LICENSE NO. CS/PTS/C-202602-071

DATE

FAILSAFE

2 July 2026

Zodiac

Smart contract

SECURITY TEST CONDUCTED BY

SMART CONTRACT AUDIT

FailSafe

Table of Contents

Executive Summary

FailSafe was engaged to conduct an elite security review of the Zodiac smart contracts. As part of this work, our experienced team delved deeply into the system architecture, which is built around a dual-vault structure designed to manage real-world asset deposits, handle rebasing yield generation, and issue shares through tokenized receipts. The objective was to identify potential vulnerabilities, assess their implications, and deliver actionable recommendations to fortify the protocol's security posture. During the audit, several interesting patterns and vulnerabilities were discovered. Notably, a high-severity flaw was identified in the vault share pricing mechanism, where the absence of first-depositor protection allowed an actor to capture pre-seeded yield and artificially inflate the share price, effectively causing a denial-of-service for standard user deposits. Additionally, a vulnerability in the rebase cap logic caused limits to round down to zero, enabling a permanent denial-of-service attack on both vaults via minute token transfers. Another significant issue involved improper state sequencing during loss recovery, allowing early redeemers to withdraw disproportionate value against stale balances while locking out subsequent users. Furthermore, several lower-severity issues were noted, including missing slippage protection in the routing logic, protocol fee evasion through transaction chunking, and gaps in input validation. These findings highlight the importance of strict mathematical precision, secure state management, and comprehensive transaction validation to prevent economic and operational impacts. The Zodiac development team is commended for their proactive approach to security. The code written thus far reflects a strong commitment to maintaining a robust security posture. Their responsiveness and willingness to engage with the audit process demonstrate a commendable dedication to improving and securing their protocol.

Following the original engagement, a redemption fee was added to `Vault1` at commit `7a5b1c21`. That change was reviewed separately and is documented in the Diff Audit section, which carries three further findings numbered 14 to 16. All three were remediated at commit `11a0b75f` and verified.

Project Details

Project	Zodiac
Repository	`https://github.com/go-equitize/zodiac-contracts`
Audit Type	Smart contract
Initial Commit	`5c667f96759ded16563d0f0f88c2762e6dfe80f5`
Final Commit	`4f80a5c702dbe92a7c9b3bd5c67111317f62feaf`
Timeline	9 June 2026 - 1 July 2026

Structure & Organization of The Security Report

Issues are tagged as “Open”, “Acknowledged”, “Partially Resolved”, “Resolved” or “Closed” depending on whether they have been fixed or addressed.

- **Open**: The issue has been reported and is awaiting remediation from developer team.
- **Acknowledged**: The developer team has reviewed and accepted the issue but has decided not to fix it.
- **Partially Resolved**: Mitigations have been applied, yet some risks or gaps still remain.
- **Resolved**: The issue has been fully addressed and no further work is necessary.
- **Closed**: The issue is deemed no longer pertinent or actionable.

Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical	The issue affects the platform in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
High	The issue affects the ability of the platform to compile or operate in a significant way.
Medium	The issue affects the ability of the platform to operate in a way that doesn’t significantly hinder its behavior.
Low	The issue has minimal impact on the platform’s ability to operate.
Info	The issue is informational in nature and does not pose any direct risk to the platform’s operation.

Methodology

Threat Modelling

We employed a threat modelling approach to identify potential attack vectors and risks associated with the smart contract(s). This involved:

1. Asset Identification: Enumerating the critical assets within the smart contract(s), such as tokens, sensitive data, access controls, and more.
2. Threat Enumeration: Identifying potential threats such as reentrancy, integer overflow/underflow, denial of service, and more.
3. Vulnerability Assessment: Assessing vulnerabilities in the context of the smart contract(s) and its interaction with external components.
4. Risk Prioritization: Prioritizing identified threats based on their severity and potential impact.

Manual Code Review

Our manual analysis involved an in-depth review of the smart contract(s) source code, focusing on:

1. Code Review: Line-by-line examination to detect vulnerabilities and ensure compliance with best practices.
2. Logic Analysis: Analyzing the smart contract(s) business logic for vulnerabilities and inconsistencies.
3. Gas Optimization: Identifying areas for gas optimization and efficiency improvements.
4. Access Control Review: Ensuring proper access controls and permission management.
5. External Dependencies: Assessing the security implications of external dependencies or oracles.

Functional Testing in Hardhat/Foundry

Functional testing was performed using Hardhat/Foundry to ensure the correctness and reliability of the smart contract(s). This included:

1. Functional Testing: Comprehensive tests covering various functionalities and edge cases.
2. Integration Testing: Verifying the interaction of smart contract(s) with other components.
3. Deployment Verification: Ensuring the correctness of smart contract(s) deployment.

Fuzzing and Invariant Testing

Where deemed necessary based on the complexity and criticality of the smart contract(s), fuzzing and invariant testing were performed to identify vulnerabilities that might not be caught through conventional methods. This included:

1. Fuzz Testing: Employing fuzzing techniques to generate invalid, unexpected, or random inputs to trigger potential vulnerabilities.
2. Invariant Testing: Verifying invariants and properties to ensure the correctness and consistency of the smart contract(s) across various scenarios.

Edge Cases Scenarios Coverage

The audit thoroughly covered a wide spectrum of edge cases, including but not limited to:

1. Extreme Inputs: Testing with extreme and boundary inputs to assess resilience.
2. Exception Handling: Evaluating how the contract(s) handle exceptional scenarios.
3. Concurrency: Assessing the contract(s) behaviour in concurrent or simultaneous interactions.
4. Non-Standard Scenarios: Analyzing non-standard use cases that might impact contract(s) behaviour.

Reporting and Recommendations

Each finding includes:

1. The location within the codebase where the issue was found.
2. A clear explanation of the vulnerability, its root cause, and its potential exploitation.
3. Code snippets or detailed instructions on how to address the vulnerability.
4. Best practices and coding guidelines to prevent similar issues in the future.
5. Wherever applicable, a PoC to demonstrate issue severity, aiding effective mitigation.

Remediation Support

1. FailSafe is available to collaborate with the development team to address and remediate identified vulnerabilities.
2. Code changes and security fixes will be reviewed and validated upon request.

In-scope

- `contracts/vaults/Vault1.sol` — 239 nSLOC — Primary
- `contracts/vaults/Vault2.sol` — 210 — Primary
- `contracts/tokens/ZToken.sol` — 112 — Primary
- `contracts/tokens/TempToken.sol` — 111 — Primary
- `contracts/router/ZTokenRouter.sol` — 65 — Primary
- `contracts/vaults/base/BaseVault.sol` — 49 — Shared
- `contracts/security/Whitelist.sol` — 44 — Primary
- `contracts/security/Blocklist.sol` — 43 — Primary
- `contracts/vaults/FeeManager.sol` — 40 — Primary
- `contracts/libraries/Lib.sol` — 38 — Library
- `contracts/libraries/Errors.sol` — 27 — Library
- `contracts/interfaces/IVault1.sol` — 22
- `contracts/interfaces/IVault2.sol` — 17
- `contracts/vaults/storage/Vault1Storage.sol` — 15
- `contracts/vaults/storage/Vault2Storage.sol` — 14
- `contracts/interfaces/IZToken.sol` — 11
- `contracts/tokens/storage/ZTokenStorage.sol` — 11
- `contracts/interfaces/ITempToken.sol` — 10
- `contracts/tokens/storage/TempTokenStorage.sol` — 9
- `contracts/vaults/storage/BaseVaultStorage.sol` — 7
- `contracts/interfaces/IBaseVault.sol` — 5
- `contracts/interfaces/IBlocklist.sol` — 5
- `contracts/interfaces/IFeeManager.sol` — 5
- `contracts/interfaces/IWhitelist.sol` — 3

Out of scope

- any other files

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved
● Critical	–	–	–	–	–
● High	1	–	–	–	1
● Medium	3	–	–	–	3
● Low	7	–	1	–	6
● Info	2	–	–	–	2
Total	**13**	–	**1**	–	**12**

The Diff Audit section covers a later change to `Vault1` and is summarised separately at the head of that section.

#	Finding	Severity	Status
1	The very first person to deposit into Vault2 can grab any yield that arrived before them, and then leave the share price so high that normal deposits stop working	● High	Resolved
2	After a loss, if the admin un-pauses Vault2 before fixing its recorded balance, the price is still too high — early redeemers take more than their share and later ones can't get out	● Medium	Resolved
3	Vault1 records nominal amounts instead of the measured balance delta — with a rebasing Fund token this bricks deposits and drifts the peg on exit	● Medium	Resolved
4	When the rebase cap is enabled and the vault is near-empty, anyone can permanently freeze both vaults with a single dust transfer — the cap rounds to zero, every rebase trips it, and one un-booked dust amount re-trips the pause after every admin unpause	● Medium	Resolved
5	redeem is missing the zero-asset guard that withdraw has, so if the share price ever computes to zero a holder can burn their shares and receive nothing, with no revert	● Low	Resolved
6	A user can make Vault2 mislabel a plain donation as ”yield” — Vault1.deposit(X, address(vault2)) sends freshly minted tempTokens straight into Vault2, which the docs claim can only ever be real yield	● Low	Resolved

#	Finding	Severity	Status
7	The ”suspicious rebase” pause doesn’t latch when it trips during a normal deposit/withdraw — the transaction reverts and undoes the pause, so the persistent paused flag and the SuspiciousRebase event don’t get recorded via that path	● Low	Acknowledged
8	The interface docs say passing amount = 0 withdraws everything, but the code actually rejects 0	● Low	Resolved
9	The negative-rebase recovery function only un-pauses the vault when there’s still a shortfall to burn — if someone has already topped the balance back up, the call quietly does nothing and leaves the vault paused while looking like it succeeded	● Low	Resolved
10	The protocol fee can be dodged by feeding yield in tiny chunks — each chunk is small enough that the fee rounds down to zero, so the treasury collects nothing	● Low	Resolved
11	The router has no slippage protection (minOut) or deadline, so a rate move between submission and execution silently gives the user a worse fill	● Low	Resolved
12	A few read-only functions return values that an outside reader can misinterpret — the tokens always report 18 decimals, and Vault1’s peg view returns zeros while paused — so wallets, explorers, and integrators can show wrong numbers even though the protocol’s own accounting is fine	● Info	Resolved
13	The ”max you can deposit/mint” functions claim an unlimited amount, but actually depositing near that amount reverts with an overflow — and ”max you can withdraw” can report more than the vault will actually pay out	● Info	Resolved

Finding 1: The very first person to deposit into Vault2 can grab any yield that arrived before them, and then leave the share price so high that normal deposits stop working

Severity: ● High

Status: Resolved

Description

Vault2 hands out shares using a price: shares = your deposit / price, where price = (assets in the vault) / (shares already issued). There's a special case: when there are no shares yet, it just gives you shares equal to your deposit, 1-for-1, and completely ignores how much is already sitting in the vault. Normally OpenZeppelin's vault code protects against exactly this, but this project replaced that math with its own (in `Lib`) and left the protection out.

Here's why that matters. Vault1 earns yield and delivers it to Vault2 by minting tempTokens into it. If that yield lands *before anyone has deposited into Vault2*, you end up in a weird state: the vault holds real value, but there are zero shares. That's the dangerous moment. This isn't exotic — at launch Vault1 can already be earning while nobody has deposited into Vault2 yet.

