// Security Assessment

03.25.2025 - 04.04.2025

Nawa Finance
Coredao

HALBORN

Nawa Finance - Coredao

Prepared by: HALBORN

Last Updated 04/02/2025

Date of Engagement: March 25th, 2025 - April 4th, 2025

Summary

0% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
38	1	5	4	5	23

TABLE OF CONTENTS

1. Summary

2. Introduction

`CoreDAO` engaged our security analysis team to conduct a comprehensive security assessment of their smart contract ecosystem. The primary aim was to meticulously assess the security architecture of the smart contracts to pinpoint vulnerabilities, evaluate existing security protocols, and offer actionable insights to bolster security and operational efficacy of their smart contract framework. Our assessment was strictly confined to the smart contracts provided, ensuring a focused and exhaustive analysis of their security features.

3. Assessment Summary

Our engagement with `CoreDAO` spanned an 8 day period, during which we dedicated one full-time security engineer equipped with extensive experience in blockchain security, advanced penetration testing capabilities, and profound knowledge of various blockchain protocols. The objectives of this assessment were to:

- Verify the correct functionality of smart contract operations.

- Identify potential security vulnerabilities within the smart contracts.

- Provide recommendations to enhance the security and efficiency of the smart contracts.

4. Test Approach And Methodology

Our testing strategy employed a blend of manual and automated techniques to ensure a thorough evaluation. While manual testing was pivotal for uncovering logical and implementation flaws, automated testing offered broad code coverage and rapid identification of common vulnerabilities. The testing process included:

- A detailed examination of the smart contracts' architecture and intended functionality.

- Comprehensive manual code reviews and walkthroughs.

- Functional and connectivity analysis utilizing tools like Solgraph.

- Customized script-based manual testing and testnet deployment using Foundry.

This executive summary encapsulates the pivotal findings and recommendations from our security assessment of `CoreDAO` smart contract ecosystem. By addressing the identified issues and implementing the recommended fixes, `CoreDAO` can significantly boost the security, reliability, and trustworthiness of its smart contract platform.

5. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

5.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker’s control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

5.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

5.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9

SEVERITY	SCORE VALUE RANGE
Low	2 - 4.4
Informational	0 - 1.9

6. SCOPE

Out-of-Scope: New features/implementations after the remediation commit IDs.

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	5	4	5	23

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - INCOMPATIBLE STRATEGY INTERFACE USAGE BREAKS REGISTRY INTROSPECTION	CRITICAL	-
HAL-02 - INCONSISTENT FEE DEDUCTION LOGIC IN SWAP FUNCTIONS	HIGH	-
HAL-03 - UNRELIABLE WITHDRAWAL FLOW AND PARTIAL FUND DELIVERY RISK	HIGH	-
HAL-04 - STRATEGY UPGRADE ALLOWS INCOMPATIBLE WANT TOKENS LEADING TO ASSET LOSS	HIGH	-

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-05 - INCOMPLETE ACCESS CONTROL AND INCONSISTENT ALLOWANCE HANDLING ON CORE LIQUIDITY FUNCTIONS	HIGH	-
HAL-06 - PANIC FAILS DUE TO PREMATURE PAUSE AND REMOVED ALLOWANCES	HIGH	-
HAL-07 - REFERRER CAN BE SELF OR OVERWRITTEN ON REPEATED DEPOSITS	MEDIUM	-
HAL-08 - UNSAFE DIRECT TOKEN TRANSFER	MEDIUM	-
HAL-09 - EARN DOES NOT FORWARD FUNDS INTO ACTIVE STRATEGY	MEDIUM	-
HAL-10 - ZAPIN DOES NOT ACCOUNT FOR DEFLATIONARY TOKENS OR TRANSFER FEES	MEDIUM	-
HAL-11 - MISSING REENTRANCY PROTECTION ON EXTERNAL CALL PATHS	LOW	-

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-12 - UNNECESSARY AND UNSAFE REPEATED ALLOWANCE GRANTS WITHOUT PROPER REVOCATION	LOW	-
HAL-13 - MISSING TWO-STEP OWNERSHIP TRANSFER AND UNSAFE RENOUNCEOWNERSHIP	LOW	-
HAL-14 - UNRESTRICTED STRATEGIST ASSIGNMENT CAN LEAD TO INVALID ROLE CONFIGURATION	LOW	-
HAL-15 - ADDING A VAULT ALLOWS DUPLICATED ENTRIES WITHOUT VALIDATION	LOW	-
HAL-16 - UNREACHABLE AND UNUSED PAUSE AND PANIC FUNCTIONALITY	INFORMATIONAL	-
HAL-17 - REFERRER CAN BE SELF OR OVERWRITTEN ON REPEATED DEPOSITS	INFORMATIONAL	-
HAL-18 - RESCUE ALLOWS WITHDRAWAL OF CORE STRATEGY ASSET	INFORMATIONAL	-

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-19 - MISSING UPPER BOUND VALIDATION IN SETREVSHAREFEES	INFORMATIONAL	-
HAL-20 - ZERO APPROVALDELAY ALLOWS IMMEDIATE STRATEGY UPGRADE	INFORMATIONAL	-
HAL-21 - ADMIN FUNCTIONALITY NOT USED	INFORMATIONAL	-
HAL-22 - REDUNDANT OVERFLOW CHECK IN TOUINT24	INFORMATIONAL	-
HAL-23 - REDUNDANT USE OF SAFEMATH WITH SOLIDITY 0.8.X	INFORMATIONAL	-
HAL-24 - INCONSISTENT SLIPPAGE DENOMINATOR PRECISION ACROSS CONTRACTS	INFORMATIONAL	-
HAL-25 - UNNECESSARY PROPOSEDTIME ASSIGNMENT IN UPGRADESTRAT	INFORMATIONAL	-

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-26 - MISLEADING DEV COMMENT ON REFERRER SETTER FUNCTION	INFORMATIONAL	-
HAL-27 - CONFUSING WITHDRAWAL FLAG NAMING AND LOGIC	INFORMATIONAL	-
HAL-28 - UNUSED CORETOKEN VARIABLE CREATES ASSET TYPE AMBIGUITY	INFORMATIONAL	-
HAL-29 - UNACCOUNTED ETH RECEIVED OUTSIDE OF DEPOSIT FLOW	INFORMATIONAL	-
HAL-30 - REDUNDANT UNBONDING CHECKS AND DUPLICATED WITHDRAWAL TRACKING	INFORMATIONAL	-
HAL-31 - UNBOUNDED APPROVE TO VAULT EXPOSES DUALCORETOKEN TO EXTERNAL DRAIN	INFORMATIONAL	-
HAL-32 - INCORRECT WITHDRAWAL FALLBACK CHECK CAN LEAD TO ETH TRANSFER FAILURE	INFORMATIONAL	-

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-33 - REDUNDANT INHERITANCE AND UNUSED FEE LOGIC IN STRATEGY	INFORMATIONAL	-
HAL-34 - MISLEADING COMMENT IN HARVEST REGARDING LP BALANCE LOGGING	INFORMATIONAL	-
HAL-35 - UNUSED STATE VARIABLES INCREASE CONTRACT SIZE AND REDUCE CLARITY	INFORMATIONAL	-
HAL-36 - MISSING SLIPPAGE LIMIT VALIDATION IN CONSTRUCTOR	INFORMATIONAL	-
HAL-37 - ZAPIN DOES NOT VALIDATE VAULT COMPATIBILITY WITH LP TOKEN	INFORMATIONAL	-
HAL-38 - NO PRICE-PER-SHARE LOGIC ENABLES YIELD DILUTION VIA 1:1 MINTING	INFORMATIONAL	-

