certora

Security Assessment
Final Report

Veda

September 2025

Prepared for Veda Team

certora

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Veda	https://github.com/Veda-Labs/boring-vault/tree/feat/instant-withdraw-accountant	3c768bd (original commit) c758db1 (fix commit)	Solidity

Project Overview

This document describes the manual code review findings of **Veda**. The following contract list is included in our scope:

```
src/base/BoringVault.sol
src/base/Roles/TellerWithMultiAssetSupport.sol
src/base/Roles/TellerWithBuffer.sol
src/base/Roles/TellerWithYieldStreaming.sol
src/base/Roles/AaveV3BufferHelper.sol
src/base/Roles/AccountantWithRateProvider.sol
src/base/Roles/AccountantWithYieldStreaming.sol
src/base/Roles/CrossChain/MessageLib.sol
src/base/Roles/CrossChain/PairwiseRateLimiter.sol
src/base/Roles/CrossChain/CrossChainTellerWithGenericBridge.sol
src/base/Roles/CrossChain/Bridges/LayerZero/LayerZeroTeller.sol
src/base/Roles/CrossChain/Bridges/LayerZero/LayerZeroTellerWithRateLimiting.sol
src/helper/GenericRateProvider.sol
src/helper/GenericRateProviderWithDecimalScaling.sol
```

The work was undertaken from **August 28, 2025,** to **September 14, 2025**. During this time, Certora's security researchers performed a manual audit of all the Solidity contracts and discovered several bugs in the codebase, which are summarized in the subsequent section.

Protocol Overview

Veda is a cross-chain, multi-asset vault protocol designed to manage deposits, yield, and liquidity modularly, leaning on its trusted roles for executing sensitive operations. At its core, it combines its `BoringVault` with extensible user-facing `Teller` contracts, yield integrations with external protocols, cross-chain messaging powered by LayerZero and dynamic exchange rates, following the yield generation.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	1	1	1
Medium	4	4	3
Low	6	6	-
Informational	8	8	3
Total	19	19	7

Severity Matrix

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

certora

Detailed Findings

ID	Title	Severity	Status
H-01	`lastVestingUpdate` is incorrectly updated, leading to premature vesting and possible MEV	High	Fixed
M-01	`updateExchangeRate()` leaves fees unaccrued	Medium	Acknowledged
M-02	postLoss() leaves the exchange rate out of sync with the share price	Medium	Fixed
M-03	Fee calculation mechanism can be exploited to inflate accumulated fees	Medium	Fixed
M-04	First depositor(s) will be unable to fully collect/vest their yield due to the way how TWAS is calculated	Medium	Fixed
L-01	Cross-chain bridge uses a burn-first approach, leaving the token supply in a state of deflation if the bridge has an unexpected delay	Low	Acknowledged
L-02	Rate providers do not handle a 0 price	Low	Acknowledged
L-03	Fee assets are not withdrawn from the underlying yield protocol prior to claiming	Low	Acknowledged
L-04	refundDeposit wrongly assumes that the deposit hash was made with NATIVE	Low	Acknowledged

L-05	Fees are taken directly from share-backed assets	Low	Acknowledged
L-06	Due to the design of the Yield distribution mechanism, malicious actors can exploit it to “sandwich” the yield streams without prior staking	Low	Acknowledged
I-01	getPendingVestingGains contains an unreachable clause	Informational	Fixed
I-02	refundDeposit directly transfers assets to the receiver which could be problematic for block-list tokens	Informational	Acknowledged
I-03	Permit functions are incorrectly handled	Informational	Acknowledged
I-04	Automatic pausing during rate update does not emit an event	Informational	Acknowledged
I-05	Withdraw function for yield Teller does not contain a `nonReentrant` modifier	Informational	Fixed
I-06	Array lengths are not compared in the vault’s `manage` function	Informational	Acknowledged
I-07	Multiple instances of silent downcasts	Informational	Acknowledged
I-08	updateExchangeRate() in Yield Streaming accountant lacks an `isPaused` check	Informational	Fixed

High Severity Issues

H-01. `lastVestingUpdate` is incorrectly updated, leading to premature vesting and possible MEV

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: AccountantWithYieldStreaming.sol	Status: Fixed	

Description:

The `lastVestingUpdate` variable is meant to track the last time that yield was vested, during an active vesting period, in order to know how much time remains until all yield is vested and accounted towards the dynamic exchange rate. However, the contract:

- Initializes this variable in the constructor
- Doesn't set its value when a vesting period begins