Why yield can reach Vault2 before the first Vault2 depositor (this is the part that's easy to wave off, so here it is precisely): the mint to Vault2 (Vault1.sol:406-409) fires on one condition only — Vault1's own Fund token balance went up (`currentBalance > lastRecordedBalance`, Vault1.sol:382). It never reads `zToken.totalSupply()`; it has no idea whether Vault2 has any depositors. The natural objection is "if money is in the system, someone is already using Vault2" — but that conflates two separate, independent deposits. Depositing Fund tokens into Vault1 and depositing tempTokens into Vault2 are different user actions, and nothing forces them to happen together or in order. Concretely, the zero-share state is reachable three ways:

- Separate layers. A user deposits Fund tokens into Vault1 and just holds the tempToken receipt in their wallet — a perfectly normal end-state; they don't have to forward it into Vault2. So Vault1 can be funded and earning while Vault2 has never been touched (`totalSupply() == 0`). A rebase then mints yield into an empty Vault2.
- Permissionless donation. The Fund token is a plain ERC-20, so anyone can transfer it straight to Vault1's address, and `detectAndSyncRebase()` (Vault1.sol:160) is callable by anyone with no whitelist. So an attacker can *manufacture* the rebase themselves with no prior Vault2 activity at all.
- Launch ordering. At deployment Vault1 can be wired and funded before the first Vault2 deposit ever happens, so the very first rebase routes yield to a zero-share Vault2.

The tell that this state is real and not theoretical: the defense-in-depth fix below is “in Vault2’s sync, skip the ratchet while `totalSupply() == 0`.” That guard only does anything *because* the code currently lets value arrive while supply is zero — if the state were impossible, the guard would be dead code.

Now the first person to deposit walks in. Because there are no shares, they get shares 1-for-1 for a tiny deposit (say 1 wei), even though the vault already holds, say, 100. Then they redeem and walk out with the whole 100. They just took yield that should have belonged to future depositors, for almost nothing.

Two honest limits, so we don’t oversell it: (1) the first depositor can only grab what was *already there* — they can’t directly steal a later person’s deposit, because if a later deposit is too small to earn even 1 share it simply reverts and that person keeps their money. (2) the attacker has to be a whitelisted address. That’s why this is HIGH, not CRITICAL.

Side effect: after the price is inflated, anyone trying to deposit less than the (now huge) price gets 0 shares, and minting 0 shares reverts — so normal small deposits just fail. The vault is effectively closed to ordinary depositors until someone deposits more than the inflated price.

Proof of Concept:

Preconditions: fees and the rebase cap are off (the deployed default). The actor must be a whitelisted address.

Capture:

1. Yield reaches Vault2 before anyone has deposited into Vault2. Concretely: a real position exists in Vault1, a positive rebase occurs, and a sync runs — Vault1 mints the yield (say 100) as tempTokens into Vault2. Now Vault2 holds 100 but zToken supply is still 0.
2. The attacker is the first to deposit into Vault2, with 1 wei. Because supply is 0, they get 1 share for that 1 wei; the 100 already inside is ignored.
3. The attacker redeems their 1 share. With 1 share and 101 in the vault, they receive ~101 — paid 1, walked out with ~101, taking the pre-seeded yield.

Lockout (separate consequence): 4. With the price now inflated against a tiny supply, any deposit smaller than the per-share price converts to 0 shares, and minting 0 shares reverts — so ordinary deposits fail until someone deposits more than the inflated price.

Source

- contracts/libraries/Lib.sol:17-24 (`convertToShares`: when there are no shares yet, it just returns the deposit 1:1 and ignores how much is already in the vault)

- contracts/vaults/Vault2.sol:309-312 (Vault2 uses this `Lib` math instead of OpenZeppelin's, so OZ's built-in protection is turned off)
- contracts/vaults/Vault1.sol:406-409 (Vault1 sends yield into Vault2 by minting, with no check that Vault2 has any depositors yet)
- contracts/vaults/Vault2.sol:383-387 (Vault2 counts that minted balance as yield)
- contracts/tokens/ZToken.sol:78-79 + contracts/libraries/Lib.sol:85-86 (minting 0 shares reverts — this is what limits the damage)

Code:

```
// Lib.sol:17-24 - the root cause: when supply is 0, it ignores how much is already in the vault
function convertToShares(uint256 assets, uint256 supply, uint256 totalAssets) internal pure returns (uint256 shares) {
    if (supply == 0) return assets;        // gives 1 share per 1 asset, no matter what's already inside
    return assets.mulDiv(supply, totalAssets, Math.Rounding.Floor);
}

// Vault1.sol:406-409 - Vault1 pushes yield into Vault2 even if Vault2 has no depositors
if (yieldAmount > 0) {
    tempToken.mint(Vault2Address, yieldAmount);
}

// Vault2.sol:383-387 - Vault2 then counts that as its balance/price
uint256 actualBalance = tempToken.balanceOf(address(this));
if (actualBalance > expectedBalance) {
    expectedBalance = actualBalance;
}
```

Impact

The first depositor pockets any yield that reached Vault2 before there were any shares — value that was meant for future depositors. That value is not necessarily the attacker's own: it can be a real rebase, or a third party / integrator pre-funding the vault, so the first depositor can walk off with someone else's money for a 1-wei stake. And while the price stays inflated, ordinary-sized deposits revert, so the vault is unusable for normal users until someone deposits an amount bigger than the inflated price. It can happen again every time the vault empties out (all shares redeemed) with a little value left behind. It needs a whitelisted address, and it can't drain a later depositor's full deposit (small deposits revert instead of being stolen) — that's why it's HIGH rather than CRITICAL.

One thing to be clear about, because it changes how urgent the fix is: the reason a victim's rounded-to-zero deposit doesn't silently hand over their money today is *incidental*, not a deliberate inflation defense. `Vault2.deposit` never checks that the shares it's about to mint are non-zero — it just calls `IZToken.mint(receiver, 0)`, and that happens to revert inside ZToken because `Lib.validateAmount` rejects 0 (contracts/tokens/ZToken.sol:79). So the protection rests on a zero-amount guard in a different contract, not on the share math itself. There is no virtual-shares offset — the standard ERC-4626 defense is simply absent. If that downstream mint ever tolerated 0, or a victim's deposit lands one unit above the rounding boundary (≥1 share

at a manipulated price), the loss becomes real. That is exactly why the fix below is the virtual-shares offset in the share math, not “rely on the mint reverting.”

Worth noting on test coverage: the team’s own inflation test (`test_Vault2_InflationAttack_MitigatedByExpectedBalance` in test/foundry/Security.t.sol:39-75) only checks the “send tempTokens straight to Vault2” door — it asserts a direct `tempToken.transfer(vault2, ...)` reverts, and its victim deposits while supply and balance are both 1 (a healthy 1:1 state), so the dangerous `supply == 0, balance > 0` state is never created. And the share-price-protection invariants in test/foundry/Invariants.t.sol bail out at the top with `if (supply == 0 || total == 0) return;` at nine separate checks (lines 413, 447, 489, 522, 552, 585, 610, 641, 738) — i.e. the properties that would catch first-depositor inflation are skipped in precisely the state where it happens. So this situation is genuinely untested.

Remediation

Primary fix (recommended) — add a virtual-shares/virtual-assets offset to the four conversion helpers in `Lib.sol`, so the `supply == 0` shortcut is removed and the first depositor is always priced against what’s already in the vault. Apply to all four (`convertToShares`, `convertToAssets`, `convertToSharesCeil`, `convertToAssetsCeil`). The change to `convertToShares` (Lib.sol:17-24):

```
function convertToShares(uint256 assets, uint256 supply, uint256 totalAssets)
    internal pure returns (uint256 shares)
{
    // remove: if (supply == 0) return assets;
    return assets.mulDiv(supply + 1, totalAssets + 1, Math.Rounding.Floor);
}
```

and the symmetric change to `convertToAssets` (Lib.sol:31-38):

```
function convertToAssets(uint256 shares, uint256 supply, uint256 totalAssets)
    internal pure returns (uint256 assets)
{
    // remove: if (supply == 0) return shares;
    return shares.mulDiv(totalAssets + 1, supply + 1, Math.Rounding.Floor);
}
```

Apply the same `+1` / `+1` (with `Ceil` rounding) to `convertToSharesCeil` (Lib.sol:45-52) and `convertToAssetsCeil` (Lib.sol:59-66). The `+1` on each side is the standard OpenZeppelin virtual-shares offset; with it, a 1-wei first deposit against a 100-token vault yields `floor(1 * 1 / 101) = 0` shares, so the attack reverts on the `mint(0)` guard instead of handing over the pile. The same offset also removes the `supply > 0, totalAssets == 0` zero-pricing case (the denominator becomes `totalAssets + 1`, never zero), so it hardens the share math against degenerate post-loss states as well. (Note: do NOT rely on “seed dead shares at initialize” as the fix here — at `Vault2.initialize` (Vault2.sol:84-106) the vault does not yet hold `MINTER_ROLE` on ZToken and has no assets, so there is nothing to seed; the offset is the applicable fix.)

Defense in depth — book the pre-seed BEFORE the first deposit prices, so the offset can act on it. This is the load-bearing detail. Vault2 must reconcile `expectedBalance` up to the actual tempToken balance even

while supply is zero, so that when the first deposit prices, `totalAssets()` already reflects the pre-seed. In `Vault2._detectAndSyncRebase` Phase 1 (Vault2.sol:383-387):

```
// Reconcile UPWARD unconditionally — including while supply == 0 — so a pre-seed is
// booked before the first deposit prices against it. Pair with the offset above.
uint256 actualBalance = tempToken.balanceOf(address(this));
if (actualBalance > expectedBalance) {
    expectedBalance = actualBalance;
    emit RebaseDetected(expectedBalance);
}
```

CRITICAL — do NOT gate this upward reconcile on `totalSupply() > 0`. Doing so holds `expectedBalance` at 0 during the first deposit, which makes the offset price `convertToShares(1, supply = 0, totalAssets = 0) = 1` (a 1-wei deposit still mints a share), and the pre-seed then attaches to that share on the next sync — re-opening roughly half the original capture. The `supply == 0` skip and the offset are mutually defeating in the exact zero-supply window this finding targets; the upward reconcile must run while supply is zero for the offset to bite. With the pre-seed booked (`totalAssets ~= P`), a 1-wei first deposit prices to `floor(1 * 1 / (P + 1)) = 0` shares and reverts on the `mint(0)` guard. Attack closed.

(If a downward reconcile is also desired for the stale-high post-loss case in the related stale-balance finding, gate ONLY the downward direction on `supply > 0` — never the upward one. Concretely: `if (actualBalance > expectedBalance) expectedBalance = actualBalance; else if (supply > 0 && actualBalance < expectedBalance) expectedBalance = actualBalance;`.)

Phase 2 (the `vault1.detectAndSyncRebase()` yield-add path, Vault2.sol:390-405) may keep its `supply > 0` guard — that path adds reported yield on top of `expectedBalance`, and while supply is zero the minted tempToken is already reflected in `actualBalance` and is captured by the upward Phase-1 reconcile above on the next interaction. The hazard is only ever the un-booked actual balance at pricing time, which Phase 1 now closes.

Also close the related injection path: in `Vault1.deposit` (Vault1.sol:171-189) and `Vault1.mint` (Vault1.sol:192-211), reject the vault-2 receiver after the existing whitelist check, e.g. `if (receiver == Vault2Address) revert Errors.InvalidAddress();` (`Vault2Address` is already a state variable on Vault1 — Vault1Storage.sol:21) — otherwise a user can mint tempTokens straight into Vault2 and have it booked as yield.

