MixBytes()

Algebra ALM Security Audit Report

DECEMBER 19, 2025

Table of Contents

1. Introduction

1.1 Disclaimer

The audit makes no statements or warranties regarding the utility, safety, or security of the code, the suitability of the business model, investment advice, endorsement of the platform or its products, the regulatory regime for the business model, or any other claims about the fitness of the contracts for a particular purpose or their bug-free status.

1.2 Executive Summary

Algebra ALM is a liquidity management protocol built on top of Algebra DEX that allows users to deposit a single token and receive vault shares representing their proportional ownership of the liquidity pool. The protocol automatically manages concentrated liquidity positions across base and limit ranges, collects trading fees and farming rewards. Rewards are distributed to vault shareholders who stake their shares in the FarmingRewardsDistributor contract. Swap fees are distributed between vault shareholders and configured fee recipients.

The ALM (Automated Liquidity Management) plugin continuously monitors the vault's current state in deposited tokens and determines whether rebalancing is needed. When rebalancing conditions are met, the plugin calls the vault's rebalance() function, which burns existing concentrated liquidity positions and mints new ones according to calculated tick ranges based on price, volatility, and inventory thresholds.

The audit was conducted over 20 days by 3 auditors, involving an in-depth manual code review and automated analysis within the scope.

During the audit, beyond examining common attack vectors and our internal checklist, we conducted a comprehensive analysis of the following areas:

- **Security of the Peripheral-Contract Pattern.** The design uses a peripheral contract that will receive user approvals. We verified that the user-token pool employs the safe transferFrom pattern pulling from msg.sender, rather than from any other address.
- **Handling of Token Decimals.** The project operates on token pairs that most likely have different decimals. We confirmed that all arithmetic correctly accounts for this characteristic.
- **Profit from Price Manipulation.** We ensured that a malicious actor cannot earn unjustified profit or halt the protocol by manipulating the vault token prices.
- **Rewards and Fees Collection Order.** We confirmed that rewards and fees are always collected before positions are dismantled.
- **Swap Callback Security.** We verified that algebraSwapCallback is correctly implemented and can only be called from the pool.
- **Native Token Handling.** We verified that native value can be handled by the AlgebraVaultDepositGuard contract, receive() function is correctly implemented.
- **Force Stop Rebalance via Intentional Revert.** BaseRebalanceManager._rebalance() executes AlgebraVault.rebalance() in a try-catch clause. We verified that the rebalance cannot be

force-stopped via intentional reverts, since AlgebraVault.rebalance() has no failure conditions.

- **State Machine Integrity under Attack.** BaseRebalanceManager has a state system that may change on each price movement. We attempted to break this state using various approaches (flash loans, preventing rebalance execution for extended periods, attempting to reach states that would revert during resolution), but in all cases the system works as expected.

Furthermore, the ALM Plugin scope was further analyzed using Savant Chat to improve attack-vector coverage and enhance the overall quality of the security analysis.

We also want to add some small notes on the rebalance logic:

The strategy's BaseRebalanceManager._removeDecimals() logic, used when allowToken1 is false, performs integer division (10^decimals) / price. For pairs with an extreme price difference, such as WBTC (paired) / SHIB (deposit), where the price of one unit of the paired token (in deposit token units) exceeds 10^decimalsSum (e.g., 1 WBTC > 10^26 wei SHIB), this division results in 0. This zero value propagates to getTickAtPrice(), causing the transaction to revert. While this does not affect standard pairs like WBTC/USDC or ETH/USDC, it renders the strategy unusable for pairs involving tokens with very low unit prices (e.g., memecoins) against high-value assets like BTC. It is recommended to document this limitation or implement a safeguard to handle such extreme price ratios gracefully. (See Low-12. Incompatibility with High-Value-Difference Pairs due to Precision Loss in _removeDecimals() for details).

The BaseRebalanceManager._getPriceAccountingDecimals() function suffers from precision loss in a specific case: when calculating inverse prices (when token1 >= token0) for pools with very high tick values where sqrtPriceX96 > 2^128 (price > ~1.8×10^19, tick > ~443,000). In this condition, the function performs FullMath.mulDiv(2^128, decimals, priceX128), where priceX128 is extremely large (≥ 2^192), causing the numerator to be smaller than the denominator and resulting in a truncated value of 0. This causes the rebalance to silently return early without executing. While the probability of encountering such extreme price ratios is very low in practice, it is recommended to avoid deploying the strategy for pairs with extremely high price ratios, as this may lead to unexpected behavior.

Additionally, it is important to note that the project is designed for standard ERC20 tokens with decimals <= 18 and without custom transfer hooks (e.g., fee-on-transfer). Usage with non-standard tokens or tokens with higher decimals may lead to unexpected behavior.

The issues identified during the audit, together with detailed explanations and suggested recommendations, are outlined in the **Findings Report** section below.

Disclaimer

The client could provide the smart contracts for the deployment by a third party. To make sure that the deployed code hasn't been modified after the last audited commit, one should conduct their own investigation and deployment verification.