This leads to the scenario where a vesting period's starting timestamp is greater than the `lastVestingUpdate` timestamp, which is used to calculate the vested amount, meaning that a great portion of the yield gets instantly unlocked, rather than gradually throughout the vesting period, since the next update to the exchange rate would detect that there is a time delta between `startTimestamp` and the last update:

```JavaScript
    function getPendingVestingGains() public view returns (uint256 amountVested) {
        uint256 currentTime = block.timestamp;

        uint256 timeSinceLastUpdate = currentTime - vestingState.lastVestingUpdate;
        uint256 totalRemainingTime = vestingState.endVestingTime -
vestingState.lastVestingUpdate;

        return amountVested =
uint256(vestingState.vestingGains).mulDivDown(timeSinceLastUpdate, totalRemainingTime);
    }
```

As it can be seen from the code block above, since vestingState.lastVestingUpdate's timestamp is earlier than the start of the vesting period, timeSinceLastUpdate would be a positive delta even in the same block that vestYield was called, leading to the premature vesting.

The function for calculating the vesting gain correctly uses the last timestamp to calculate the new unlocked yield, however, updating the `lastUpdateTimestamp` happens only if there is already a running yield period, not when a new one starts:

JavaScript

```
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 w/o reincluding the pending gains (done 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);
    }
```

This also allows malicious actors to exploit the current design, by sandwiching the `vestYield` call and stealing a portion (or the majority) of the yield within the same block.

Recommendations: Set `lastVestingUpdate` to the current timestamp (block.timestamp) every time `vestYield` is called.

Customer's response: Fixed in daee10b.

Fix Review: Fix confirmed.

Medium Severity Issues

M-01. `updateExchangeRate()` leaves fees unaccrued

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: AccountantWithRateProvider.sol	Status: Acknowledged	

Description:

Inside AccountantWithRateProviders.sol the main updateExchangeRate() function uses an emergency pausing mechanism in case any of the sanity checks for the update delay or the upper/lower bounds is broken:

```
JavaScript
function _beforeUpdateExchangeRate(uint96 newExchangeRate)
        internal
        view
        returns (
            bool shouldPause,
            AccountantState storage state,
            uint64 currentTime,
            uint256 currentExchangeRate,
            uint256 currentTotalShares
        )
    {
        state = accountantState;
        if (state.isPaused) revert AccountantWithRateProviders__Paused();
        currentTime = uint64(block.timestamp);
        currentExchangeRate = state.exchangeRate;
        currentTotalShares = vault.totalSupply();
        shouldPause = currentTime < state.lastUpdateTimestamp +
state.minimumUpdateDelayInSeconds
            || newExchangeRate >
currentExchangeRate.mulDivDown(state.allowedExchangeRateChangeUpper, 1e4)
```

```
            || newExchangeRate <
currentExchangeRate.mulDivDown(state.allowedExchangeRateChangeLower, 1e4);
    }
```

However, even if the sanity checks are broken, the contract still updates the global state as it pauses, which is done intentionally:

JavaScript

```
 function updateExchangeRate(uint96 newExchangeRate) external virtual requiresAuth {
        (
            bool shouldPause,
            AccountantState storage state,
            uint64 currentTime,
            uint256 currentExchangeRate,
            uint256 currentTotalShares
        ) = _beforeUpdateExchangeRate(newExchangeRate);
        if (shouldPause) {
            // Instead of reverting, pause the contract. This way the exchange rate updater
is able to update the exchange rate
            // to a better value, and pause it.
            state.isPaused = true;
        } else {
            _calculateFeesOwed(state, newExchangeRate, currentExchangeRate,
currentTotalShares, currentTime);
        }

        newExchangeRate = _setExchangeRate(newExchangeRate, state);
        state.totalSharesLastUpdate = uint128(currentTotalShares);
        state.lastUpdateTimestamp = currentTime;

        emit ExchangeRateUpdated(uint96(currentExchangeRate), newExchangeRate, currentTime);
    }
```

However, fees accrual is skipped and the `lastUpdateTimestamp` is moved forward, meaning any fees that should have accrued between the last update and the update causing the pause are skipped. Also, updating `lastUpdateTimestamp` means that we will have to wait for the

minimum delay before updating the exchange rate to a correct price, potentially stalling the contract

Recommendations: When the contract gets paused due to some sanity check being broken, do not update any state. When the exchange rate normalises, the fees for the past period can be calculated accordingly

Customer's response: Acknowledged.