8. FINDINGS & TECH DETAILS

8.1 (HAL-01) INCOMPATIBLE STRATEGY INTERFACE USAGE BREAKS REGISTRY INTROSPECTION

// CRITICAL

Description

In the `NawaRegistry` contract, the `vaultsInfo` function relies on the assumption that all registered strategies conform to the `INawaStrategyV2` interface, which includes functions such as `wantUnderlyingToken()` and `revShareToken()` .
However, the actual strategies used across the system implement various other interfaces (`IStrategy` , `IDualCoreStrategy` , `StratManager`) and **do not implement** the `wantUnderlyingToken()` or `revShareToken()` functions. As a result, when `vaultsInfo` attempts to query strategy metadata, it will **revert at runtime**, breaking off-chain integrations, registry queries, or any UI components relying on this data.
This renders the registry introspection logic unreliable and introduces a silent failure risk when new vaults are added—if the strategy does not conform, the registry will accept it, but then fail when trying to read from it.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (10.0)

Recommendation

Standardize all strategies to implement a common metadata interface like `INawaStrategyV2` , or refactor `vaultsInfo` to dynamically detect capabilities using `try/catch` patterns or interface flags. If not all strategies are expected to support the same metadata, the registry should:
1. **Validate interface compliance** before calling the functions.
2. Fallback to optional metadata fields where appropriate.
3. Maintain a registry version or vault type flag to differentiate strategy implementations.
Alternatively, modify the registry to store static metadata (e.g. `revShareToken` , `underlyingToken`) during vault registration to avoid querying untrusted or incompatible contracts at runtime.

8.2 (HAL-02) INCONSISTENT FEE DEDUCTION LOGIC IN SWAP FUNCTIONS // HIGH

Description

The `NawaRouter` contract implements protocol-level fees across its swap functions, but inconsistently applies the fee logic based on the swap type. This misalignment leads to swap failures, slippage inaccuracies, and broken constant-product invariants in AMM pools.
For *exact input* swaps, such as `swapExactTokensForTokens` or `swapExactETHForTokens` , the function calculates the `amounts` array using the full `amountIn` , but then subtracts the fee before transferring tokens to the pair. Since `getAmountsOut` assumes the full `amountIn` reaches the pool, reducing it after the fact leads to insufficient liquidity provisioning, which may result in output values below `amountOutMin` , failing slippage checks or producing less than expected tokens.
For ***exact output*** swaps, such as `swapTokensForExactTokens` or `swapTokensForExactETH` , the required input is calculated using `getAmountsIn` , returning the precise amount the pool needs. However, the contract subtracts the fee from this required amount, leading to an underfunded pool that cannot fulfill the output target. The fee should be added **on top** of the required input, not subtracted from it.
This problem extends to `SupportingFeeOnTransferTokens` variants, where inputs such as `msg.value` or `amountIn` are used for swap expectations, but the fee is deducted **after** or at the wrong point in logic, breaking the swap flow or producing unintended outcomes.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:M (8.8)

Recommendation

Refactor all swap functions in the `NawaRouter` contract to consistently apply fee logic based on the swap type:

- For exact input swaps, subtract the fee from the input ***before*** computing the output.
- For exact output swaps, calculate the required input and add the fee on top.
- Ensure that `TransferHelper` calls or ETH wrapping/unwrapping reflect the actual amount sent to the pair, and that all `amountOutMin` or `amountInMax` checks are made based on the post-fee effective input.

This correction guarantees alignment with AMM expectations and preserves slippage guarantees across all routing paths.

8.3 (HAL-03) UNRELIABLE WITHDRAWAL FLOW AND PARTIAL FUND DELIVERY RISK

// HIGH

Description

The `NawaVaultV2`, `NawaVaultUpgradable`, and `NawaVault` contracts all share a common withdrawal pattern that is vulnerable to underdelivery from the strategy. In the `_withdraw` logic, the user's entitled withdrawal amount `r` is computed as:

```
r = (balance() * _shares) / totalSupply;
```

If the vault does not hold enough tokens to fulfill this amount (`b < r`), it attempts to withdraw the difference from the strategy. After the external call, it determines how much was actually received:

```
uint256 _diff = balance().sub(b);
```

If the returned amount is less than requested, the vault silently adjusts `r` to a lower value:

```
r = b.add(_diff);
```

This behavior introduces several reliability and safety concerns:
1. **User Underpayment**: The user receives fewer tokens than their calculated share without any visibility or tracking of the shortfall. The logic assumes it's acceptable to return a lesser amount, but the comment `// CHECK THIS` correctly flags that this is a potentially dangerous and non-transparent behavior.
2. **Silent Strategy Failure**: No validation is performed to ensure the strategy fulfills its obligation. A malfunctioning, misconfigured, or malicious strategy can underdeliver with no consequences or alert, shifting the burden to the vault and harming user expectations.
3. **SafeTransfer Reverts**: If the vault still doesn't hold enough tokens to complete the transfer—even after attempting to pull from the strategy—the `safeTransfer` to the user will fail, reverting the transaction. This may lead to stuck withdrawals with no graceful fallback.
This problem is particularly amplified in the `BitfluxStrategy`, where the `withdraw` logic intentionally performs:

```
uint256 amount = Math.min(requestedAmount, poolBalance);
```

This means the strategy may return **less than the requested withdrawal**, and it leaves it up to the vault to decide how to handle that shortfall. If the vault doesn't reject underdelivery or track the deficit, users are silently underpaid or blocked entirely.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:H (8.1)

Recommendation

Enforce reliable withdrawal mechanics through the following improvements:

1. **Require full delivery by default**: Revert when the strategy fails to provide the required amount:

```
require(_diff >= _withdrawAmount, "Strategy underdelivered");
```

1. **Track underdelivery if partial withdrawals are intended**: If the design supports partial fulfillment, explicitly track the shortfall per user and offer a secondary mechanism to claim the remainder later.
2. **Make behavior explicit**: If partial withdrawals are allowed, document this clearly and ensure the front-end reflects it. Use events to log the shortfall transparently.
3. **In `BitfluxStrategy`**, avoid using `Math.min()` for withdraw amounts unless the vault supports partial delivery. Instead, use functions like `calculateRemoveLiquidityOneToken` to pre-validate the exact amount that the pool will return and ensure that it can fulfill the expected amount or revert otherwise.
4. **Introduce optional withdrawal thresholds**: Allow vaults to define a `minExpected` threshold below which a withdrawal automatically fails, reducing edge case behavior.

These changes ensure users are not shortchanged due to strategy-level failures and enforce clearer boundaries between the vault's logic and the responsibilities of the strategy. Without this, the vault operates on an implicit trust model that weakens its guarantees.

8.4 (HAL-04) STRATEGY UPGRADE ALLOWS INCOMPATIBLE WANT TOKENS LEADING TO ASSET LOSS

// HIGH

Description

In `NawaVaultV2` , `NawaVaultUpgradable` , and `NawaVault` , the `proposeStrat` and `upgradeStrat` flow and the `setStrategy` on `SolvVault` contract does not verify that the new strategy's `want()` token matches the vault's current `want()` token. Although the contract assumes compatibility by calling `strategy.migrate()` after transferring the full balance of the old `want()` to the new strategy, this behavior is not enforced or validated.