1.3 Project Overview

Summary

Title	Description
Client	Algebra
Category	DEX
Project	ALM
Type	Solidity
Platform	EVM
Timeline	10.11.2025 - 16.12.2025

Scope of Audit

File	Link
contracts/AlgebraVault.sol	AlgebraVault.sol
contracts/AlgebraVaultDepositGuard.sol	AlgebraVaultDepositGuard.sol
contracts/AlgebraVaultFactory.sol	AlgebraVaultFactory.sol
contracts/FarmingRewardsDistributor.sol	FarmingRewardsDistributor.sol
contracts/common/Enum.sol	Enum.sol
contracts/lib/AlgebraVaultDeployer.sol	AlgebraVaultDeployer.sol
contracts/lib/FarmingRewardsDistributorDeployer.sol	FarmingRewardsDistributorDeployer.sol
contracts/lib/OracleLibrary.sol	OracleLibrary.sol
contracts/lib/UV3Math.sol	UV3Math.sol
packages/alm/contracts/AlmPlugin.sol	AlmPlugin.sol
packages/alm/contracts/RebalanceManager.sol	RebalanceManager.sol

File	Link
packages/alm/contracts/base/BaseRebalanceManager.sol	BaseRebalanceManager.sol

Versions Log

Date	Commit Hash	Note
10.11.2025	57d820afa1d58bf89073e668f5608942d90188c7	Initial Commit (ALM Vault)
10.11.2025	6a5bcc44abfb90c3edb05bbea7efec233b5bd257	Initial Commit (ALM Plugin)
16.11.2025	d637339f968d67f175e8cb56ce3ae54a69bdefee	Re-audit Commit (ALM Vault)
16.11.2025	facb3310cff162942ccba662d26821457ad20dac	Re-audit Commit (ALM Plugin)
16.11.2025	9fc58be6a39ed8ca984284b0dee2b1d7bdbd91d0	Re-audit Commit (ALM Plugin)

1.4 Security Assessment Methodology

Project Flow

Stage	Scope of Work
Interim audit	**Project Architecture Review:** · Review project documentation · Conduct a general code review · Perform reverse engineering to analyze the project's architecture based solely on the source code · Develop an independent perspective on the project's architecture · Identify any logical flaws in the design OBJECTIVE: UNDERSTAND THE OVERALL STRUCTURE OF THE PROJECT AND IDENTIFY POTENTIAL SECURITY RISKS.
	Code Review with a Hacker Mindset: · Each team member independently conducts a manual code review, focusing on identifying unique vulnerabilities. · Perform collaborative audits (pair auditing) of the most complex code sections, supervised by the Team Lead. · Develop Proof-of-Concepts (PoCs) and conduct fuzzing tests using tools like Foundry, Hardhat, and BOA to uncover intricate logical flaws. · Review test cases and in-code comments to identify potential weaknesses. OBJECTIVE: IDENTIFY AND ELIMINATE THE MAJORITY OF VULNERABILITIES, INCLUDING THOSE UNIQUE TO THE INDUSTRY.
	Code Review with a Nerd Mindset: · Conduct a manual code review using an internally maintained checklist, regularly updated with insights from past hacks, research, and client audits. · Utilize static analysis tools (e.g., Slither, Mythril) and vulnerability databases (e.g., Solodit) to uncover potential undetected attack vectors. OBJECTIVE: ENSURE COMPREHENSIVE COVERAGE OF ALL KNOWN ATTACK VECTORS DURING THE REVIEW PROCESS.

Stage	Scope of Work
	Consolidation of Auditors' Reports: · Cross-check findings among auditors · Discuss identified issues · Issue an interim audit report for client review OBJECTIVE: COMBINE INTERIM REPORTS FROM ALL AUDITORS INTO A SINGLE COMPREHENSIVE DOCUMENT.
Re-audit	**Bug Fixing & Re-Audit:** · The client addresses the identified issues and provides feedback · Auditors verify the fixes and update their statuses with supporting evidence · A re-audit report is generated and shared with the client OBJECTIVE: VALIDATE THE FIXES AND REASSESS THE CODE TO ENSURE ALL VULNERABILITIES ARE RESOLVED AND NO NEW VULNERABILITIES ARE ADDED.
Final audit	**Final Code Verification & Public Audit Report:** · Verify the final code version against recommendations and their statuses · Check deployed contracts for correct initialization parameters · Confirm that the deployed code matches the audited version · Issue a public audit report, published on our official GitHub repository · Announce the successful audit on our official X account OBJECTIVE: PERFORM A FINAL REVIEW AND ISSUE A PUBLIC REPORT DOCUMENTING THE AUDIT.

1.5 Risk Classification

Severity Level Matrix

Severity	Impact: High	Impact: Medium	Impact: Low
Likehood: High	Critical	High	Medium
Likehood: Medium	High	Medium	Low
Likehood: Low	Medium	Low	Low

Impact

- **High** – Theft from 0.5% OR partial/full blocking of funds (>0.5%) on the contract without the possibility of withdrawal OR loss of user funds (>1%) who interacted with the protocol.
- **Medium** – Contract lock that can only be fixed through a contract upgrade OR one-time theft of rewards or an amount up to 0.5% of the protocol's TVL OR funds lock with the possibility of withdrawal by an admin.
- **Low** – One-time contract lock that can be fixed by the administrator without a contract upgrade.

Likelihood

- **High** – The event has a 50–60% probability of occurring within a year and can be triggered by any actor (e.g., due to a likely market condition that the actor cannot influence).
- **Medium** – An unlikely event (10–20% probability of occurring) that can be triggered by a trusted actor.
- **Low** – A highly unlikely event that can only be triggered by the owner.

Action Required

- **Critical** – Must be fixed as soon as possible.
- **High** – Strongly advised to be fixed to minimize potential risks.
- **Medium** – Recommended to be fixed to enhance security and stability.
- **Low** – Recommended to be fixed to improve overall robustness and effectiveness.

Finding Status

- **Fixed** – The recommended fixes have been implemented in the project code and no longer impact its security.
- **Partially Fixed** – The recommended fixes have been partially implemented, reducing the impact of the finding, but it has not been fully resolved.
- **Acknowledged** – The recommended fixes have not yet been implemented, and the finding remains unresolved or does not require code changes.