Note on the residual first-depositor donation: once the pre-seed is booked, an honest first depositor whose deposit is smaller than the pre-seed receives floor-rounded (possibly zero) shares — i.e. they donate into the pooled value rather than being robbed by an attacker. This is the standard, accepted virtual-offset trade-off and is not a theft vector; the actor-driven capture is fully closed. If the team wants to eliminate even the donation, seed a minimal first position operationally at launch (deposit a small fixed amount as the protocol before opening deposits) so the `supply == 0` state never coincides with a non-zero pre-seed in production.

Finding 2: After a loss, if the admin un-pauses Vault2 before fixing its recorded balance, the price is still too high — early redeemers take more than their share and later ones can't get out

Severity: ● Medium

Status: Resolved

Description

After a loss event Vault2 is paused, and its recorded balance (`expectedBalance`) is left at the old, too-high number while the real tempToken balance has dropped (the recovery burns the shortfall). The intended recovery, which the docs spell out (recover `reconcileExpectedBalance()` `unpause()`), fixes the recorded balance before resuming. The gap: nothing in the code enforces that order — `unpause()` only checks the admin role, and the sync can only ratchet the recorded balance *up*, so it cannot self-correct a stale-high value.

So if the admin un-pauses before reconciling, withdrawals price off the inflated recorded balance. Concretely: 100 shares, real balance dropped to 80, but `expectedBalance` still 100. The first redeemer of 50 shares is paid 50 (their fair post-loss share was ~40) and drains the real balance to 30; the next redeemer of 50 shares is owed 50 but only 30 is left, so their transfer reverts — they're locked out. One honest user's funds are redistributed to a faster one, and the slower user can't exit.

This is not a stolen key and not attacker-manufactured — it needs the admin to slip on the documented two-step order during an already-stressful recovery. But documentation isn't an on-chain guard: the code provides zero protection against that slip, and the consequence is loss of user principal. That's why it's a real MEDIUM, not just an operational note.

Proof of Concept:

Setup: Vault2 has 100 zToken shares and `expectedBalance` of 100. A loss event drops the real tempToken balance to 80; recovery burns the shortfall, so the real balance is now ~80 but `expectedBalance` is still 100 (stale-high).

1. The admin calls `unpause()` before calling `reconcileExpectedBalance()` — nothing in the code stops this ordering.
2. User A redeems 50 shares. Priced against the stale `expectedBalance` (100), they are paid 50 — more than their fair post-loss share of ~40. The real balance drops from 80 to 30.
3. User B redeems their 50 shares. They are owed 50, but only 30 remains, so the transfer reverts — User B is locked out and cannot exit.

4. Net result: User A took more than their share, User B can't withdraw at all — one honest user's loss became another's gain, with no attacker and no stolen key.

Source

- contracts/vaults/Vault2.sol:242, 276 (the sync only ever adjusts the recorded balance upward, never down)
- contracts/vaults/Vault2.sol:252, 285 (withdrawals are checked against the recorded balance, not the real one)
- contracts/vaults/Vault2.sol:256, 289 (the actual payout comes from the real, lower balance)
- contracts/vaults/base/BaseVault.sol:74-76 (`unpause()` has no check that the books were fixed first)

Code:

```
// Vault2.sol:383-387 - Phase 1 only ever ratchets UP, so a stale-high expectedBalance never self-corrects:
uint256 actualBalance = tempToken.balanceOf(address(this));
if (actualBalance > expectedBalance) {        // only the "balance went up" case is handled
    expectedBalance = actualBalance;
    emit RebaseDetected(expectedBalance);
}

// Vault2.sol:252-256 (redeem path mirrors this) - withdraw checks the stale-high expectedBalance,
// but pays out of the real (lower) token balance:
Lib.validateBalance(expectedBalance, assets);                // priced against the inflated recorded balance
expectedBalance -= assets;
IZToken(address(zToken)).burn(owner, _shares);
IERC20(address(tempToken)).safeTransfer(receiver, assets);    // actual payout from the real, lower balance

// BaseVault.sol:74-76 - unpause() has no precondition that the books were reconciled first:
function unpause() external override onlyRole(ADMIN_ROLE) {
    _unpause();
}
```

Impact

During the recovery window the price is wrong: early redeemers get more than their fair post-loss share and drain the limited real balance, and later redeemers are locked out (their transfer reverts for lack of funds). One user's loss becomes another's gain. It happens silently, triggered by a plausible admin ordering mistake with no on-chain backstop.

Remediation

Make the stale-high state self-healing so it can't depend on operator discipline. Best option: let the sync reconcile *downward* too. In `Vault2._detectAndSyncRebase` Phase 1 (Vault2.sol:383-387) the code only handles `actualBalance > expectedBalance`; add the symmetric case so a drop is also picked up:

```
uint256 actualBalance = tempToken.balanceOf(address(this));
if (actualBalance != expectedBalance) {
    expectedBalance = actualBalance;     // reconcile both up and down
    emit RebaseDetected(expectedBalance);
}
```

Because the sync runs at the top of every deposit/withdraw/redeem, a stale-high value then self-corrects on the first interaction regardless of admin ordering. (A downward reconcile is safe here — Vault2's real balance only drops via the recovery burn, and the donation guard concerns balance going *up*, not down.) Alternatively, gate the resume: since `BaseVault.unpause()` is shared and not `virtual`, add a Vault2-specific recovery-unpause that requires `expectedBalance == tempToken.balanceOf(address(this))` (or simply reconciles before unpausing).

Finding 3: Vault1 records nominal amounts instead of the measured balance delta — with a rebasing Fund token this bricks deposits and drifts the peg on exit

Severity: ● Medium

Status: Resolved

Description

Vault1 tracks its Fund token holdings arithmetically: it books the *requested* amount on every operation rather than the amount actually received or sent. On deposit it does `rwaToken.safeTransferFrom(msg.sender, this, assets)` then `tempToken.mint(receiver, assets)` and `lastRecordedBalance += assets` (Vault1.sol:183-185), using the nominal `assets` in all three. The design assumes the Fund token transfers exactly, i.e. that a transfer of `assets` moves the recipient's `balanceOf` by exactly `assets`.

That assumption does not hold for a shares-based rebasing token, which is the intended asset. Such a token stores balances as `shares * multiplier / 1e18` and converts on every transfer with two floor divisions:

```
1  shares          = floor(amount * 1e18 / multiplier)   // rounds down
2  received balance = floor(shares * multiplier / 1e18)  // rounds down again  => <= amount
```

When the multiplier is not exactly `1e18`, the transfer credits up to 1 wei less than `amount`. So after a deposit, `lastRecordedBalance == assets` but the vault's real `rwaToken.balanceOf(this)` rose by only `assets - 1`. Now `balanceOf < lastRecordedBalance`, and the next `_detectAndSyncRebase` reads that shortfall as a **negative rebase** and pauses the vault (Vault1.sol:412-417) — even though no rebase occurred.

Because `ZTokenRouter.depositRWA` calls `vault1.deposit()` then `vault2.deposit()` in one transaction (ZTokenRouter.sol:151-156), and Vault2's own sync calls `vault1.detectAndSyncRebase()`, the phantom shortfall is seen inside the same deposit: Vault1 pauses, returns -1, Vault2 pauses, and the whole transaction reverts with `ContractPaused`. The deposit path is bricked on the very first deposit at any non-unity multiplier. Because the transaction reverts, on-chain state is left clean (balances 0, both vaults unpaused) — so the symptom is "deposits revert while redeeming existing positions still works," not a latched pause.

The same nominal-vs-measured mismatch exists on the exit and fee paths, in the opposite direction. `withdraw`/`redeem` do `rwaToken.safeTransfer(receiver, assets)` then `lastRecordedBalance -= assets` (Vault1.sol:239-241, 270-272), and the positive-rebase fee leg does `rwaToken.safeTransfer(feeManager, feeAmount)` with `lastRecordedBalance = currentBalance - feeAmount` (Vault1.sol:398-402). With a rebasing token the outbound transfer moves the vault's `balanceOf` by up to 1 wei less than the nominal amount, while the book is decremented by the full nominal — so `balanceOf` drifts *above* `lastRecordedBalance`. This drift is strictly positive, so it cannot pause the vault, but it accumulates as a wei-scale over-statement that the next sync books as

phantom yield to Vault2, and it violates the documented peg invariant (`lastRecordedBalance == balanceOf == TempToken supply`). Its sharper consequence is on a full exit: when the last holder withdraws everything, the token leaves a wei of dust in the vault so `balanceOf` is 1 while `lastRecordedBalance` and tempToken supply are 0 — the next deposit reads that dust as a positive rebase and mints it into Vault2 while zToken supply is still 0, recreating the pre-seed state that Finding 1 depends on (a re-arm on every vault emptying).

Proof of Concept:

Preconditions: the Fund token is a shares-based rebasing token with a multiplier ≠ 1e18 (currently 1.111e18). Deposit path:

1. A first deposit of `assets` is made (via the router or directly into Vault1). Vault1 books `lastRecordedBalance += assets` and mints `assets` tempTokens, but the vault's real Fund token balance rose by `assets - 1` (double-floor rounding in the token).
2. The next `_detectAndSyncRebase` (the router's Vault2 leg triggers it in the same tx) sees `currentBalance (assets - 1) < lastRecordedBalance (assets)`, classifies it as a negative rebase, pauses Vault1, returns -1.
3. Vault2 pauses on the -1 and the whole deposit transaction reverts. Deposits are unusable; existing redemptions still work. This reproduces at any non-unity multiplier, including a 1-bps rebase.

Exit path:

1. A holder withdraws `assets`. The vault sends `assets` (token credits the receiver `assets - 1`), so the vault's `balanceOf` falls by `assets - 1`, but `lastRecordedBalance -= assets` (full nominal). Now `balanceOf > lastRecordedBalance` by ~1 wei.
2. Repeat / full exit: after all holders leave, the dust remains in the vault with `lastRecordedBalance == 0`; the next deposit's sync books it as a positive rebase into a zero-supply Vault2 (Finding 1 re-arm).

Source

- contracts/vaults/Vault1.sol:183-185 (`deposit`: `safeTransferFrom(...); tempToken.mint(receiver, assets); lastRecordedBalance += assets` — all nominal)
- contracts/vaults/Vault1.sol:204-207 (`mint`: same nominal pattern)
- contracts/vaults/Vault1.sol:239-241, 270-272 (`withdraw`/`redeem`: `safeTransfer(receiver, assets); lastRecordedBalance -= assets` — nominal on the way out)
- contracts/vaults/Vault1.sol:398-402 (positive-rebase fee leg: `safeTransfer(feeManager, feeAmount)` with `lastRecordedBalance = currentBalance - feeAmount`)

- contracts/vaults/Vault1.sol:412-417 (negative-rebase branch: any `currentBalance < lastRecordedBalance` pauses the vault — this is what the 1-wei deposit shortfall trips)
- contracts/router/ZTokenRouter.sol:151-156 (`depositRWA` runs `vault1.deposit()` then `vault2.deposit()` in one tx, so the phantom shortfall is detected inside the deposit and reverts it)

Code:

```
// Vault1.deposit (Vault1.sol:183-185) - books nominal `assets`, not the measured balance delta:
rwaToken.safeTransferFrom(msg.sender, address(this), assets);
tempToken.mint(receiver, assets);              // with a rebasing token, balanceOf rose by only assets-1
lastRecordedBalance = lastRecordedBalance + assets;

// Vault1._detectAndSyncRebase (Vault1.sol:412-417) - the 1-wei shortfall is read as a negative rebase:
} else if (currentBalance < lastRecordedBalance) {
    _pause();
    emit NegativeRebaseDetected(lastRecordedBalance - currentBalance);
    return -1;
}

// Vault1.withdraw (Vault1.sol:239-241) - nominal on exit, opposite-direction drift:
tempToken.burn(owner, assets);
rwaToken.safeTransfer(receiver, assets);       // balanceOf falls by only assets-1
lastRecordedBalance = lastRecordedBalance - assets;
```

Impact

Availability: with the intended rebasing Fund token at any non-unity multiplier, the deposit path is bricked on the first deposit — no user can convert Fund tokens to zTokens through Vault1 or the router — until the accounting is fixed. No funds are lost (the transaction reverts atomically), and existing positions can still be redeemed. On the exit/fee side the mismatch is wei-scale and cannot pause the vault, but it breaks the documented 1:1 peg invariant, mints a trickle of unearned yield to Vault2, and re-creates the Finding 1 pre-seed state each time the vault is fully emptied. Overall a Medium: severe deposit-availability impact under the intended asset, plus an invariant/peg-integrity defect on exit, with no direct fund loss.