This leads to a critical flaw: **if the new strategy operates on a different `want()` token**, and the vault still transfers the previous `want()` balance to it, those funds will be locked inside the new strategy with no recovery mechanism—especially since none of the currently deployed strategies implement custom `migrate()` logic to handle mismatched tokens.

The vulnerability arises from the implicit assumption that:

1. The new strategy will either operate on the same `want` token or
2. Implement a custom `migrate()` function that handles conversion, forwarding, or recovery of the old token.

However, none of the strategies in scope implement a `migrate()` function with this behavior, violating the assumption and enabling potential permanent asset loss if a strategy is upgraded to one with a different underlying asset.

This problem is exacerbated by the fact that `upgradeStrat` :

- Calls `retireStrat()` on the old strategy (to handle cleanup),
- Transfers **all** vault-held `want()` tokens to the new strategy unconditionally,
- Calls `migrate()` on the new strategy.

If the new strategy uses a different `want()` token or fails to correctly forward or recover the old one, these funds are permanently stuck.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (7.5)

Recommendation

Introduce a strict compatibility check in the `proposeStrat` or `upgradeStrat` function to ensure the new strategy's `want()` token matches the current one:

```
require(IStrategy(_implementation).want() == want(), "want mismatch");
```

Alternatively, or in addition:

- Modify the `migrate()` interface to accept the previous `want` token as an argument, enabling explicit handling of non-matching tokens.
- Require all strategy implementations to handle incompatible `want()` tokens by transferring the old `want()` funds back to a recovery address (e.g., the vault) or another approved recovery handler.
- In the vault logic, if the new strategy has a different `want`, do **not** transfer the old `want` to the strategy. Instead, keep the funds within the vault and rely on the existing `rescueTokens()` functionality, which already disallows reclaiming the current `want()` but can recover other tokens.

For `SolvVault` and `CoreVault` contracts consider implementing a `propose` / `ugprade` system or make sure the new `want()` is the same as the old strategy `want()` .
By implementing one or more of these safeguards, the upgrade path becomes resistant to misconfigured or malicious strategies and avoids scenarios in which vault assets become irrecoverably locked.

8.5 (HAL-05) INCOMPLETE ACCESS CONTROL AND INCONSISTENT ALLOWANCE HANDLING ON CORE LIQUIDITY FUNCTIONS

// HIGH

Description

In the `BitfluxStrategy` contract, the `deposit` function lacks an access control check to ensure that only the vault can trigger it. This contrasts with the `withdraw` function, which properly enforces `require(msg.sender == vault, ...)` . As a result, **any external actor can call `deposit()`**, initiating liquidity operations on the strategy's behalf and potentially affecting vault accounting.

Furthermore, both `addLiquidity` and `removeLiquidity` are public and callable by any address. This introduces several problematic scenarios:

- **Arbitrary liquidity movements**: Any external user can deposit assets into the `bitfluxPool` by calling `addLiquidity` or withdraw them via `removeLiquidity`, including specifying arbitrary withdrawal amounts.
- **Broken liquidity flow**: A malicious or careless caller can:
  - Call `removeLiquidity()` to withdraw LP funds,
  - Follow up with `deposit()` or `addLiquidity()` to return them,
  - Repeatedly toggle the state of pool exposure, leading to operational unpredictability or external manipulation if pool behavior (e.g. yield, inflation) reacts to participation duration or timing.
- **No allowance handling in public liquidity functions**: `addLiquidity` and `removeLiquidity` do not internally call `_giveAllowances` , which can cause revert failures if approvals are not already set. This contradicts the pattern seen in other parts of the strategy and makes direct function calls unreliable.

The combination of open access, missing token approvals, and missing vault verification leads to a fragile and potentially exploitable design surface, even if the economic incentive to do so is low.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (7.5)

Recommendation

Harden the liquidity and execution flow with the following changes:

1. **Restrict `deposit()`** to the vault with a `require(msg.sender == vault)` check, aligning it with `withdraw()`.
2. **Make `addLiquidity()` and `removeLiquidity()` internal or onlyVault** if they are meant to be part of vault-controlled flows. If external exposure is necessary, wrap them with proper authorization.
3. **Ensure allowance safety**: If these functions remain callable, invoke `_giveAllowances()` inside them or explicitly require that approvals are already in place. Consistency is critical to avoid runtime reverts.
4. **Protect against liquidity manipulation**: If calling `addLiquidity()` or `removeLiquidity()` inappropriately could alter yield distributions, inflation calculations, or harvest logic, enforce stricter control to avoid malicious timing attacks or flash deposit cycles.

These fixes will enforce proper separation of control, improve predictability of the strategy's asset flow, and reduce surface area for unintended interactions.

8.6 (HAL-06) PANIC FAILS DUE TO PREMATURE PAUSE AND REMOVED ALLOWANCES

// HIGH

Description

In the `BitfluxStrategy` contract, the `panic()` function is designed to execute an emergency exit by withdrawing funds from the pool and halting strategy operations. However, it calls `pause()` before performing the withdrawal:

```
function panic() external onlyVault {
    pause();
    removeLiquidity();
}
```

The `pause()` function invokes `_removeAllowances()` , which sets token allowances (including for LP tokens) to zero. As a result, when `removeLiquidity()` is executed immediately after, the `bitfluxPool.removeLiquidityOneToken()` call fails, because the pool cannot transfer the required LP tokens due to the revoked allowance.

This makes the `panic()` function unusable in practice—it will always revert when attempting to remove liquidity in emergency conditions, precisely when it is needed most.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:N/Y:N (7.5)

Recommendation

Reorder the operations in `panic()` to preserve the necessary allowances until after liquidity is withdrawn:

```
function panic() external onlyVault {
    removeLiquidity(); // withdraw while allowances are active
    pause();           // then revoke allowances
}
```

This ensures the strategy can successfully exit its positions before disabling further interactions. Alternatively, if revoking allowances pre-withdrawal is a security requirement, the function must temporarily re-grant them, though this adds unnecessary complexity. Emergency flows must be reliable and self-contained—ensuring that they don't rely on assumptions invalidated by earlier steps. This correction restores `panic()` 's intended behavior and secures user funds during protocol-wide shutdowns.

8.7 (HAL-07) REFERRER CAN BE SELF OR OVERWRITTEN ON REPEATED DEPOSITS

// MEDIUM

Description

In the `NawaVaultV2` contract, the `_deposit` function allows users to set a `_referrer` address during deposit without enforcing that it must differ from `msg.sender` . This permits users to set themselves as their own referrer, which is likely unintended and may result in meaningless or exploitative referral data in `NawaReferrerV2` .

Additionally, the `user.referrer` value is overwritten on every deposit, regardless of whether a referrer was previously recorded. This behavior contradicts standard referral mechanisms, where the referrer is typically locked on the first deposit. As a result, users may cycle through different referrers across multiple deposits, leading to inconsistent or misleading rev-share tracking, particularly when `revShareEnabled` is true and external accounting functions are triggered via `_recordDepositRevShare` or `_recordWithdrawRevShare` .

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

Add a check to ensure `_referrer` is not equal to `msg.sender` to prevent self-referrals. To preserve the integrity of the referral system, consider locking the referrer address on the user's first deposit and prevent it from being overwritten on subsequent deposits. This will ensure consistent tracking for revenue-sharing and avoid state corruption or gaming of the referral system.