M-02. `postLoss()` leaves the exchange rate out of sync with the share price

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: AccountantWithYieldStreaming.sol	Status: Fixed	

Description:

The `postLoss()` function of the `AccountantWithYieldStreaming.sol` is meant to simply update the exchange rate based on the losses and reflect them on either the unvested gains or on the current assets in the system. The function firstly calls `_updateExchangeRate()`, which internally calls `_collectFees()` , which seems correct at first since we want to collect the fees thus far, before reflecting the loss. However, the function also equalizes the parent contract's exchange rate to the share price:

```
JavaScript
   function _collectFees() internal {
        AccountantState storage state = accountantState;
        uint256 currentTotalShares = vault.totalSupply();
        uint64 currentTime = uint64(block.timestamp);

        //calculate fees using function inherited from `AccountantWithRateProviders`
        _calculateFeesOwed(
            state, uint96(vestingState.lastSharePrice), state.exchangeRate,
currentTotalShares, currentTime
        );

                            //@audit the share price we set here is the price without the loss
we post
        state.exchangeRate = uint96(vestingState.lastSharePrice);
        state.totalSharesLastUpdate = uint128(currentTotalShares);
        state.lastUpdateTimestamp = currentTime;
    }
```

Posting the loss and updating the share price happens after the above code is executed, thus it leaves the new share price with a lower value than the exchange rate. This means that next time `_updateExchangeRate()` is invoked, the old exchange rate used to calculate the fees will be stale. This can lead to a scenario where the new exchange rate is higher than the current one, but lower than the one saved before `postLoss`, which means fees will be overcharged with regard to the stale rate instead of the rate with the loss reflected on it, leading to an indirect loss for the user.

Recommendations: After posting a loss, update the parent exchange rate to match the share price

Customer's response: Fixed in ea03631.

Fix Review: Fix confirmed.

M-03. Fee calculation mechanism can be exploited to inflate accumulated fees

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: AccountantWithYieldStreaming.sol	Status: Fixed	

Description: Due to the way of how the current fee calculation mechanism is set up, malicious actors can take advantage of it to artificially inflate the accumulated fees via the YieldStreaming accountant. Since fee calculation is based on either the lastShares or the currentShares, and `state.lastUpdateTimestamp` is updated only during fee collections, certain contract states can be exploited to artificially inflate fees.

During times in which yield hasn't been vested/streamed, the `state.lastUpdateTimestamp` variable isn't updated as well. The next time that a vestYield call takes place, a malicious actor can deposit a large amount of funds in the same block which would update the `state.totalSharesLastUpdate` variable, but not the `state.lastUpdateTimestamp`. Said malicious actor needs to hold the funds for only 1 block, i.e. withdraw in the next one, but the fee accrual mechanism would "count" it as if the supply was present during the whole period where no yield was vested.

Let's take a hypothetical scenario as an example. Currently there is a vault with `10_000e18` token units as both assets and `10_000e18` as supply shares.

No yield has been vested for the last 14 days. A malicious actor deposits a large amount of shares in the vault in the same block that a new vestYield call has occurred (doesn't have to frontrun it). They deposit a large amount of the token, let's say `10_000_000e18`, subsequent operations performed in the same block, for example they can also perform another deposit for 1 wei, would also once again trigger the `_updateExchangeRate` call:

```JavaScript
 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 w/o reincluding the pending gains (done 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);
    }
```

Considering that there's no `newlyVested` funds, the `_collectFees()` call won't be executed, and we would only update `state.totalSharesLastUpdate` which would also include the `10_000_000e18` funds deposited by the malicious actor.

The only thing left to do is for the malicious actor to withdraw the funds in the next block. When the `_updateExchangeRate` takes place prior to the funds being withdrawn, we would have some funds vested as one block has passed. When the `_collectFees()` call gets triggered, we would be using the current shares which include the 10M malicious funds, as well as the last ones which also do:

```javascript
function _calculateFeesOwed(
        AccountantState storage state,
        uint96 newExchangeRate,
        uint256 currentExchangeRate,
        uint256 currentTotalShares,
        uint64 currentTime
    ) internal virtual {
        // Only update fees if we are not paused.
        // Update fee accounting.
        (uint256 newFeesOwedInBase, uint256 shareSupplyToUse) = _calculatePlatformFee(
            state.totalSharesLastUpdate,
            state.lastUpdateTimestamp,
            state.platformFee,
            newExchangeRate,
            currentExchangeRate,
            currentTotalShares,
            currentTime
        );
```

Considering that both supplies amount to 10_010_000e18, that amount will be used to calculate the fees owed using the state.lastUpdateTimestamp which hasn't been updated in 14 days. Considering an exchange rate of 1e18 (for simplicity purposes), and a 3% fee (0.03e4), the contract would accumulate 11_797e18 tokens in fees, even though the 10_000_000e18 funds were held for only one block. This would effectively temporarily DoS the claim fees functionality.