Remediation

1. On deposit and mint, record the measured amount actually received instead of the nominal argument. In `Vault1.deposit` / `Vault1.mint`:

   ```
   uint256 balanceBefore = rwaToken.balanceOf(address(this));
   rwaToken.safeTransferFrom(msg.sender, address(this), assets);
   uint256 received = rwaToken.balanceOf(address(this)) - balanceBefore;
   Lib.validateAmount(received);
   tempToken.mint(receiver, received);
   lastRecordedBalance = lastRecordedBalance + received;
   ```

   This keeps `lastRecordedBalance == balanceOf == TempToken supply`, so the rounding gap can never be mis-read as a negative rebase.

2. Apply the symmetric measured-delta pattern on the way out. In `withdraw`/`redeem`, decrement `lastRecordedBalance` by the measured amount actually sent (`balanceBefore − balanceAfter`) rather than the nominal `assets`; do the same for the fee-leg transfer in `_detectAndSyncRebase` (set `lastRecordedBalance` from the measured post-transfer balance, with a zero-fee branch). This keeps the book aligned with the real balance on exit and prevents the positive dust drift and the Finding 1 re-arm on full exit.

3. Alternatively/additionally, document that the Fund token must be an exact-credit ERC-20 — but since the intended asset is a shares-based rebasing token, the measured-delta approach is the applicable fix, not a documented constraint.

Finding 4: When the rebase cap is enabled and the vault is near-empty, anyone can permanently freeze both vaults with a single dust transfer — the cap rounds to zero, every rebase trips it, and one un-booked dust amount re-trips the pause after every admin unpause

Severity: ● Medium

Status: Resolved

Description

Vault1 pauses itself if a rebase looks suspiciously large. The size limit is `lastRecordedBalance * maxRebaseBps / 10000`. With integer math, when the balance is small this rounds down to **zero** — so the limit becomes 0, and then any increase at all, even 1 wei, exceeds it and pauses the vault. The vulnerable balance is anything below `10000 / maxRebaseBps` base units (up to ~10000 wei at a small cap), i.e. only while the vault is essentially empty. Note this also misfires with no attacker at all: at low TVL the very first legitimate yield is "larger" than a zero limit, so an ordinary positive rebase pauses the vault — a self-inflicted DoS, not just a griefing vector.

Two open doors make this reachable by anyone: (1) the underlying is a normal ERC-20, so anyone can send dust straight to Vault1 with no permission; and (2) `detectAndSyncRebase()` (Vault1.sol:160) has no access control. So an attacker sends 1-2 wei to Vault1, calls `detectAndSyncRebase()`, the limit (0) is exceeded, and Vault1 pauses. While Vault1 is paused it returns `-1` to Vault2's sync, and Vault2 pauses too; `syncExpectedBalance()` (Vault2.sol:159) lets anyone trigger that cascade. Both vaults stop.

The pause is durable: the standalone `detectAndSyncRebase()` returns normally (it does not revert), so the pause commits. And it does not even cost fresh dust each round — the cap branch does `return 0` (Vault1.sol:391) *before* `lastRecordedBalance` is updated (Vault1.sol:398), so the donated dust is never booked; the balance permanently reads higher than recorded, and the next call after any admin unpause re-trips the pause. One dust transfer plus repeated free calls is an indefinite freeze, as long as the vault stays near-empty.

The cap-pause itself is intended behavior, but three things here are not: the limit rounding to zero, the pause persisting through this path, and the cascade to Vault2 — which README:63 actually mis-states as "Vault2 does not auto-pause on this path" (it does: a paused Vault1 returns -1, and Vault2.sol:390-405 pauses on that).

Proof of Concept:

Preconditions: the cap is enabled (admin has set maxRebaseBps > 0, e.g. 1%) and the vault balance is small enough that the limit rounds to 0 — for a 1% cap that means lastRecordedBalance below 100 base

units, which only holds while the vault is near-empty (and the cap branch needs lastRecordedBalance > 0, so a tiny non-zero position like 1 wei).

1. Anyone sends 1-2 wei of the underlying token to Vault1 (a plain ERC-20 transfer — no whitelist, no permission).
2. Anyone calls `detectAndSyncRebase()`. Vault1's balance went up; the size limit rounds to 0; the increase exceeds it, so Vault1 pauses. The pause sticks because this entry point returns normally instead of reverting.
3. Anyone calls `syncExpectedBalance()` (or any Vault2 operation runs): Vault1 returns -1 because it is paused, and Vault2 pauses too. Both vaults are now frozen.
4. The admin unpauses. The dust was never booked into lastRecordedBalance (the cap branch returns before that update), so the balance still reads higher than recorded — the next `detectAndSyncRebase()` re-trips the pause. No new dust is needed; repeat after every unpause.

Source

- contracts/vaults/Vault1.sol:160-164 (`detectAndSyncRebase()` is open to anyone — only `nonReentrant`, no whitelist/role)
- contracts/vaults/Vault1.sol:385-392 (the cap: `maxRebase = lastRecordedBalance * maxRebaseBps / 10000`; if a rebase exceeds it, pause and `return 0` — before `lastRecordedBalance` is updated at :398)
- contracts/vaults/Vault1Storage.sol:33 (`maxRebaseBps` defaults to 0 — the cap is off unless an admin enables it)
- contracts/vaults/Vault2.sol:390-405 (when Vault1 returns -1 because it's paused, Vault2 pauses itself too)
- contracts/vaults/Vault2.sol:159-163 (`syncExpectedBalance()` is open to anyone, so the cascade can be triggered permissionlessly)

Code:

```
// Vault1.sol:160-164 - anyone can trigger the rebase check (no whitelist/role):
function detectAndSyncRebase() external override nonReentrant returns (int256 yieldAmount) {
    if (paused()) return -1;
    yieldAmount = _detectAndSyncRebase();
    return yieldAmount;
}

// Vault1.sol:385-392 - the cap rounds to 0 on a small balance, then pauses and returns BEFORE line 398 books the dust:
if (maxRebaseBps > 0 && lastRecordedBalance > 0) {
    uint256 maxRebase = (lastRecordedBalance * maxRebaseBps) / BASIS_POINTS;  // rounds to 0 when balance is tiny
    if (rebaseAmount > maxRebase) {
        _pause();
        emit SuspiciousRebase(rebaseAmount, maxRebase);
        emit VaultPaused("Suspicious rebase: amount exceeds maxRebaseBps");
        return 0;                          // returns here - lastRecordedBalance (line 398) is never updated
```

```
    }
}

// Vault2.sol:390-405 — a paused Vault1 returns -1, and Vault2 pauses itself too (the cascade):
try vault1.detectAndSyncRebase() returns (int256 yieldAmount) {
    ...
    } else if (yieldAmount < 0) {
        if (!paused()) { _pause(); }
        emit NegativeRebaseDetected();
    }
} catch { if (!paused()) { _pause(); } emit NegativeRebaseDetected(); }
```

Impact

For one dust transfer plus gas, anyone can keep both vaults paused — no deposits, withdrawals, or redemptions for anyone — and re-trip it after every unpause faster than an admin can keep up. No funds are stolen, and the admin can permanently neutralize it with one call (`setMaxRebaseBps(0)`), so it is a recoverable denial-of-service. It is inactive unless the cap has been enabled, and only durably exploitable while the vault balance is near-empty.

Remediation

1. Stop the limit from rounding to zero, at Vault1.sol:385-392. Either clamp it:

   ```
   uint256 maxRebase = (lastRecordedBalance * maxRebaseBps) / BASIS_POINTS;
   if (maxRebase == 0) maxRebase = 1;            // never a zero limit
   ```

   or only apply the cap above a minimum balance, so the percentage cannot round to 0:

   ```
   if (maxRebaseBps > 0 && lastRecordedBalance >= MIN_BALANCE_FOR_CAP) { ... }
   ```

   with `MIN_BALANCE_FOR_CAP >= BASIS_POINTS / maxRebaseBps`. This closes the dust-trip regardless of who can call the trigger.

2. Add access control to the triggers: apply the existing `onlyWhitelisted` modifier (BaseVault.sol:30) or a keeper role to `Vault1.detectAndSyncRebase()` (Vault1.sol:160) and `Vault2.syncExpectedBalance()` (Vault2.sol:159).

3. Correct README:63, which states Vault2 does not auto-pause on this path; it does cascade once Vault1 is paused.

Finding 5: `redeem` is missing the zero-asset guard that `withdraw` has, so if the share price ever computes to zero a holder can burn their shares and receive nothing, with no revert

Severity: ● Low

Status: Resolved

Description

The two exit functions guard their inputs inconsistently. `withdraw` checks the asset amount isn't zero (Vault2.sol:238). `redeem` checks the *share* amount but never the resulting asset amount (Vault2.sol:271). So if the share-to-asset rate ever computes to zero — which happens when the recorded balance (`totalAssets`) is 0 while shares still exist — a `redeem` call sails through: it burns the holder's shares (Vault2.sol:288) and transfers them 0 (Vault2.sol:289), with no error. The holder loses their shares for nothing. The only thing keeping impact low is that reaching `totalAssets == 0` with shares outstanding is a degenerate state — it needs a near-total loss of backing plus a recovery that leaves the recorded balance at 0 — so it's a real but narrow-trigger defect, an oversight in input validation rather than a likely event.

Proof of Concept:

1. The recorded balance is 0 while zToken supply is still > 0 (e.g. a loss-recovery reconciles the balance down to 0 while shares remain).
2. A holder calls redeem(shares). The asset amount computes to 0 (shares × 0 / supply).
3. The share-amount check passes; there is no asset-amount check.
4. The holder's shares are burned and 0 is transferred — they lose their shares for nothing, with no revert.

Source

- contracts/vaults/Vault2.sol:238 (`withdraw` calls `Lib.validateAmount(assets)` — reverts on a zero asset amount)
- contracts/vaults/Vault2.sol:271 (`redeem` only calls `Lib.validateAmount(_shares)` — never checks the asset amount)
- contracts/vaults/Vault2.sol:283-289 (`redeem` computes `assets`, then burns shares and transfers `assets` — which can be 0)

Code:

```
// withdraw() - guards the asset amount:
Lib.validateAmount(assets);                // Vault2.sol:238 - reverts if assets == 0

// redeem() - guards only the share amount, not the asset amount:
Lib.validateAmount(_shares);               // Vault2.sol:271
...
assets = convertToAssets(_shares);         // == 0 when totalAssets() == 0
IZToken(address(zToken)).burn(owner, _shares);             // burns the holder's shares
IERC20(address(tempToken)).safeTransfer(receiver, assets);  // transfers 0 - no revert
```

Impact

In the degenerate state where the recorded balance is 0 but shares still exist, a holder calling `redeem` burns their shares and gets 0 assets back with no warning. It's a real loss path, but gated behind a rare near-total-loss + recovery state, so it's low-likelihood.

