Scanned Code Report

# AUDITAGENT

Powered by NETHERMIND SECURITY

## Code Info

Auditor Scan

| | |
|---|---|
| Scan ID<br>7 | Date<br>February 16, 2026 |
| Organization<br>StoneVault | Repository<br>stonevault |
| Branch<br>master | Commit Hash<br>9948f644...bad2879c |

## Contracts in scope

src/StoneVaultCore.sol
src/StoneZapIn.sol
src/StoneZapInManual.sol
src/StoneZapOut.sol
src/interfaces/ICurve.sol

## Code Statistics

| | | |
|---|---|---|
| Findings<br>9 | Contracts Scanned<br>5 | Lines of Code<br>2165 |

## Findings Summary


Total Findings

- High Risk (0)
- Medium Risk (1)
- Low Risk (8)
- Info (0)
- Best Practices (0)

## Code Summary

The Stone Vault protocol is a decentralized yield-aggregator designed to generate returns on a balanced portfolio of stablecoins. The core of the protocol is the `StoneVaultCore` contract, which functions as a vault accepting deposits of three specific stablecoins: DAI, LUSD, and crvUSD.

Upon deposit, the protocol allocates these assets to different established DeFi lending and yield protocols to optimize returns:
- DAI is supplied to Spark Lend.
- LUSD is supplied to the Aave V3 pool.
- crvUSD is deposited into a Curve ERC4626 vault.

In exchange for their deposit, users receive `SVT` (Stone Vault Token), an ERC20 token representing their share in the vault. The value of SVT is designed to appreciate over time as yield accrues from the underlying protocols. The deposit mechanism requires users to provide a balanced amount of all three stablecoins; the contract will only pull the minimum value supplied across the three tokens to maintain the portfolio's balance. A small fee of 0.01% is applied to both deposits and withdrawals. The protocol also includes an `emergencyWithdraw` function, allowing users to retrieve assets from the operational protocols even if one or more of the integrated platforms experience issues.