Recommendations: When `vestYield` is called, make sure that fees are calculated even in the case where no vesting is currently being streamed.

Customer's response: Fixed in 294dd0d.

Fix Review: Fix confirmed.

M-04. First depositor(s) will be unable to fully collect/vest their yield due to the way how TWAS is calculated

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: AccountantWithYieldStreaming.sol	Status: Fixed	

Description:

Upon contract deployment, the constructor sets values for multiple variables, including the block.timestamp as the value for all timestamp-based variables.

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

        //initialize supply observations
        supplyObservation.cumulativeSupply = 0;
        supplyObservation.cumulativeSupplyLast = 0;
        supplyObservation.lastUpdateTimestamp = uint128(block.timestamp);

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

This also includes the startVestingTime which usually signals whenever the current or last vesting period has started..

After the contract has been deployed, the difference between the contract deployment timestamp and the first deposit will also go towards the calculation of the `timeSinceLastVest` part of the TWAS calculations, which will dilute the rewards that need to be claimed by the first deposited, i.e. everyone which has deposited until the first `vestYield` call.

Currently TWAS is calculated in the following way:

```JavaScript
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;

    }
```

- If the contract was deployed at timestamp X;
- The first deposit occurred at timestamp Y (X + 200_000);
- And the first vestYield call occurred at timestamp Z (Y + 50_000)

The timeSinceLastVest will be calculated as “now - X”, instead of “now - Y”, this means that the 200_000 seconds (this an example, it could be more or less), will be counted towards the timeSinceLastVest, i.e. timeframe in which the cumulative supply was part of the vault, even though this wasn’t the case. This will further dilute the TWAS and lower the maximum daily yield which could be distributed to the first depositor(s).

Recommendations: One option would be to set the `startVestingTime` to the timestamp of the first deposit (by including a clause which checks this).

Customer's response: Fixed in 14365fb.

Fix Review: Fix confirmed.

Low Severity Issues

L-01. Cross-chain bridge uses a burn-first approach, leaving the token supply in a state of deflation if the bridge has an unexpected delay

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: CrossChainTellerWithGenericBridge.sol	Status: Acknowledged	

Description:

All Cross-chain bridge operations are burn-first, meaning they do an `exit` on the source chain before even sending the message to the destination

```
JavaScript
function _bridge(uint96 shareAmount, address to, bytes calldata bridgeWildCard, ERC20
feeToken, uint256 maxFee)
        internal
    {
        // Since shares are directly burned, call `beforeTransfer` to enforce before transfer
hooks.
        beforeTransfer(msg.sender, address(0), msg.sender);

        // Burn shares from sender
        vault.exit(address(0), ERC20(address(0)), 0, msg.sender, shareAmount);

        // Send the message.
        MessageLib.Message memory m = MessageLib.Message(shareAmount, to);
        // `messageToUnit256` reverts on overflow, eventhough it is not possible to overflow.
        // This was done for future proofing.
        uint256 message = m.messageToUint256();

        bytes32 messageId = _sendMessage(message, bridgeWildCard, feeToken, maxFee);

        emit MessageSent(messageId, shareAmount, to);
    }
```

If there are any channel and message failures or other issues with the bridge, e.g if a message is sent to an allowed `to` chain, but on the destination chain, the source of the message is disallowed, we have a state of global deflation. While it might not cause a great impact to exchange rates and supplies, it is an inconvenience and removes the ability to allow users to refund their bridge operation in case the message takes too long to go through, considering that failing LZ V2 messages need to be manually retried.

Recommendations: Escrow the bridged tokens in the contract and then burn them on a successful receive, either via a secondary message or monitoring of emitted events and a dedicated burner role.

Customer's response: Acknowledged.