1.6 Summary of Findings

Findings Count

Severity	Count
Critical	0
High	0
Medium	1
Low	17

Findings Statuses

ID	Finding	Severity	Status
M-1	Missing Rebalance Bookkeeping Updates After Successful Rebalance	Medium	Fixed
L-1	Inaccurate Percentage Calculations Due to Unclaimed Fees Leading to Suboptimal Rebalancing Decisions	Low	Acknowledged
L-2	Unnecessary Gap Between Base and Limit Positions in Full Range Rebalancing	Low	Acknowledged
L-3	Hardcoded Minimum Gas Threshold Preventing Dynamic Adjustment	Low	Fixed
L-4	Incorrect State Return When Rebalance Not Needed	Low	Fixed
L-5	Missing Whitelist Check for Reward Tokens in `getReward()`	Low	Fixed
L-6	Checks-Effects-Interactions Pattern Violation in `_getReward()`	Low	Fixed
L-7	Rounding Error Allows Theft of User Deposits via Donation Attack	Low	Fixed
L-8	Risk Parameter Setters Lack Cross-Validation	Low	Fixed
L-9	Typo in Comment	Low	Fixed

L-10	Unsafe ETH Transfer Using `transfer()` Instead of `call()`	Low	Fixed
L-11	Unused Code	Low	Fixed
L-12	Incompatibility with High-Value-Difference Pairs due to Precision Loss in `_removeDecimals()`	Low	Acknowledged
L-13	Incorrect Tick Rounding for Negative Exact Multiples Causes Misaligned Liquidity Ranges	Low	Acknowledged
L-14	Missing Validation Allows Staking Token to Be Added as Reward Token	Low	Fixed
L-15	Quadratic Time Complexity in `getReward()` Function	Low	Fixed
L-16	Unsafe `approve()` Usage in Constructor	Low	Fixed
L-17	Missing Tick Bounds Validation in Range Calculations	Low	Acknowledged

2. Findings Report

2.1 Critical

Not Found

2.2 High

Not Found

2.3 Medium

M-1	Missing Rebalance Bookkeeping Updates After Successful Rebalance		
Severity	Medium	**Status**	Fixed in 9fc58be6

Description

In `BaseRebalanceManager._rebalance()`, after a successful `IAlgebraVault.rebalance()` call the contract does not update `lastRebalanceTimestamp`, `lastRebalanceCurrentPrice`, and `state = newState`.

These values are used by `BaseRebalanceManager._decideRebalance()` and `BaseRebalanceManager._updateStatus()` to enforce rebalance throttling and key state/price-based transitions. If they are not updated, the manager behaves as if a rebalance never happened:

- `minTimeBetweenRebalances` throttling in `BaseRebalanceManager._decideRebalance()` is effectively skipped because `lastRebalanceTimestamp` stays unchanged;
- `BaseRebalanceManager._updateStatus()` only evaluates `priceShiftTrigger` when `lastRebalanceCurrentPrice != 0`, so price-shift driven rebalances may never be triggered;
- `BaseRebalanceManager._updateStatus()` falls back to its "initial" branch when `lastRebalanceCurrentPrice == 0`, while `state` is never updated to `newState` after a successful rebalance, distorting the state-machine behavior.

Recommendation

We recommend updating `lastRebalanceTimestamp`, `lastRebalanceCurrentPrice`, and `state` after a successful `IAlgebraVault.rebalance()` call in `BaseRebalanceManager._rebalance()`.

2.4 Low

L-1	Inaccurate Percentage Calculations Due to Unclaimed Fees Leading to Suboptimal Rebalancing Decisions		
Severity	Low	**Status**	Acknowledged

Description

In `BaseRebalanceManager._obtainTWAPs()`, the function calls `IAlgebraVault(vault).getTotalAmounts()` which includes unclaimed fees (`tokensOwed`) from positions. These unclaimed fees are then used to calculate `percentageOfDepositToken` and `percentageOfDepositTokenUnused`, which are critical for determining rebalancing state and decisions. Additionally, `_getDepositTokenVaultBalance()` returns only the current vault balance, which does not include fees that will be collected and added to the vault balance when `collectFees()` is called. However, when the vault's `rebalance()` function is called, it first collects fees via `collectFees()`, which distributes a portion of fees to protocol/affiliate recipients, leaving only the remaining portion in the vault (increasing the vault's token balances). This means the rebalancing actually operates on a different portfolio base than what was used for decision-making: `getTotalAmounts()` includes unclaimed fees that will be partially distributed, while `depositTokenBalance` excludes fees that will be added to the vault balance.

This discrepancy causes `percentageOfDepositToken` to be calculated on an inflated portfolio base (denominator includes unclaimed fees), while `percentageOfDepositTokenUnused` is calculated on an understated base (numerator excludes fees that will be collected, while denominator is inflated). This leads to suboptimal rebalancing decisions. Position ranges are computed assuming more capital is available than will actually be present after fee distribution, resulting in ranges that may be slightly wider or narrower than optimal, and target prices that don't perfectly match the actual available capital. While the absolute difference is usually small, in edge cases with high fee accumulation, this mismatch can affect state transitions (OverInventory/UnderInventory/Normal) when percentages are near threshold boundaries.