Remediation

1. Add the same zero-asset guard `withdraw` has to `redeem`: after `assets = convertToAssets(_shares)`, `Lib.validateAmount(assets)` (revert if 0). This is the direct fix for the asymmetry.
2. Optionally, also revert or auto-pause when `totalAssets() == 0 && supply > 0`, so the vault can't sit in a state where every redemption pays nothing.

Finding 6: A user can make Vault2 mislabel a plain donation as “yield” — `Vault1.deposit(X, address(vault2))` sends freshly minted tempTokens straight into Vault2, which the docs claim can only ever be real yield

Severity: ● Low

Status: Resolved

Description

Vault2 keeps a number, `expectedBalance`, of how much tempToken it thinks it holds. The rule the team wrote down is simple: “if my real balance is higher than `expectedBalance`, the extra is yield Vault1 sent me, so count it as yield.” They block outsiders from sending tempTokens into Vault2 directly to protect this. The problem: when you deposit into Vault1 you get to choose who receives the tempTokens, and nothing stops you from choosing Vault2. So you call `Vault1.deposit(X, address(vault2))`, pay X of the real asset, and Vault1 mints X tempTokens into Vault2. The block on direct transfers doesn’t catch this because minting goes through a different path than transfers. Vault2 then sees its balance went up and records X as “yield” — but it isn’t yield, it’s just your donation. So the team’s written rule (“the extra is always real yield”) is not actually true. That’s the whole issue. It is not a way to steal money: you gave away X and got nothing back — no shares, no receipt. There is even an easier way to do the same thing without permission (just send the underlying asset straight to Vault1, which the rebase logic then books as yield). We report this separately only because (1) it breaks a rule the team explicitly documented, and (2) their own test for this only checks the direct-transfer door, never this deposit door.

Proof of Concept:

Preconditions: the actor is whitelisted, and Vault2’s own address is whitelisted (the deployment adds Vault2 to the shared whitelist so the yield path can mint to it).

1. The actor calls `Vault1.deposit(X, address(vault2))`. The only receiver check is “is Vault2 whitelisted?” — it is — so it passes.
2. Vault1 pulls X of the underlying from the actor and mints X tempTokens to Vault2. The transfer restriction is skipped because this is a mint, not a transfer. The actor receives no zTokens.
3. On Vault2’s next sync, it sees its balance went up by X and records X as “yield” — but it was a plain donation, not yield. The documented “any excess is always real yield” invariant is now false. The actor simply gave away X and got nothing back.

Source

- contracts/vaults/Vault1.sol:171-189 (`deposit` lets you choose any whitelisted `receiver`; line 184 mints to it; no check that `receiver != Vault2`)
- contracts/tokens/TempToken.sol:189-192 (the mint case returns before the `to == vault2` block at :199-205, so minting into Vault2 is not blocked)
- contracts/vaults/Vault2.sol:383-387 (Phase 1 counts any extra balance as “yield”)
- contracts/README.md:49 and “contracts/Zodiac Contracts Design Doc.txt”:144-145 (the docs say this extra balance is “always legitimate yield from vault1”)

Code:

```
// Vault1.deposit lets you pick the receiver, with no check that it isn't Vault2:
if (!whitelist.isWhitelisted(receiver)) revert Errors.NotWhitelisted();
tempToken.mint(receiver, assets);          // receiver can be address(vault2)

// In TempToken, a mint is handled first and returns early, so the "block transfers into vault2" rule never runs:
if (from == address(0)) {                  // this is a mint
    super._update(from, to, value);
    return;                                // returns here, before the to == vault2 check below
}
...
if (to == vault2) { ... }                  // never reached for a mint
```

Impact

The team’s documented promise that Vault2’s extra balance “is always real yield” is false — a user can make Vault2 count a donation as yield. But on its own nobody profits: the user pays X and receives nothing. Any actual loss would require the share price to already be in the manipulable first-depositor / inflated-price state, and an even simpler permissionless version (just send the underlying to Vault1, which books it as yield) already exists. So this is a broken assumption plus a missing test, not a standalone way to lose funds. That’s why it’s LOW.

Remediation

1. In `Vault1.deposit` (Vault1.sol:171-189) and `Vault1.mint` (Vault1.sol:192-211), reject the vault-2 receiver right after the existing whitelist check: `if (receiver == Vault2Address) revert Errors.InvalidAddress();` (`Vault2Address` is already a Vault1 state variable — Vault1Storage.sol:21; `Errors.InvalidAddress` exists — Errors.sol:7). Normal user deposits should never target Vault2; the yield path mints to Vault2 internally.
2. Make Vault2’s accounting trust only the reported yield, not the raw balance. In `Vault2._detectAndSyncRebase` (Vault2.sol:381-406), the Phase-2 path already credits `expectedBalance` from the

`int256` returned by `vault1.detectAndSyncRebase()`; the leak is the Phase-1 block (Vault2.sol:383-387) that ratchets `expectedBalance` up to any raw `tempToken.balanceOf(address(this))` increase. Restrict Phase 1 so it cannot absorb a balance that did not come from a reported rebase (at minimum, guard it with `IZToken(address(zToken)).totalSupply() > 0`), so an unsolicited mint into Vault2 is not silently booked as yield.

3. Add a regression test that calls `Vault1.deposit(X, address(vault2))` and asserts it reverts (after the fix) — the existing inflation test (`test_Vault2_InflationAttack_MitigatedByExpectedBalance`, test/foundry/Security.t.sol:39-75) only asserts that a direct `tempToken.transfer(address(vault2), ...)` reverts; nothing in the suite exercises the deposit-into-vault2 mint channel, so it is currently untested.
4. Adding a virtual-shares offset to the Vault2 share math also neutralizes the share-price impact of any donation booked this way, reducing this to a pure invariant/documentation issue.
5. Fix the misleading comment that documents this exact false guarantee: the Vault2 contract header (contracts/vaults/Vault2.sol:20) states “expectedBalance guards against donation attacks via direct tempToken transfers.” That is only half true — it blocks direct transfers (the receipt-token transfer rules reject them) but does NOT block the mint-into-Vault2 path described above, which the same accounting still books as yield. Reword it to say the guard covers direct transfers only and that mints originated through Vault1’s deposit/mint receiver must be blocked separately (per remediation step 1).

Finding 7: The “suspicious rebase” pause doesn’t latch when it trips during a normal deposit/withdraw — the transaction reverts and undoes the pause, so the persistent paused flag and the SuspiciousRebase event don’t get recorded via that path

Severity: ● Low

Status: Acknowledged

Description

When a rebase exceeds the cap, Vault1 calls `_pause()` inside `_detectAndSyncRebase` and returns. Through the standalone `detectAndSyncRebase()` that pause persists (and the SuspiciousRebase event stands). But through a normal deposit/withdraw, the very next line is `if (paused()) revert` (Vault1.sol:181) — and that revert unwinds the whole transaction, including the `_pause()` and the event. So on the user path the persistent paused flag never sticks and the event is rolled back; `lastRecordedBalance` is also left unchanged, so the same condition is just re-evaluated next time. Importantly, the *protective* effect still holds on this path — the user’s deposit/withdraw reverts, so no operation goes through at the suspicious rate. What’s unreliable is only the latched state and the alert, depending on which entry point tripped the cap.

Proof of Concept:

Setup: the cap is enabled (`maxRebaseBps > 0`) and a rebase large enough to exceed it is pending on Vault1.

1. Path A (user op): a user calls `deposit`. The sync trips the cap and calls `_pause()`, but the very next line `if (paused()) revert` reverts the whole transaction — so `_pause()` and the `SuspiciousRebase` event are rolled back. After the tx: `paused()` is still false, no event was emitted, monitoring sees nothing — yet the deposit failed.
2. Path B (standalone): instead call `detectAndSyncRebase()` directly. Same cap-trip, same `_pause()`, but there is no trailing revert — so the pause commits and `SuspiciousRebase` is emitted. After the tx: `paused()` is true and the event is on-chain.
3. Same underlying condition, two different observable outcomes. An attacker can pick Path A to keep tripping the breaker silently (every honest deposit just reverts like a normal failed tx) while alerting keyed on the event / paused flag stays blind.

Source

- contracts/vaults/Vault1.sol:385-392 (the cap branch calls `_pause()` and `return 0` — without updating `lastRecordedBalance` at :398)

- contracts/vaults/Vault1.sol:171-189 (deposit/withdraw run the sync, then `if (paused()) revert` — which rolls the `_pause()` back)
- contracts/vaults/Vault1.sol:160-164 (the standalone `detectAndSyncRebase()` returns normally, so the pause persists there)

Code:

```
// Vault1.sol:385-392 - inside _detectAndSyncRebase, the cap-trip pauses and returns:
_pause();
emit SuspiciousRebase(rebaseAmount, maxRebase);
emit VaultPaused("Suspicious rebase: amount exceeds maxRebaseBps");
return 0;

// Vault1.sol:180-181 - but on the deposit/withdraw path, the line right after the sync reverts the whole tx,
// which UNDOES the _pause() and the events above:
_detectAndSyncRebase();
if (paused()) revert Errors.ContractPaused();   // revert rolls back the pause + SuspiciousRebase emit

// Vault1.sol:160-164 - via the standalone entry point there is no trailing revert, so the pause persists here:
function detectAndSyncRebase() external override nonReentrant returns (int256 yieldAmount) {
    if (paused()) return -1;
    yieldAmount = _detectAndSyncRebase();       // pause from the cap-trip commits; event stands
    return yieldAmount;
}
```

Impact

No funds at risk and no attacker leverage on the value side — the user operation reverts either way, so the suspicious rebase is not processed. The gap is observability/robustness: the persistent paused flag and the SuspiciousRebase event are recorded only when the cap is tripped via the standalone sync, not via a user op — so monitoring/alerting and any tooling that reads the paused flag can miss the event. There's a sharper edge to this when paired with the dust-cap condition (a near-empty vault where the cap rounds to zero): an attacker effectively gets to *choose* whether the trip is visible or silent. Tripping via the standalone `detectAndSyncRebase()` latches the pause and emits SuspiciousRebase (loud); doing the same thing through `deposit` reverts the whole call, so the breach leaves no event and no flag (silent) while still blocking that deposit. Used the silent way, every honest deposit reverts and looks like an ordinary failed transaction, so the protocol is effectively down for deposits while alerting keyed on SuspiciousRebase/the paused flag sees nothing.

Remediation

1. Record the cap-trip in a way that survives the user-path revert (e.g. commit the pause/flag in a separate sub-call that isn't unwound by the outer revert), so the breaker latches and emits consistently regardless of entry point.

Finding 8: The interface docs say passing amount = 0 withdraws everything, but the code actually rejects 0

Severity: ● Low

Status: Resolved

Description

The interface comment tells integrators “pass amount = 0 to withdraw the full balance,” but the actual `withdrawFees` rejects 0 and reverts. So the documented convenience behavior doesn’t exist — only the doc is wrong, the code is fine.