L-02. Rate providers do not handle a 0 price

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: GenericRateProvider.sol GenericRateProviderWithDecimalScaling.sol	Status: Acknowledged	

Description:

The generic rate providers are simple contracts meant to return the quote token rates with the base assets of the vault. Under the hood they are made to be compatible with multiple oracles such as Chainlink, Redstone, etc.

```
JavaScript
   function getRate() public virtual view returns (uint256) {
        bytes memory callData = abi.encodeWithSelector(
            selector,
            staticArgument0,
            staticArgument1,
            staticArgument2,
            staticArgument3,
            staticArgument4,
            staticArgument5,
            staticArgument6,
            staticArgument7
        );
        bytes memory result = target.functionStaticCall(callData);

        if (signed) {
            //if target func() returns an int, we get the result and then cast it to a
uint256
            int256 res = abi.decode(result, (int256));
            if (res < 0) revert GenericRateProvider__PriceCannotBeLtZero();

            return uint256(res);

        } else {
```

```
            return abi.decode(result, (uint256));
        }
```

The contracts properly manage the possible negative rate value returns from integer calls, however they do not consider the possible 0 price return. As the rates are used when quoting from the accountant contract, calculations will easily get skewed or lead to a revert due to division by 0

Recommendations: Check for the 0 price, reject it and handle appropriately in the calling contract if it needs some return value, like a backup last price or a secondary oracle call

Customer's response: Acknowledged.

L-03. Fee assets are not withdrawn from the underlying yield protocol prior to claiming

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: AccountantWithRateProviders.sol	Status: Acknowledged	

Description:

The base `AccountantWithRateProviders.sol` contract is the accountant containing the fee claiming logic and can work as a standalone accountant to any vault. However, for the underlying `Teller.sol` contracts that integrate with it, some use buffers to deposit the incoming assets into external yield protocols in order to earn and thus, impact the exchange rate.

Fees are calculated on every rate update and are stored as a variable, which is emptied when `claimFees()` gets called, which simply sends the asset to the designated address

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

        AccountantState storage state = accountantState;
        if (state.isPaused) revert AccountantWithRateProviders__Paused();
        if (state.feesOwedInBase == 0) revert AccountantWithRateProviders__ZeroFeesOwed();

        // Determine amount of fees owed in feeAsset.
        uint256 feesOwedInFeeAsset;
        RateProviderData memory data = rateProviderData[feeAsset];
        if (address(feeAsset) == address(base)) {
            feesOwedInFeeAsset = state.feesOwedInBase;
        } else {
            uint8 feeAssetDecimals = ERC20(feeAsset).decimals();
            uint256 feesOwedInBaseUsingFeeAssetDecimals =
                _changeDecimals(state.feesOwedInBase, decimals, feeAssetDecimals);
            if (data.isPeggedToBase) {
                feesOwedInFeeAsset = feesOwedInBaseUsingFeeAssetDecimals;
```

```
            } else {
                uint256 rate = data.rateProvider.getRate();
                feesOwedInFeeAsset = feesOwedInBaseUsingFeeAssetDecimals.mulDivDown(10 **
feeAssetDecimals, rate);
            }
        }
        // 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 problem here for the tellers with buffers is that the vault, which is the sender of the `claimFees()` call via a `manage()` call, does not hold any of the funds, which are still generating yield somewhere. The contract also does not withdraw the needed assets from those yield protocols, thus every attempt to claim fees would fail, unless they are manually withdrawn via a `manage()` call, done beforehand.

Recommendations: Implement asset withdrawal from the underlying buffer's yield protocol. If the intention is to have this function for vaults that keep the funds, introduce a secondary counterpart function that deals with buffers atomically, as having 2 separate operations for claiming is error prone.

Customer's response: Acknowledged.

L-04. `refundDeposit` wrongly assumes that the deposit hash was made with NATIVE

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: TellerWithMultiAssetSupport.sol	Status: Acknowledged	

Description:

Whenever users make a deposit to a Teller which has some `shareLockPeriod` those deposits are eligible for a refund within that period. This is done by "registering" the deposits as part of the `publicDepositHistory` mapping as part of the `_afterPublicDeposit` function.

The problem is, when it comes to native deposits, the `depositAsset` is set to the nativeWrapper, and then passed to the `_afterPublicDeposit` function, and is registered with the depositAsset set to the nativeWrapper as part of the `publicDepositHistory.`

JavaScript

```
 if (address(depositAsset) == NATIVE) {
            if (msg.value == 0) revert TellerWithMultiAssetSupport__ZeroAssets();
            nativeWrapper.deposit{value: msg.value}();
            // Set depositAmount to msg.value.
            depositAmount = msg.value;
            nativeWrapper.safeApprove(address(vault), depositAmount);
            // Update depositAsset to nativeWrapper.
            depositAsset = nativeWrapper;
            // Set from to this address since user transferred value.
            from = address(this);

  } else {
            if (msg.value > 0) revert TellerWithMultiAssetSupport__DualDeposit();
            from = msg.sender;
        }
        _afterPublicDeposit(msg.sender, depositAsset, depositAmount, shares,
shareLockPeriod);
    }
```

But when a deposit needs to be refunded via the `refundDeposit` function, the function flow assumes that the depositAsset will be provided as NATIVE, but if that's the case, the function would fail as such a depositHash won't be discovered since it was made with nativeWrapper as the depositAsset.