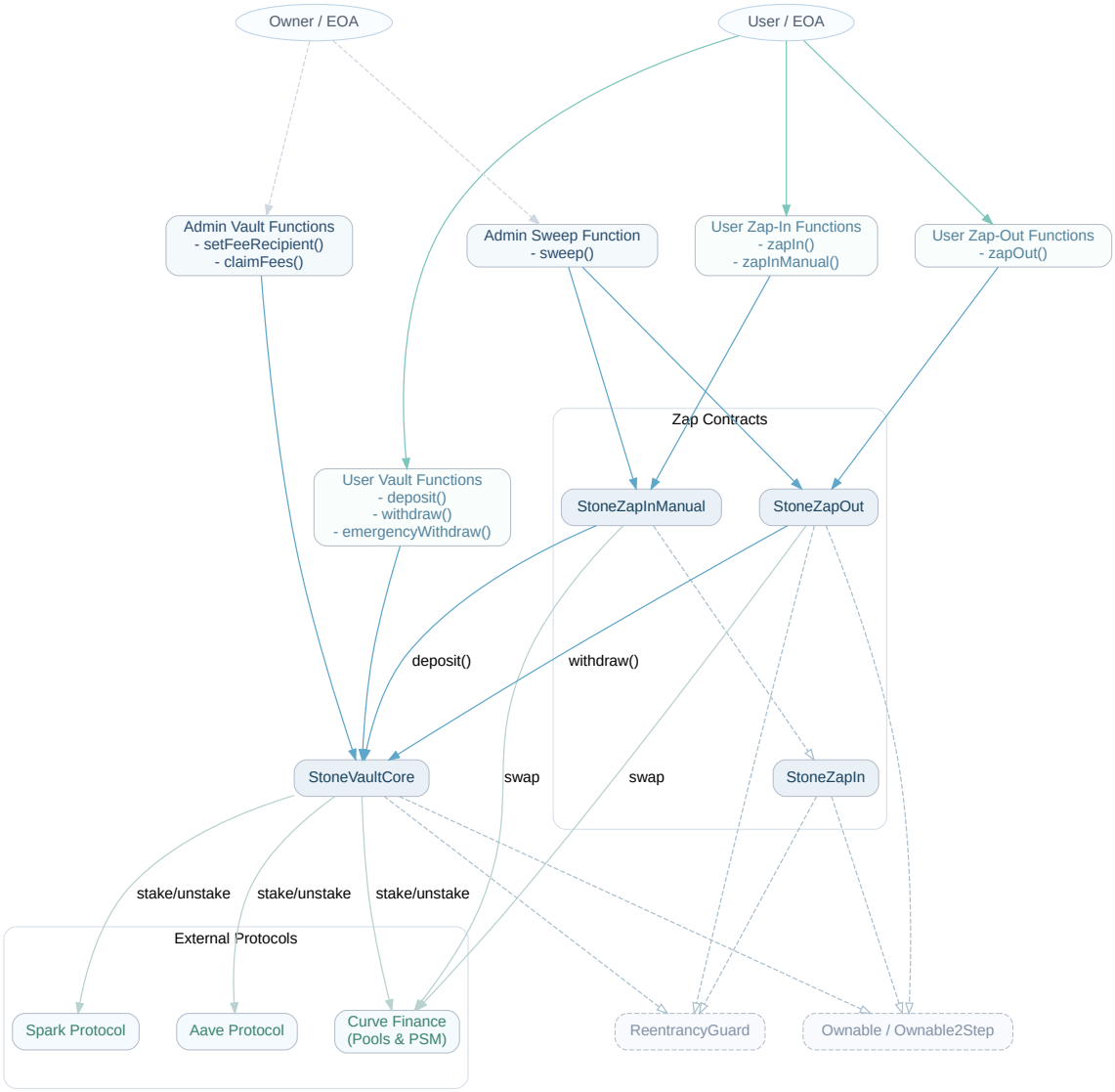
To enhance user experience, the protocol includes a suite of "zap" contracts:
- **`StoneZapIn`**: Simplifies the deposit process by allowing users to deposit a single stablecoin (e.g., USDC, USDT, DAI). The contract automatically handles the necessary swaps into the required DAI, LUSD, and crvUSD basket before depositing into the core vault.
- **`StoneZapInManual`**: An advanced version of the zap-in contract that gives users or front-ends the ability to specify custom swap routes and amounts, enabling optimizations to minimize slippage.
- **`StoneZapOut`**: Facilitates withdrawals by allowing users to burn their SVT tokens and receive their funds back as a single stablecoin of their choice, automating the reverse swap process.

### Entry Points & Actors

The primary actor interacting with the protocol is the **User**.

- `deposit(amounts, minSharesOut, receiver)`: A User deposits a balanced basket of DAI, LUSD, and crvUSD to mint SVT shares.
- `withdraw(shares, receiver, minAmountsOut)`: A User burns their SVT shares to redeem a proportional amount of the underlying stablecoins.
- `emergencyWithdraw(shares, receiver, skipProtocols)`: A User initiates a best-effort withdrawal, bypassing any specified protocols that may have failed.
- `zapIn(inputToken, amount, receiver, ...)`: A User deposits a single stablecoin, which is automatically swapped and deposited into the vault.
- `zapInManual(inputToken, amount, pathAmounts, ...)`: A User performs an optimized zap-in by specifying the swap distribution for the input token.
- `zapOut(shares, outputToken, receiver, ...)`: A User burns SVT shares to withdraw and receive the proceeds as a single, specified stablecoin.

## Code Diagram



1 of 9 Findings

src/StoneVaultCore.sol

**Final emergencyWithdraw can burn totalSupply to zero while leaving protocol positions, permanently locking funds**

Medium Risk

In `emergencyWithdraw`, shares are burned unconditionally before (best-effort) external withdrawals, and failures are swallowed via `try/catch`. If the caller is redeeming the entire remaining share supply (i.e., `shares == totalSupply()`), any skipped protocol (via `skipProtocols`) or any protocol withdrawal failure will leave interest-bearing positions inside the vault while `totalSupply()` becomes zero.