Issue example

Consider a vault with state transition thresholds:

- `underInventoryThreshold = 7800` (78%)
- `normalThreshold = 8100` (81%)
- `overInventoryThreshold = 9100` (91%)
- `simulate = 9400` (94%)

Vault portfolio (WETH/USDT pair, price: 1 WETH = 2000 USDT):

- 100 WETH in positions (deposit token, across both base and limit ranges)
- 12,800 USDT in positions (paired token, equivalent to 6.4 WETH at current price, across both base and limit ranges)
- 0.7 WETH in unclaimed fees (`tokensOwed0` aggregated across base and limit positions) – ~0.7% of the total WETH in positions
- 50 USDT in unclaimed fees (`tokensOwed1` aggregated across base and limit positions, equivalent to 0.025 WETH) – ~0.4% of the total USDT in positions
- Protocol fee: 20% of fees
- Affiliate fee: 10% of fees

Current behavior - with unclaimed fees included:

- `getTotalAmounts()` returns: 100.7 WETH (100 + 0.7) deposit token and 12,850 USDT (12,800 + 50, equivalent to 6.425 WETH) paired token (includes unclaimed fees in both tokens)
- `depositTokenBalance = 0` (no deposit token on vault balance, fees not yet collected)
- Total portfolio value (in WETH equivalent): 100.7 + 6.425 = 107.125 WETH
- `percentageOfDepositToken` = (100.7 / 107.125) × 10000 ≈ **9400** (94.00%+)
- `percentageOfDepositTokenUnused` = (0 / 107.125) × 10000 = **0** (0%)
- Assuming current `state == State.Normal`: Since `percentageOfDepositToken > simulate` (≈ 9400), the system incorrectly determines `State.OverInventory`

After fee collection (actual rebalancing base):

- `collectFees()` collects 0.7 WETH and 50 USDT
- 0.14 WETH + 10 USDT goes to protocol (20%)
- 0.07 WETH + 5 USDT goes to affiliate (10%)
- 0.49 WETH + 35 USDT stays in the vault (added to vault balance, 70% of fees)
- Actual portfolio: 100.49 WETH (100 + 0.49) deposit token and 12,835 USDT (12,800 + 35, equivalent to 6.4175 WETH) paired token
- Actual `depositTokenBalance` ≈ 0.49 WETH (fees added to vault balance)
- Total portfolio value (in WETH equivalent): 100.49 + 6.4175 ≈ 106.91 WETH
- Correct `percentageOfDepositToken` ≈ (100.49 / 106.91) × 10000 ≈ **~9395–9399** (< 9400, i.e. just below `simulate` but above `normalThreshold`)
- Correct `percentageOfDepositTokenUnused` ≈ (0.49 / 106.91) × 10000 ≈ **~45–50** (≈0.5%)
- With these post-fee values, the correct state should remain: `State.Normal`

The same discrepancy can also affect transitions between `State.UnderInventory` and `State.Normal` when `percentageOfDepositToken` is close to the `underInventoryThreshold` / `normalThreshold` boundaries.

Recommendation

We recommend implementing a function in the vault that returns `AlgebraVault.getTotalAmounts()` without including unclaimed fees, and using these calculations for percentage computations.

Client's Commentary

We do not consider this problem impactful enough comparing to the efforts it would take to fix this.

L-2	Unnecessary Gap Between Base and Limit Positions in Full Range Rebalancing		
Severity	Low	**Status**	Acknowledged

Description

In `BaseRebalanceManager._getRangesWithoutState()`, the code can create a gap of `tickSpacing` between the base and limit positions in two cases when the current tick is already a valid rounded tick:

- When `_allowToken1 == true` and `currentTick == tickRounded`
- When `_allowToken1 == false` and `currentTick == tickRounded`

In both situations this is inconsistent with the alternative branches where positions touch each other, and contradicts the function's purpose of providing near-full-range coverage.

1. Case `allowToken1 == true`, `currentTick` is positive:

- `tickSpacing = 10`
- `currentTick = 20` (already rounded, so `tickRounded = 20`)
- `MIN_TICK = -887272`
- `MAX_TICK = 887272`

Bounds are:

- `baseLower = MIN_TICK` (rounded)
- `baseUpper = 20`
- `limitLower = 20 + 10 = 30`
- `limitUpper = MAX_TICK` (rounded)

Final positions are `[MIN_TICK, 20)` and `[30, MAX_TICK)`, leaving ticks `[20, 30)` without liquidity coverage.

2. Case `allowToken1 == true`, `currentTick` is negative:

- `tickSpacing = 10`
- `currentTick = -20` (already rounded, so `tickRounded = -20`)
- `MIN_TICK = -887272`
- `MAX_TICK = 887272`

Bounds are:

- `baseLower = MIN_TICK` (rounded)
- `baseUpper = -20`
- `limitLower = -20 + 10 = -10`
- `limitUpper = MAX_TICK` (rounded)

  Final positions are `[MIN_TICK, -20)` and `[-10, MAX_TICK)`, leaving ticks `[-20, -10)` without liquidity coverage.

3. Case `_allowToken1 == false`, `currentTick` is positive:

- `tickSpacing = 10`
- `currentTick = 20` (already rounded, so `tickRounded = 20`)
- `MIN_TICK = -887272`
- `MAX_TICK = 887272`

Bounds are:

- `limitLower = MIN_TICK` (rounded)
- `limitUpper = 20 - 10 = 10`
- `baseLower = 20`
- `baseUpper = MAX_TICK` (rounded)

Final positions are [MIN_TICK, 10) and [20, MAX_TICK), leaving ticks [10, 20) without liquidity coverage.

4. Case _allowToken1 == false, currentTick is negative:

- tickSpacing = 10
- currentTick = -20 (already rounded, so tickRounded = -20)
- MIN_TICK = -887272
- MAX_TICK = 887272

Bounds are:

- limitLower = MIN_TICK (rounded)
- limitUpper = -20 - 10 = -30
- baseLower = -20
- baseUpper = MAX_TICK (rounded)

Final positions are [MIN_TICK, -30) and [-20, MAX_TICK), leaving ticks [-30, -20) without liquidity coverage.

Recommendation

We recommend removing additional tick spacing when current tick is already rounded.