JavaScript

```
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);
    }
```

Taking the above into consideration, the line which determines if the depositAsset is NATIVE and switches it to the nativeWrapper if it is, is obsolete.

Recommendations:

Either remove the above-mentioned line or move it above the `publicDepositHistory` hash checks.

Customer's response: Acknowledged.

L-05. Fees are taken directly from share-backed assets

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: AccountantWithYieldStreaming.sol AccountantWithRateProvider.sol	Status: Acknowledged	

Description:

Currently, the protocol has fees disabled on all of their live instances, however fees are meant to be collected based on positive incoming new exchange rates and are taken as a percentage of the incoming tokens, to uniformly distribute them among the depositors instead of charging a flat fee:

```
JavaScript
function _calculatePlatformFee(
        uint128 totalSharesLastUpdate,
        uint64 lastUpdateTimestamp,
        uint16 platformFee,
        uint96 newExchangeRate,
        uint256 currentExchangeRate,
        uint256 currentTotalShares,
        uint64 currentTime
    ) internal view returns (uint256 platformFeesOwedInBase, uint256 shareSupplyToUse) {
        shareSupplyToUse = currentTotalShares;
        // Use the minimum between current total supply and total supply for last update.
        if (totalSharesLastUpdate < shareSupplyToUse) {
            shareSupplyToUse = totalSharesLastUpdate;
        }

        // 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);
        }
    }
 state.lastUpdateTimestamp = currentTime;
    }
```

As it can be seen, the fee is taken as the time delta of the yearly % fee on the incoming yield. This is true for both the rate provider and the yield streaming contracts. For e.g, if we have a 100 assets yield, those 100 assets are accounted for towards the exchangeRate and then the fee is taken from them.

This means that the exchange rate is changed with regards to 100 assets, even though 2 of them could have left, meaning there are only 98 assets backing the inflated exchange rate:

JavaScript

```
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 w/o reincluding the pending gains (done in
`newlyVested`)
            uint256 _totalAssets =
uint256(vestingState.lastSharePrice).mulDivDown(currentShares, ONE_SHARE);
            //@audit the share price gets updated and then fees are taken
            //This new share price includes the fee assets, but they will be withdrawnn
            //without burning shares
            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);
    }
```

This would leave the last person to redeem their shares unable to do so, as their shares will attempt to access those assets that were taken as a fee, for example:

1. Bob and Alice both deposit 1000e6 USDC at 1:1 for 1000 shares each, assume timestamp 0
2. 100 seconds later there is 10e6 yield, the exchange rate does not update yet since the yield is not vested yet
3. After 100 more seconds, another exchange rate update is invoked (assume manually for simplicity, the contract allows it) and the exchange rate is updated. Assuming all 10e6 yield got vested, the new exchange rate is calculated as (2000e6 + 10e6) / 2000e6 = 1.005e6 and fee collection is triggered
4. _calculatePlatformFee uses 1.0e6 as the exchange rate since it is the smaller past rate, does 2000e6 * 1 = 2000e6 assets from which to take a fee. 200 seconds have passed since the first deposits, assuming a 4% fee, divided by 365 days in seconds, we get roughly 500 wei in fees.
5. The protocol claims their fees
6. Now both Bob and Alice both hold 1000 shares worth 1005 USDC
7. If both try to redeem, the first one whose transaction gets executed will receive their 1005 USDC, while the other one will revert since the total assets in the protocol are not the 2010e6 USDC needed to cover the shares, but 2010e6 - 500, because 500 went to the protocol as fees.
8. The last withdrawer cannot take their tokens

The counter-scenario to the above is that if all users withdraw their assets, the protocol cannot claim its fees as there are no assets left due to the shares being redeemed. The same is also true if a whale exists the protocol and the total fees were based on their deposit, since now the assets needed to back those fees exited the vault.

Recommendations: Instead of firstly updating the share price and then collecting fees based on the last exchange rate, first collect the fees, then calculate the new exchange rate based not only on the incoming yield but also on the assets that have now been reserved out of the vault for fees. This way the exchange rate will accurately represent the total assets available to the depositors and the fees will be correctly taken from all depositors' yield equally.