Because all exit paths (`withdraw` / `emergencyWithdraw`) require burning shares and there is no admin rescue for protocol positions, any remaining `sparkDAI` / `aaveLUSD` / `scrvUSD` positions become permanently unrecoverable once `totalSupply()==0`.

Vulnerable flow:

```
// amounts for some protocols may be 0 due to user-chosen skips or runtime failures

// 3. BURN shares IMMEDIATELY
_burn(msg.sender, shares);

// 5. Try to withdraw from each protocol (best-effort)
// failures are caught and converted into skips
try ISparkLendPool(sparkLend).withdraw(...) { ... } catch { effectiveSkips |= 1; }
try IAavePool(aavePool).withdraw(...) { ... } catch { effectiveSkips |= 2; }
try scrvUSD.withdraw(...) { ... } catch { effectiveSkips |= 4; }
```

Concrete exploit / failure scenario:
- Assume a bank-run leaves a single holder with all remaining shares (`shares == totalSupply()`), while at least one protocol is paused/failing (or the user sets `skipProtocols` for any reason).
- The holder calls `emergencyWithdraw(shares=totalSupply(), receiver, skipProtocols)`.
- Shares are burned to zero supply, but one or more protocol positions remain because they were skipped or the external call reverted and was caught.
- Result: `totalSupply()==0` while `_getSparkValue()` and/or `_getAaveValue()` and/or `_getCurveValue()` is still non-zero, leaving assets locked in external protocols with no remaining shares to redeem them.

Real-world impact:
- Permanent loss of the remaining protocol-held funds (cannot be withdrawn by anyone once supply is zero).
- The vault can be effectively “bricked” for future users because `totalAssets()` can remain > 0 while no shares exist, and there is no mechanism to unwind these protocol positions without shares.

This directly violates the project invariant that when `totalSupply() == 0` the vault must hold no interest-bearing positions.

Severity Note:
- A single holder can plausibly own all shares (bank-run or consolidation).
- At least one protocol withdraw can fail or the user can choose to skip it.
- No other external mechanism exists to pull assets from protocols besides share-burning withdraw paths.

2 of 9 Findings

src/StoneVaultCore.sol

## Incorrect Proportional Asset Calculation in Withdrawals Leads to Value Drain

Low Risk

The `withdraw` and `emergencyWithdraw` functions incorrectly calculate the proportional amount of each underlying asset a user is entitled to. The formulas add the `VIRTUAL_ASSETS_OFFSET` (1 wei) to each of the three asset components individually before determining the proportional amount based on the user's shares. The correct approach, consistent with ERC4626 principles, is to calculate the total value of assets a user's shares represent and then determine the proportional split of each underlying token. The current implementation calculates the total withdrawal value based on `totalAssets + 3` instead of the correct `totalAssets + 1` (the virtual asset offset). This error systematically allows users to withdraw slightly more assets than their shares represent, causing a slow but steady drain of value from the vault. This harms all remaining liquidity providers by decreasing the value of their shares over time and breaks a core invariant of the protocol.

Code snippet from the `withdraw` function showing the flawed calculation for each asset component:

```
amounts[0] = shares.mulDiv(
    _getSparkValue() + VIRTUAL_ASSETS_OFFSET, // Incorrectly adds offset here
    supply + 10 ** DECIMALS_OFFSET,
    Math.Rounding.Floor
);
amounts[1] = shares.mulDiv(
    _getAaveValue() + VIRTUAL_ASSETS_OFFSET, // And here
    supply + 10 ** DECIMALS_OFFSET,
    Math.Rounding.Floor
);
amounts[2] = shares.mulDiv(
    _getCurveValue() + VIRTUAL_ASSETS_OFFSET, // And here
    supply + 10 ** DECIMALS_OFFSET,
    Math.Rounding.Floor
);
```

The same flawed logic is present in the `emergencyWithdraw` function.

