Veda-Labs

Boring-Vault

Security Assessment Report

Version: 2.1

December, 2025

Contents

Introduction 2
- Disclaimer 2
- Document Structure 2
- Overview 2

Security Assessment Summary 3
- Scope 3
- Approach 3
- Coverage Limitations 4
- Findings Summary 4

Detailed Findings 5

Summary of Findings 6
- Loss Evasion And Share Price Manipulation Before Posting Loss 8
- `refundDeposit()` Ignores Share Price Changes Leading To Protocol Accounting Imbalances 11
- Potential Fund Loss As Fees Owed Do Not Reduce Total Assets 13
- `lastUpdateTimestamp` Not Updated When No Pending Vesting Gains 15
- Platform Fees Overcharged After Vesting Ends Due To Incorrect `timeDelta` Calculation 17
- Potential Divide-by-Zero In `_updateExchangeRate()` 19
- Vesting Gains Released Early Because `lastVestingUpdate` Not Updated In `vestYield()` 21
- Rate Truncation In `GenericRateProviderWithDecimalScaling` Causes Distorted Share Distribution 23
- Bridging Bypasses `beforeTransfer()` Restrictions Allowing Denied Addresses To Receive Shares 25
- New Deposits Reset All User Share Lock Times 27
- Cross-Chain Bridge Bypasses Destination Chain Vault `cap` 29
- Precision Loss In `_changeDecimals()` Leads To Distorted Share Distribution 30
- Inconsistent Exchange Rate After `postLoss()` Leads To Inflated Platform Fees 32
- TWAS Calculation Leading To Unrealistic Daily Yield Leading To Revert During Vesting 34
- Inconsistent Fee Collection Between `_updateExchangeRate()` And `resetHighwaterMark()` 36
- Missing `beforeTransfer()` Check On Deposits Allows Denied Addresses to Mint Shares 38
- Cross-Chain Deposits Bypass Destination Chain Share Lock Periods 39
- Zero `shareLockPeriod` Prevents Immediate Withdrawals And Refunds 40
- Missing Exchange Rate Update Before Active Vest Check In `vestYield()` 42
- `GenericRateProvider` Does Not Provide Ways To Utilize Oracle Rate Validations 44
- Unwanted Reverts In `depositAndBridge()` And `depositAndBridgeWithPermit()` If Vault `cap` Is Set 46
- Missing `isPaused` Checks In `pause()` And `unpause()` Could Lead To Offchain Inconsistencies 48
- Miscellaneous General Comments 49
- Restricted Users Can Still Deposit Into The Vault 52
- Misssing Version Function In The `Teller` 53
- Missing Referral Address Support In Deposits 54
- Rounding Induced Vesting Loss And Exchange Rate Update Edge Cases 55

A Test Suite 57

B Vulnerability Severity Classification 58

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Veda-Labs components in scope. The review focused solely on the security aspects of the Solidity implementation of the contracts, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Veda-Labs components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Veda-Labs components in scope.

Overview

Boring Vaults are flexible vault contracts that allow for intricate strategies, secured by both onchain and offchain mechanisms.

The BoringVault architecture is made up of:

- **BoringVault**: A barebones vault contract that outsources complex functionality to external contracts.
- **Manager**: Limits the possible strategies BoringVaults can use, without large gas overheads, or unnecessary risk.
- **Teller**: Facilitates user deposits and withdrawals in/out of the BoringVault.
- **Accountant**: Provides a safe share price for Teller interactions via offchain oracles.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Boring-Vault and OAppAuth repositories.

The scope of this time-boxed review was strictly limited to the following files at Boring-Vault commit 3c768bd and OAppAuth commit 5beb4df:

- `BoringVault.sol`
- `TellerWithMultiAssetSupport.sol`
- `TellerWithBuffer.sol`
- `AccountantWithRateProviders.sol`
- `AccountantWithYieldStreaming.sol`
- `GenericRateProvider.sol`
- `GenericRateProviderWithDecimalScaling.sol`
- `CrossChainTellerWithGenericBridge.sol`
- `MessageLib.sol`
- `PairwiseRateLimiter.sol`
- `LayerZeroTeller.sol`
- `LayerZeroTellerWithRateLimiting.sol`
- `TellerWithYieldStreaming.sol`
- `OAppAuth/*`

The fixes of the identified issues were assessed at Boring-Vault commit c758db1.

Additional retesting has been performed specifically on `AccountantWithYieldStreaming.sol` at PR-552, commit 568d946.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 27 issues during this assessment. Categorised by their severity:

- High: 3 issues.
- Medium: 5 issues.
- Low: 13 issues.
- Informational: 6 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Veda-Labs components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
BOV-01	Loss Evasion And Share Price Manipulation Before Posting Loss	High	Closed
BOV-02	`refundDeposit()` Ignores Share Price Changes Leading To Protocol Accounting Imbalances	High	Closed
BOV-03	Potential Fund Loss As Fees Owed Do Not Reduce Total Assets	High	Closed
BOV-04	`lastUpdateTimestamp` Not Updated When No Pending Vesting Gains	Medium	Resolved
BOV-05	Platform Fees Overcharged After Vesting Ends Due To Incorrect `timeDelta` Calculation	Medium	Closed
BOV-06	Potential Divide-by-Zero In `_updateExchangeRate()`	Medium	Closed
BOV-07	Vesting Gains Released Early Because `lastVestingUpdate` Not Updated In `vestYield()`	Medium	Closed
BOV-08	Rate Truncation In `GenericRateProviderWithDecimalScaling` Causes Distorted Share Distribution	Low	Closed
BOV-09	Bridging Bypasses `beforeTransfer()` Restrictions Allowing Denied Addresses To Receive Shares	Low	Closed
BOV-10	New Deposits Reset All User Share Lock Times	Low	Closed
BOV-11	Cross-Chain Bridge Bypasses Destination Chain Vault `cap`	Low	Closed
BOV-12	Precision Loss In `_changeDecimals()` Leads To Distorted Share Distribution	Low	Closed
BOV-13	Inconsistent Exchange Rate After `postLoss()` Leads To Inflated Platform Fees	Low	Resolved
BOV-14	TWAS Calculation Leading To Unrealistic Daily Yield Leading To Revert During Vesting	Low	Resolved
BOV-15	Inconsistent Fee Collection Between `_updateExchangeRate()` And `resetHighwaterMark()`	Low	Resolved
BOV-16	Missing `beforeTransfer()` Check On Deposits Allows Denied Addresses to Mint Shares	Low	Closed
BOV-17	Cross-Chain Deposits Bypass Destination Chain Share Lock Periods	Low	Closed
BOV-18	Zero `shareLockPeriod` Prevents Immediate Withdrawals And Refunds	Low	Closed
BOV-19	Missing Exchange Rate Update Before Active Vest Check In `vestYield()`	Low	Closed
BOV-20	`GenericRateProvider` Does Not Provide Ways To Utilize Oracle Rate Validations	Informational	Closed
BOV-21	Unwanted Reverts In `depositAndBridge()` And `depositAndBridgeWithPermit()` If Vault `cap` Is Set	Informational	Closed

BOV-22	Missing `isPaused` Checks In `pause()` And `unpause()` Could Lead To Offchain Inconsistencies	Informational	Closed
BOV-23	Miscellaneous General Comments	Informational	Resolved
BOV-24	Restricted Users Can Still Deposit Into The Vault	Low	Resolved
BOV-25	Misssing Version Function In The `Teller`	Informational	Resolved
BOV-26	Missing Referral Address Support In Deposits	Informational	Resolved
BOV-27	Rounding Induced Vesting Loss And Exchange Rate Update Edge Cases	Medium	Closed

BOV-01	Loss Evasion And Share Price Manipulation Before Posting Loss		
Asset	AccountantWithYieldStreaming.sol TellerWithMultiAssetSupport.sol TellerWithYieldStreaming.sol		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

There is no restriction preventing a user from withdrawing and redepositing within the same block. This creates a potential for sandwich attacks when `postLoss()` is pending in the mempool.

When users deposit, the function `_afterPublicDeposit()` is called. It sets the `shareUnlockTime` in the user's `beforeTransferData`, ensuring that users cannot deposit and withdraw within the same transaction or block.

TellerWithMultiAssetSupport._afterPublicDeposit()

```
function _afterPublicDeposit(
    address user,
    ERC20 depositAsset,
    uint256 depositAmount,
    uint256 shares,
    uint256 currentShareLockPeriod
) internal {
    // Increment then assign as its slightly more gas efficient.
    uint256 nonce = ++depositNonce;
    // Only set share unlock time and history if share lock period is greater than 0.
    if (currentShareLockPeriod > 0) {
        beforeTransferData[user].shareUnlockTime = block.timestamp + currentShareLockPeriod;
        publicDepositHistory[nonce] = keccak256(
            abi.encode(user, depositAsset, depositAmount, shares, block.timestamp, currentShareLockPeriod)
        );
    }
    emit Deposit(nonce, user, address(depositAsset), depositAmount, shares, block.timestamp, currentShareLockPeriod);
}
```

When the function `withdraw()` is called, it triggers `beforeTransfer()` to verify that the required delay has passed.

TellerWithMultiAssetSupport.beforeTransfer()

```
function beforeTransfer(address from, address to, address operator) public view virtual {
    if (
        beforeTransferData[from].denyFrom || beforeTransferData[to].denyTo
            || beforeTransferData[operator].denyOperator
            || (permissionedTransfers && !beforeTransferData[operator].permissionedOperator)
    ) {
        revert TellerWithMultiAssetSupport__TransferDenied(from, to, operator);
    }
    if (beforeTransferData[from].shareUnlockTime > block.timestamp) {
        revert TellerWithMultiAssetSupport__SharesAreLocked();
    }
}
```

Due to no restricions on withdrawals and deposits within the same block, a user who has already deposited some amount (and its `shareUnlockTime` is already passed) could withdraw before `postLoss()` is executed, avoiding the share price decrease, and then redeposit the same amount after `postLoss()` to acquire more shares at the reduced price.