Proof of Concept:

1. An integrator reads the `IFeeManager.withdrawFees` doc, which states `amount = 0` withdraws the entire balance.
2. They call `withdrawFees(token, to, 0)` expecting a full sweep.
3. `Lib.validateAmount(0)` reverts with `InvalidAmount` — the documented “0 = withdraw all” behavior does not exist, and the call fails instead of sweeping the balance.

Source

contracts/interfaces/IFeeManager.sol:28,31 (the doc says `amount = 0` withdraws the full balance) vs contracts/vaults/FeeManager.sol:56 (`Lib.validateAmount(amount)` rejects 0)

Code:

```
// IFeeManager.sol:27-32 - the interface documents amount = 0 as "withdraw the full balance":
/// @dev Requires ADMIN_ROLE. Pass `amount = 0` to withdraw the full balance.
/// @param amount Amount to withdraw; 0 withdraws the entire balance.
function withdrawFees(address token, address to, uint256 amount) external;

// FeeManager.sol:54-56 - but the implementation rejects 0 outright via validateAmount:
Lib.validateAddress(token);
Lib.validateAddress(to);
Lib.validateAmount(amount);                // reverts InvalidAmount when amount == 0 - the documented behavior is
    impossible
```

Impact

An integrator who follows the docs and passes 0 gets a revert instead of a full withdrawal. No funds at risk.

Remediation

1. Either implement the documented “0 = withdraw all” behavior, or fix the doc to say the amount must be non-zero.

Finding 9: The negative-rebase recovery function only un-pauses the vault when there's still a shortfall to burn — if someone has already topped the balance back up, the call quietly does nothing and leaves the vault paused while looking like it succeeded

Severity: ● Low

Status: Resolved

Description

When Vault1 takes a loss it pauses itself and the admin is meant to call `recoverFromNegativeRebase` to burn the shortfall and resume. The catch: all of that function's work — burning the excess, fixing `lastRecordedBalance`, emitting the event, and the `_unpause()` — is wrapped in a single `if (currentBalance < receiptSupply)` block, with no `else`. So the function only does anything when there is still a shortfall.

The problem is that the balance can recover on its own before the admin gets to that call. The underlying is a plain ERC-20, so anyone can send Fund tokens straight to Vault1 (no permission, no role), and a rebasing asset can also simply rebound. Either way, by the time the admin calls recovery, `currentBalance` is already back to or above `receiptSupply`. The `if` is false, the body is skipped, and the function returns successfully — but it burned nothing, emitted nothing, and never un-paused. The admin's transaction "succeeds," yet the vault is still frozen.

And it doesn't sit there quietly. While Vault1 is paused, its sync returns -1, and Vault2's own sync treats that as a negative rebase and re-pauses itself every time it runs. So one silent no-op on Vault1 keeps both vaults down until someone notices and calls the plain `unpause()` by hand.

This isn't a stolen key and isn't an attacker getting paid — it's a misleading function that reports success while leaving the system paused, with a permissionless donation as the trigger. No funds are lost and the admin can always fall back to the separate `unpause()`, which is why it's LOW rather than higher.

Proof of Concept:

1. Vault1 takes a negative rebase and pauses itself; `lastRecordedBalance` is left unchanged, so the books still show a shortfall.
2. Before the admin runs recovery, the Fund token balance is restored — anyone sends Fund tokens directly to Vault1 (a plain ERC-20 transfer, no whitelist needed), or the rebasing asset rebounds — so `currentBalance` is now >= `receiptSupply`.
3. The admin calls `recoverFromNegativeRebase(burnFrom)`. Both guards pass (vault is paused, TempToken isn't), but `currentBalance < receiptSupply` is false, so the body is skipped. The call returns successfully having burned nothing and emitted nothing — and never un-paused.

4. The vault is still paused. On the next Vault2 sync, Vault1 returns -1, and Vault2 re-pauses too. Both stay down until the admin notices and calls the plain `unpause()` separately.

Source

- contracts/vaults/Vault1.sol:120-137 (`recoverFromNegativeRebase`: everything, including `_unpause()`, sits inside `if (currentBalance < receiptSupply)`; there is no `else` and no unconditional un-pause)
- contracts/vaults/Vault1.sol:412-417 (a negative rebase pauses the vault and leaves `lastRecordedBalance` unchanged, so the recovery precondition is set up)
- contracts/vaults/base/BaseVault.sol:74-76 (`unpause()` is the only other way out — a separate manual admin call)
- contracts/vaults/Vault2.sol:390-405 (while Vault1 stays paused it returns -1, so Vault2 keeps re-pausing itself on every sync)

Code:

```
// Vault1.recoverFromNegativeRebase — the only un-pause lives inside the shortfall branch:
function recoverFromNegativeRebase(address burnFrom) external onlyRole(ADMIN_ROLE) {
    if (!paused()) revert Errors.VaultNotPaused();
    if (tempToken.paused()) revert Errors.ContractPaused();

    uint256 currentBalance = rwaToken.balanceOf(address(this));
    uint256 receiptSupply = tempToken.totalSupply();

    if (currentBalance < receiptSupply) {       // false once the balance is topped back up
        uint256 excess = receiptSupply - currentBalance;
        require(tempToken.balanceOf(burnFrom) >= excess, "Insufficient balance to burn");
        tempToken.burn(burnFrom, excess);
        lastRecordedBalance = currentBalance;
        emit RebaseDetected(currentBalance);
        _unpause();                             // never reached when currentBalance >= receiptSupply
    }
    // no else: nothing happens, no event, vault stays paused, call still "succeeds"
}
```

Impact

During an incident recovery, the admin calls the function that is supposed to resume the vault, the call succeeds, and the vault stays paused anyway — with no event or error to signal the no-op. Both vaults stay frozen (Vault2 keeps re-pausing off Vault1's paused state) until someone realizes the recovery didn't actually resume anything and calls the plain `unpause()` by hand. It's extended downtime during an already-stressful moment, not a loss of funds, and there's always the manual `unpause()` fallback — so it's LOW.

Remediation

Make the function resume the vault even when there's nothing to burn, instead of silently doing nothing. In `recoverFromNegativeRebase` (Vault1.sol:120-137), add the already-restored case:

```
if (currentBalance < receiptSupply) {
    uint256 excess = receiptSupply - currentBalance;
    require(tempToken.balanceOf(burnFrom) >= excess, "Insufficient balance to burn");
    tempToken.burn(burnFrom, excess);
    lastRecordedBalance = currentBalance;
    emit RebaseDetected(currentBalance);
    _unpause();
} else {
    // peg already restored: nothing to burn, but still resume and re-sync the books
    lastRecordedBalance = currentBalance;
    emit RebaseDetected(currentBalance);
    _unpause();
}
```

Alternatively, if a no-op should not silently look like success, revert with a clear error (e.g. `Errors.NothingToRecover`) when `currentBalance >= receiptSupply`, so the admin gets an explicit signal to use the plain `unpause()` instead of believing recovery completed.

Also fix the comment that oversells this function. The NatSpec (contracts/vaults/Vault1.sol:116-119) says the function "restores the 1:1 peg" and that after it the admin should reconcile and unpause Vault2 — implying it always resumes Vault1. It does not: when the balance is already restored it returns without unpausing. Update the comment to state that it only acts (burns + unpauses) while a shortfall exists, and to direct the admin to the plain `unpause()` otherwise, matching whichever code fix above is applied.

Finding 10: The protocol fee can be dodged by feeding yield in tiny chunks — each chunk is small enough that the fee rounds down to zero, so the treasury collects nothing

Severity: ● Low

Status: Resolved

Description

The fee on a rebase is `rebaseAmount * fee% / 10000`, using integer math that rounds down. If a single rebase is small enough, that fee rounds all the way down to **0**, and the entire amount is passed through as yield with no fee taken. Since anyone can trigger a rebase and the underlying can be added in any size, a large amount of yield can be fed in as many tiny rebases, each below the threshold, each paying zero fee. The leftover from rounding isn't saved up either — the recorded balance moves forward after each chunk — so it never adds up into a later fee-bearing rebase. Result: the protocol collects ~0 fee on yield delivered this way.

Proof of Concept:

Say the fee is 1%, so the fee on a rebase is rebaseAmount / 100, which rounds to 0 for any rebase under 100 units.

1. Instead of letting one big rebase of R happen, add the underlying to Vault1 in chunks of 99 units.
2. Trigger the rebase after each chunk: 99 × 1% rounds to 0 fee, so all 99 goes to Vault2 as yield, and the recorded balance moves up so nothing is carried over.
3. Repeat until all of R is delivered. The treasury collected roughly nothing instead of the intended 1% of R.

Source

- contracts/vaults/Vault1.sol:395 (fee = rebaseAmount × fee% / 10000 — integer division, rounds down)
- contracts/vaults/Vault1.sol:396,406-409 (whatever isn't fee becomes yield to Vault2; if the fee is 0, the whole thing is yield)
- contracts/vaults/Vault1.sol:160-164 (anyone can trigger a rebase, so anyone can control how it's chunked)

Code:

```
// Vault1.sol:395-409 - fee is floor-rounded per rebase; a small enough chunk makes feeAmount == 0,
// so the whole chunk passes through as yield, and lastRecordedBalance advances (no remainder carried over):
uint256 feeAmount  = (rebaseAmount * fees) / BASIS_POINTS;   // integer division -> 0 for a small rebaseAmount
uint256 yieldAmount = rebaseAmount - feeAmount;

lastRecordedBalance = currentBalance - feeAmount;           // moves forward; nothing is carried to the next chunk

if (feeAmount > 0) {
    rwaToken.safeTransfer(feeManager, feeAmount);
}
if (yieldAmount > 0) {
    tempToken.mint(Vault2Address, yieldAmount);             // when feeAmount==0, the entire chunk becomes yield
}

// Vault1.sol:160-164 - and anyone can trigger each chunk's rebase, so the attacker controls the chunk size:
function detectAndSyncRebase() external override nonReentrant returns (int256 yieldAmount) {
    if (paused()) return -1;
    yieldAmount = _detectAndSyncRebase();
    return yieldAmount;
}
```

Impact

The protocol loses its fee revenue — it leaks to Vault2 holders instead of the treasury, potentially nearly all of it on yield delivered in small chunks. Only matters once a non-zero fee is set (it's 0 by default, but the production revenue model would turn it on).

Remediation

1. Carry the rounding leftover forward (track a running remainder and fold it into the next fee), so chunking can't escape the fee.
2. And/or compute the fee against the total un-fee'd amount rather than each chunk, and/or restrict who can trigger rebases (`detectAndSyncRebase` is currently open to anyone).
3. Document the smallest rebase that's economically meaningful given the fee rate.

Finding 11: The router has no slippage protection (`minOut`) or deadline, so a rate move between submission and execution silently gives the user a worse fill

Severity: ● Low

Status: Resolved

Description

Both router functions take only an input amount. The output is computed when the transaction executes, against Vault2's current share price, which shifts on every rebase. With no `minOut`, the user accepts whatever rate prevails when the tx is mined; with no deadline, the tx can sit in the mempool and execute much later at a stale rate. This is the standard slippage-and-deadline protection a DEX-style router is expected to have, and it's missing here. Note both entry points are `onlyWhitelisted` (ZTokenRouter.sol:140,186), so the adversary set is limited to whitelisted participants rather than the open mempool — which bounds the sandwich exposure and is part of why this is LOW.