3 of 9 Findings

src/StoneVaultCore.sol

**Deposit can mint 0 SVT shares (and still pull/stake user funds) if the vault is pre-seeded with protocol tokens before first mint**

Low Risk

`deposit()` does not enforce that `shares > 0` before transferring funds and staking.

Shares are computed using `_convertToShares(totalDepositValue)`:

```
function _convertToShares(uint256 assets) internal view returns (uint256) {
    uint256 supply = totalSupply();
    uint256 totalAssets_ = totalAssets();
    return assets.mulDiv(
        supply + 10 ** DECIMALS_OFFSET,
        totalAssets_ + VIRTUAL_ASSETS_OFFSET,
        Math.Rounding.Floor
    );
}
```

If `totalSupply() == 0` while `totalAssets() > 0` (e.g., someone directly transfers `sparkDAI`, `aaveLUSD`, or `scrvUSD` tokens to the vault before any SVT is minted), the first depositor's shares become:

- `shares = assets * 1000 / (totalAssets + 1)` (floor)

With `MIN_DEPOSIT_AMOUNT_PER_TOKEN = 1e18`, the smallest allowed deposit has `assets ≈ 3e18` (three tokens), so `assets * 1000 ≈ 3e21`. If the vault has been pre-seeded to `totalAssets() > 3e21` (~3000 units with 18 decimals), the computed `shares` rounds down to `0`.

`deposit()` proceeds even when `shares == 0`:

```
shares = _convertToShares(totalDepositValue);
...
DAI.safeTransferFrom(msg.sender, address(this), depositPerToken);
LUSD.safeTransferFrom(msg.sender, address(this), depositPerToken);
CRVUSD.safeTransferFrom(msg.sender, address(this), depositPerToken);
...
_stakeAll(netAmounts);
_mint(receiver, shares);
```

This can result in a user's deposit being transferred in and staked while receiving 0 SVT, leaving the user with no redeemable balance (since `maxRedeem()` is `balanceOf(owner)`), effectively donating those funds to the vault state.

Severity Note:
- Caller sets minSharesOut = 0 (or a value below the actual computed shares).
- Vault is pre-seeded with enough aTokens/scrvUSD so that _convertToShares rounds to zero for the given deposit size.

4 of 9 Findings

src/StoneVaultCore.sol

### ERC4626 preview/limit mismatch can make SVT accounting overestimate Curve withdrawability and revert normal `withdraw()`

• Low Risk

SVT's accounting uses the Curve vault's `previewRedeem()` for valuation:

```
function _getCurveValue() internal view returns (uint256) {
    uint256 balance = scrvUSD.balanceOf(address(this));
    if (balance == 0) return 0;
    return scrvUSD.previewRedeem(balance);
}
```

But normal exits use an actual ERC4626 withdrawal that may be subject to limits/pauses/liquidity constraints in the external vault:

```
function _unstakeFromCurve(uint256 amount, address to) internal returns (uint256) {
    uint256 beforeBal = CRVUSD.balanceOf(to);
    scrvUSD.withdraw(amount, to, address(this));
    uint256 received = CRVUSD.balanceOf(to) - beforeBal;
    _enforceWithdrawReceived(2, amount, amount, received);
    return received;
}
```

Per ERC4626 semantics, preview functions can diverge from what is immediately withdrawable (e.g., if `maxWithdraw()` is lower than the requested amount, the vault is paused, or liquidity constraints exist). In that case, `_getCurveValue()` can overstate what can actually be withdrawn right now, causing `withdraw()` to compute a Curve-leg `amounts[2]` that the external `scrvUSD.withdraw(amounts[2], ...)` cannot satisfy and thus reverts.

This creates a situation where `totalAssets()`/pro-rata math indicates assets exist, but normal `withdraw()` can still fail due to external ERC4626 withdraw constraints.

Severity Note:
- scrvUSD conforms to ERC4626 and may enforce maxWithdraw or pause/redemption limits that cause withdraw() to revert.
- previewRedeem(balance) can report a value higher than what is immediately withdrawable.