Furthermore, users can unfairly evade losses and manipulate share pricing. For example, suppose there are 100 assets and 100 shares, giving a share price of 1. A loss of 20 is about to be applied. If no vesting gains are available, the entire 20 loss will be deducted from total assets:

AccountantWithYieldStreaming.postLoss()

```
function postLoss(uint256 lossAmount) external requiresAuth {
    if (accountantState.isPaused) revert AccountantWithRateProviders__Paused();

    if (block.timestamp < lastStrategistUpdateTimestamp + accountantState.minimumUpdateDelayInSeconds) {
        revert AccountantWithYieldStreaming__NotEnoughTimePassed();
    }

    //ensure most up to date data
    _updateExchangeRate(); //vested gains are moved to totalAssets, only unvested remains in `vestingState.vestingGains`

    if (vestingState.vestingGains >= lossAmount) {
        //remaining unvested gains absorb the loss
        vestingState.vestingGains -= uint128(lossAmount);
    } else {
        uint256 principalLoss = lossAmount - vestingState.vestingGains;

        //wipe out remaining vesting
        vestingState.vestingGains = 0;

        //reduce share price to reflect principal loss
        uint256 currentShares = vault.totalSupply();
        if (currentShares > 0) {
            uint128 cachedSharePrice = vestingState.lastSharePrice;
            vestingState.lastSharePrice =
                uint128((totalAssets() - principalLoss).mulDivDown(ONE_SHARE, currentShares));

            uint256 lossBps =
                uint256(cachedSharePrice - vestingState.lastSharePrice).mulDivDown(10_000, cachedSharePrice);

            //verify the loss isn't too large
            if (lossBps > maxDeviationLoss) {
                accountantState.isPaused = true;
                emit Paused();
            }
        }
    }

    //update state timestamp
    lastStrategistUpdateTimestamp = uint64(block.timestamp);

    emit LossRecorded(lossAmount);
}
```

Now, consider three possible scenarios:

- Scenario 1: No withdrawals before the loss. The share price drops to `(100 - 20) / 100 = 0.8`.
- Scenario 2: 40 assets are withdrawn just before the loss. The share price becomes `(60 - 20) / 60 = 0.66`. If this drop exceeds `maxDeviationLoss`, the protocol pauses.
- Scenario 3: 90 assets are withdrawn just before the loss. Since `totalAssets()` falls below 20, the `postLoss()` call reverts.

In addition, the system is exposed to timing risks with rebasing tokens like `stETH`. Because exchange rate synchronization requires manual updates (`postLoss()` or `updateExchangeRate()`), there are exploitable windows where exchange rates become stale. Sophisticated users can monitor rebase events and front-run administrative actions, withdrawing before a negative rebase is applied or depositing before a positive rebase is recognised, to gain an unfair advantage. While share lock periods prevent instant withdrawals after deposits, they do not mitigate these timing-based attacks.

Recommendations

Enforce a minimum cooldown period between a user's withdrawal and subsequent deposit to mitigate this risk, or alternatively, introduce a two step withdrawal process.

Resolution

The development team has closed the issue with the following comment:

> *Given the business needs of this contract the recommended fixes would break the intended design.As per our usual protocol, rate updates (and losses) are submitted via private mempools (when available). Additionally, in order to take advantage of this, a user would have to be in the vault already, meaning their capital is actively contributing to earning yield for the vault and locked. Strategies being run here will be very safe and stable (lending w/o loops), meaning that the likelihood of a loss being posted is very, very low.*

BOV-02	`refundDeposit()` Ignores Share Price Changes Leading To Protocol Accounting Imbalances		
Asset	`TellerWithMultiAssetSupport.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The refund logic in `refundDeposit()` mishandles changes in share price, creating financial inconsistencies. This can either trap excess assets in the protocol or erode the loss reserve, weakening its ability to cover losses.

The function `refundDeposit()` can be called by `STRATEGIST_MULTISIG_ROLE` to refund a pending deposit. It works by burning the user's shares (equal to the amount minted at deposit time) and transferring back the same deposit amount.

TellerWithMultiAssetSupport.refundDeposit()

```
function refundDeposit(
    uint256 nonce,
    address receiver,
    address depositAsset,
    uint256 depositAmount,
    uint256 shareAmount,
    uint256 depositTimestamp,
    uint256 shareLockUpPeriodAtTimeOfDeposit
) external requiresAuth {
    if ((block.timestamp - depositTimestamp) >= shareLockUpPeriodAtTimeOfDeposit) {
        // Shares are already unlocked, so we can not revert deposit.
        revert TellerWithMultiAssetSupport__SharesAreUnLocked();
    }
    bytes32 depositHash = keccak256(
        abi.encode(
            receiver, depositAsset, depositAmount, shareAmount, depositTimestamp, shareLockUpPeriodAtTimeOfDeposit
        )
    );
    if (publicDepositHistory[nonce] != depositHash) revert TellerWithMultiAssetSupport__BadDepositHash();

    // Delete hash to prevent refund gas.
    delete publicDepositHistory[nonce];

    // If deposit used native asset, send user back wrapped native asset.
    depositAsset = depositAsset == NATIVE ? address(nativeWrapper) : depositAsset;
    // Burn shares and refund assets to receiver.
    vault.exit(receiver, ERC20(depositAsset), depositAmount, receiver, shareAmount);

    emit DepositRefunded(nonce, depositHash, receiver);
}
```

The issue is that this refund logic ignores changes in share price between deposit and refund. Two scenarios arise:

- If the price increases between deposit and refund, some assets will remain locked in the protocol and become inaccessible.
- If the price decreases between deposit and refund, the protocol's loss reserve will shrink incorrectly, leaving less than required to cover losses.

For example:

- Initially, there are 900 assets and 900 shares, with share price at 1.

- A user deposits 100 assets and receives 100 shares.
- Later, a 200 asset loss occurs. Share price drops to `(1000 - 200) / 1000 = 0.8`. Total supply is 1000 shares worth 800 assets, with 200 assets absorbed as loss.
- If `refundDeposit()` is called, 100 shares are burnt and 100 assets are refunded.
- Supply becomes 900 shares, and protocol balance is 900 assets. The share price remains 0.8, so 900 shares are worth 720 assets, leaving only 180 assets reserved for loss.
- This misaligns with the original 200 loss, meaning the protocol no longer holds enough to cover it.

A similar inconsistency occurs if share price increases, leaving excess assets stuck in the protocol.

Recommendations

The refund logic should account for price changes since the deposit. Instead of refunding the exact deposit amount, the value should be adjusted based on the current share price.

Resolution

The development team has closed the issue with the following comment:

> *This is working as intended. Refunds are meant to be refunds, not adjusted for current market rates.*

BOV-03	Potential Fund Loss As Fees Owed Do Not Reduce Total Assets		
Asset	`AccountantWithRateProviders.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

In a scenario where all users withdraw their deposits and sufficient fees have been claimed, it would be possible for depositors to lose their funds.

Whenever `_calculateFeesOwed()` is called, it updates `AccountantState.feesOwedInBase`:

AccountantWithRateProviders._calculateFeesOwed()

```
function _calculateFeesOwed(
    AccountantState storage state,
    uint96 newExchangeRate,
    uint256 currentExchangeRate,
    uint256 currentTotalShares,
    uint64 currentTime
) internal virtual {
    // ....
    state.feesOwedInBase += uint128(newFeesOwedInBase);
}
```

The `claimFees()` function can then be called by the vault to transfer the owed fees to the `payoutAddress`:

AccountantWithRateProviders.claimFees()

```
function claimFees(ERC20 feeAsset) external {
    if (msg.sender != address(vault)) revert AccountantWithRateProviders__OnlyCallableByBoringVault();

    // ... calculates feesOwedInFeeAsset ...

    // Zero out fees owed.
    state.feesOwedInBase = 0;
    // Transfer fee asset to payout address.
    feeAsset.safeTransferFrom(msg.sender, state.payoutAddress, feesOwedInFeeAsset);

    emit FeesClaimed(address(feeAsset), feesOwedInFeeAsset);
}
```

The claimed fees are deducted directly from the vault's balance, which holds user deposits. However, the total asset value used for share price calculations does not account for these claimed fees. This creates a discrepancy where the sum of user deposits and accrued fees exceeds the actual protocol balance.

This can lead to three issues:

1. If all users withdraw their shares, the last withdrawing users may not be able to withdraw their funds as the vault balance would be too low, resulting in a loss of their funds.
2. If all users withdraw their shares, the protocol may have no assets left to claim fees, effectively losing the owed fees.
3. A user withdrawing after fee claims may receive fewer assets than if they withdrew before fee claims.

Recommendations

Deduct owed fees from the total assets so that the share price accurately reflects both user deposits and fees, preventing inconsistencies in withdrawals and fee collection.

Resolution

The development team has closed the issue with the following comment:

> *Fees are taken into account when posting rate updates. These are calculated off-chain by strategists. New exchange rates will have the fees already deducted.*

BOV-04	`lastUpdateTimestamp` Not Updated When No Pending Vesting Gains		
Asset	`AccountantWithYieldStreaming.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The fee accounting logic skips timestamp updates when no yields vest. This inflates future fee calculations, causing the protocol to overcharge during later vesting periods.