8.8 (HAL-08) UNSAFE DIRECT TOKEN TRANSFER

// MEDIUM

Description

In both `NawaVault` and `NawaVaultUpgradable`, the `upgradeStrat` function performs a direct ERC20 `transfer` when moving tokens from the vault to the newly approved strategy. This approach does not utilize `safeTransfer`, which wraps the call and verifies the return value, reverting on failure. Some non-compliant or proxy-wrapped tokens (including BNB derivatives or fee-on-transfer tokens) may return false or no value at all without reverting, causing silent failures and inconsistent asset accounting.
Using raw `transfer` makes the upgrade process vulnerable to undetected token loss, leaving funds stuck in the vault or in limbo during a strategy upgrade.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

Replace all direct `token.transfer` calls within `upgradeStrat` with `SafeERC20.safeTransfer` to ensure proper low-level handling and reversion on failure. This change guarantees that only successful transfers allow the strategy upgrade to proceed and protects user funds during the handoff between strategies.

8.9 (HAL-09) EARN DOES NOT FORWARD FUNDS INTO ACTIVE STRATEGY // MEDIUM

Description

In the `NawaVaultUpgradable` contract, the `earn()` function is meant to transfer available vault funds to the strategy for yield generation. However, the current implementation only transfers the tokens to the strategy address:

```
IERC20Upgradeable(want()).safeTransfer(strategy, bal);
```

It does **not** follow up with a call to `strategy.deposit()`, which is required to actually deploy the funds within the strategy. As a result, when `earn()` is called directly (e.g. by a keeper or automated bot), the funds remain idle on the strategy contract rather than being actively put to work. This breaks the intended compounding mechanism and leads to reduced capital efficiency.
When the full flow is triggered from `deposit()`, the strategy does receive the `deposit()` call, ensuring the funds are utilized. However, `earn()` is intended to be an explicit optimization hook, and its incomplete behavior introduces inconsistency and economic inefficiency.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:M/Y:N (5.0)

Recommendation

Update the `earn()` function to include a call to `strategy.deposit()` immediately after transferring the funds:

```
IERC20Upgradeable(want()).safeTransfer(strategy, bal);
IStrategy(strategy).deposit();
```

This ensures consistent behavior between direct and internal flows, and guarantees that all funds sent to the strategy are immediately activated. If the goal is to decouple fund transfer and strategy logic for modularity, it should be made explicit via naming (e.g. `flushToStrategy()` vs `earn()`), and access should be controlled accordingly.

8.10 (HAL-10) ZAPIN DOES NOT ACCOUNT FOR DEFLATIONARY TOKENS OR TRANSFER FEES

// MEDIUM

Description

In the `BitfluxZap` contract, the `zapIn` function assumes that the full `_amount` specified by the user will be received by the contract when executing:

```
TransferHelper.safeTransferFrom(tokenIn, msg.sender, address(this), _amount);
```

Immediately afterward, it uses `_amount` directly when calling `addLiquidity`, assuming that the full value is available. However, if `tokenIn` is a deflationary or fee-on-transfer token, the contract will receive **less than `_amount`**, causing one of two issues:
1. **Underfunding**: If the received amount is less than expected and `addLiquidity` attempts to use the original `_amount`, it may revert due to insufficient balance.
2. **Overdraw from leftovers**: If the contract holds surplus of the same token from prior operations, the discrepancy may be silently covered using leftover balance, causing another user's excess tokens to be used for a new user's operation.
This leads to unreliable behavior, potential cross-user fund leakage, and inconsistencies in how zap operations are executed across token types.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

Capture the actual amount received using token balance deltas before and after the transfer:

```
uint256 before = IERC20(tokenIn).balanceOf(address(this));
TransferHelper.safeTransferFrom(tokenIn, msg.sender, address(this), _amount);
uint256 after = IERC20(tokenIn).balanceOf(address(this));
uint256 actualReceived = after - before;
```

Then use `actualReceived` for all liquidity logic instead of the original `_amount`. This ensures compatibility with deflationary tokens and prevents the contract from unintentionally relying on pre-existing balances to fulfill user interactions.

8.11 (HAL-11) MISSING REENTRANCY PROTECTION ON EXTERNAL CALL PATHS // LOW

Description

In the `CoreVault` , `NawaVaultV2` , and `SolvZap` contracts, multiple externally accessible functions interact with external contracts without using a `nonReentrant` modifier or equivalent protection, exposing the system to reentrancy risks. These risks may not originate directly from the vaults or zaps themselves but from the behavior of the connected contracts or tokens—especially in scenarios involving misbehaving strategies, wrapped assets, or external protocol integrations.
In `CoreVault` , functions such as `deposit` , `initiateUnbond` , `withdraw` , and `withdrawDirect` interact directly with strategy contracts and update state variables. If the connected strategy or token behaves maliciously or re-enters the vault, these functions may be called again before internal state is safely finalized, leading to potential double-spending or accounting inconsistencies.
In `NawaVaultV2` , the `withdrawZap` function performs complex interactions involving token transfers and potentially zap logic, but lacks the `nonReentrant` protection found on its base `withdraw` function. This inconsistency increases the risk that certain withdrawal paths are exposed to unexpected external behaviors.
In the `SolvZap` contract, the functions `zapInWithWBTC()` and `zapInWithSolvBTCb()` perform external calls to third-party protocols such as `bitfluxPool.swap()` or `solvStaking.createSubscription()` . If any of these contracts behave in a reentrant fashion—intentionally or due to internal logic bugs—there is no guard in place to prevent reentrant access to state or logic within the zap contract.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

Apply the `nonReentrant` modifier to all relevant external functions that:

- Perform external calls (e.g., `transfer`, `approve`, `deposit`, external protocol invocations).
- Update or rely on shared contract state that could be re-entered or manipulated mid-execution.

Specifically:

- In `CoreVault`: Add `nonReentrant` to `deposit`, `initiateUnbond`, `withdraw`, and `withdrawDirect`.
- In `NawaVaultV2`: Add `nonReentrant` to `withdrawZap` for consistency and protection.

- In `SolvZap`: Add `nonReentrant` to `zapInWithWBTC` and `zapInWithSolvBTCb`, as these call untrusted external systems.

For the `earn()` function in `NawaVaultV2`, although it doesn't change vault state, it sends tokens to a strategy and calls `strategy.deposit()`. To minimize exposure in the event of a malicious or broken strategy, consider either restricting this function to a trusted role (e.g., `onlyOwner`, `onlyKeeper`) or protecting it with `nonReentrant`.
These changes ensure consistent enforcement of call protection across all high-risk execution paths and reduce the possibility of indirect reentrancy vulnerabilities arising from connected contracts.

8.12 (HAL-12) UNNECESSARY AND UNSAFE REPEATED ALLOWANCE GRANTS WITHOUT PROPER REVOCATION

// LOW

Description

In the `BitfluxStrategy` contract, the `_giveAllowances` function is invoked multiple times across various functions after the constructor, including during routine execution of core actions like deposits and withdrawals. However, this is redundant and inefficient, as both `want()` and `LP_TOKEN()` are immutable references without setters, and under the standard ERC20 implementation, once an allowance is set to `type(uint256).max` , the allowance remains at maximum and does **not decrease** on `transferFrom` unless explicitly changed.
The repeated calls to `_giveAllowances()` introduce unnecessary gas consumption on each call path, with no functional gain, especially since the approvals are already permanently set during the constructor. Furthermore, the approval mechanism does not include any corresponding calls to `_removeAllowances()` after operations are completed, which leaves the allowances at their maximum regardless of intent—this undermines any security argument for dynamic approval management.
In scenarios where security is a concern (e.g., a potentially unsafe or upgradable `bitfluxPool`), the strategy would benefit from **temporary** approvals. However, in the current implementation, this logic is incomplete and misleading:

- `_giveAllowances` is declared `public`, making it callable by any external actor. This violates encapsulation and exposes the contract to external manipulation of approval state.
- There is no `_removeAllowances` invoked post-operation, meaning approvals are never revoked.
- The approval is overly broad: e.g., the deposit logic grants allowance to both `want` and `LP_TOKEN`, even though depositing does not require use of the `LP_TOKEN`.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

Refactor the allowance logic for clarity, efficiency, and security:
1. **Restrict `_giveAllowances` visibility** to `internal` or `private` to prevent external abuse.