5 of 9 Findings

src/StoneZapIn.sol

**slippageBps is not enforced on Maker Light PSM legs (sellGem/buyGem), allowing successful zapIn with unbounded loss vs user-selected slippage**

Low Risk

The zap routes that use Maker's Light PSM do not apply the user-provided `slippageBps` at all. Curve swaps are protected via `min_dy` computed from `_pegMinOut(...)`, but PSM swaps are executed without any minimum-out check and without verifying the received amount post-swap.

Vulnerable code paths:

```
// _zapInUSDC: USDC -> DAI via PSM (NO minOut enforcement)
uint256 daiBefore = DAI.balanceOf(address(this));
USDC.forceApprove(lightPsm, t1);
ILightPsm(lightPsm).sellGem(address(this), t1);
uint256 daiReceived = DAI.balanceOf(address(this)) - daiBefore;
```

```
// _zapInCRVUSD: USDC -> DAI via PSM (NO minOut enforcement)
USDC.forceApprove(lightPsm, usdcForDai);
ILightPsm(lightPsm).sellGem(address(this), usdcForDai);
uint256 daiReceived = DAI.balanceOf(address(this)) - daiBefore;
```

```
// _zapInDAI: DAI -> USDC via PSM (NO minOut / no check usdcReceived == usdcWanted)
uint256 usdcWanted = t3 / 1e12;
DAI.forceApprove(lightPsm, t3);
ILightPsm(lightPsm).buyGem(address(this), usdcWanted);
uint256 usdcReceived = USDC.balanceOf(address(this)) - usdcBefore;
```

Exploit scenario (realistic under config changes / adverse PSM conditions):
1) A user calls `zapIn(USDC, amount, receiver, minSharesOut=0, slippageBps=50, ...)` expecting each swap to respect ~0.5% max loss.
2) The PSM applies a large fee/spread (e.g., governance increases `tout` / `tin`, or the PSM implementation otherwise returns less than a 1:1 peg).
3) The USDC->DAI leg executes anyway (no `minOut` parameter and no post-check against `_pegMinOut`).
4) The transaction can still succeed and mint shares (especially if `minSharesOut` is set too low), but the user has paid an arbitrarily large hidden loss on the PSM portion while the function still advertises slippage protection via `slippageBps`.

Impact:
- Users can experience losses that exceed `slippageBps` while still getting a successful `zapIn` (no revert), i.e., the slippage guarantee is partially illusory.
- This is particularly dangerous because many users set `minSharesOut` to 0 (or too low), relying primarily on `slippageBps` as their safety bound.
- Breaks the stated invariant that "for every internal Curve/PSM swap, tokens received are at least `_pegMinOut(...)`".

Severity Note:
- Maker Light PSM fees/spread (tin/tout) can be increased by Maker governance beyond typical ~0% values.
- Users may set minSharesOut to 0 or too low, relying mainly on slippageBps for per-leg protection.
- buyGem semantics deliver exactly gemAmt of USDC on success and require sufficient DAI including fees; otherwise the call reverts.

6 of 9 Findings  src/StoneZapIn.sol  src/StoneZapInManual.sol

## Anyone can steal any pre-existing DAI/LUSD/crvUSD held by the zap contract via the public “unused token refund”

Low Risk

The zap contracts unconditionally transfer *all* remaining balances of DAI/LUSD/crvUSD to the user-chosen `receiver` after a vault deposit, without isolating/refunding only the amounts created by the current zap call.

```
function _returnUnusedTokens(address receiver) internal {
    uint256 daiBalance = DAI.balanceOf(address(this));
    uint256 lusdBalance = LUSD.balanceOf(address(this));
    uint256 crvusdBalance = CRVUSD.balanceOf(address(this));

    if (daiBalance > 0) {
        DAI.safeTransfer(receiver, daiBalance);
    }
    if (lusdBalance > 0) {
        LUSD.safeTransfer(receiver, lusdBalance);
    }
    if (crvusdBalance > 0) {
        CRVUSD.safeTransfer(receiver, crvusdBalance);
    }
}
```

Because the function uses the contract’s full token balances (not “balance delta” attributable to the current caller), **any** DAI/LUSD/crvUSD that exists on the zap contract for any reason (e.g., a user mistakenly transfers tokens directly to the zap contract address, or tokens are airdropped/sent as “rescue funds”) becomes claimable by the next caller.