During the exchange rate update, if there are no pending vesting gains, `newlyVested` will be zero. In this case, the body of the `if` statement is skipped, meaning `_collectFees()` is not called and `state.lastUpdateTimestamp` remains unchanged:

AccountantWithYieldStreaming._updateExchangeRate()

```
function _updateExchangeRate() internal {
    _updateCumulative();

    // Calculate how much has vested since `lastVestingUpdate`
    uint256 newlyVested = getPendingVestingGains();

    uint256 currentShares = vault.totalSupply();
    if (newlyVested > 0) {
        // Update share price without reincluding pending gains
        uint256 _totalAssets = uint256(vestingState.lastSharePrice).mulDivDown(currentShares, ONE_SHARE);
        vestingState.lastSharePrice = uint128((_totalAssets + newlyVested).mulDivDown(ONE_SHARE, currentShares));

        _collectFees();

        // Move vested amount from pending to realized
        vestingState.vestingGains -= uint128(newlyVested);
        vestingState.lastVestingUpdate = uint128(block.timestamp);
    }

    AccountantState storage state = accountantState;
    state.totalSharesLastUpdate = uint128(currentShares);

    emit ExchangeRateUpdated(vestingState.lastSharePrice);
}
```

If fees are collected later when new yields vest, `_calculatePlatformFee()` computes the platform fee using `timeDelta = currentTime - lastUpdateTimestamp`. Since `lastUpdateTimestamp` was not updated during periods with no vesting, `timeDelta` includes periods where no yields were active, resulting in overestimated fees.

The root cause is that `lastUpdateTimestamp` is only updated when `_collectFees()` is called, which only happens when `newlyVested > 0`.

Recommendations

Update `lastUpdateTimestamp` even if no new vesting occurs, or update it every time `vestYield()` is called to ensure platform fees are accurately calculated.

Resolution

This issue has been addressed in commit daee10b. It was fixed by moving `lastVestingUpdate` outside of the block that checks for gains; the variable now always updates.

BOV-05	Platform Fees Overcharged After Vesting Ends Due To Incorrect `timeDelta` Calculation		
Asset	`AccountantWithRateProviders.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The platform fee calculation depends on deposit timing, producing inconsistent and inflated charges. Identical economic scenarios can yield different fees, undermining fairness and predictability of protocol costs.

During exchange rate updates, `_collectFees()` is invoked, which calls `_calculateFeesOwed()` and then `_calculatePlatformFee()`. The platform fee is calculated as `platformFeesAnnual * timeDelta / 365 days`, where `timeDelta` is `currentTime - lastUpdateTimestamp`:

AccountantWithRateProviders._calculatePlatformFee()

```
// Determine platform fees owned.
if (platformFee > 0) {
    uint256 timeDelta = currentTime - lastUpdateTimestamp;
    uint256 minimumAssets = newExchangeRate > currentExchangeRate
        ? shareSupplyToUse.mulDivDown(currentExchangeRate, ONE_SHARE)
        : shareSupplyToUse.mulDivDown(newExchangeRate, ONE_SHARE);
    uint256 platformFeesAnnual = minimumAssets.mulDivDown(platformFee, 1e4);
    platformFeesOwedInBase = platformFeesAnnual.mulDivDown(timeDelta, 365 days);
}
```

Here, `currentTime` is always equal to `block.timestamp`, regardless of whether it is after the vesting end time. This should instead use `min(block.timestamp, VestingState.endVestingTime)`, since fees should only apply when yields are actively vesting.

This leads to an inconsistency. Consider a vesting period of 10 days:

- On day 7, a deposit is made.
- On day 12, another deposit is made. At this point, gains from day 0 to day 10 should already be fully vested. During calculating the platform fee when the second deposit is made, `_calculateFeesOwed()` uses `timeDelta = 12 - 7 = 5 days`. However, only 3 days (from day 7 to day 10) had gains available, so the fee is overestimated.

Comparing two scenarios:

- Scenario 1: Deposit on day 7, another on day 12 → fee based on `platformFeesAnnual * 7 / 365 + platformFeesAnnual * 5 / 365`.
- Scenario 2: Deposit on day 7, another on day 10, and another on day 12 → fee based on `platformFeesAnnual * 7 / 365 + platformFeesAnnual * 3 / 365 + 0`.

Thus, the same economic situation produces different fees depending on when deposits occur, causing inconsistent results.

Recommendations

Use `min(block.timestamp, VestingState.endVestingTime)` in the function `_calculatePlatformFee()` to calculate the `timeDelta`, since fees should only apply when yields are actively vesting.

Resolution

The development team has closed the issue with the following comment:

> *Platform fees are charged based on AUM, regardless of performance.*

BOV-06	Potential Divide-by-Zero In `_updateExchangeRate()`		
Asset	`AccountantWithYieldStreaming.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The exchange rate update logic can trigger a divide-by-zero when vesting gains exist but no shares remain. This could halt core protocol functions such as deposits and yield processing.

During the exchange rate update, the function calculates the newly vested amount by calling `getPendingVestingGains()`. It then adds this amount to `_totalAssets` and divides by `totalSupply()` to determine the new share price:

AccountantWithYieldStreaming._updateExchangeRate()

```
function _updateExchangeRate() internal {
    _updateCumulative();

    // Calculate how much has vested since `lastVestingUpdate`
    uint256 newlyVested = getPendingVestingGains();

    uint256 currentShares = vault.totalSupply();
    if (newlyVested > 0) {
        // Update the share price without reincluding the pending gains (handled in `newlyVested`)
        uint256 _totalAssets = uint256(vestingState.lastSharePrice).mulDivDown(currentShares, ONE_SHARE);
        vestingState.lastSharePrice = uint128((_totalAssets + newlyVested).mulDivDown(ONE_SHARE, currentShares));

        _collectFees();

        // Move vested amount from pending to realized
        vestingState.vestingGains -= uint128(newlyVested); // remove from pending
        vestingState.lastVestingUpdate = uint128(block.timestamp); // update timestamp
    }

    AccountantState storage state = accountantState;
    state.totalSharesLastUpdate = uint128(currentShares);

    emit ExchangeRateUpdated(vestingState.lastSharePrice);
}
```

If `totalSupply()` is zero, a divide-by-zero will occur. This can happen when all users have withdrawn, burning all shares (`totalSupply() = 0`), while some vesting gains are still pending (`newlyVested > 0`). In that scenario, the following line will revert:

Problematic Division Line

```
vestingState.lastSharePrice = uint128((_totalAssets + newlyVested).mulDivDown(ONE_SHARE, currentShares));
```

As a result, any function that relies on `_updateExchangeRate()` will fail and revert, rendering subsequent deposits, vesting yield, and loss postings nonfunctional, which can cause the protocol as a whole to become inoperative.

Recommendations

Refactor the code to handle the case when `currentShares` is zero to prevent a divide-by-zero error, ensuring safe calculation of `lastSharePrice` even when no shares remain.

Resolution

The development team has acknowledged the issue with the following comment:

> *In practice, if every single user left the vault at the same time and yield remained, something has gone very wrong and the vault would be paused, blocking deposits anyways.*

BOV-07	Vesting Gains Released Early Because `lastVestingUpdate` Not Updated In `vestYield()`		
Asset	`AccountantWithYieldStreaming.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The vesting logic incorrectly anchors calculations to contract creation time instead of the vesting start. This causes yields to vest prematurely, breaking intended schedules and overstating user rewards.

When the function `vestYield()` is called, it updates `vestingState.startVestingTime` and `vestingState.endVestingTime` but does not update `vestingState.lastVestingUpdate`.

Excerpt from AccountantWithYieldStreaming.vestYield()

```
//update vesting timestamps
vestingState.startVestingTime = uint64(block.timestamp);
vestingState.endVestingTime = uint64(block.timestamp + duration);
```

This means `vestingState.lastVestingUpdate` remains set to the contract creation time:

AccountantWithYieldStreaming Constructor

```
//initialize vesting state
vestingState.lastSharePrice = startingExchangeRate;
vestingState.vestingGains = 0;
vestingState.lastVestingUpdate = uint128(block.timestamp);
vestingState.startVestingTime = uint64(block.timestamp);
vestingState.endVestingTime = uint64(block.timestamp);
```

The issue arises when `getPendingVestingGains()` is called. It calculates `amountVested` based on `timeSinceLastUpdate = currentTime - vestingState.lastVestingUpdate`, which incorrectly uses the contract creation timestamp for `vestingState.lastVestingUpdate` rather than the actual start time of vesting. This results in yield being vested earlier than intended.

Excerpt from AccountantWithYieldStreaming.getPendingVestingGains()

```
//time that has passed since last update
uint256 timeSinceLastUpdate = currentTime - vestingState.lastVestingUpdate;

//total remaining vesting period when we last updated
uint256 totalRemainingTime = vestingState.endVestingTime - vestingState.lastVestingUpdate;

//vest it linearly over the remaining time
//return amountVested = (vestingState.vestingGains * timeSinceLastUpdate) / totalRemainingTime;
return amountVested = uint256(vestingState.vestingGains).mulDivDown(timeSinceLastUpdate, totalRemainingTime);
```

For example:

- Contract created at day 100.
- User deposits at day 105.
- Yield vested at day 108, with vesting scheduled until day 120.
- Another deposit occurs at day 112.

The contract calculates:
`vestingGains * (112 - 100) / (120 - 100) = vestingGains * 12 / 20`

But the correct calculation should be:
`vestingGains * (112 - 108) / (120 - 108) = vestingGains * 4 / 12`

This discrepancy is significant, as `12/20` of the yield is vested instead of the intended `4/12`.

Note that this issue is not limited to the first yield vesting after contract creation. It can also occur when all prior yield has already vested and `vestYield()` is called again after some time. In this scenario, `vestingState.lastVestingUpdate` reflects the last time `_updateExchangeRate()` was called and `getPendingVestingGains()` returned a nonzero value.