2. **Avoid redundant approvals** in core execution paths—if `want` and `LP_TOKEN` do not change and allowances are set in the constructor, no further action is needed.
3. If dynamic approval is truly desired for risk isolation, implement a `grantAllowance` modifier that wraps sensitive operations:

```
modifier grantAllowance() {
    _giveAllowances();
    _;
    _removeAllowances();
}
```

1. Apply **granular approvals** only for tokens relevant to the operation—e.g., only approve `want` during `deposit`, not `LP_TOKEN`.
2. Keep `_giveAllowances` only in `pause` and `unpause`, where reinitializing allowance makes operational sense.

This approach maintains security boundaries, eliminates redundant gas usage, and clarifies the intent of token approval management.

8.13 (HAL-13) MISSING TWO-STEP OWNERSHIP TRANSFER AND UNSAFE RENOUNCEOWNERSHIP

// LOW

Description

Multiple contracts in the system rely on the `Ownable` pattern to control privileged operations, but currently allow single-step ownership transfers via `transferOwnership`. This introduces the risk of accidentally or maliciously setting an incorrect or invalid address as the new owner, potentially rendering the contract permanently inaccessible from a governance perspective.
Additionally, the inherited `Ownable` implementation exposes the `renounceOwnership` function, which permanently removes the owner. If unintentionally called, this would leave the contract without any privileged access, disabling upgrades, recovery paths, or administrative controls, depending on the role of ownership in the specific contract.
This is particularly critical in upgradeable vaults and strategy contracts where ownership governs sensitive operations such as pausing, upgrading strategies, or rescuing funds.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (2.0)

Recommendation

Replace the current ownership transfer mechanism with a two-step pattern (`proposeOwner` and `acceptOwnership`) to ensure the new owner explicitly accepts the role before the transfer is finalized. Additionally, override and disable `renounceOwnership` to prevent accidental removal of the owner role unless explicitly intended by governance design.

8.14 (HAL-14) UNRESTRICTED STRATEGIST ASSIGNMENT CAN LEAD TO INVALID ROLE CONFIGURATION

// LOW

Description

In the `StratManager` contract, the `setStrategist` function is callable by the current `strategist`, allowing them to unilaterally update the role. However, this opens the door to misconfiguration, such as assigning the strategist role to an invalid address (e.g. `address(0)`), either maliciously or by mistake. Since there are no access guards or validation on the new address, this may render the strategist role unusable or break off-chain integrations relying on this role.
Moreover, the strategist address appears to be unused within the contract itself, making the role effectively inert and raising questions about its relevance. If the role is intended to have operational authority in the future or in derived contracts, this weakness should be addressed preemptively.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (2.0)

Recommendation

Restrict the `setStrategist` function to `onlyOwner` and validate the input to prevent assignment of invalid addresses:

```
function setStrategist(address _strategist) external onlyOwner {
    require(_strategist != address(0), "Invalid address");
    strategist = _strategist;
}
```

If the role is not being used or planned for future use, consider removing the strategist variable and function entirely to reduce unnecessary surface area.

8.15 (HAL-15) ADDING A VAULT ALLOWS DUPLICATED ENTRIES WITHOUT VALIDATION

// LOW

Description

In the `NawaRegistry` contract, the `addVaults` function appends new vault addresses to an internal array without checking for duplicates. This allows the same vault to be registered multiple times, leading to inconsistent or redundant state when retrieving vault data or iterating through the registry.
This design flaw can cause UI issues, unnecessary data inflation, or misinterpretation of vault statistics. Additionally, there is no gas efficiency benefit to allowing duplicates, and failing to prevent them may complicate off-chain consumers or future registry logic.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (2.0)

Recommendation

Prevent duplicate vault entries by using a `mapping(address => bool)` to track registered vaults, or by adopting `EnumerableSet` from OpenZeppelin, which provides an efficient and reliable way to manage unique sets of addresses.
For example:

```
EnumerableSet.AddressSet private _vaultSet;

function addVaults(address[] calldata vaults) external onlyOwner {
    for (uint256 i = 0; i < vaults.length; i++) {
        if (_vaultSet.add(vaults[i])) {
            // Only add to array if not already present
            _vaultArray.push(vaults[i]);
        }
    }
}
```

This ensures vaults are registered once and simplifies downstream consumption of registry data.

8.16 (HAL-16) UNREACHABLE AND UNUSED PAUSE AND PANIC FUNCTIONALITY

// INFORMATIONAL

Description

In the `CoreVault` , `SolvStrategy` , and `SolvVault` contracts, functions related to emergency handling such as `pause()` , `unpause()` , and `panic()` are defined in the strategies but never called by their respective vaults. Additionally, the `paused` state variable is not referenced in any of the contract logic, meaning there is no actual behavioral difference between paused and unpaused states.
As implemented, this renders the pause functionality non-functional (dead code), and emergency functions like `panic()` —which are designed to withdraw funds back to the vault—cannot be triggered in practice. This is particularly problematic in a live environment where mechanisms for responding to unexpected strategy failures, bugs, or external exploits must be callable in a timely and deterministic manner.
Furthermore, without proper linkage from the vault to the strategy, the access control guarding `panic` / `pause` (e.g. `onlyVault`) is ineffective, and the emergency mechanisms remain inaccessible even to privileged actors.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:M/I:H/D:N/Y:N (1.8)

Recommendation

Implement proper hooks in the `CoreVault` , `SolvVault` , and any other owning vault contracts to expose `pause()` , `unpause()` , and `panic()` functionality on their respective strategies. Additionally:

- Ensure the `paused` state is actually enforced in relevant functions (e.g. block deposits, earnings, or withdrawals if paused).
- Consider adding `onlyOwner`, `onlyGovernance`, or `onlyEmergencyAdmin` role checks to vault-level pause triggers if fine-grained access control is needed.

This makes the emergency controls operational, enforceable, and aligned with the intended design for fault tolerance and safety.

8.17 (HAL-17) REFERRER CAN BE SELF OR OVERWRITTEN ON REPEATED DEPOSITS

// INFORMATIONAL

Description

In the `NawaVaultV2` contract, the `_deposit` function allows users to set a `_referrer` address during deposit without enforcing that it must differ from `msg.sender`. This permits users to set themselves as their own referrer, which is likely unintended and may result in meaningless or exploitative referral data in `NawaReferrerV2`.
Additionally, the `user.referrer` value is overwritten on every deposit, regardless of whether a referrer was previously recorded. This behavior contradicts standard referral mechanisms, where the referrer is typically locked on the first deposit. As a result, users may cycle through different referrers across multiple deposits, leading to inconsistent or misleading rev-share tracking, particularly when `revShareEnabled` is true and external accounting functions are triggered via `_recordDepositRevShare` or `_recordWithdrawRevShare`.