Concrete exploit scenario:
1) A victim accidentally transfers 100,000 DAI to `StoneZapInManual` (or `StoneZapIn`) directly.
2) An attacker observes this balance on-chain.
3) The attacker calls `zapInManual(..., receiver=attacker)` (or `zapIn`) with a small but valid amount that succeeds through `vault.deposit(...)`.
4) After the vault call, `_returnUnusedTokens(attacker)` executes and transfers **the entire** DAI/LUSD/crvUSD balance on the zap contract—including the victim’s mistakenly sent 100,000 DAI—to the attacker.

Impact:
- Direct theft of any DAI/LUSD/crvUSD that ends up on the zap contract outside of the current caller’s intended flow.
- This effectively bypasses the intended “only owner can sweep residual tokens” expectation for these three assets, since the public zap functions can be used as an unrestricted sweep mechanism by choosing `receiver`.

Severity Note:
- A third party mistakenly sends DAI/LUSD/CRVUSD to the zap contract address outside the normal zap flow.
- The zap call path remains valid so that vault.deposit succeeds and triggers the refund.

7 of 9 Findings  src/StoneZapIn.sol  src/StoneZapOut.sol

## `_pegMinOut()` rounds down before applying slippage for 18->6 conversions, weakening min-out protection (can reach 0 for tiny legs)

Low Risk

Both `StoneZapIn` and `StoneZapOut` compute `min_dy` via `_pegMinOut()` using integer division when converting from 18 decimals to 6 decimals:

```
function _pegMinOut(
    uint256 amountIn,
    uint8 fromDec,
    uint8 toDec,
    uint256 bps
) internal pure returns (uint256) {
    uint256 fairQuote;
    if (fromDec <= toDec) {
        fairQuote = amountIn * 10 ** (toDec - fromDec);
    } else {
        fairQuote = amountIn / 10 ** (fromDec - toDec);
    }
    return (fairQuote * (BPS_DENOMINATOR - bps)) / BPS_DENOMINATOR;
}
```

For 18->6, this effectively does:
- `fairQuote = floor(amountIn / 1e12)` (truncate first)
- then applies bps: `minOut = floor(fairQuote * (10000 - bps) / 10000)`

This ordering loses up to 1 unit of the 6-decimal token relative to an equivalent computation that applies bps before the final downscale (because the truncated remainder `< 1e12` is discarded before slippage is applied).

Additionally, when `fairQuote == 1` and `bps > 0`, `minOut` becomes 0 (since `floor(1 * (10000-bps) / 10000) == 0`). That happens whenever `amountIn` is in `[1e12, 2e12)` (i.e., 0.000001 to <0.000002 of an 18-decimal token).

These 18->6 min-outs are used across multiple swap legs (examples):

```
// StoneZapOut: DAI -> USDT (18->6)
ICurve3Pool(curve3Pool).exchange(0, 2, daiAmt, _pegMinOut(daiAmt, 18, 6, slippageBps));

// StoneZapOut: LUSD -> USDC (18->6)
ICurveLusd(curveLusdPool).exchange_underlying(0, 2, lusdAmt, _pegMinOut(lusdAmt, 18, 6,
slippageBps));

// StoneZapIn: LUSD -> USDC (18->6)
ICurveLusd(curveLusdPool).exchange_underlying(0, 2, t3, _pegMinOut(t3, 18, 6, slippageBps));
```

Therefore, for 18->6 conversions the computed `min_dy` is systematically rounded down (and can become 0 for very small swap amounts), meaning the enforced per-leg minimum can be weaker than the nominal bps setting by an amount attributable purely to integer truncation.