For example, continuing from the previous scenario: if a deposit is made on day 120, all vested yield will be claimed, leaving pending vesting gains at zero. Later, if a new yield with a 50 day duration starts vesting on day 180 and a deposit is made on day 190, the released yield would be calculated as:

Incorrect Calculation
```
new vestingGains * (190 - 120) / (180 + 50 - 120)
```

However, it should correctly be calculated based on the new vesting period:

Correct Calculation
```
new vestingGains * (190 - 180) / (180 + 50 - 180)
```

Recommendations

Update `vestingState.lastVestingTime` when yield is vested to ensure correct vesting behaviour.

Resolution

The development team has closed the issue with the following comment:

> *This update happens inside `_updateExchangeRate`, which is called when yield is vested.*

BOV-08	Rate Truncation In `GenericRateProviderWithDecimalScaling` Causes Distorted Share Distribution		
Asset	`AccountantWithRateProviders.sol GenericRateProviderWithDecimalScaling.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `GenericRateProviderWithDecimalScaling` contract uses floor division when scaling rates from higher to lower decimals, causing precision loss that unfairly disadvantages users depositing quote tokens.

Consider a vault using `DAI` as a base and `cDAI` as a quote, and `cDAI`'s exchange rate, which is in 28 decimals. When it is scaled down to quote token decimals (8 for `cDAI`), the division truncates precision and creates unequal share distribution.

GenericRateProviderWithDecimalScaling.getRate()

```
function getRate() public override view returns (uint256) {
    // ...

    if (inputDecimals > outputDecimals) {
        // @audit Floor division loses precision for quote tokens
        return rate / 10 ** (inputDecimals - outputDecimals);
    }
    // ...
}
```

For example, using the cDAI/DAI rate provider, which returns `24642927985426739` wei per `cDAI` (approximately 0.0246 cDAI per ETH), the `getRate()` function will truncate that rate to `2464292` when scaling from 28 decimals down to 8 decimals for `cDAI` (this is following the documentation for what rate providers should return).

AccountantWithRateProviders.setRateProviderData()

```
/**
* @notice Update the rate provider data for a specific `asset`.
* @dev Rate providers must return rates in terms of `base` or
* an asset pegged to base and they must use the same decimals
* as `asset`.
* @dev Callable by OWNER_ROLE.
*/
function setRateProviderData(ERC20 asset, bool isPeggedToBase, address rateProvider) external requiresAuth {
   rateProviderData[asset] =
      RateProviderData({isPeggedToBase: isPeggedToBase, rateProvider: IRateProvider(rateProvider)});
   emit RateProviderUpdated(address(asset), isPeggedToBase, rateProvider);
}
```

The truncated rate is then used in share calculations through `getRateInQuote()`:

AccountantWithRateProviders.getRateInQuote()

```
function getRateInQuote(ERC20 quote) public view virtual returns (uint256 rateInQuote) {
    // ...
    uint256 quoteRate = data.rateProvider.getRate(); // @audit Returns truncated rate (225 instead of 225949367088607)
    uint256 oneQuote = 10 ** quoteDecimals;
    rateInQuote = oneQuote.mulDivDown(exchangeRateInQuoteDecimals, quoteRate);
}
```

Since the truncated rate becomes the denominator in `getRateInQuote()`, a smaller denominator results in a larger rate value, which then becomes the denominator in the final share calculation. This causes quote token depositors to receive fewer shares than they should.

Recommendations

Remove decimal scaling from `GenericRateProviderWithDecimalScaling` and handle decimal differences directly in `getRateInQuote()` calculations using `FixedPointMathLib` operations, preserving full precision throughout the computation.

Resolution

The development team has acknowledged the issue with no comment.

BOV-09	Bridging Bypasses `beforeTransfer()` Restrictions Allowing Denied Addresses To Receive Shares		
Asset	`CrossChainTellerWithGenericBridge.sol` `TellerWithMultiAssetSupport.sol` `OAppAuthReceiver.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The share transfer restrictions enforced by `beforeTransfer()` can be bypassed via bridging. This loophole allows denied addresses to still receive shares, undermining access controls and enforcement of transfer restrictions.

There is a hook `beforeTransfer()` to check if shares are locked, or if `from`, `to`, or `operator` are denied in `beforeTransferData`:

TellerWithMultiAssetSupport.beforeTransfer()

```
function beforeTransfer(address from, address to, address operator) public view virtual {
    if (
        beforeTransferData[from].denyFrom || beforeTransferData[to].denyTo
            || beforeTransferData[operator].denyOperator
            || (permissionedTransfers && !beforeTransferData[operator].permissionedOperator)
    ) {
        revert TellerWithMultiAssetSupport__TransferDenied(from, to, operator);
    }
    if (beforeTransferData[from].shareUnlockTime > block.timestamp) {
        revert TellerWithMultiAssetSupport__SharesAreLocked();
    }
}
```

This hook is triggered when transferring shares between accounts. However, it can be bypassed through the bridging mechanism.

For example, if Alice is denied as a `to` address on chain A (`beforeTransferData[Alice].denyTo = true`), no one can transfer shares to Alice on chain A. But users can bypass this by bridging their shares from chain A to chain B, and then bridging back to Alice on chain A. When the message is received on chain A, the contract mints shares directly to Alice without verifying if Alice is denied:

OAppAuthReceiver.lzReceive()

```
function lzReceive(
    Origin calldata _origin,
    bytes32 _guid,
    bytes calldata _message,
    address _executor,
    bytes calldata _extraData
) public payable virtual {
    // Ensures that only the endpoint can attempt to lzReceive() messages to this OApp.
    if (address(endpoint) != msg.sender) revert OnlyEndpoint(msg.sender);

    // Ensure that the sender matches the expected peer for the source endpoint.
    if (_getPeerOrRevert(_origin.srcEid) != _origin.sender) revert OnlyPeer(_origin.srcEid, _origin.sender);

    // Call the internal OApp implementation of lzReceive.
    _lzReceive(_origin, _guid, _message, _executor, _extraData);
}
```

CrossChainTellerWithGenericBridge._completeMessageReceive()

```
function _completeMessageReceive(bytes32 messageId, uint256 message) internal {
    MessageLib.Message memory m = message.uint256ToMessage();

    // Mint shares to message.to
    vault.enter(address(0), ERC20(address(0)), 0, m.to, m.shareAmount);

    emit MessageReceived(messageId, m.shareAmount, m.to);
}
```

The bridging process bypasses the denying mechanism, allowing restricted addresses to receive shares.

Recommendations

Implement checks in the bridging process so that bypassing the denying mechanism this way is prevented.

Resolution

The development team has closed the issue with the following comment:

> *Even if shares are minted on another chain, funds cannot be sent on that chain.*

BOV-10	New Deposits Reset All User Share Lock Times		
Asset	`TellerWithMultiAssetSupport.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `_afterPublicDeposit()` function resets the entire user's share unlock time on every deposit when `currentShareLockPeriod > 0`, creating a denial-of-service condition. Users who make multiple deposits before their shares unlock will have ALL their shares (including previously deposited shares nearing unlock) locked for the full period again.

Every deposit resets the user's global share unlock time, not just for the newly deposited shares:

TellerWithMultiAssetSupport._afterPublicDeposit()

```
function _afterPublicDeposit(
    address user,
    ERC20 depositAsset,
    uint256 depositAmount,
    uint256 shares,
    uint256 currentShareLockPeriod
) internal {
    uint256 nonce = ++depositNonce;
    if (currentShareLockPeriod > 0) {
        // @audit - resets unlock time for user shares, not just new deposit!
        beforeTransferData[user].shareUnlockTime = block.timestamp + currentShareLockPeriod;
        publicDepositHistory[nonce] = keccak256(
            abi.encode(user, depositAsset, depositAmount, shares, block.timestamp, currentShareLockPeriod)
        );
    }
    emit Deposit(nonce, user, address(depositAsset), depositAmount, shares, block.timestamp, currentShareLockPeriod);
}
```

The `beforeTransfer()` hook checks the global unlock time against all user shares:

TellerWithMultiAssetSupport.beforeTransfer()

```
function beforeTransfer(address from, address to, address operator) public view virtual {
    // ...
    if (beforeTransferData[from].shareUnlockTime > block.timestamp) {
        revert TellerWithMultiAssetSupport__SharesAreLocked();
    }
}
```

Users are in a liquidity dilemma - they cannot deposit during favourable market conditions without resetting their share unlock timer, and they cannot withdraw shares nearing unlock if they've recently deposited. This is mitigated by their ability to deposit under a different address, however.

Recommendations

Implement per-deposit or per-batch share locking instead of global user-level locking, allowing previously deposited shares to unlock on their original schedule while only applying lock periods to newly deposited shares.

Resolution

The development team has closed the issue with the following comment:

> *This is intentional.*

BOV-11	Cross-Chain Bridge Bypasses Destination Chain Vault `cap`		
Asset	`CrossChainTellerWithGenericBridge.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

Users can bypass vault caps on destination chains by using the cross-chain bridging functionality. The `_completeMessageReceive()` function mints shares directly without any `cap` validation, allowing users to exceed the intended vault limits on the destination chain through bridging operations.

When bridged shares arrive on the destination chain, `_completeMessageReceive()` mints them without checking the vault cap:

CrossChainTellerWithGenericBridge._completeMessageReceive()

```
function _completeMessageReceive(bytes32 messageId, uint256 message) internal {
    MessageLib.Message memory m = message.uint256ToMessage();

    // @audit - mints shares directly without cap validation!
    vault.enter(address(0), ERC20(address(0)), 0, m.to, m.shareAmount);

    emit MessageReceived(messageId, m.shareAmount, m.to);
}
```

Vault caps become meaningless in multi-chain deployments, as users can always circumvent them through cross-chain bridging, potentially breaking economic assumptions and risk management controls.