When allowToken1 == true:

```
if (twapResult.currentTick == tickRounded) {
--  ranges.limitLower = _tickSpacing + twapResult.currentTick;
++  ranges.limitLower = twapResult.currentTick;
} else {
    ranges.limitLower = tickRoundedDown + _tickSpacing;
}
```

When allowToken1 == false:

```
--  if (twapResult.currentTick == tickRounded) {
--    tickRoundedDown = twapResult.currentTick - _tickSpacing;
--  }
```

Client's Commentary

We acknowledge. This is an intended behaviour

L-3	Hardcoded Minimum Gas Threshold Preventing Dynamic Adjustment		
Severity	Low	**Status**	Fixed in facb3310

Description

In BaseRebalanceManager._rebalance(), the minimum gas threshold of 1600000 is hardcoded, preventing dynamic adjustment if gas costs change due to network conditions, contract upgrades, or changes in vault implementation. This inflexibility may require a contract redeployment if the hardcoded value becomes insufficient, leading to potential rebalancing failures or unnecessary gas waste.

Recommendation

We recommend making the minimum gas threshold configurable by adding a state variable and setter function with proper access control.

Client's Commentary

Client: Fixed in facb3310cff162942ccba662d26821457ad20dac

MixBytes: The require statement enforcing a minimum gas threshold has been removed.

L-4	Incorrect State Return When Rebalance Not Needed		
Severity	Low	**Status**	Fixed in facb3310

Description

In `BaseRebalanceManager._updateStatus()`, when `percentageOfDepositTokenUnused <= depositTokenUnusedThreshold` (indicating that rebalancing is not needed), the function always returns `State.Normal` regardless of the current state. This is incorrect when the current state is `OverInventory` or `UnderInventory` and the `percentageOfDepositToken` indicates that the system should remain in the current state.

This issue does not affect the code logic, as rebalancing does not occur when `needToRebalance = false` is returned. However, it returns an incorrect state value, which may cause confusion when monitoring or debugging the system's state.

Recommendation

We recommend returning the current `state` instead of always returning `State.Normal`:

```
if (
    twapResult.percentageOfDepositTokenUnused
        <= thresholds.depositTokenUnusedThreshold
) {
--  return (false, State.Normal); // no rebalance needed
++  return (false, state); // no rebalance needed
} else {
    return (true, state);
}
```

Client's Commentary

Fixed in 50d2aa7fe52385710bdae42f8a8c834a17ab4538

L-5	Missing Whitelist Check for Reward Tokens in getReward()		
Severity	Low	**Status**	Fixed in d637339f

Description

The FarmingRewardsDistributor contract maintains a whitelist of reward tokens via the rewardTokens array, which is populated by managers through addReward(). However, the _getReward() function does not verify that the provided token addresses are actually in the whitelist before processing them.

This allows users to pass arbitrary token addresses. As a result, _calculateClaimable() updates userInfo.lastTimeUpdated = block.timestamp even for non-whitelisted tokens, polluting the user's state with invalid data.

Recommendation

We recommend implementing a whitelist check using EnumerableSet.AddressSet from OpenZeppelin to efficiently validate reward tokens via contains() function.

```
function _getReward(
    address _user,
    address[] memory _rewardTokens
) internal whenNotPaused returns (uint256[] memory claimableAmounts) {
    claimableAmounts = new uint256[](_rewardTokens.length);

    for (uint256 i; i < _rewardTokens.length; i++) {
        address token = _rewardTokens[i];
        if (!rewardTokensSet.contains(token)) revert InvalidRewardToken();

        RewardData storage r = rewardData[token];
        _updateReward();
        _calculateClaimable(_user, token);
        // ... rest of the logic
    }
}
```

Alternatively, additional mapping can be used:

```
mapping(address => bool) public isRewardToken;

function _getReward(...) {
    for (uint256 i; i < _rewardTokens.length; i++) {
        address token = _rewardTokens[i];
        if (!isRewardToken[token]) revert InvalidRewardToken();
        // ... rest of the logic
    }
}
```

Client's Commentary

Fixed in 98f90b909e04e4c5cbab09a4f8dab95fc845cd85

L-6	Checks-Effects-Interactions Pattern Violation in `_getReward()`		
Severity	Low	**Status**	Fixed in d637339f

Description

The `FarmingRewardsDistributor._getReward()` function violates the checks-effects-interactions pattern by performing an external call (`safeTransfer`) before updating the contract state.

```
function _getReward(
    address _user,
    address[] memory _rewardTokens
) internal whenNotPaused returns (uint256[] memory claimableAmounts) {
    claimableAmounts = new uint256[](_rewardTokens.length);

    for (uint256 i; i < _rewardTokens.length; i++) {
        address token = _rewardTokens[i];
        RewardData storage r = rewardData[token];
        _updateReward();
        _calculateClaimable(_user, token);
        if (claimable[token][_user] > 0) {
            claimableAmounts[i] = claimable[token][_user];

            IERC20(token).safeTransfer(_user, claimable[token][_user]);
            r.amount -= claimable[token][_user]; // @audit state update after
            emit RewardPaid(_user, token, claimable[token][_user]);
            claimable[token][_user] = 0; // @audit state update after
        }
    }
}
```

The external call `safeTransfer()` is made before state updates (`r.amount -= ...` and `claimable[token][_user] = 0`). According to the checks-effects-interactions pattern, all state changes should be completed before making external calls to prevent reentrancy vulnerabilities.

Recommendation

We recommend following the checks-effects-interactions pattern by updating state before making external calls.

Client's Commentary

Fixed in f3a58dfdbf498cb1027c7b6e65856b1f7d3f4157

L-7	Rounding Error Allows Theft of User Deposits via Donation Attack		
Severity	Low	**Status**	Fixed in d637339f

Description

The AlgebraVault.deposit() function is vulnerable to a rounding-based attack where an attacker can steal user deposits by manipulating the vault's state. Due to Solidity's integer division rounding down, an attacker can cause legitimate deposits to result in 0 shares, while the deposited tokens remain in the vault.