8 of 9 Findings

src/StoneZapOut.sol

## DAI dust left in contract when zapping out to USDC or crvUSD due to 18→6 decimal truncation

Low Risk

In zapOut, when the output token is USDC or crvUSD, the DAI leg is converted to USDC via the Maker PSM using a truncated 18→6 conversion. The code computes usdcWanted = _to6(daiAmt) (integer division by 1e12) and then buys exactly usdcWanted USDC, approving up to the full daiAmt. Any remainder daiAmt - (usdcWanted * 1e12) is left behind on the contract and never swapped or sent to the receiver. This happens in both paths:

_zapOutToUSDC:
uint256 usdcWanted = _to6(daiAmt);
DAI.forceApprove(lightPsm, daiAmt);
ILightPsm(lightPsm).buyGem(address(this), usdcWanted);

_zapOutToCRVUSD:
uint256 usdcWanted = _to6(daiAmt);
DAI.forceApprove(lightPsm, daiAmt);
ILightPsm(lightPsm).buyGem(address(this), usdcWanted);

Because usdcWanted floors to 6 decimals, unless daiAmt is an exact multiple of 1e12 wei, a positive DAI remainder (up to 1e12−1 wei ≈ 1e−6 DAI) remains on the contract after swaps. The final transfer uses amountOut = balanceAfter(outputToken) − balanceBefore(outputToken), so this leftover DAI is not sent to the receiver. Consequently, the contract's DAI balance increases after a successful zapOut to USDC or crvUSD, violating the stated post-conditions that balances remain unchanged or within a dust threshold. Typical remainders (≤1e12−1 wei) are far above the DUST_THRESHOLD_18 = 1000 wei in the invariant, so the breach is observable in practice. This breaks invariants requiring zero net token accumulation by the zap-out helper (indexes 0, 1, 3, 4).

9 of 9 Findings

src/StoneZapInManual.sol

**Manual path bounds can become infeasible for small `amount` due to integer truncation, causing guaranteed reverts**

Low Risk

`zapInManual()` enforces that each of the three `pathAmounts[i]` lies within a min/max range derived from `amount` using integer division:

```
uint256 minPath = (amount * MIN_PATH_BPS) / 10000;
uint256 maxPath = (amount * MAX_PATH_BPS) / 10000;
if (
    pathAmounts[0] < minPath || pathAmounts[0] > maxPath ||
    pathAmounts[1] < minPath || pathAmounts[1] > maxPath ||
    pathAmounts[2] < minPath || pathAmounts[2] > maxPath
) {
    revert PathImbalanceTooHigh();
}
```

Because `minPath`/`maxPath` are truncated, for sufficiently small `amount` values the bounds can collapse to a single integer value, making it impossible to satisfy both:
1) `pathAmounts[0] + pathAmounts[1] + pathAmounts[2] == amount`, and
2) each `pathAmounts[i]` within `[minPath, maxPath]`.

Concrete example (as stated by the hypothesis):
- `amount = 4`
- `minPath = floor(4 * 2833 / 10000) = 1`
- `maxPath = floor(4 * 3833 / 10000) = 1`

So each `pathAmounts[i]` must equal 1, but then the sum is `1 + 1 + 1 = 3 != 4`, and all calls revert.

This creates rounding-driven “infeasible windows” where `zapInManual()` is unusable for certain small inputs purely due to precision truncation in the bound computation.

## Disclaimer

Kindly note, no guarantee is being given as to the accuracy and/or completeness of any of the outputs the AuditAgent may generate, including without limitation this Report. The results set out in this Report may not be complete nor inclusive of all vulnerabilities. The AuditAgent is provided on an 'as is' basis, without warranties or conditions of any kind, either express or implied, including without limitation as to the outputs of the code scan and the security of any smart contract verified using the AuditAgent.

Blockchain technology remains under development and is subject to unknown risks and flaws. This Report does not indicate the endorsement of any particular project or team, nor guarantee its security. Neither you nor any third party should rely on this Report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset.

To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this Report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.