Recommendations

Add a way of checking the destination vault's `cap` settings before allowing further shares to be minted.

Resolution

The development team has acknowledged the issue with the following comment:

> *In an effort to keep share bridging as simple as possible, we are choosing to accept this. In the majority of cases, caps are meant to be pre-deposit vaults and lifted when cross-chain is enabled.*

BOV-12	Precision Loss In `_changeDecimals()` Leads To Distorted Share Distribution		
Asset	`AccountantWithRateProviders.sol` `TellerWithMultiAssetSupport.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `_changeDecimals()` function in `AccountantWithRateProviders` uses floor division when converting exchange rates between different decimal precisions, causing systematic precision loss that benefits depositors at the expense of existing shareholders when depositing assets with fewer decimals than the base token:

AccountantWithRateProviders._changeDecimals()

```
function _changeDecimals(uint256 amount, uint8 fromDecimals, uint8 toDecimals) internal pure returns (uint256) {
    if (fromDecimals == toDecimals) {
        return amount;
    } else if (fromDecimals < toDecimals) {
        return amount * 10 ** (toDecimals - fromDecimals);
    } else {
        return amount / 10 ** (fromDecimals - toDecimals); //  @audit - floor division loses precision
    }
}
```

This function is called during `getRateInQuote()` when converting exchange rates:

AccountantWithRateProviders.getRateInQuote()

```
function getRateInQuote(ERC20 quote) public view virtual returns (uint256 rateInQuote) {
    if (address(quote) == address(base)) {
        rateInQuote = accountantState.exchangeRate;
    } else {
        RateProviderData memory data = rateProviderData[quote];
        uint8 quoteDecimals = ERC20(quote).decimals();
        uint256 exchangeRateInQuoteDecimals = _changeDecimals(accountantState.exchangeRate, decimals, quoteDecimals);
        // @audit - rest of function uses truncated exchangeRateInQuoteDecimals
    }
}
```

This directly affects share calculations during deposits:

TellerWithMultiAssetSupport._erc20Deposit()

```
function _deposit(ERC20 depositAsset, uint256 depositAmount, uint256 minimumMint, address from, address to, Asset memory asset)
    internal virtual returns (uint256 shares) {
    // ...
    shares = depositAmount.mulDivDown(ONE_SHARE, accountant.getRateInQuoteSafe(depositAsset));
    // @audit - uses truncated rate from _changeDecimals
}
```

From the above, it can be seen that the share will be higher due to this truncation. This precision loss occurs on every cross-decimal deposit after yield events, creating a systematic value extraction mechanism.

Recommendations

Replace the naive division with `FixedPointMathLib` operations that support controlled rounding direction by rounding exchange rates UP for deposits and DOWN for withdrawals to always favor the vault.

Resolution

The development team has acknowledged the issue with no comment.

BOV-13	Inconsistent Exchange Rate After `postLoss()` Leads To Inflated Platform Fees		
Asset	`AccountantWithRateProviders.sol AccountantWithYieldStreaming.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The loss posting logic leaves the exchange rate state out of sync, causing fees to be calculated from outdated values. This leads to overstated assets and excessive platform fee charges after losses.

When the function `postLoss()` is called, if vesting gains are less than the loss amount, the share price is reduced to reflect the principal loss:

Excerpt from AccountantWithYieldStreaming.postLoss()

```
if (vestingState.vestingGains >= lossAmount) {
    //....
} else {
    uint256 principalLoss = lossAmount - vestingState.vestingGains;

    //wipe out remaining vesting
    vestingState.vestingGains = 0;

    //reduce share price to reflect principal loss
    uint256 currentShares = vault.totalSupply();
    if (currentShares > 0) {
        uint128 cachedSharePrice = vestingState.lastSharePrice;
        vestingState.lastSharePrice =
            uint128((totalAssets() - principalLoss).mulDivDown(ONE_SHARE, currentShares));

        //.....
    }
```

In this process, only `vestingState.lastSharePrice` is updated, while `AccountantState.exchangeRate` remains unchanged. This causes an issue when the high water mark is reset after posting loss. The function `resetHighwaterMark()` forwards the outdated `state.exchangeRate` as both `newExchangeRate` and `currentExchangeRate` to `_calculateFeesOwed()`:

AccountantWithRateProviders.resetHighwaterMark()

```
function resetHighwaterMark() external virtual requiresAuth {
    //.....
    _calculateFeesOwed(state, state.exchangeRate, state.exchangeRate, currentTotalShares, currentTime);
    //.....
}
```

Inside `_calculateFeesOwed()`, the function `_calculatePlatformFee()` is used to compute platform fees. It relies on comparing `newExchangeRate` and `currentExchangeRate` to determine `minimumAssets`:

Excerpt from AccountantWithRateProviders._calculatePlatformFee()

```
// Determine platform fees owned.
if (platformFee > 0) {
    uint256 timeDelta = currentTime - lastUpdateTimestamp;
    uint256 minimumAssets = newExchangeRate > currentExchangeRate
        ? shareSupplyToUse.mulDivDown(currentExchangeRate, ONE_SHARE)
        : shareSupplyToUse.mulDivDown(newExchangeRate, ONE_SHARE);
    uint256 platformFeesAnnual = minimumAssets.mulDivDown(platformFee, 1e4);
    platformFeesOwedInBase = platformFeesAnnual.mulDivDown(timeDelta, 365 days);
}
```

Since both values are equal, `minimumAssets` is calculated using the outdated `state.exchangeRate`. This results in overvalued assets and excessive fee charges.

For example:

- Current exchange rate = 1
- `postLoss()` sets `vestingState.lastSharePrice` = 0.9, but `AccountantState.exchangeRate` remains at 1
- `resetHighwaterMark` is called and passes `state.exchangeRate` (1) to `_calculateFeesOwed`
- `_calculatePlatformFee` evaluates `minimumAssets` at 1 instead of 0.9
- Fees are calculated on an inflated value, charging more than intended

The issue arises because the exchange rate is not updated after posting a loss, leaving `state.exchangeRate` out of sync with `vestingState.lastSharePrice`, or because `state.exchangeRate` is forwarded to `_calculateFeesOwed()` while that value lags behind the `vestingState.lastSharePrice`, or during resetting high watermark, it does not update the exchange rate.

Recommendations

Update the exchange rate after posting a loss.

Resolution

This issue has been addressed in commit ea03631 by performing fee collection on every update to keep variables in sync.

BOV-14	TWAS Calculation Leading To Unrealistic Daily Yield Leading To Revert During Vesting		
Asset	`AccountantWithYieldStreaming.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The TWAS calculation anchors to contract creation time, producing unrealistically low averages during the first or delayed vesting events. This inflates daily yield checks and can cause unintended reverts, disrupting yield distribution.

When the `vestYield()` function is called, it calculates the time-weighted average supply (TWAS) to validate that the daily yield in BPS does not exceed the `maxDeviationYield`:

Excerpt from AccountantWithYieldStreaming.vestYield()

```
// Use TWAS to validate the yield amount:
uint256 averageSupply = _getTWAS();
uint256 _totalAssets = averageSupply.mulDivDown(vestingState.lastSharePrice, ONE_SHARE);
uint256 dailyYieldAmount = yieldAmount.mulDivDown(1 days, duration);
uint256 dailyYieldBps = dailyYieldAmount.mulDivDown(10_000, _totalAssets);

if (dailyYieldBps > maxDeviationYield) {
    // maxDeviationYield is in bps
    revert AccountantWithYieldStreaming__MaxDeviationYieldExceeded();
}
```

The `_getTWAS()` function calculates the TWAS based on the difference in cumulative supply and the time elapsed since `startVestingTime`:

AccountantWithYieldStreaming._getTWAS()

```
function _getTWAS() internal view returns (uint256) {
    //handle first yield event
    if (supplyObservation.cumulativeSupply == 0) {
        return vault.totalSupply();
    }

    uint64 timeSinceLastVest = uint64(block.timestamp) - vestingState.startVestingTime;

    if (timeSinceLastVest == 0) {
        return vault.totalSupply(); // If no time passed, return current supply
    }

    // TWAS = (current cumulative - last vest cumulative) / time elapsed
    uint256 cumulativeDelta = supplyObservation.cumulativeSupply - supplyObservation.cumulativeSupplyLast;
    return cumulativeDelta / timeSinceLastVest;
}
```

When a deposit is made and `vestYield()` is called for the first time, `timeSinceLastVest` can be very large because `vestingState.startVestingTime` is set to the contract creation time. This results in an artificially low `averageSupply`, which in turn inflates `dailyYieldBps`. If `dailyYieldBps > maxDeviationYield`, the function will revert unexpectedly.

The root cause is that TWAS is being calculated from the contract creation time, rather than from the first deposit (or first share mint). This can produce unrealistic TWAS values at the beginning of vesting.

In summary, after some time has passed since the contract's creation, a first vesting may trigger a revert due to an inflated `dailyYieldBps` that exceeds `maxDeviationYield`. This issue may also arise whenever a new vesting period starts following the completion of a previous one, the greater the time gap between the end of the prior vesting and the start of the new vesting, the higher the risk, resulting in the same potential for reversion.

Recommendations

Calculate `timeSinceLastVest` based on the timestamp of the first deposit (or first share minted) instead of the contract creation time. This will produce a more accurate average supply and prevent false `maxDeviationYield` reverts.

Resolution

This issue has been addressed in commit 14365fb by implementing a new function only callable by the `Teller` that sets the timestamp to that of the first deposit.

