaultComposerMulti: zero OFT");
207:
208:         OVaultComposerMultiStorage storage $ = _getOVaultComposerMultiStorage();
209:         require($.depositPipes[asset] != pipe, "OVaultComposerMulti: same pipe");
210:
211:         // Verify pipe accepts this asset
212:         address pipeAsset = IDepositPipe(pipe).asset();
213:         if (pipeAsset != asset) revert InvalidPipe();
214:
215:         _addAsset(asset, pipe); // updates mapping and array
216:@>       $.assetOFTs[asset] = assetOFT;
217:
218:         // Approve the asset OFT for compose messages
219:@>       $.approvedOFTs[assetOFT] = true;
220:
221:         emit DepositPipeRegistered(asset, pipe, assetOFT);
222:     }
```

This leaves the old OFT in an approved state within the `approvedOFTs` mapping, enabling it to potentially trigger `VaultComposerBase.lzCompose` and `handleCompose` if it receives cross-chain messages. While `setOFTApproval` exist for manual revocation, they are timelocked and must be executed separately.

The issue stems from the lack of validation or automatic cleanup in `registerDepositPipe` when overwriting `assetOFTs[asset]`. If an OFT is updated (e.g., due to a security patch), the old one remains approved unless explicitly disabled. This could allow malicious or erroneous compose messages from the old OFT to be processed, as `_isApprovedOFT` would still validate positively.

Modify `registerDepositPipe` to automatically handle updates: If `assetOFTs[asset] != address(0)`, fetch the old OFT, set `approvedOFTs[oldOFT] = false`.

## [L-10] Hardcoded `minAmountLD` may cause loss in cross-chain redemptions

The `redeemAndSend` function in `OVaultComposerMulti.sol`, redeems shares via the redemption pipe, performs an initial slippage check, then sets `minAmountLD` to 0 before calling `_send`. This overrides the user's provided minimum, bypassing OFT's built-in slippage protection in `_debitView`. Same behavior in `_handleRedeemShares`.

```
File: OVaultComposerMulti.sol
154:         _assertSlippage(assets, _sendParam.minAmountLD);
155:         _sendParam.amountLD = assets;
156:@>       _sendParam.minAmountLD = 0;
```

```
File: OVaultComposerMulti.sol
406:         if (sendParam.dstEid != 0) {
407:             sendParam.amountLD = assets;
408:@>           sendParam.minAmountLD = 0; // Already protected
409:
410:             // Approve exact amount for cross-chain send
411:             IERC20($.underlyingAsset).approve($.underlyingAssetOFT, assets);
412:
413:             _send($.underlyingAssetOFT, sendParam, redeemer);
```

LayerZero OFT handles decimal differences across chains by removing dust in `OFTCore._removeDust`, flooring amounts to multiples of `decimalConversionRate`. With `minAmountLD = 0`, transfers always proceed, but users may receive less than redeemed if dust is removed, leaving remnants in the composer.

This risk increases if `OVaultComposerMulti.setUnderlyingAsset` changes the underlying to a token with different decimals (e.g., from 6 to 18), altering the conversion rate and potentially discarding larger dust amounts. Without a non-zero `minAmountLD`, there's no reversion on unexpected losses, leading to silent under-delivery.

To address this, implement dynamic slippage handling when underlyingAsset cross-chain transfers in `redeemAndSend` and `_handleRedeemShares`:

## [L-11] Malicious fulfillment calls delaying asset recovery indefinitely

In the `RedemptionPipe` contract, the functions `fulfillFastRedeems` and `fulfillRedeems` (accessible only to the `FULFILL_MANAGER_ROLE`) unconditionally update `lastRedemptionTime` to the current `block.timestamp` after processing the input arrays, even if the arrays are empty or no actual redemptions are fulfilled.