BVSS

AO:S/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:U (1.0)

Recommendation

Add a check to ensure `_referrer` is not equal to `msg.sender` to prevent self-referrals. To preserve the integrity of the referral system, consider locking the referrer address on the user's first deposit and prevent it from being overwritten on subsequent deposits. This will ensure consistent tracking for revenue-sharing and avoid state corruption or gaming of the referral system.

8.18 (HAL-18) RESCUE ALLOWS WITHDRAWAL OF CORE STRATEGY ASSET

// INFORMATIONAL

Description

In the `DualCoreStrategy` contract, the `rescueToken` function allows the contract owner to withdraw any ERC20 token from the strategy. However, there is no restriction preventing the withdrawal of `dualCoreToken` , which is the primary asset managed by the strategy. Allowing `dualCoreToken` to be withdrawn arbitrarily bypasses the strategy's staking and unbonding mechanisms and poses a serious custodial risk.

If the owner mistakenly or maliciously calls `rescueToken(dualCoreToken, ...)` , they could remove funds that are actively bonded, awaiting unbonding, or otherwise committed in the strategy's flow. This undermines user trust and could disrupt accounting both in the vault and in the external `dualCoreVault` .

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.0)

Recommendation

Add a conditional check in the `rescueToken` function to prevent the withdrawal of `dualCoreToken` :

```
require(_token != address(dualCoreToken), "Cannot rescue core token");
```

This ensures that only unrelated or accidentally sent tokens can be rescued, and the strategy's core asset remains protected under the proper withdrawal flow.

8.19 (HAL-19) MISSING UPPER BOUND VALIDATION IN SETREVSHAREFEES

// INFORMATIONAL

Description

In both the `FeeManager` and `StratFeeManager` contracts, the `setRevShareFees` function allows fee values to be set without enforcing an upper bound. This opens the possibility for misconfiguration or abuse, where the fee value could exceed 100% of the allocated revenue, leading to incorrect or excessive fee deductions.
Each contract uses a fixed `FEE_DIVISOR` (e.g., 1e3 or 1e4) as the denominator for fee calculations, but there is no check to ensure the configured fee is ≤ `FEE_DIVISOR` . As a result, a value higher than the divisor could produce unintended outcomes or revert downstream logic when distributing rewards or fees.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.0)

Recommendation

Add a validation in `setRevShareFees` to enforce that the fee value is not greater than `FEE_DIVISOR` . For example:

```
require(_fee <= FEE_DIVISOR, "Fee exceeds maximum allowed");
```

This ensures predictable fee behavior, prevents misconfiguration, and upholds consistency across fee-bearing contracts.

8.20 (HAL-20) ZERO APPROVALDELAY ALLOWS IMMEDIATE STRATEGY UPGRADE

// INFORMATIONAL

Description

In the `NawaVaultV2`, `NawaVaultUpgradable`, and `NawaVault` contracts, the `approvalDelay` parameter is passed to the constructor without requiring it to be non-zero. This value defines the minimum delay between calling `proposeStrat` and executing `upgradeStrat`. If `approvalDelay` is initialized to zero, a malicious or compromised strategist can bypass the intended upgrade delay protection by immediately proposing and upgrading a new strategy within the same transaction or block.

The delay mechanism is intended to serve as a time-based safeguard against rushed or malicious upgrades, giving governance or observers time to react. A zero value effectively disables this protection.

BVSS

AO:S/AC:L/AX:L/R:P/S:U/C:N/A:L/I:N/D:N/Y:N (0.3)

Recommendation

Require a minimum non-zero value for `approvalDelay` during contract construction to enforce a meaningful delay window for strategy upgrades. For example, reject values below a minimum threshold (e.g., `>= 1 day`) in the constructor to ensure the upgrade mechanism cannot be executed instantly.

8.21 (HAL-21) ADMIN FUNCTIONALITY NOT USED

// INFORMATIONAL

Description

The `NawaRouter` contract declares an `admin` variable that is never used throughout the entire contract logic. This creates unnecessary state storage and can mislead developers or auditors into assuming there's administrative control or restricted functionality tied to this variable. Moreover, it may imply a false security model where no such privilege exists in practice, or it could indicate incomplete or deprecated functionality that was never properly removed.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Remove the unused `admin` variable from the `NawaRouter` contract to reduce bytecode size, clarify contract intent, and avoid confusion around administrative control. If the variable is reserved for future use, document it explicitly and protect it with access control logic.

8.22 (HAL-22) REDUNDANT OVERFLOW CHECK IN TOUINT24

// INFORMATIONAL

Description

The `BytesLib` library includes a `require(_start + 3 >= _start, 'toUint24_overflow')` check in the `toUint24` function. This check is intended to catch overflows in the addition of `_start + 3`. However, the contract uses Solidity version `>=0.8.0`, which includes built-in overflow checking for arithmetic operations by default. As such, this manual check is redundant and has no functional effect beyond unnecessary gas consumption.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Remove the redundant `require(_start + 3 >= _start, 'toUint24_overflow')` check in `toUint24` from the `BytesLib` library. Solidity 0.8+ will automatically revert on overflow, and simplifying the code improves readability and saves gas.

8.23 (HAL-23) REDUNDANT USE OF SAFEMATH WITH SOLIDITY 0.8.X

// INFORMATIONAL

Description

Contracts such as `NawaVaultV2` and others across the codebase make extensive use of the `SafeMath` library for arithmetic operations. However, Solidity versions `>=0.8.0` natively include built-in overflow and underflow protection, making the use of `SafeMath` unnecessary. This redundancy adds minor bytecode bloat and decreases code readability without providing any additional security benefits.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Remove the `SafeMath` import and its usage across all contracts compiled with Solidity 0.8.x or above. Native arithmetic operations (e.g. `a + b`, `a - b`, `a * b`) are safe and revert on overflow/underflow by default in these versions.

8.24 (HAL-24) INCONSISTENT SLIPPAGE DENOMINATOR PRECISION ACROSS CONTRACTS

// INFORMATIONAL

Description

The contracts `SolvZap`, `BitfluxZap`, and `FeeManager` define different denominator values for slippage and fee precision calculations. For instance, one uses `SLIPPAGE_DENOMINATOR = 1e4`, while another uses `1e3`, and `FeeManager` also uses `1e3` as its basis for percentage logic. This inconsistency introduces unnecessary complexity and increases the risk of misinterpretation or misconfiguration when reading, maintaining, or integrating with these contracts.
Varying precision standards across contracts also create the potential for logic errors when shared parameters or configurations are passed between systems (e.g. a 50 basis point slippage tolerance being interpreted as 0.5% or 5% depending on the denominator used).

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Standardize the denominator used for slippage and fee calculations across all contracts (preferably to `1e4`, representing basis points) to ensure consistency, reduce ambiguity, and prevent subtle arithmetic bugs. Update associated logic to align with the chosen precision level, and document the format clearly in each relevant contract.

8.25 (HAL-25) UNNECESSARY PROPOSEDTIME ASSIGNMENT IN UPGRADESTRAT

// INFORMATIONAL

Description

In both `NawaVault` and `NawaVaultUpgradable`, the `upgradeStrat` function assigns a zero value to `stratCandidate.proposedTime` after the strategy upgrade is completed. However, this reset operation is redundant because the check `stratCandidate.implementation != address(0)` is already used to control the upgrade flow and prevent re-execution. Since resetting the timestamp has no functional effect beyond that point, it incurs unnecessary gas costs and clutters the upgrade logic.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Remove the `stratCandidate.proposedTime = 0` assignment from the `upgradeStrat` function. The logic controlling strategy replacement is already safeguarded by the `implementation` field, making this assignment redundant and inefficient.