BOV-15	Inconsistent Fee Collection Between `_updateExchangeRate()` And `resetHighwaterMark()`		
Asset	`AccountantWithRateProviders.sol` `AccountantWithYieldStreaming.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Fee collection behaviour is inconsistent: `_updateExchangeRate()` skips fees when no yield vests, while `resetHighwaterMark()` charges fees regardless. This discrepancy can lead to unjustified fee accruals without corresponding gains.

In `_updateExchangeRate()`, fees are collected only if `newlyVested > 0`, meaning that new yield has vested. In that case, `_collectFees()` is invoked:

AccountantWithYieldStreaming._updateExchangeRate()

```
function _updateExchangeRate() internal {
    _updateCumulative();

    // Calculate how much has vested since `lastVestingUpdate`
    uint256 newlyVested = getPendingVestingGains();

    uint256 currentShares = vault.totalSupply();
    if (newlyVested > 0) {
        // Update the share price without reincluding pending gains
        uint256 _totalAssets = uint256(vestingState.lastSharePrice).mulDivDown(currentShares, ONE_SHARE);
        vestingState.lastSharePrice = uint128((_totalAssets + newlyVested).mulDivDown(ONE_SHARE, currentShares));

        _collectFees();

        // Move vested amount from pending to realized
        vestingState.vestingGains -= uint128(newlyVested);
        vestingState.lastVestingUpdate = uint128(block.timestamp);
    }

    AccountantState storage state = accountantState;
    state.totalSharesLastUpdate = uint128(currentShares);

    emit ExchangeRateUpdated(vestingState.lastSharePrice);
}
```

By contrast, when `resetHighwaterMark()` is called, `_calculateFeesOwed()` is executed, which collects fees regardless of whether new yield has vested:

AccountantWithRateProviders.resetHighwaterMark()

```
function resetHighwaterMark() external virtual requiresAuth {
    AccountantState storage state = accountantState;

    if (state.exchangeRate > state.highwaterMark) {
        revert AccountantWithRateProviders__ExchangeRateAboveHighwaterMark();
    }

    uint64 currentTime = uint64(block.timestamp);
    uint256 currentTotalShares = vault.totalSupply();
    _calculateFeesOwed(state, state.exchangeRate, state.exchangeRate, currentTotalShares, currentTime);
    state.totalSharesLastUpdate = uint128(currentTotalShares);
    state.highwaterMark = accountantState.exchangeRate;
    state.lastUpdateTimestamp = currentTime;

    emit HighwaterMarkReset();
}
```

This creates an inconsistency as `_updateExchangeRate()` skips fee collection when no new vesting occurs, while `resetHighwaterMark()` collects fees unconditionally. As a result, fees may be charged even when no additional yield has accrued.

Recommendations

Align the behavior of `resetHighwaterMark()` with `_updateExchangeRate()` by collecting fees only when new vesting has occurred.

Resolution

This issue has been addressed in commit ea03631 by performing fee collection on every update to keep variables in sync.

BOV-16	Missing `beforeTransfer()` Check On Deposits Allows Denied Addresses to Mint Shares
Asset	`TellerWithYieldStreaming.sol`
Status	**Closed:** See Resolution
Rating	Severity: Low Impact: Low Likelihood: Low

Description

The deny-list check is inconsistently applied: withdrawals trigger `beforeTransfer()`, but deposits do not. This allows blocked addresses to bypass restrictions by minting shares directly, undermining the access control mechanism.

When withdrawing, the function `beforeTransfer()` is used to check whether the `to` address is denied:

TellerWithYieldStreaming.withdraw()

```
function withdraw(ERC20 withdrawAsset, uint256 shareAmount, uint256 minimumAssets, address to)
    external
    override
    requiresAuth
    returns (uint256 assetsOut)
{
    // update vested yield before withdraw
    _getAccountant().updateExchangeRate();
    beforeTransfer(msg.sender, address(0), msg.sender);
    assetsOut = _withdraw(withdrawAsset, shareAmount, minimumAssets, to);

    emit Withdraw(address(withdrawAsset), shareAmount);
}
```

However, this check is not applied during deposits, when shares are minted to the `to` address.
For example, if `beforeTransferData[Bob].denyTo = true`, Bob is blocked from receiving shares via transfer.
But Bob can still call `deposit()` and have shares minted to his account. Minting should be treated as a transfer, just as burning is treated as a transfer (which already has the `beforeTransfer()` hook in withdrawals).

Recommendations

Deposits should also include the `beforeTransfer()` hook, consistent with the withdrawal mechanism.

Resolution

The development team has closed the issue with the following comment:

> *If blocked on withdraws, blocked users donate to the vault if they choose to deposit.*

BOV-17	Cross-Chain Deposits Bypass Destination Chain Share Lock Periods		
Asset	`LayerZeroTeller.sol CrossChainTellerWithGenericBridge.sol TellerWithMultiAssetSupport.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Users can bypass share lock periods on destination chains by using `depositAndBridge()` instead of direct deposits. The `_completeMessageReceive()` function mints shares directly without applying the destination chain's share lock period, while direct deposits properly enforce locks through `_afterPublicDeposit()`:

CrossChainTellerWithGenericBridge._completeMessageReceive()

```
function _completeMessageReceive(bytes32 messageId, uint256 message) internal {
    MessageLib.Message memory m = message.uint256ToMessage();

    // @audit - shares minted directly - NO share lock applied!
    vault.enter(address(0), ERC20(address(0)), 0, m.to, m.shareAmount);

    emit MessageReceived(messageId, m.shareAmount, m.to);
}
```

In contrast, direct deposits on the destination chain correctly enforce share lock periods:

TellerWithMultiAssetSupport.deposit()

```
function deposit(ERC20 depositAsset, uint256 depositAmount, uint256 minimumMint)
    external payable ... returns (uint256 shares) {
    Asset memory asset = _beforeDeposit(depositAsset);
    // ...
    shares = _erc20Deposit(depositAsset, depositAmount, minimumMint, msg.sender, msg.sender, asset);

    // @audit - share lock is properly applied here
    _afterPublicDeposit(msg.sender, depositAsset, depositAmount, shares, shareLockPeriod);
}
```

The above behaviour creates unfair advantages for cross-chain users over direct depositors.

Recommendations

Call `_afterPublicDeposit()` in `_completeMessageReceive()` with the destination chain's `shareLockPeriod` to ensure consistent share lock enforcement between direct and bridged deposits.

Resolution

The development team has closed the issue with the following comment:

> *`depositAndBridge` is only enabled when `shareLock` is 0. This is true for both source and destination chains.*

BOV-18	Zero `shareLockPeriod` Prevents Immediate Withdrawals And Refunds		
Asset	`TellerWithMultiAssetSupport.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The authority can change the `shareLockPeriod` to zero, which allows immediate withdrawal after a deposit.

TellerWithMultiAssetSupport.setShareLockPeriod()

```
function setShareLockPeriod(uint64 _shareLockPeriod) external requiresAuth {
    if (_shareLockPeriod > MAX_SHARE_LOCK_PERIOD) revert TellerWithMultiAssetSupport__ShareLockPeriodTooLong();
    shareLockPeriod = _shareLockPeriod;
}
```

If set to zero, a deposit made after this change cannot be withdrawn immediately and also cannot be refunded by `STRATEGIST_MULTISIG_ROLE` using `refundDeposit` in a scenario.

For example, if the `shareLockPeriod` is 5 days and Alice deposits, then `beforeTransferData[Alice].shareUnlockTime` will be equal to `block.timestamp + 5 days`. After 1 day, the authority sets `shareLockPeriod` to zero. Alice deposits again, but since the period is zero, the following code does not execute, leaving her second deposit without a recorded public deposit history:

Excerpt from TellerWithMultiAssetSupport._afterPublicDeposit()

```
if (currentShareLockPeriod > 0) {
    beforeTransferData[user].shareUnlockTime = block.timestamp + currentShareLockPeriod;
    publicDepositHistory[nonce] = keccak256(
        abi.encode(user, depositAsset, depositAmount, shares, block.timestamp, currentShareLockPeriod)
    );
}
```

As a result, Alice cannot withdraw her second deposit immediately, despite `shareLockPeriod` being zero, because the following check in `beforeTransfer()` will use the entry from 5 days ago:

Excerpt from TellerWithMultiAssetSupport.beforeTransfer()

```
if (beforeTransferData[from].shareUnlockTime > block.timestamp) {
    revert TellerWithMultiAssetSupport__SharesAreLocked();
}
```

In addition, the `STRATEGIST_MULTISIG_ROLE` cannot refund it, as the logic requires a matching `publicDepositHistory`. So Alice must wait until the `shareUnlockTime` for her first deposit has passed.

Excerpt from TellerWithMultiAssetSupport.refundDeposit()

```
bytes32 depositHash = keccak256(
    abi.encode(
        receiver, depositAsset, depositAmount, shareAmount, depositTimestamp, shareLockUpPeriodAtTimeOfDeposit
    )
);
if (publicDepositHistory[nonce] != depositHash) revert TellerWithMultiAssetSupport__BadDepositHash();
```

The root cause is that deposits made when `shareLockPeriod` is zero do not create a public deposit history entry, breaking withdrawal and refund logic.

Recommendations

Update the code so that `shareUnlockTime` is tracked for each individual deposit and ensure that a public deposit history is created even when `shareLockPeriod` is set to zero.

Resolution

The development team has acknowledged the issue with no comment.

BOV-19	Missing Exchange Rate Update Before Active Vest Check In `vestYield()`		
Asset	`AccountantWithYieldStreaming.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Updating the exchange rate before enforcing the delay lets callers clear all pending vesting and sidestep `minimumUpdateDelayInSeconds`. This effectively disables the intended rate-limit on strategic updates, enabling timing/fee manipulation and policy bypass.