```
File: RedemptionPipe.sol
445:     function fulfillFastRedeems(address[] calldata owners, uint256[] calldata shares,
uint256[] calldata customFees)
446:         external
447:         onlyRole(FULFILL_MANAGER_ROLE)
448:     {
...
465:         $.lastRedemptionTime = block.timestamp;
466:     }
```

This timestamp is used in the `RedemptionPipe.recoverAssets()` function to enforce a recovery delay:

```
File: RedemptionPipe.sol
197:         require($.treasury != address(0), "RedemptionPipe: treasury not set");
198:         require(block.timestamp > $.lastRedemptionTime + $.recoveryDelay, "RedemptionPipe:
recovery delay not met");
```

A malicious, compromised holder of `FULFILL_MANAGER_ROLE` can call these functions with empty parameters (e.g., `owners = []`, `shares = []`), resetting the recovery timer each time. This prevents the timelock from executing `recoverAssets()`, effectively blocking asset recovery indefinitely. The issue arises because the `lastRedemptionTime` update is not conditioned on the successful fulfillment of at least one redemption, allowing trivial calls to manipulate the state.

Only update `lastRedemptionTime` if at least one redemption was successfully processed.

## [L-12] Asset config inconsistency between composer and redemption pipe

The `setUnderlyingAsset` function in `OVaultComposerMulti.sol` updates the `underlyingAsset` without validating it against the `underlyingAsset` in `RedemptionPipe.sol` (immutable post-initialization). It also doesn't update or validate `underlyingAssetOFT`, risking inconsistencies where `OVaultComposerMulti` approves and sends the wrong token via an incompatible OFT:

```
File: OVaultComposerMulti.sol
158:         // Approve exact amount for cross-chain send
159:         IERC20($.underlyingAsset).approve($.underlyingAssetOFT, assets);
```

The redemption pipe's `underlyingAsset` is set during `initialize` and exposed via a getter, but the composer doesn't sync with it during updates.

```
File: OVaultComposerMulti.sol
263:     function setUnderlyingAsset(address _underlyingAsset) external onlyTimelock {
264:         require(_underlyingAsset != address(0), "OVaultComposerMulti: zero asset");
265:
266:         OVaultComposerMultiStorage storage $ = _getOVaultComposerMultiStorage();
267:         address oldAsset = $.underlyingAsset;
268:         $.underlyingAsset = _underlyingAsset;
269:         emit UnderlyingAssetUpdated(oldAsset, _underlyingAsset);
270:     }
```

Changing the composer's `underlyingAsset` to a different token desynchronizes the system: `IRedemptionPipe.redeem` transfers the pipe's asset, but the composer approves/sends assuming its own configuration, causing reverts (e.g., approval on an incorrect token or OFT token incompatibility).

```
File: RedemptionPipe.sol
326:         // Transfer assets to user (fee stays with liquidity provider)
327:         $.underlyingAsset.safeTransferFrom($.liquidityProvider, receiver, assetsAfterFee);
```

There's no derivation of `underlyingAsset` from `IOFT(underlyingAssetOFT).token()` (unlike `SHARE_ERC20` in `VaultComposerBase`), missing automatic linkage. Similarly, `setRedemptionPipe` doesn't sync the composer's `underlyingAsset` or `underlyingAssetOFT`, increasing desync risks during pipe changes.

It is recommended:

- Modify `OVaultComposerMulti.setUnderlyingAsset` to accept `_underlyingAssetOFT` parammeter, derive `_underlyingAsset = IOFT(_underlyingAssetOFT).token()`, and update both storage variables atomically.
- Add validation in `setUnderlyingAsset`: `require(_underlyingAsset == IRedemptionPipe(redemptionPipe).underlyingAsset(), "");` to ensure consistency.
- In `OVaultComposerMulti.setRedemptionPipe`, fetch `newUnderlyingAsset = IRedemptionPipe(_redemptionPipe).underlyingAsset()` and update composer's `underlyingAsset` and `underlyingAssetOFT` (prompt for new OFT if needed).
- In `OVaultComposerMulti.initialize`, set `underlyingAsset = IRedemptionPipe(_redemptionPipe).underlyingAsset()` and validate `_underlyingAssetOFT.token() == _underlyingAsset`.

## [L-13] Lack of Pyth confidence interval usage

The `PythPriceOracle` contract integrates with Pyth Network for price feeds but fails to incorporate the `confidence` interval (`conf` field) when fetching and converting prices. Pyth documentation recommends using the confidence interval to compute a conservative price range (e.g., `price ± conf`) to protect users and the protocol from volatility. Ignoring this can lead to over- or undervaluation of assets, particularly in deposits where incoming assets are priced to mint shares. For instance, during volatility, using the raw `price` without adjusting for `conf` might overvalue deposits, minting excess shares and diluting existing holders if the true price is lower.

It is recommended to use the confidence interval to protect your users from these unusual market conditions.

## [L-14] Missing validation of asset decimals in `setPriceId` and `setPriceIds`

Both `setPriceId` and its batch variant `setPriceIds` allow the caller to manually specify the `decimals` value for each asset. However, the functions do not verify that the provided `decimals` match the actual ERC-20 token metadata (`IERC20Metadata(asset).decimals()`). If an incorrect value is configured (for example, setting `18` instead of `6` for USDC), it would silently distort price scaling and result in incorrect NAV calculations or redemption ratios. Since the function is role-restricted, this is a governance misconfiguration risk rather than a direct exploit, but the potential impact is high.

Add a validation step to ensure that `decimals` provided by the role holder match the actual token metadata:

```
require(decimals == IERC20Metadata(asset).decimals(), "PythOracle: decimals mismatch");
```

## [L-15] `DepositPipe.convertToAssets()` understates deposit assets due to rounding

The view path `convertToAssets(shares)` chains multiple truncating conversions that can collectively **favor the depositor**, returning fewer `assets` than the true economic cost:

- `_convertToAssets(shares)` uses **Ceil** (protocol-favored) to compute `underlyingValue18`.
- `_normalizeFromDecimals18(underlyingValue18)` then divides by a 10^Δ scale when `underlyingDecimals < 18`, which **floors** and reduces `underlyingValueNative`.
- `priceOracle.convertAmount(underlying -> deposit)` performs further integer math that typically **floors** again. The function stitches these steps as shown here.

Net effect: the two downward roundings (normalization and price conversion) can outweigh the initial ceiling, so the preview may **understate** `assets` and slightly dilute existing holders when users rely on the view to size deposits.

- For deposit/preview-mint paths (including this view), round **up** when converting from 18 to native decimals, e.g. replace division by `Math.ceilDiv(value18, scaleFactor)` in the 18 ->native normalization helper.
- Refactor the oracle conversion used here to compute the price ratio with a **single high-precision** `mulDiv` **and one rounding at the end** (use `Floor` / `Ceil` consistently by operation), avoiding intermediate truncations.

## [L-16] Rounding in `DepositPipe.mint()` favors depositors

When users mint a fixed amount of vault shares via `DepositPipe.mint()`, the conversion chain introduces multiple downward roundings that collectively favor the depositor.

The function sequence:

```
uint256 underlyingValue18 = _convertToAssets(shares);
uint256 underlyingValueNative = _normalizeFromDecimals18(underlyingValue18);
assets = priceOracle.convertAmount(underlyingAsset, depositAsset, underlyingValueNative);
```

Includes two rounding points:

1. `_normalizeFromDecimals18()` performs integer division ( `value18 / scaleFactor` ), truncating decimals - always rounding **down**, which reduces the required amount of underlying assets.
2. `convertAmount()` in `PythPriceOracle` also floors intermediate ratios, potentially further reducing the calculated `assets` value.

As a result, a depositor may provide slightly fewer tokens than the actual economic value of the requested shares, leading to a small NAV inflation and dilution of existing holders.

- In `_normalizeFromDecimals18()`, use ceiling division when converting from 18 decimals to native decimals for deposit operations:

`solidity return Math.ceilDiv(value18, scaleFactor);` * Review `PythPriceOracle.convertAmount()` to ensure a single consistent rounding direction.

## [L-17] Missing liquidity and balance invariants in `withdraw()`

The `withdraw()` function lacks liquidity and post-balance checks that are present in `redeem()`. In `redeem()`, the contract verifies that the liquidity provider holds sufficient assets **before** the transfer and asserts that its post-transfer balance matches the expected value. In contrast, `withdraw()` performs the transfer without verifying either condition.

While this asymmetry is unlikely to create a direct exploit, it weakens invariant safety and makes debugging liquidity inconsistencies more difficult, especially when dealing with non-standard ERC-20 tokens or rounding edge cases.