Proof of Concept:

1. A user submits `depositRWA(rwaAmount)` (or `redeemZToken`) expecting the share price they saw when they signed. The call carries only the input amount — there is no `minOut` to bound the result and no `deadline` to bound the timing.
2. Before the tx is mined, the Vault2 share price shifts (a rebase syncs, or another whitelisted party's tx reorders ahead of it). The output is recomputed at execution time against the new price.
3. The call still succeeds and the user receives the worse amount — there is no minimum-output check to revert on, and no deadline to revert a tx that sat in the mempool and executed much later at a stale rate. The user silently gets a worse fill than intended.

Source

- contracts/router/ZTokenRouter.sol:140 (`depositRWA` — `onlyWhitelisted`, takes only an input amount; no `minOut`, no deadline)
- contracts/router/ZTokenRouter.sol:186 (`redeemZToken` — same)
- contracts/vaults/Vault2.sol:182,309-312 (output is priced at execution time against the live, rebase-sensitive share price)

Code:

```
// ZTokenRouter.sol:140 — depositRWA takes only an input amount: no minOut, no deadline:
function depositRWA(uint256 rwaAmount) external nonReentrant onlyWhitelisted whenNotEmergency returns (uint256
     zTokenAmount) {
    ...
    zTokenAmount = vault2.deposit(vrAmount, msg.sender);   // output priced live at execution; caller cannot bound it
}

// ZTokenRouter.sol:186 — redeemZToken, same shape: only an input amount, no minOut / no deadline:
function redeemZToken(uint256 zTokenAmount) external nonReentrant onlyWhitelisted whenNotEmergency returns (uint256
     rwaAmount) {
    ...
    rwaAmount = vault1.redeem(vrAmount, msg.sender, address(this));   // realized rate unknown until mined
}
```

Impact

Users routing through this contract can receive materially less than expected, and are exposed to ordering/sandwich and stale-execution by other whitelisted parties. No insolvency — it's user value loss and a missing safety rail.

Remediation

1. Add `minOut` (revert if the realized output is below the user's minimum) and a `deadline` (revert if mined past it) to both router functions.

Finding 12: A few read-only functions return values that an outside reader can misinterpret — the tokens always report 18 decimals, and Vault1's peg view returns zeros while paused — so wallets, explorers, and integrators can show wrong numbers even though the protocol's own accounting is fine

Severity: ● Info

Status: Resolved

Description

This groups the read-only functions whose output an outside reader can misread. None of them affects the protocol's own math — every write path re-reads live state before it computes — so this is purely about what wallets, explorers, and integrators see.

First, decimals. Neither token sets its own `decimals()`, so both report 18. If the underlying asset isn't 18 decimals (say it's 6), anything outside the protocol reading `decimals()` will misinterpret balances by a huge factor — a million-x off for a 6-decimal asset.

Second, the two peg views on Vault1 don't agree and one of them returns placeholder numbers. `validatePeg()` returns `(false, 0, 0)` whenever the vault is paused — the 0s are a sentinel meaning "can't validate right now," not the real Fund token balance or receipt supply. A frontend that displays those two numbers as actual balances would show 0/0 during a pause. `getPegStatus()`, on the other hand, returns the real live balance and supply but never looks at the pause state, so it can report `isPegged = true` while the vault is paused. A UI reading one and not the other, or treating the `validatePeg` zeros as real, will show a misleading picture exactly when something is already wrong (during a pause).

Proof of Concept:

1. Decimals: deploy with a 6-decimal underlying. A wallet/explorer reads `ZToken.decimals()` and gets 18 (the inherited default, never overridden). It displays a balance of 1,000,000 units as `1.0` instead of `1,000,000` — off by 10^12.
2. Peg views during a pause: pause Vault1. A frontend calls `validatePeg()` and receives `(false, 0, 0)` — if it renders those last two values as the Fund token balance and receipt supply, it shows `0 / 0` even though the real balances are non-zero.
3. At the same moment, calling `getPegStatus()` returns the real balances and `isPegged = true` (it never checks `paused()`). So the two views disagree — one says "0/0, not valid," the other says "pegged" — exactly while the vault is paused and something is already wrong.

Source

- contracts/tokens/ZToken.sol, contracts/tokens/TempToken.sol (neither overrides `decimals()`, so both inherit the default of 18)
- contracts/vaults/Vault1.sol:351-359 (`validatePeg` returns `(false, 0, 0)` when the vault is paused — the 0s are a “not valid” signal, not the real balance and supply)
- contracts/vaults/Vault1.sol:362-367 (`getPegStatus` returns the live balance/supply and peg flag but ignores `paused()`, so it can report “pegged” while the vault is actually paused — the two peg views can disagree)

Code:

```
// ZToken.sol / TempToken.sol — neither overrides decimals(), so both inherit the ERC20 default of 18,
// regardless of the underlying asset's decimals. (No decimals() function is declared in either file.)

// Vault1.sol:351–359 — validatePeg returns (false, 0, 0) while paused: the 0s are a sentinel, not real balances:
function validatePeg() public view returns (bool isValid, uint256 rwaBalance, uint256 receiptSupply) {
    if (paused()) {
        return (false, 0, 0);              // a reader that shows these as balances sees 0/0 during a pause
    }
    rwaBalance = rwaToken.balanceOf(address(this));
    receiptSupply = tempToken.totalSupply();
    isValid = (rwaBalance == receiptSupply);
    return (isValid, rwaBalance, receiptSupply);
}

// Vault1.sol:362–367 — getPegStatus ignores paused() entirely, so it can report isPegged == true while paused:
function getPegStatus() external view returns (uint256 rwaBalance, uint256 receiptSupply, bool isPegged) {
    rwaBalance = rwaToken.balanceOf(address(this));
    receiptSupply = tempToken.totalSupply();
    isPegged = (rwaBalance == receiptSupply);   // no paused() check — the two peg views can disagree
    return (rwaBalance, receiptSupply, isPegged);
}
```

Impact

Wallets/UIs/integrators can display or price amounts wrong — by a large factor if the underlying isn’t 18 decimals, or by showing zeroed/contradictory peg numbers during a pause. The protocol’s own accounting and all write functions are unaffected, since they never trust these views; the risk is entirely in what off-chain readers show.

Remediation

1. Override `decimals()` on both tokens to match the underlying asset, or clearly document that only 18-decimal assets are supported.

2. Make the two peg views consistent: either have `validatePeg` return the real balance/supply (and a separate `isValid`/`isPaused` flag) instead of zeroing them, or have `getPegStatus` also reflect `paused()`, so a reader can't get a "pegged while paused" answer or mistake the `validatePeg` sentinel zeros for real balances. At minimum, document that `validatePeg` returns `(false, 0, 0)` while paused and that `getPegStatus` does not account for pause.

Finding 13: The “max you can deposit/mint” functions claim an unlimited amount, but actually depositing near that amount reverts with an overflow — and “max you can withdraw” can report more than the vault will actually pay out

Severity: ● Info

Status: Resolved

Description

The ERC-4626 standard says `maxDeposit`/`maxMint` should return the largest amount you could actually deposit. Here they just return the maximum possible number — but if you tried to mint near that, the internal multiply overflows and reverts. So the “max” is a number the vault can’t actually honor. Separately, `maxWithdraw` reports a value without applying the same liquidity ceiling that the real withdraw enforces, so it can promise more than the vault will pay out when the recorded balance is higher than the real one. These are spec-conformance issues — tools and integrators that trust these “max” numbers get wrong answers.

Proof of Concept:

1. An integrator calls `maxMint(user)` (or `maxDeposit`). It returns `type(uint256).max` — implying any amount is mintable.
2. The integrator sizes a mint near that value and calls `mint`. Internally `previewMint` runs the amount through `Lib.convertToAssetsCeil`, whose `mulDiv` overflows at that magnitude, so the call reverts — the “max” was never actually honorable.
3. Separately, when `expectedBalance` is higher than the real token balance (e.g. the post-loss window), `maxWithdraw(owner)` returns the asset value of the owner’s shares with no liquidity ceiling — a number larger than `withdraw` will actually pay out. An integrator sizing a withdrawal from it submits a tx that reverts for lack of funds.

Source

- contracts/vaults/Vault2.sol:343-352 (`maxDeposit`/`maxMint` return the maximum possible number)
- contracts/vaults/Vault2.sol:211, 326-329 (but mint runs the amount through the conversion math)
- contracts/libraries/Lib.sol:59-66 (the conversion multiplies, which overflows near the max)

Code:

```
// Vault2.sol:343-352 - maxDeposit/maxMint just return the largest possible number (not a real, honorable limit):
function maxDeposit(address) public view override returns (uint256) {
    if (paused() || emergencyMode) return 0;
    return type(uint256).max;
}
function maxMint(address) public view override returns (uint256) {
    if (paused() || emergencyMode) return 0;
    return type(uint256).max;
}

// Vault2.sol:355-358 - maxWithdraw reports the asset value of the owner's shares with no liquidity ceiling,
// so it can promise more than withdraw() will actually pay out:
function maxWithdraw(address owner) public view override returns (uint256) {
    if (paused() || emergencyMode) return 0;
    return convertToAssets(IZToken(address(zToken)).balanceOf(owner));
}

// Lib.sol:59-66 - minting near type(uint256).max runs through this multiply, which overflows and reverts:
function convertToAssetsCeil(uint256 shares, uint256 supply, uint256 totalAssets) internal pure returns (uint256 assets)
     {
    if (supply == 0) return shares;
    return shares.mulDiv(totalAssets, supply, Math.Rounding.Ceil);   // overflow on shares ~ uint256.max
}
```

Impact

Integrators/aggregators that size a deposit or withdrawal from these “max” values get misled; the failure is a clean revert, no funds lost.

Remediation

1. Return realistic limits from `maxDeposit`/`maxMint` (values that won’t overflow the conversion at the current state), and have `maxWithdraw` reflect the same liquidity ceiling that withdraw actually enforces.
2. Fix the comments that present these as usable maximums. The NatSpec on `maxDeposit`/`maxMint` (contracts/vaults/Vault2.sol:342-352) reads as if the returned value is an amount you could actually deposit/mint, but the code returns the largest possible number, which reverts on overflow if used. State plainly that these return an unbounded sentinel and must not be passed straight into a deposit/mint, or change the code to return a real bound per step 1 so the comment becomes true.

Diff Audit — Vault1 Redemption Fee

A redemption fee was added to `Vault1` after the original audit closed. On every `withdraw` and `redeem`, a configurable percentage of the amount is now sent to the shared `FeeManager` and the remainder goes to the receiver. The existing rebase-yield fee was renamed from `fees` to `yieldFees` to distinguish the two. This section reviews that change.

Delta Commit	`7a5b1c21e25ccb4d16b9bea9fc6235e5a4d7b6aa` — “Added redemptionFee mechanism and renamed fee to yieldFee” (7 July 2026).
Scope	`Vault1.sol`, `IVault1.sol` and `Vault1Storage.sol` — 80 lines touched, 36 lines of new code. This review covers the change and the surrounding code it affects; the wider system is covered by the original audit.
Deployment	`Vault1` proxy `0x74D25113A93fF7Dd3dfA3ec00E750d2aE5AF32C8` on Ethereum Sepolia (chainId 11155111).
Remediation Commit	`11a0b75f0400b38a5a6b01d5418cfb632e2c8f72` — “Fixed auditing finding 14, 15 and 16” (5 August 2026).
Findings	Three: one Medium, one Low, one Info. All resolved.

The storage change is handled correctly: `redemptionFees` is appended after the existing variables and the reserved gap is reduced from 50 to 49, so the layout stays compatible for an in-place upgrade. The fee arithmetic itself is sound — the fee and the net amount always add back to the requested amount, and the balance-delta bookkeeping introduced for Finding 3 still holds with two transfers instead of one.