In the `vestYield()` function, the current logic first checks for an active vest. If a vest exists, it enforces a minimum delay since the last strategic update before updating the exchange rate:

AccountantWithYieldStreaming.vestYield()

```
// Only check if there's an active vest
if (vestingState.vestingGains > 0) {
    if (block.timestamp < lastStrategistUpdateTimestamp + accountantState.minimumUpdateDelayInSeconds) {
        revert AccountantWithYieldStreaming__NotEnoughTimePassed();
    }
}

// Update the exchange rate, then validate if everything checks out
_updateExchangeRate();
```

However, during the exchange rate update, `getPendingVestingGains()` is called to calculate the remaining vested yield. If this amount equals `vestingState.vestingGains`, then all pending vesting is cleared:

AccountantWithYieldStreaming._updateExchangeRate()

```
function _updateExchangeRate() internal {
    _updateCumulative();

    // Calculate how much has vested since `lastVestingUpdate`
    uint256 newlyVested = getPendingVestingGains();

    uint256 currentShares = vault.totalSupply();
    if (newlyVested > 0) {
        // Update the share price without reincluding pending gains
        uint256 _totalAssets = uint256(vestingState.lastSharePrice).mulDivDown(currentShares, ONE_SHARE);
        vestingState.lastSharePrice = uint128((_totalAssets + newlyVested).mulDivDown(ONE_SHARE, currentShares));

        _collectFees();

        // Move vested amount from pending to realized
        vestingState.vestingGains -= uint128(newlyVested);
        vestingState.lastVestingUpdate = uint128(block.timestamp);
    }

    AccountantState storage state = accountantState;
    state.totalSharesLastUpdate = uint128(currentShares);

    emit ExchangeRateUpdated(vestingState.lastSharePrice);
}
```

This implies that if the exchange rate is updated first in the function `vestYield()`, all vesting yield may be released, for example, when the current time is past the vesting end. In this case, the delay requirement between strategic updates is passed, as it only applies when there is a non-zero vested balance remaining.

Recommendations

Move the call to `_updateExchangeRate()` before the active vest check in the function `vestYield()`.

Resolution

The development team has acknowledged the issue with the following comment:

> *In practice the mininum update delay is kept* `<=` *the minimum stream duration, ensuring that if there are gains left, we check this block. If no gains are left, the stratgeist should be able to post an update.*

BOV-20	`GenericRateProvider` Does Not Provide Ways To Utilize Oracle Rate Validations
Asset	`GenericRateProvider.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `GenericRateProvider` contract has a fundamental limitation in its price feed decoding logic, which could lead to an inability to utilize price feed validation logic. The current implementation only supports functions that return a single `int256` or `uint256` value, preventing the use of recommended oracle patterns that include staleness protection.

GenericRateProvider.getRate()

```
function getRate() public virtual view returns (uint256) {
    bytes memory result = target.functionStaticCall(callData);

    // ...

    if (signed) {
        // @audit Only supports single int256 return value
        int256 res = abi.decode(result, (int256));
        if (res < 0) revert GenericRateProvider__PriceCannotBeLtZero();
        return uint256(res);
    } else {
        // @audit Only supports single uint256 return value
        return abi.decode(result, (uint256));
    }
}
```

For Chainlink price feeds, this limitation forces users to use the deprecated `latestAnswer()` function:

Chainlink Oracle - Deprecated Function

```
// Deprecated function - lacks freshness checks
function latestAnswer() external view returns (int256);
```

Instead of the recommended latestRoundData() function:

Chainlink Oracle - Recommended Function

```
// @audit Recommended function with staleness protection
function latestRoundData() external view returns (
    uint80 roundId,
    int256 answer,        // The actual price data
    uint256 startedAt,
    uint256 updatedAt,    // Critical for freshness validation
    uint80 answeredInRound
);
```

By defaulting to the use of `latestAnswer()`, the system loses critical protection against:

- Stale price data: No way to verify when the price was last updated
- Oracle downtime: No mechanism to detect if the oracle has stopped updating
- Invalid rounds: No access to round metadata for validation

Recommendations

Extend `GenericRateProvider` to support multiple return values and implement proper staleness checks, or create specialized oracle adapters that handle the full recommended oracle interface including freshness validation.

Resolution

The development team has acknowledged the issue with no comment.

BOV-21	Unwanted Reverts In `depositAndBridge()` And `depositAndBridgeWithPermit()` If Vault `cap` Is Set
Asset	`CrossChainTellerWithGenericBridge.sol TellerWithMultiAssetSupport.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `depositAndBridge()` and `depositAndBridgeWithPermit()` functions will revert when the vault `cap` is set and a deposit would temporarily exceed it, even though the shares are immediately burned for bridging. The `cap` check occurs during share minting, but doesn't account for the immediate burning that follows.

Both functions call `_erc20Deposit()` which enforces the vault cap:

CrossChainTellerWithGenericBridge.depositAndBridge()

```
function depositAndBridge(...) external payable requiresAuth nonReentrant ... returns (uint256 sharesBridged) {
    Asset memory asset = _beforeDeposit(depositAsset);

    // @audit - cap check happens here - doesn't consider immediate burning
    sharesBridged = _erc20Deposit(depositAsset, depositAmount, minimumMint, msg.sender, msg.sender, asset);

    // ...

    // @audit - shares are immediately burned here
    _bridge(uint96(sharesBridged), to, bridgeWildCard, feeToken, maxFee);
}
```

The `_erc20Deposit()` function checks the `cap` against the temporary total:

TellerWithMultiAssetSupport._erc20Deposit()

```
function _erc20Deposit(...) internal virtual returns (uint256 shares) {
    uint112 cap = depositCap;
    shares = depositAmount.mulDivDown(ONE_SHARE, accountant.getRateInQuoteSafe(depositAsset));

    // ...

    if (cap != type(uint112).max) {
        // @audit - reverts even though shares will be immediately burned!
        if (shares + vault.totalSupply() > cap) revert TellerWithMultiAssetSupport__DepositExceedsCap();
    }

    vault.enter(from, depositAsset, depositAmount, to, shares); // Temporary mint
}
```

Recommendations

Modify the `cap` check logic to account for bridge operations where shares are immediately burned, or bypass the `cap` check entirely for `depositAndBridge()` functions since they don't permanently increase the vault's total supply.

Resolution

The development team has acknowledged the issue with the following comment:

This is intended behavior. If a cap is set we do not want to mint shares, regardless of where they are going.

BOV-22	Missing `isPaused` Checks In `pause()` And `unpause()` Could Lead To Offchain Inconsistencies
Asset	`AccountantWithRateProviders.sol` `TellerWithMultiAssetSupport.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

When the `isPaused` state variable is being set in both `AccountantWithRateProviders` and `TellerWithMultiAssetSupport`, there is no check that a change from the current contract state is taking place.

AccountantWithRateProviders.pause() and unpause()

```
/**
 * @notice Pause this contract, which prevents future calls to `updateExchangeRate`, and any safe rate
 *         calls will revert.
 * @dev Callable by MULTISIG_ROLE.
 */
function pause() external requiresAuth {
    accountantState.isPaused = true;
    emit Paused();
}

/**
 * @notice Unpause this contract, which allows future calls to `updateExchangeRate`, and any safe rate
 *         calls will stop reverting.
 * @dev Callable by MULTISIG_ROLE.
 */
function unpause() external requiresAuth {
    accountantState.isPaused = false;
    emit Unpaused();
}
```

Even though this does not present a direct issue for the onchain functionality of the contract, the events are still emitted, which may lead to misleading state updates in offchain components, resulting in bad UI/UX.

Recommendations

Check the state of `isPaused` and either revert or return without event emission if a duplicate state change is invoked.

Resolution

The development team has acknowledged the issue with no comment.

BOV-23	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Highwater Mark Assignment Uses Storage Access Instead Of Cached State Reference**

   Related Asset(s): AccountantWithRateProviders.sol

   The `AccountantWithRateProviders` contract creates the `state` storage reference but omits to use it at one point.

   AccountantWithRateProviders.resetHighwaterMark()

   ```
   function resetHighwaterMark() external requiresAuth {
       AccountantState storage state = accountantState; // @audit storage reference
       // ... validation checks ...

       _calculateFeesOwed(state, state.exchangeRate, state.exchangeRate, currentTotalShares, currentTime);

       state.totalSharesLastUpdate = uint128(currentTotalShares);
       // @audit not using reference
       state.highwaterMark = accountantState.exchangeRate;
       state.lastUpdateTimestamp = currentTime;
   }
   ```

   Replace `accountantState.exchangeRate` with `state.exchangeRate` to use the storage reference:

   ```
   state.totalSharesLastUpdate = uint128(currentTotalShares);
   // @audit Modified as per recommendation
   state.highwaterMark = state.exchangeRate;
   state.lastUpdateTimestamp = currentTime;
   ```

2. **Code Duplication Between LayerZero Teller Implementations Could Be Extracted To A Base Contract**

   Related Asset(s): LayerZeroTeller.sol

   The `LayerZeroTeller` and `LayerZeroTellerWithRateLimiting` contracts contain extensive code duplication that could be extracted to a base contract to improve maintainability and reduce the risk of inconsistencies. The vast majority of their functionality is identical, differing only in rate-limiting checks within two functions.