```
// redeem(): has pre- and post-conditions
uint256 lpBalance = $.underlyingAsset.balanceOf($.liquidityProvider);
require(lpBalance >= assetsAfterFee, "RedemptionPipe: insufficient liquidity");

$.underlyingAsset.safeTransferFrom($.liquidityProvider, receiver, assetsAfterFee);

uint256 expectedLpBalance = lpBalance - assetsAfterFee;
require(
    $.underlyingAsset.balanceOf($.liquidityProvider) == expectedLpBalance,
    "RedemptionPipe: liquidity provider balance mismatch"
);
```

```
// withdraw(): missing both checks
$.underlyingAsset.safeTransferFrom($.liquidityProvider, receiver, actualNetAssets);
```

Mirror the same invariant checks in `withdraw()` to maintain symmetry and ensure predictable LP accounting:

```
uint256 lpBalance = $.underlyingAsset.balanceOf($.liquidityProvider);
require(lpBalance >= actualNetAssets, "RedemptionPipe: insufficient liquidity");

$.underlyingAsset.safeTransferFrom($.liquidityProvider, receiver, actualNetAssets);

uint256 expectedLpBalance = lpBalance - actualNetAssets;
require(
    $.underlyingAsset.balanceOf($.liquidityProvider) == expectedLpBalance,
    "RedemptionPipe: liquidity provider balance mismatch"
);
```

## [L-18] `previewDeposit` / `previewMint` improperly revert due to user or global limits

Finding a full description.

According to [EIP-4626](https://eips.ethereum.org/EIPS/eip-4626), the `previewDeposit` and `previewMint` functions:

> MUST NOT account for deposit or mint limits like those returned from maxDeposit or maxMint, and should always act as though the operation would be accepted regardless of approvals or limits.

However, in DepositPipe implementation, both `previewDeposit` and `previewMint` include checks for supply limits. These checks cause **reverts** if a user or the vault has reached their deposit limits, which **violates ERC-4626 expectations**. A similar issue also exists in RedemptionPipe if it is expected to be ERC**4626** compliant.

**Recommendations**

Remove all limit checks from `previewDeposit` and `previewMint` functions.

## [L-19] Processing later in `fulfillFastRedeems` provides an advantage

In `fulfillFastRedeems`, being later in the `owners` array is more advantageous. This is because withdrawal fees generated from fast withdrawals processed earlier accumulate, so those processed later can receive the fees generated earlier.

The order of `fulfillFastRedeems` parameters is determined by `customFees`. To process requests using `customFees` first, they are placed at the front of the `owners` and `shares` arrays. In other words, there is no guarantee that they will be called in the order in which fast withdrawal requests were made.

```
function fulfillFastRedeems(address[] calldata owners, uint256[] calldata shares, uint256[]
calldata customFees)
    external
    onlyRole(FULFILL_MANAGER_ROLE)
{
    ...
    for (uint256 i = 0; i < length; i++) {
        if (shares[i] > 0) {
            // Calculate assets for fee calculation
@>          uint256 assets = convertToAssets(shares[i]);
            // Use custom fee basis points if provided, otherwise use default
@>          uint256 feeBps = customFees.length > i ? customFees[i] : $.fees.fastRedeemFeeBps;
            require(feeBps <= BASIS_POINTS, "RedemptionPipe: Incorrect Custom Fee");
            uint256 fee = (assets * feeBps) / BASIS_POINTS;
@>          _fulfillFastRedeem(owners[i], shares[i], fee);
        }
    }
    $.lastRedemptionTime = block.timestamp;
}

function _fulfillFastRedeem(address owner, uint256 shares, uint256 fee) internal {
    ...

    // Calculate assets based on current NAV
@>  uint256 assets = convertToAssets(shares);

    uint256 assetsAfterFee = assets - fee;

    // Update NAV. Fees stay with liquidityProvider, only decrease NAV by user's portion
    uint256 navDecrease = assetsAfterFee;
@>  $.navOracle.decreaseTotalAssets(navDecrease);

    // Update pending
    request.shares -= shares;

    // Burn shares held in custody
@>  $.shareManager.burnSharesFromSelf(shares);

    ...
}
```

**Recommendations**

In `fulfillFastRedeems`, first calculate the amount of assets and fees for all withdrawal requests. Calculate all requests to be processed in this call using the same totalShare and NAV. Once all calculations are complete, iterate again and call `_fulfillFastRedeem`.