8.26 (HAL-26) MISLEADING DEV COMMENT ON REFERRER SETTER FUNCTION

// INFORMATIONAL

Description

In the `NawaVaultV2` contract, the comment above the setter function for updating the `NawaReferrer` contract incorrectly states:

```
/// @dev Ability to change the zap address
```

However, the function modifies the reference to the `NawaReferrerV2` contract, not a zap address. This can cause confusion for developers, auditors, and integrators, especially in a system involving multiple zap-related components. Misleading comments reduce code maintainability and increase the risk of misconfiguration.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Update the in-code comment to accurately describe the function's behavior. For instance, change the line to:

```
/// @dev Ability to change the NawaReferrer contract address
```

This improves clarity and avoids confusion between zap logic and referral logic.

8.27 (HAL-27) CONFUSING WITHDRAWAL FLAG NAMING AND LOGIC

// INFORMATIONAL

Description

In the `CoreVault` contract, the boolean `canWithdraw` flag controls whether a user can withdraw funds after initiating an unbonding process. However, the current naming and logic are counterintuitive:

- `canWithdraw` is set to `false` when the user calls `initiateUnbond`, despite that action initiating the withdrawal process.
- The `withdraw` function allows withdrawals only if `canWithdraw` is `false`, using the condition `require(!request.canWithdraw, ...)`.
- After a successful withdrawal, `canWithdraw` is then set to `true`.

This inverted logic is misleading, as the variable name suggests that `true` should enable withdrawals, but the implementation treats `false` as the signal to proceed. This naming mismatch introduces unnecessary confusion and increases the risk of errors.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Either rename the flag to `hasWithdrawn` to reflect its actual semantics, or restructure the logic so that `canWithdraw` being `true` correctly signals eligibility to withdraw. Both options improve clarity and reduce the cognitive load for reviewers and developers.

8.28 (HAL-28) UNUSED CORETOKEN VARIABLE CREATES ASSET TYPE AMBIGUITY

// INFORMATIONAL

Description

The `CoreVault` contract declares an immutable variable `coreToken` of type `IERC20Upgradeable`, suggesting that the vault is intended to manage ERC20 tokens. However, both the `deposit` and `withdraw` functions operate using native ETH:

- `deposit` is marked as `payable` and uses `msg.value` to handle incoming funds.
- `withdraw` uses a low-level `call` to transfer ETH back to the user.

This inconsistency introduces ambiguity regarding the vault's intended asset model. The presence of an ERC20 `coreToken` implies a token-based system, yet no interactions with it exist in the logic. This can confuse developers, integrators, and auditors about the actual type of asset held, and may lead to integration bugs, incorrect assumptions about ERC20 approvals, or misdirected user funds. Additionally, the unused `coreToken` variable increases the contract's storage size unnecessarily and may be misleading during audits or upgrades.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Clarify the asset model by either removing the unused `coreToken` variable if the vault is strictly designed to operate with native ETH, or fully integrate it into the deposit and withdrawal logic if the vault is intended to work with ERC20 tokens. Consistency between the declared state and operational logic is essential to avoid misinterpretation and design-level bugs.

8.29 (HAL-29) UNACCOUNTED ETH RECEIVED OUTSIDE OF DEPOSIT FLOW

// INFORMATIONAL

Description

The `CoreVault` contract includes a `receive()` function that allows the contract to accept ETH directly. However, when ETH is sent via this fallback mechanism, it bypasses the `deposit()` function entirely. As a result, no user accounting is performed, no shares are minted, and no internal state is updated.

This design flaw creates a scenario where ETH can be unintentionally or maliciously sent to the contract and becomes unaccounted for. Since the vault does not track these funds, they are effectively stuck—neither attributed to any user nor recoverable through standard withdrawal mechanisms.

Moreover, the presence of a `receive()` function implicitly signals to users and integrators that direct ETH transfers are supported, which contradicts the vault's intended operational flow.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Restrict the use of the `receive()` function by enforcing that only the strategy contract (or other known internal address) may send ETH directly to the vault, or otherwise revert all unsolicited transfers:

```
receive() external payable {
    require(msg.sender == address(strategy), "Direct ETH transfer not allowed");
}
```

Alternatively, remove the `receive()` function entirely if the strategy does not send ETH back via low-level transfers, forcing all ETH inflows to go through the `deposit()` function, which handles user accounting properly.

This ensures consistency, prevents stuck funds, and avoids user confusion or integration errors.

8.30 (HAL-30) REDUNDANT UNBONDING CHECKS AND DUPLICATED WITHDRAWAL TRACKING

// INFORMATIONAL

Description

In the `CoreVault` contract, the `withdraw` function enforces an internal unbonding delay using a hardcoded `UNBONDING_PERIOD` and a local `unbondRequests` mapping to track user withdrawal eligibility. However, if the vault is integrated with `DualCoreStrategy` , the strategy already maintains its own unbonding logic via `pendingWithdrawals` and validates withdrawal readiness using the `isWithdrawalReady` function.

This results in duplication of both time-based checks and state tracking between the vault and the strategy:

- The vault checks whether `block.timestamp >= request.unbondTime + UNBONDING_PERIOD`.
- The strategy separately tracks unbond timestamps and enforces readiness via `isWithdrawalReady`.

This layered validation is unnecessary and increases gas usage, code complexity, and the likelihood of mismatches between the two layers if logic diverges. Moreover, if the strategy is trusted to handle unbonding delays, repeating those checks in the vault offers no additional safety.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Eliminate the redundant `UNBONDING_PERIOD` check and the `unbondRequests` mapping from the vault when working with `DualCoreStrategy` . Instead, rely on the strategy's `isWithdrawalReady` method to determine if a withdrawal is permitted. If early reverts are desirable at the vault level, directly call `require(strategy.isWithdrawalReady(msg.sender), "...")` during `withdraw` . This reduces duplication and offloads time-sensitive logic to the component already responsible for managing it.

8.31 (HAL-31) UNBOUNDED APPROVE TO VAULT EXPOSES DUALCORETOKEN TO EXTERNAL DRAIN

// INFORMATIONAL

Description

In the `DualCoreStrategy` contract, the `unstake` function calls `dualCoreToken.approve(dualCoreVault, type(uint256).max)` in order to pre-authorize the vault to transfer tokens for efficiency. While this avoids repeated approval transactions for each unstake, it also introduces a critical security risk.

If the `DualCoreStrategy` contract contains any vulnerability that allows an unauthorized or manipulated call to trigger a withdrawal or similar interaction with the `dualCoreVault`, the vault could be exploited to drain the entire `dualCoreToken` balance held by the strategy. Because the allowance is effectively unlimited, any faulty logic or reentrancy that gives control over the vault's withdrawal pathway becomes significantly more dangerous, with no cap on the potential damage.

This pattern also violates the principle of least privilege, granting full control to the external vault over valuable tokens even when most interactions require only limited transfers.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Avoid setting an unbounded allowance unless the external contract is fully trusted, immutable, and does not expose any third-party-controlled execution paths. Prefer approving only the specific amount needed at the time of unstake:

```
dualCoreToken.approve(dualCoreVault, _amount);
```

Alternatively, reset the allowance to zero after each use if repeated approvals are undesirable and the vault allows re-approval. This ensures tighter control over token flows and minimizes the risk surface in the event of misbehavior or compromise in either the strategy or the vault.