Function Duplications

```
// @audit Same struct, state variables, errors, events, and immutables
struct Chain { bool allowMessagesFrom; bool allowMessagesTo; uint128 messageGasLimit; }
mapping(uint32 => Chain) public idToChains;
address internal immutable lzToken;

// @audit All admin functions are identical across both contracts
function addChain(uint32 chainId, bool allowMessagesFrom, bool allowMessagesTo, address targetTeller, uint128
    ↪ messageGasLimit) external requiresAuth {
    // Identical implementation
}

function removeChain(uint32 chainId) external requiresAuth {
    // @audit Identical implementation
}

function allowMessagesFromChain(uint32 chainId, address targetTeller) external requiresAuth {
    // @audit Identical implementation
}

function allowMessagesToChain(uint32 chainId, address targetTeller, uint128 messageGasLimit) external requiresAuth {
    // @audit Identical implementation
}

function stopMessagesFromChain(uint32 chainId) external requiresAuth {
    // @audit Identical implementation
}

function stopMessagesToChain(uint32 chainId) external requiresAuth {
    // @audit Identical implementation
}

function setChainGasLimit(uint32 chainId, uint128 messageGasLimit) external requiresAuth {
    // @audit Identical implementation
}

function _previewFee(uint256 message, bytes calldata bridgeWildCard, ERC20 feeToken) internal view override returns (uint256
    ↪ fee) {
    // @audit Completely identical implementation including all validation logic
}
```

Over 90% of the code is duplicated across both contracts, creating a maintenance burden and increasing the risk of bugs when updates are applied inconsistently. Any changes to core LayerZero functionality must be replicated across both contracts.

Create a `BaseLayerZeroTeller` abstract contract containing all shared functionality (chain management, fee handling, core LayerZero operations), then have `LayerZeroTellerWithRateLimiting` inherit from it and override only `_lzReceive()` and `_sendMessage()` to add rate limiting checks.

3. **Misleading Comment For `lastUpdateTimestamp` Field**

   Related Asset(s): AccountantWithRateProviders.sol

   The comment `@param lastUpdateTimestamp the block timestamp of the last exchange rate update` is somewhat inaccurate and potentially misleading, since `AccountantState.lastUpdateTimestamp` is also updated during calls to `resetHighwaterMark()`, even though that function does not modify the exchange rate.

AccountantState
```
 * @param lastUpdateTimestamp the block timestamp of the last exchange rate update
 * @param isPaused whether or not this contract is paused
 * @param minimumUpdateDelayInSeconds the minimum amount of time that must pass between
 *       exchange rate updates, such that the update won't trigger the contract to be paused
 * @param platformFee the platform fee
 * @param performanceFee the performance fee
 */
struct AccountantState {
    address payoutAddress;
    uint96 highwaterMark;
    uint128 feesOwedInBase;
    uint128 totalSharesLastUpdate;
    uint96 exchangeRate;
    uint16 allowedExchangeRateChangeUpper;
    uint16 allowedExchangeRateChangeLower;
    uint64 lastUpdateTimestamp;
    bool isPaused;
    uint24 minimumUpdateDelayInSeconds;
    uint16 platformFee;
    uint16 performanceFee;
}
```

Revise the comment to more accurately reflect the behavior.

4. **Misleading Comment When Initializing `lastStrategistUpdateTimestamp`**

   Related Asset(s): AccountantWithYieldStreaming.sol

   The comment does not match the actual implementation. This does not affect the protocol's functionality, as the function `vestYield()` already checks `vestingState.vestingGains > 0`.

   ```
   //initialize strategist update time to 0 so first posts are valid
   lastStrategistUpdateTimestamp = uint64(block.timestamp);
   ```

   Update the comment to accurately reflect the code behaviour, or adjust the initialisation of `lastStrategistUpdateTimestamp` to zero.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team's responses to the raised issues above are as follows:

1. **Highwater Mark Assignment Uses Storage Access Instead Of Cached State Reference**

   No changes were made.

2. **Code Duplication Between LayerZero Teller Implementations Could Be Extracted To A Base Contract**

   No changes were made.

3. **Misleading Comment For `lastUpdateTimestamp` Field**

   No changes were made.

4. **Misleading Comment When Initializing `lastStrategistUpdateTimestamp`**

   The misleading comment in `AccountantWithYieldStreaming` was fixed in 03b2204.

BOV-24	Restricted Users Can Still Deposit Into The Vault		
Asset	`TellerWithMultiAssetSupport.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Users on the deny list are still able to deposit into the vault.

In the function `_erc20Deposit()`, there is no enforcement to restrict deposits to only allowed users:

TellerWithMultiAssetSupport._erc20Deposit()

```
function _erc20Deposit(
    ERC20 depositAsset,
    uint256 depositAmount,
    uint256 minimumMint,
    address from,
    address to,
    Asset memory asset
) internal virtual returns (uint256 shares) {
    uint112 cap = depositCap;
    if (depositAmount == 0) revert TellerWithMultiAssetSupport__ZeroAssets();
    shares = depositAmount.mulDivDown(ONE_SHARE, accountant.getRateInQuoteSafe(depositAsset));
    shares = asset.sharePremium > 0 ? shares.mulDivDown(1e4 - asset.sharePremium, 1e4) : shares;
    if (shares < minimumMint) revert TellerWithMultiAssetSupport__MinimumMintNotMet();
    if (cap != type(uint112).max) {
        if (shares + vault.totalSupply() > cap) revert TellerWithMultiAssetSupport__DepositExceedsCap();
    }
    vault.enter(from, depositAsset, depositAmount, to, shares);
    _afterDeposit(depositAsset, depositAmount);
}
```

This undermines the deny-list mechanism, as restricted addresses can bypass intended controls and continue participating in the vault.

Recommendations

Ensure that deny-list checks are consistently enforced during the deposit process, similar to transfers and withdrawals.

Resolution

This issue has been addressed in commit c758db1 by preventing deny-listed users from depositing.

BOV-25	Misssing Version Function In The `Teller`
Asset	`LayerZeroTeller.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `Teller` contract does not currently expose a method to identify its deployed version. This makes it difficult to distinguish between different deployments or upgrades.

Recommendations

Introduce a `version()` function in the Teller contract that returns the contract's version number, enabling easier tracking, debugging, and integration management.

Resolution

This issue has been addressed in commit c758db1 by implementing function `version()`.

BOV-26	Missing Referral Address Support In Deposits
Asset	`TellerWithMultiAssetSupport.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The current deposit implementation does not support attaching referral addresses. Without this feature, the protocol cannot natively track referrals.

Recommendations

Extend the deposit logic to include an optional referral address, enabling the system to record and leverage referrals for incentive mechanisms.

Resolution

This issue has been addressed in commit c758db1 by adding a `referral` parameter to the relevant functions.

BOV-27	Rounding Induced Vesting Loss And Exchange Rate Update Edge Cases		
Asset	`AccountantWithYieldStreaming.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The vault's yield vesting and exchange rate update logic can silently lose vested yield due to integer rounding and timestamp updates occurring when no effective share price increase is realised.

Originally, newly vested yield was applied by incrementally increasing the share price using integer division. When the per-share increment rounded down to zero, the system still deducted the corresponding amount from `vestingGains`, effectively burning yield. This scenario is plausible at any supply level when `newlyVested` is small relative to total shares and `_updateExchangeRate()` is called frequently.

An attempted fix that recomputed the share price from total assets suffered from the same rounding behaviour - recomputing and flooring the share price still resulted in no observable increase while vesting gains were consumed.

A subsequent mitigation attempted to only apply vesting when it resulted in an actual share price increase. However, the implementation still updated the vesting timestamp even when no gains were applied. This caused a second-order loss - time elapsed without releasing yield was effectively discarded, reducing future vesting amounts.

To address this, the protocol introduced a *virtual share price* tracked at RAY precision. Yield is now accumulated in this high-precision variable and only materialized into the user-facing share price when converted, significantly reducing rounding loss. Timestamp updates were also restricted to cases where yield is actually posted.

While the virtual share price approach resolves the primary rounding loss issue, two edge cases remain:

1. **Vault empties mid-vest.** If total supply drops to zero during an active vesting period, the virtual share price is not updated. A subsequent depositor could capture the remaining vesting yield.
2. **Precision loss on principal loss recalculation.** In rare scenarios where a principal loss exceeds the buffer, the virtual share price is recomputed using `totalAssets()`, which is derived from the rounded-down real share price rather than the virtual one. This can discard a small amount of accumulated RAY-level precision.

Historically, the issue could lead to permanent loss of vested yield due to rounding and timestamp mismanagement. The current design largely eliminates this risk. The remaining edge cases may result in minor yield misallocation or negligible precision loss in rare and operationally unlikely scenarios.

Recommendations

Implement the following:

- Update virtual share price even when total supply is zero, or explicitly carry forward vesting state until a non-zero supply exists.
- Base loss recalculations directly on the virtual share price to preserve accumulated precision.

Resolution

The finding has been closed with the following comment provided by the development team:

> *"After some internal discussion, we are ok with accepting these 2 edge cases. We are not concerned about the vault emptying completely mid-vest, and we are also not concerned with this level of precision in the rare scenario when postLoss will actually be used."*

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `forge` framework was used to perform these tests and the output is given below.

```
Ran 5 tests for test/SigP_POCs.t.sol:SigP_POCsTest
[PASS] test_SigP_IncorrectFeeCalculationInTotalAssetsLeadsToAssetLoss() (gas: 694468)
[PASS] test_SigP_PrecissionLossDueToChangeDecimalTruncation() (gas: 732215)
[PASS] test_SigP_PrecissionLossDueToRateProviderTruncation() (gas: 729394)
[PASS] test_SigP_RebasingTokenLoss() (gas: 501255)
[PASS] test_SigP_RefundLeadsToAssetLoss() (gas: 613185)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 8.92s (23.71s CPU time)

Ran 1 test suite in 8.92s (8.92s CPU time): 5 tests passed, 0 failed, 0 skipped (5 total tests)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

		Likelihood		
		Low	Medium	High
Impact	High	Medium	High	Critical
Impact	Medium	Low	Medium	High
Impact	Low	Low	Low	Medium

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].