Customer's response: Acknowledged.

L-06. Due to the design of the Yield distribution mechanism, malicious actors can exploit it to "sandwich" the yield streams without prior staking

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: AccountantWithYieldStreaming.sol	Status: Acknowledged	

Description:

The yield distribution/streaming mechanism can be exploited by either front-running the vestYield call, or depositing after it, and then withdrawing when it's fully vested. The problem with the current design of the mechanism is that the yield being vested is based on the deposits that were already made and are "staked" for some time.

When the yield which was generated upon those deposits is to be distributed, a malicious actor can "sandwich" it by staking for only 1 to 7 days (depending on the distribution period) and dilute all other liquidity providers / depositors which have likely staked for longer times.

Due to the way that the exchange rate is calculated, the accrued yield is gradually being vested as part of the totalAssets, meaning that users could enter the vault prior or at the beginning of a vesting period, and exit when it's fully vested and dilute all other LPs by taking the majority of the yield that they have accumulated.

Recommendations:

Since this is inherently the design of the architecture, the STRATEGIST needs to constantly vestYield and update it, and for that yield not to have any stepwise jumps so that there's no incentive to be exploited.

Customer's response: Acknowledged.

Informational Issues

I-01. `getPendingVestingGains` contains an unreachable clause

Description:

`getPendingVestingGains()` simply returns the vested yield from the period that is not yet added to the active exchange rate. It uses a simple linear formula, guarded by 2 sanity checks:

- the first one checking if the period is over and returning the entire amount early
- the second one checking if the last update time is past the period end and if the vesting gains are 0, returning 0

The problem here is the second check:

```javascript
 if (currentTime >= vestingState.endVestingTime) {
    return vestingState.vestingGains; // Return ALL remaining unvested gains
}

//if we haven't updated yet or no gains to vest
if (vestingState.lastVestingUpdate >= vestingState.endVestingTime ||
vestingState.vestingGains == 0) {
    return 0;
}
```

If the current time is not greater than the end of the vesting period, this means there is no way for the last vesting update to be past the end of the period, since `currentTime > lastUpdate` always holds.

Recommendations: Remove the unneeded

`vestingState.lastVestingUpdate >= vestingState.endVestingTime` clause.

Customer's response: Fixed in 6a18e76.

Fix Review: Fix confirmed.

I-02. `refundDeposit` directly transfers assets to the receiver which could be problematic for block-list tokens

Description:

The `refundDeposit` function, meant for reimbursing deposits while they are still in their locked period, uses a push mechanism for forcefully burning the user's shares and sending them directly to them:

```javascript
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.
        //@audit we directly send the assets to the address
        vault.exit(receiver, ERC20(depositAsset), depositAmount, receiver, shareAmount);

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

However, the function treats the `receiver` address as the address of the user whose deposit we are refunding and not as a separate address for the funds, as it is used in deriving the `depositHash`. In the case where the asset used for the deposit contains a blocklisting mechanism, a user's deposit might become unrefundable.

Recommendations: Use a pull mechanism for the refund, as in locking the refunded funds in a separate mapping tied to the user address and adding a separate function that allows users to retrieve their assets.

Customer's response: Acknowledged.

I-03. Permit functions are incorrectly handled

Description:

`depositWithPermit` is a function, meant to allow using signatures to execute deposits. However, permits are handled via passing `msg.sender` as the address to send the assets and to receive the shares, making it another entry-point and not an on-behalf-of function. While intentional behavior, the function serves no purpose and can be confusing due to its naming implication

Recommendations: Either remove this unneeded entry point or reimplement it to allow deposit on behalf of somebody else.

Customer's response: Acknowledged.

I-04. Automatic pausing during rate update does not emit an event

Description:

The `AccountantWithRateProvider.sol` is meant to pause itself if it detects that the new exchange rate being updated is breaking a sanity check, intentionally forcing a manual unpause. However, there is no event emitted when pausing, meaning it has to be manually checked.

Recommendations: Emit the necessary pause event when emergency pausing.

Customer's response: Acknowledged.

I-05. Withdraw function for yield Teller does not contain a `nonReentrant` modifier

Description:

The `TellerWithYieldStreaming.sol` is an extension of the previous tellers, which is made to be compatible with the `AccountantWithYieldStreaming.sol` via invoking exchange rate updates on every deposit and withdrawal.

Unlike its base counterpart, this contract's `withdraw` function does not contain the `nonReentrant` modifier:

```
JavaScript
function withdraw(ERC20 withdrawAsset, uint256 shareAmount, uint256 minimumAssets, address
to)
        external
        override
        requiresAuth
        //@audit nonReentant missing
        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);
    }
```

As the contract stands right now, there is no threat as it follows the CEI pattern and the transfer of tokens is the last operation of the transaction. However, it is a good precaution to have against other reentrancy types. It also deviates from the original implementation

Recommendations: Implement the missing modifier

Customer's response: Fixed in f4239bb.

Fix Review: Fix confirmed.

I-06. Array lengths are not compared in the vault's `manage` function

Description:

The `BoringVault#manage()` function has a multi-call counterpart meant to execute multiple ordered external calls. However, the lengths of the passed arrays are not verified to match, potentially reverting without indicating a reason.

JavaScript

```
function manage(address[] calldata targets, bytes[] calldata data, uint256[] calldata values)
        external
        requiresAuth
        returns (bytes[] memory results)
    {
        uint256 targetsLength = targets.length;
        results = new bytes[](targetsLength);
        for (uint256 i; i < targetsLength; ++i) {
            results[i] = targets[i].functionCallWithValue(data[i], values[i]);
        }
    }
```

Recommendations: Implement a simple array length check.

Customer's response: Acknowledged.

I-07. Multiple instances of silent downcasts

Description:

There are multiple instances in the AccountantWithYieldStreaming.sol, as well as AccountantWithRateProviders.sol which downcast uint256 to uint128, as well as uint128 to uint96, without checking if the value surpasses it.

```JavaScript
// AccountantWithRateProviders (Constructor)
totalSharesLastUpdate: uint128(vault.totalSupply()),

// AccountantWithRateProviders (resetHighWaterMark and updateExchangeRate)
state.totalSharesLastUpdate = uint128(currentTotalShares);

// AccountantWithRateProviders (_calculateFeesOwed)
state.feesOwedInBase += uint128(newFeesOwedInBase);

// AccountantWithYieldStreaming (vestYield)
vestingState.vestingGains = uint128(yieldAmount);

// AccountantWithYieldStreaming (postLoss)
vestingState.vestingGains -= uint128(lossAmount);
vestingState.lastSharePrice =
uint128((totalAssets() - principalLoss).mulDivDown(ONE_SHARE, currentShares));

// AccountantWithYieldStreaming (updateExchangeRate)
vestingState.lastSharePrice = uint128((_totalAssets + newlyVested).mulDivDown(ONE_SHARE,
currentShares));
vestingState.vestingGains -= uint128(newlyVested);
state.totalSharesLastUpdate = uint128(currentShares);

// AccountantWithYieldStreaming (_collectFees)
state.totalSharesLastUpdate = uint128(currentTotalShares);

_calculateFeesOwed(
```

```
            state, uint96(vestingState.lastSharePrice), state.exchangeRate,
currentTotalShares, currentTime);

state.exchangeRate = uint96(vestingState.lastSharePrice);
```

Recommendations: Implement checks which make sure that the amount-in-question isn't higher than the amount to which we're down-casting it to.

Customer's response: Acknowledged.

I-08. updateExchangeRate() in Yield Streaming accountant lacks an `isPaused` check

Description:

As an example, in the AccountantWithRateProviders, if `updateExchangeRate` is to be called whenever the vault is paused, it will revert. This check is performed in the `_beforeUpdateExchangeRate` :

```
JavaScript
function updateExchangeRate(uint96 newExchangeRate) external virtual requiresAuth {
        (
            bool shouldPause,
            AccountantState storage state,
            uint64 currentTime,
            uint256 currentExchangeRate,
            uint256 currentTotalShares
        ) = _beforeUpdateExchangeRate(newExchangeRate);


 function _beforeUpdateExchangeRate(uint96 newExchangeRate)
        internal
        view
        returns (...
        )
    {
        state = accountantState;
        if (state.isPaused) revert AccountantWithRateProviders__Paused();
```

This check isn't implemented as part of the `updateExchangeRate,` although the function should be invoked only through the deposit/withdraw, as well as vestYield/postLoss flows, which contain it.

Recommendations: Implement an "isPaused" check as part of the updateExchangeRate.

Customer's response: Fixed in 30008b9.

Fix Review: Fix confirmed.

Recommendations

- Add respective versioning view functions to be able to track different contract versions;
- Similar systems implement respective referrer functionalities to incentivize and promote system widespread and usage;
- Break down the deny list functionality to ease up usage and avoid conflicts when the contract itself could be on the deny list;

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.