The shares calculation in AlgebraVault.deposit() uses integer division that rounds down:

```
if (_totalSupply != 0) {
    uint256 priceForPool = _getConservativePrice(price, twap, auxTwap, true);
    uint256 pool0PricedInToken1 = pool0.mul(priceForPool).div(PRECISION);
    shares = shares.mul(_totalSupply).div(pool0PricedInToken1.add(pool1));
} else {
    shares = shares.mul(MIN_SHARES);
}
```

If shares rounds down to 0, the function mints 0 shares to the user, but the deposited tokens remain in the vault. When the attacker withdraws their shares, they receive a proportional share of ALL tokens in the vault, including the tokens deposited by users who received 0 shares.

Attack Scenario:

1. Attacker deposits 1 wei and receives 1000 shares (MIN_SHARES), totalSupply = 1000
2. Attacker transfers 1000e6 tokens directly to the vault
3. Legitimate user attempts to deposit 1e6 tokens:
   - shares = 1e6 * 1000 / (1000e6 + 1) = 0 (rounds down to 0)
   - _mint(user, 0) is called, but 1e6 tokens remain in the vault
   - Vault balance: 1000e6 + 1 + 1e6 = 1001000001 wei
4. Attacker withdraws all 1000 shares and receives ALL tokens, including the user's 1e6 deposit:

```
uint256 unusedAmount0 = IERC20(token0)
    .balanceOf(address(this))
    .mul(shares)
    .div(_totalSupply);
uint256 unusedAmount1 = IERC20(token1)
    .balanceOf(address(this))
    .mul(shares)
    .div(_totalSupply);
    if (unusedAmount0 > 0) IERC20(token0).safeTransfer(to, unusedAmount0);
    if (unusedAmount1 > 0) IERC20(token1).safeTransfer(to, unusedAmount1);
```

This issue is classified **Low** severity because while it theoretically allows an attacker to steal user deposits, in practice most users will interact with the vault through

AlgebraVaultDepositGuard, which includes a slippage check. If a user receives 0 shares, the transaction will revert when minimumProceeds > 0, which is the expected behavior. The attack is only viable if users deposit directly to the vault (bypassing AlgebraVaultDepositGuard) or set minimumProceeds = 0 in the guard, both of which are unlikely scenarios.

Recommendation

We recommend increasing the MIN_SHARES constant from 1000 to 1e6 and adding a check to prevent minting 0 shares. Increasing MIN_SHARES to 1e6 significantly increases the cost of the attack - with MIN_SHARES = 1e6, if the first deposit is 1 wei for 1e6 shares, an attacker would need to donate more than 1e12 tokens to cause a 1e6 token deposit to result in 0 shares, making the attack economically unfeasible.

```
if (_totalSupply != 0) {
    uint256 priceForPool = _getConservativePrice(price, twap, auxTwap, true);
    uint256 pool0PricedInToken1 = pool0.mul(priceForPool).div(PRECISION);
    shares = shares.mul(_totalSupply).div(pool0PricedInToken1.add(pool1));

++  if (shares == 0) revert InvalidDeposit();
} else {
    shares = shares.mul(MIN_SHARES);
}
```

Client's Commentary

Fixed in 9f5f362a3723e9ec6fe8686fd30a22948653e1d8

L-8	Risk Parameter Setters Lack Cross-Validation		
Severity	Low	**Status**	Fixed in d637339f

Description

Multiple administrator-only setter functions do not check that the new values remain coherent with the rest of the configuration, which allows inconsistent states and even permanent DoS:

- `setDtrDelta` only enforces `_dtrDelta <= 10000`; if the value becomes greater than `underInventoryThreshold`, `_decideRebalance` reverts on `thresholds.underInventoryThreshold - thresholds.dtrDelta`, bricking every rebalance attempt.
- `setPercentages` validates `limitReservePct` only against the current `thresholds.simulate`, so a later `setTriggers` call can raise `simulate` and silently leave `limitReservePct` out of bounds for `_getPriceBounds`.
- `setHighVolatility`, `setSomeVolatility`, and `setExtremeVolatility` never maintain `some <= high <= extreme`, allowing inverted thresholds that break `_decideRebalance` heuristics.
- The same three setters (`setHighVolatility`, `setSomeVolatility`, `setExtremeVolatility`) also lack an upper bound (e.g., `<= 10000`), so administrators can configure meaningless values above 100% volatility, nullifying the guardrails that `_calcPercentageDiff` relies on.

Recommendation

We recommend introducing strict invariants for every setter (e.g., `_dtrDelta < thresholds.underInventoryThreshold`, `limitReservePct <= 10000 - newSimulate`, `some <= high <= extreme`, each of `_someVolatility`, `_highVolatility`, `_extremeVolatility <= 10000`) and adding regression tests that cover applying these updates in different orders.

Client's Commentary:

Fixed in b2dca6683810364e16204dd31c5839fcca424926

L-9	Typo in Comment		
Severity	Low	**Status**	Fixed in d637339f

Description

There is a typo in a comment in the AlgebraVault._withdrawFromPosition() function. The word "aleady" should be "already".

```
// this function is always called after _cleanPositions is *aleady* called
```

Recommendation

We recommend fixing the typo by changing "aleady" to "already" in the comment.