Note on the live deployment. The redemption fee is not yet live on Sepolia, and the deployed implementation predates this change, so no user was affected by any of the three findings. All three were fixed before deployment; each resolution below was verified against the remediation commit.

Finding 14: The vault's preview and max functions still quote pre-fee amounts, so integrators are told they will receive more than they do

Severity: ● Medium

Status: Resolved

Description

`withdraw` and `redeem` now deduct a fee before paying the receiver, but the ERC-4626 view functions were not updated to match. `previewRedeem` and `previewWithdraw` still quote amounts 1:1, `maxWithdraw` and `maxRedeem` still report the full receipt-token balance, and both exit functions still return the gross amount while transferring the net.

The practical consequence is that anything reading those views is told a number the user will not receive. ERC-4626 exists so that routers, aggregators, lending markets and accounting tools can quote a vault without reading its internals; the standard's contract is that `previewRedeem` does not overstate what will actually arrive, and that `withdraw(assets)` delivers `assets`. Both now hold only when the fee is zero.

Changing the `Withdraw` event to report the net amount is a reasonable choice on its own — it matches what the receiver actually got. The issue is narrower than the event: it is that the quoting surface and the settlement no longer agree, and nothing in the interface signals the gap.

Source

`contracts/vaults/Vault1.sol:374–381` — the previews are unchanged and fee-blind:

```
function previewWithdraw(uint256 assets) public pure returns (uint256) {
    return assets;
}

function previewRedeem(uint256 shares) public pure returns (uint256) {
    return shares;
}
```

`contracts/vaults/Vault1.sol:396–404` — the maximums report the full balance, with no allowance for the fee:

```
function maxWithdraw(address owner) public view returns (uint256) {
    if (paused()) return 0;
    return tempToken.balanceOf(owner);
}
```

`contracts/vaults/Vault1.sol:284–295` — the fee is deducted, the net is transferred, and the gross is returned:

```
uint256 feeAmount = (assets * redemptionFees) / BASIS_POINTS;
uint256 netAssets = assets - feeAmount;
...
rwaToken.safeTransfer(receiver, netAssets);
...
emit Withdraw(msg.sender, receiver, owner, netAssets, assets);
return assets;                    // gross, not netAssets
```

Impact

- An integrator quoting `previewRedeem` and crediting a user that amount over-credits them by the fee on every redemption.
- A caller that trusts the returned value rather than measuring its own balance records more than it received. Finding 15 is this exact case inside Zodiac's own router.
- `withdraw(assets)` no longer delivers `assets`. Integrations that size an exact-output withdrawal get a short fill with no error.
- `maxWithdraw` promises slightly more than the vault will pay out, so a withdrawal sized from it under-delivers rather than reverting.

Remediation

Make the quoting surface agree with settlement. The smallest change that achieves this:

```
function previewRedeem(uint256 shares) public view returns (uint256) {
    return shares - (shares * redemptionFees) / BASIS_POINTS;
}
```

`previewWithdraw` should return the gross receipt-token amount needed to net the requested `assets`, and `maxWithdraw` should report the post-fee amount. Returning `netAssets` from `withdraw` and `redeem` instead of the gross would also close Finding 15 at the same time, and is worth preferring for that reason. Whichever shape is chosen, documenting in `IVault1` that this vault charges a redemption fee helps integrators who read the interface before the implementation.

Resolution

Resolved at commit `11a0b75f`. The quoting surface now accounts for the fee: `previewWithdraw` returns the gross receipt amount needed to net the requested assets, while `previewRedeem` and `maxWithdraw` report post-fee figures. Settlement was aligned to match, with each exit delivering exactly what its quote promises and returning the corresponding amount. `maxRedeem` is correctly left at the full balance because it is denominated in shares, and the interface now documents the fee.

Finding 15: The router's minimum-payout check reads the vault's pre-fee figure, so a user can receive less than the minimum they asked for

Severity: ● Low

Status: Resolved

Description

`ZTokenRouter.redeemZToken` lets a caller set a floor on what they receive and is meant to revert if the payout falls below it. It compares that floor against the value `Vault1.redeem` returns, which is the amount before the fee. The user receives the amount after. So the payout can fall below the caller's floor without the check firing.

The router was not modified by this change. The behaviour follows from the return value described in Finding 14: the vault reports gross, the router treats that as delivered, and the fee sits invisibly between the two. Because the check compares a figure the user never receives, it permits the shortfall rather than blocking it — the failure is quiet rather than a revert.

This is the protection added for Finding 11 in the original review. That fix remains correct as written; the fee simply moved the number it was reading.

Source

`contracts/router/ZTokenRouter.sol` — `redeemZToken` bounds the returned value, not the received one:

```
1  rwaAmount = vault1.redeem(vrAmount, msg.sender, address(this));
2
3  if (rwaAmount < minRwaOut) revert Errors.SlippageExceeded();
4
5  emit RedeemedZToken(msg.sender, zTokenAmount, rwaAmount);
```

`contracts/vaults/Vault1.sol:335` — `redeem` returns the gross `assets` while transferring `netAssets` to the receiver.

Impact

- A caller who sets a floor can receive less than that floor, with the transaction succeeding. The size of the shortfall is the fee.

- The `RedeemedZToken` event reports the gross amount, so an off-chain consumer reconciling router activity records more than was paid out.
- Because the fee is currently zero and this commit is not yet deployed, no user has been affected. The protection stops holding the moment a non-zero fee goes live.

Remediation

Have the router measure what it actually received rather than trusting the reported figure:

```
uint256 balanceBefore = rwaToken.balanceOf(msg.sender);
vault1.redeem(vrAmount, msg.sender, address(this));
rwaAmount = rwaToken.balanceOf(msg.sender) - balanceBefore;

if (rwaAmount < minRwaOut) revert Errors.SlippageExceeded();
```

Alternatively, returning the net amount from `Vault1.redeem` and `Vault1.withdraw` fixes this without touching the router, and is the same change suggested in Finding 14 — one edit closes both.

Resolution

Resolved at commit `11a0b75f`. `redeemZToken` now records the caller's Fund token balance before the redemption and derives the payout from the measured difference, so `minRwaOut` bounds what the user actually receives rather than a reported figure. Both available fixes were applied — `Vault1.redeem` also returns the net amount now — so the check holds even if the two ever diverge again.

Finding 16: The redemption fee rounds away to nothing on very small withdrawals, where the yield fee carries the remainder forward

Severity: ● Info

Status: Resolved

Description

The redemption fee is calculated with plain integer division and no accumulator, so on a withdrawal small enough that the fee rounds to zero, no fee is charged and nothing is carried forward. The yield fee in the same contract does keep a running remainder for exactly this reason — it was added in response to Finding 10 of the original review. The redemption fee was written without it.

The practical exposure is very small and worth stating plainly. On the Fund token, which has 18 decimals, the fee only rounds away on amounts around a millionth of a trillionth of a token. Avoiding a meaningful fee this way would take more transactions than is remotely feasible, and the gas cost of each one exceeds the fee avoided by many orders of magnitude — at any withdrawal size, since both the fee and the number of pieces scale with the amount. There is no configuration of the current deployment where this is worth doing.

The finding is raised for two reasons rather than for revenue protection. The first is consistency: the same contract now handles the same rounding problem two different ways, which is the kind of divergence that gets copied forward. The second is portability. The threshold depends entirely on the asset's decimals, and `DeployNewAsset.s.sol` exists to point this vault at other assets. On an asset with very few decimals — some tokenised securities use two, or none — the amounts stop being negligible.

Source

`contracts/vaults/Vault1.sol:284` and `:324` — the redemption fee truncates with nothing retained:

```
uint256 feeAmount = (assets * redemptionFees) / BASIS_POINTS;
```

`contracts/vaults/Vault1.sol:466-468` — the yield fee, for contrast, carries the remainder into the next collection:

```
uint256 feeNumerator = rebaseAmount * yieldFees + feeRemainder;
uint256 feeAmount    = feeNumerator / BASIS_POINTS;
feeRemainder         = feeNumerator % BASIS_POINTS;
```

Impact

- On an 18-decimal asset such as the current Fund token, the uncollected amounts are too small to matter and the behaviour is not worth exploiting.
- On an asset with very few decimals, the same code would let a user avoid the fee by splitting a withdrawal, at a cost low enough to be worthwhile.
- Two different rounding conventions for two fees in one contract is a maintenance hazard independent of the amounts involved.

Remediation

Mirror the pattern already used for the yield fee, with a separate accumulator so the two fees do not interfere:

```
uint256 feeNumerator      = assets * redemptionFees + redemptionFeeRemainder;
uint256 feeAmount         = feeNumerator / BASIS_POINTS;
redemptionFeeRemainder    = feeNumerator % BASIS_POINTS;
```

`redemptionFeeRemainder` would be appended to storage and the reserved gap reduced accordingly, the same way `redemptionFees` was. If the Zodiac development team would rather not add state for an amount this small, then documenting that the redemption fee is not exact at very small amounts, and confirming the vault will only ever be used with high-decimal assets, is a reasonable alternative.

Resolution

Resolved at commit `11a0b75f`. A `redemptionFeeRemainder` accumulator now carries the truncated fee forward, mirroring the yield-fee treatment and kept deliberately separate so the two carries never borrow from each other. A withdrawal split into pieces now collects the same fee as a single one.

Disclaimer

This security report (“Report”) is provided by FailSafe (“Tester”) for the exclusive use of the client (“Client”). The scope of this assessment is limited to the security testing services performed against the systems, applications, or environments supplied by the Client. This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer, and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This Report, provided in connection with the Services set forth in the Agreement, shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This Report may not be transmitted, disclosed, referred to, or relied upon by any person for any purpose, nor may copies be delivered to any other person other than the Company, without FailSafe’s prior written consent in each instance.

This Report is not, nor should it be considered, an “endorsement” or “disapproval” of any particular project, system, or team. This Report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts FailSafe to perform security testing. This Report does not provide any warranty or guarantee regarding the absolute security or bug-free nature of the technology analyzed, nor does it provide any indication of the technology’s proprietors, business, business model, or legal compliance.

This Report should not be used in any way to make decisions around investment or involvement with any particular project. This Report in no way provides investment advice, nor should it be leveraged as investment advice of any sort. This Report represents an extensive testing process intended to help our customers identify potential security weaknesses while reducing the risks associated with complex systems and emerging technologies.

Technology systems, applications, and cryptographic assets present a high level of ongoing risk. FailSafe’s position is that each company and individual are responsible for their own due diligence and continuous security practices. FailSafe’s goal is to help reduce attack vectors and the high level of variance associated with utilizing new and evolving technologies, and in no way claims any guarantee of security or functionality of the systems we agree to test.

The security testing services provided by FailSafe are subject to dependencies and are under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. The testing process may include false positives, false negatives, and other unpredictable results. The services may access and depend upon multiple layers of third-party technologies.

ALL SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT

PERMITTED UNDER APPLICABLE LAW, FAILSAFE HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, TESTING REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, FAILSAFE SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, FAILSAFE MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE.

WITHOUT LIMITATION TO THE FOREGOING, FAILSAFE PROVIDES NO WARRANTY OR DISCLAIMER UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER FAILSAFE NOR ANY OF FAILSAFE'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. FAILSAFE WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, TESTING REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, TESTING REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO ANY OTHER PERSON WITHOUT FAILSAFE'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF FAILSAFE CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED TESTING REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.