Client's Commentary

Fixed in be187a6fb083e9cc7977130ce1b65fdc994fa8d0

L-10	Unsafe ETH Transfer Using transfer() Instead of call()		
Severity	Low	**Status**	Fixed in d637339f

Description

The AlgebraVaultDepositGuard._forwardWithdraw() function uses transfer() to send ETH to recipients, which is unsafe and can cause transactions to fail. The transfer() function forwards a fixed 2300 gas stipend, which may be insufficient for contracts with complex receive() or fallback() functions.

```
if (withdrawNative) {
    // the vault temporarily custodies the withdrawn amounts
    (amount0, amount1) = algebraVault.withdraw(shares, address(this));
    if (token0 == WRAPPED_NATIVE) {
        IWRAPPED_NATIVE(WRAPPED_NATIVE).withdraw(amount0);
        payable(to).transfer(amount0); // @audit-issue `transfer` is used
        IERC20(token1).safeTransfer(to, amount1);
    } else {
        IWRAPPED_NATIVE(WRAPPED_NATIVE).withdraw(amount1);
        payable(to).transfer(amount1); // @audit-issue `transfer` is used
        IERC20(token0).safeTransfer(to, amount0);
    }
}
```

Additionally, starting with Solidity compiler version 0.8.31, the transfer() method for addresses is deprecated and will prevent contract compilation (Solidity documentation).

Recommendation

We recommend replacing transfer() with call() while following the checks-effects-interactions pattern. The state changes should be completed before making external calls, and proper error handling should be implemented.

Client's Commentary

Fixed in 11e57ed873e86eeb8c1a17f0b07c838ea7bb6802

L-11	Unused Code		
Severity	Low	**Status**	Fixed in d637339f

Description

There are two instances of unused code in the codebase:

1. **Unused** managers **mapping**: The managers mapping in FarmingRewardsDistributor is declared but never used.

```
/// @notice Addresses approved to call mint
mapping(address => bool) public override managers;
```

2. **Unused** Enum.sol **file**: The common/Enum.sol file and its Operation enum are declared but never used in the codebase.

```
/// @title Enum - Collection of enums
contract Enum {
    enum Operation {Call, DelegateCall}
}
```

Recommendation

We recommend removing the unused managers mapping and its corresponding interface function, as well as the unused Enum.sol file and Operation enum, if they're not needed for future functionality.

Client's Commentary

Fixed in 12869152d51b08a9db4049b4bf9e29900c220cab

L-12	Incompatibility with High-Value-Difference Pairs due to Precision Loss in `_removeDecimals()`		
Severity	Low	**Status**	Acknowledged

Description

The `_removeDecimals` function performs price inversion using integer division: `(10 ** decimals) / amount`. This logic is used when `allowToken1` is false (i.e., when the Deposit Token is Token0) to convert the price from "Paired per Deposit" to "Deposit per Paired". However, if the price of the Paired token (in Deposit token units) exceeds `10 ** decimalsSum` (where `decimalsSum = depositDecimals + pairedDecimals`), the result of the division is truncated to `0`. This zero value is then passed to `getTickAtPrice`, which calls `TickMath.getTickAtSqrtRatio(0)`. Since `0 < MIN_SQRT_RATIO`, the function reverts with `IAlgebraPoolErrors.priceOutOfRange()`.

This issue renders the strategy unusable for trading pairs with extreme price disparities, such as `WBTC (paired) / SHIB (deposit)`, where the price of one unit of the paired token can exceed the precision threshold (`1 WBTC > 10^26 wei SHIB`).

Recommendation

Fixing this issue within the current architecture is non-trivial due to `uint256` precision limitations: integer arithmetic cannot represent values smaller than 1 (which occur when inverting extremely large prices), requiring a fundamental redesign to use fixed-point arithmetic (e.g., Q64.96) or a different scaling approach. It is recommended to explicitly acknowledge this limitation in the documentation and advise against deploying the strategy for pairs with extreme price ratios (where `Price_raw > 10^depositDecimals`). This includes pairs involving memecoins with high supply and low unit price, where the price difference can trigger this error.

Client's Commentary

We acknowledge this one.

L-13	Incorrect Tick Rounding for Negative Exact Multiples Causes Misaligned Liquidity Ranges		
Severity	Low	**Status**	Acknowledged

Description

The `BaseRebalanceManager.roundTickToTickSpacingConsideringNegative()` function incorrectly rounds negative ticks that are exact multiples of the tick spacing. The function unconditionally subtracts `_tickSpacing` for all negative ticks to implement floor rounding, but this causes exact multiples (e.g., `-60` with spacing `60`) to be shifted down unnecessarily (resulting in `-120` instead of `-60`).

Since this function is used to calculate `commonTick`, `tickForHigherPrice`, and `tickForLowerPrice` in `BaseRebalanceManager._getRangesWithState()` and `tickRoundedDown` in `BaseRebalanceManager._getRangesWithoutState()`, the misalignment propagates to the final liquidity ranges. While additional adjustments in some states (e.g., `State.Normal`) may partially compensate for this error, the incorrect rounding causes liquidity ranges to be shifted one full spacing below the intended position. However, the practical impact of this issue is limited, as the misalignment is constrained to a single `tickSpacing` unit, and since liquidity positions are already bound to tick boundaries defined by `tickSpacing`, such a shift typically has minimal effect on liquidity placement or fee collection efficiency in most scenarios.

Recommendation

We recommend modifying `BaseRebalanceManager.roundTickToTickSpacingConsideringNegative()` to only subtract `_tickSpacing` when the input tick is not an exact multiple of `_tickSpacing`.

Client's Commentary

We do not consider this problem impactful enough comparing to the efforts it would take to fix this.

L-14	Missing Validation Allows Staking Token to Be Added as Reward Token		
Severity	Low	**Status**	Fixed in d637339f

Description

The `FarmingRewardsDistributor.addReward()` function does not validate that the reward token is different from the staking token. If a manager accidentally adds the staking token as a reward token, the reward calculation logic will incorrectly treat staked tokens as rewards.

Recommendation

We recommend adding a check in `addReward()` to prevent adding the staking token as a reward token by validating that `_rewardToken != stakingToken` before adding it to the reward tokens array.

Client's Commentary

Fixed in 7155546bf7abcb725dbb7860da5d324ae26efeaa

L-15	Quadratic Time Complexity in `getReward()` Function		
Severity	Low	**Status**	Fixed in `d637339f`

Description

The `FarmingRewardsDistributor._getReward()` function has quadratic time complexity when processing multiple reward tokens. The function calls `_updateReward()` inside a loop that iterates over each reward token, and `_updateReward()` itself iterates over all reward tokens to update their reward calculations.

This inefficiency increases gas costs significantly as the number of reward tokens grows, making the function expensive to call and potentially causing transactions to fail due to gas limits.

Recommendation

We recommend moving the `_updateReward()` call outside of the loop that iterates over reward tokens.

Client's Commentary

Fixed in 5e679d615b0735b0b6f7a35ef070c70f5a6feee6

L-16	Unsafe approve() Usage in Constructor		
Severity	Low	**Status**	Fixed in d637339f

Description

The AlgebraVault constructor uses the standard approve() function to grant infinite approval to the NFT manager for both token0 and token1. However, some ERC20 tokens (such as USDT) do not allow changing an existing non-zero approval to a non-zero value, which can cause the approve() call to revert.

Using approve() directly can cause the contract deployment to fail when dealing with tokens that have non-standard approval behavior, preventing the vault from being deployed successfully.

Recommendation

We recommend replacing approve() with forceApprove() from OpenZeppelin's SafeERC20 library in the AlgebraVault constructor.

Client's Commentary

Fixed in 3603f391b198871491300db8ab97462d426a2155

L-17	Missing Tick Bounds Validation in Range Calculations		
Severity	Low	**Status**	Acknowledged

Description

The `BaseRebalanceManager._getRangesWithState()` and `BaseRebalanceManager._getRangesWithoutState()` functions compute position ranges by adding or subtracting `tickSpacing` from various ticks derived from `twapResult.currentTick` and price bounds. In several branches, they perform unchecked arithmetic such as:

In `_getRangesWithState()`:

- `ranges.baseUpper = int24(ranges.baseUpper + _tickSpacing);`
- `ranges.baseLower = int24(ranges.baseLower + _tickSpacing);`
- `ranges.baseUpper = currentTickIsRound ? twapResult.currentTick - _tickSpacing : _tickSpacing + ranges.baseUpper;`
- `ranges.limitUpper = currentTickIsRound ? twapResult.currentTick : _tickSpacing + ranges.limitUpper;`

In `_getRangesWithoutState()`:

- `tickRoundedDown = twapResult.currentTick - _tickSpacing;`
- `ranges.baseUpper = tickRoundedDown + _tickSpacing;`
- `ranges.limitLower = _tickSpacing + twapResult.currentTick;`
- `ranges.limitLower = tickRoundedDown + _tickSpacing;`

These operations do not ensure that the resulting ticks remain within the valid range [`TickMath.MIN_TICK`, `TickMath.MAX_TICK`]. If the current price is already near the upper or lower global tick, adding or subtracting `tickSpacing` can push one of the computed ticks outside this range (for example, `TickMath.MAX_TICK + tickSpacing` or `TickMath.MIN_TICK - tickSpacing`).

These out-of-range ticks are then propagated to `IAlgebraVault.rebalance()`, which eventually calls the pool's `mint()` / `burn()` functions. The pool enforces strict tick bounds and will revert when supplied with ticks outside [`MIN_TICK`, `MAX_TICK`]. In such a scenario, any rebalance attempt that goes through the affected branch of `BaseRebalanceManager._getRangesWithState()` or `BaseRebalanceManager._getRangesWithoutState()` will consistently revert until the price moves back inside a safe zone or the configuration is changed, preventing rebalancing at price extremes.

This issue is classified **Low** severity because it only manifests when the pool price moves extremely close to the global tick bounds (`TickMath.MIN_TICK` / `TickMath.MAX_TICK`), which is unlikely for well-configured pools.

Recommendation

We recommend adding explicit clamping or bounds checks when adjusting ticks by `tickSpacing` in both `BaseRebalanceManager._getRangesWithState()` and `BaseRebalanceManager._getRangesWithoutState()`, ensuring that all computed ticks are always constrained to the [`TickMath.MIN_TICK`, `TickMath.MAX_TICK`] interval before calling `IAlgebraVault.rebalance()`.

Client's Commentary

We acknowledge this one. We accept this along with №12

3. About MixBytes

MixBytes is a leading provider of smart contract audit and research services, helping blockchain projects enhance security and reliability. Since its inception, MixBytes has been committed to safeguarding the Web3 ecosystem by delivering rigorous security assessments and cutting-edge research tailored to DeFi projects.

Our team comprises highly skilled engineers, security experts, and blockchain researchers with deep expertise in formal verification, smart contract auditing, and protocol research. With proven experience in Web3, MixBytes combines in-depth technical knowledge with a proactive security-first approach.

Why MixBytes

- **Proven Track Record:** Trusted by top-tier blockchain projects like Lido, Aave, Curve, and others, MixBytes has successfully audited and secured billions in digital assets.
- **Technical Expertise:** Our auditors and researchers hold advanced degrees in cryptography, cybersecurity, and distributed systems.
- **Innovative Research:** Our team actively contributes to blockchain security research, sharing knowledge with the community.

Our Services

- **Smart Contract Audits:** A meticulous security assessment of DeFi protocols to prevent vulnerabilities before deployment.
- **Blockchain Research:** In-depth technical research and security modeling for Web3 projects.
- **Custom Security Solutions:** Tailored security frameworks for complex decentralized applications and blockchain ecosystems.

MixBytes is dedicated to securing the future of blockchain technology by delivering unparalleled security expertise and research-driven solutions. Whether you are launching a DeFi protocol or developing an innovative dApp, we are your trusted security partner.

Contact Information

https://mixbytes.io/

https://github.com/mixbytes/audits_public

hello@mixbytes.io

https://x.com/mixbytes