8.32 (HAL-32) INCORRECT WITHDRAWAL FALLBACK CHECK CAN LEAD TO ETH TRANSFER FAILURE

// INFORMATIONAL

Description

In the `DualCoreStrategy` contract, the fallback logic after calling `dualCoreVault.withdraw()` includes the following condition:

```
if (received == 0 && address(this).balance < amount) {
    revert(...);
}
```

This check is intended to detect failure scenarios where the strategy either receives no ETH or an insufficient amount to satisfy the withdrawal request. However, the logic is flawed. The condition only reverts when **both**:

1. `received == 0`
2. `address(this).balance < amount`

This means if **some** ETH is received (`received != 0`) but the contract's balance is still **less than `amount`**, the code continues, and the final low-level `.call{value: amount}` will fail, as it attempts to transfer more ETH than the contract holds. This causes unexpected reverts and breaks downstream logic.

Furthermore, the in-code comment does not accurately reflect the implemented logic, adding confusion during auditing or maintenance.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Fix the conditional to revert if **either**:

- No ETH was received (`received == 0`), or
- The contract's ETH balance is still insufficient to fulfill the request.

Update the condition to:

```
if (received == 0 || address(this).balance < amount) {
    revert("Insufficient ETH received from vault");
}
```

Alternatively, if the goal is to ensure that ETH was received **and** the total balance is insufficient, the logic should explicitly reflect that and be documented accordingly:

```
if (received != 0 && address(this).balance < amount) {
    revert("Partial ETH received but not enough to fulfill withdrawal");
}
```

Also update or clarify the in-code comment to accurately describe the intention and behavior of the fallback logic. This ensures that withdrawal execution is predictable, and reverts happen before unsafe or doomed ETH transfers are attempted.

8.33 (HAL-33) REDUNDANT INHERITANCE AND UNUSED FEE LOGIC IN STRATEGY

// INFORMATIONAL

Description

The `BitfluxStrategy` contract inherits from both `StratManager` and `FeeManager` , yet the `FeeManager` contract itself already inherits from `StratManager` , making the direct inheritance of `StratManager` redundant. Furthermore, none of the fee-related functionality provided by `FeeManager` —such as fee variables or setter functions—is used within `BitfluxStrategy` . This suggests either an incomplete integration or unnecessary inheritance that adds confusion and bloat to the contract structure.
Such design introduces ambiguity about whether fees are intended to be part of the strategy logic, and misleads developers into assuming fee mechanisms are active when they are not. It also increases bytecode size and audit surface without functional benefit.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Remove the direct inheritance from `StratManager` , and if fee distribution is not required in `BitfluxStrategy` , also remove the unused `FeeManager` base contract. If fee functionality is planned for future use, document it explicitly and integrate its logic accordingly.

8.34 (HAL-34) MISLEADING COMMENT IN HARVEST REGARDING LP BALANCE LOGGING

// INFORMATIONAL

Description

In the `BitfluxStrategy` contract, the `harvest` function includes a comment stating that it will "log" the final LP token balance:

```
// log final lp balance
```

However, there is no event emitted or actual logging performed—only a call to `balanceOf(lpToken)` is made. This comment is misleading and may confuse maintainers or auditors into thinking the function is emitting stateful output when it is not.
Such inconsistencies reduce code clarity, especially in functions that handle sensitive operations like compounding or reporting yield, where accurate recordkeeping is often expected.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Update or remove the misleading comment. If logging is intended, emit an appropriate event to record the final LP balance. For example:

```
emit LPBalanceReported(IERC20(lpToken).balanceOf(address(this)));
```

Otherwise, revise the comment to reflect that the balance is simply being read, not logged.

8.35 (HAL-35) UNUSED STATE VARIABLES INCREASE CONTRACT SIZE AND REDUCE CLARITY

// INFORMATIONAL

Description

In the `NawaVaultUpgradable` contract, the declared state variables `min`, `max`, and `zapAddress` are not used anywhere in the contract logic. These variables serve no operational purpose and appear to be remnants of a past implementation or placeholders for future features.
Leaving unused variables in deployed contracts results in:

- Increased contract size and deployment cost.
- Reduced readability and higher cognitive overhead during audits and maintenance.
- Misleading expectations about available configuration or functionality.

This also creates confusion around whether the contract supports zap-based deposits or enforces participation bounds, which it currently does not.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Remove the unused `min`, `max`, and `zapAddress` state variables from the `NawaVaultUpgradable` contract unless they are explicitly reserved for upcoming features. If they are intended for future use, add a comment clarifying that intent and ensure no dead logic is compiled into production deployments.

8.36 (HAL-36) MISSING SLIPPAGE LIMIT VALIDATION IN CONSTRUCTOR

// INFORMATIONAL

Description

In the `SolvZap` contract, the `setSlippageLimit` function correctly includes a validation to ensure the provided slippage value does not exceed the configured `SLIPPAGE_DENOMINATOR` . However, the constructor sets an initial slippage limit without performing the same bounds check. This inconsistency allows deployment with an invalid or dangerously high slippage limit that would otherwise be rejected post-deployment.

This creates a gap in safety and consistency between initialization and runtime configuration. It may also introduce silent misconfigurations that affect execution behavior during the contract's initial usage.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Apply the same validation in the constructor as used in `setSlippageLimit` :

```
require(_slippageLimit <= SLIPPAGE_DENOMINATOR, "Invalid slippage limit");
```

This ensures the slippage configuration is safe and bounded from the moment of deployment, preventing misbehavior in early interactions and maintaining consistency in contract assumptions.

8.37 (HAL-37) ZAPIN DOES NOT VALIDATE VAULT COMPATIBILITY WITH LP TOKEN

// INFORMATIONAL

Description

In the `BitfluxZap` contract, the `zapIn` function proceeds to add liquidity and then calls `vault.depositZap()` under the assumption that the vault accepts the specific LP token produced by the liquidity pool. However, there is no validation to ensure that the vault's `want()` token actually matches the expected `lpToken`.

If a misconfigured vault is passed—one that accepts a different `want()` token—the `depositZap` call may revert or, depending on the vault implementation, silently accept the LP tokens without properly crediting the user. This creates a risk of:

1. **Transaction reverts**: If the vault performs validation inside `depositZap`, the call fails.
2. **Fund lock/loss**: If the vault accepts the tokens but does not recognize or credit them, the LP tokens remain locked in the vault, and the user receives no vault shares in return.

The vulnerability is made worse if the vault's `depositZap` logic changes in the future, or if multiple LP tokens are supported across vaults, increasing the risk of incorrect token routing.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Add an explicit validation step in `zapIn` to ensure the vault is compatible with the LP token:

```
require(vault.want() == address(lpToken), "Vault does not match LP token");
```

This guards against misconfiguration and ensures the zap flow is only used with vaults that are designed to accept the LP tokens being deposited. Additionally, consider enforcing post-conditions, such as verifying that vault shares are minted for the user, to ensure no silent failures occur during the deposit process.

8.38 (HAL-38) NO PRICE-PER-SHARE LOGIC ENABLES YIELD DILUTION VIA 1:1 MINTING

// INFORMATIONAL

Description

In both `CoreVault` and `SolvVault`, the `deposit()` function mints vault shares at a fixed 1:1 ratio to the amount deposited, without taking into account the current value of the underlying assets managed by the strategy. This is done through logic such as:

```
_mint(msg.sender, msg.value); // CoreVault example
```

There is no implementation of a dynamic share price or `pricePerShare` mechanism that adjusts based on the vault's total underlying balance relative to the total share supply. As a result, **new users can deposit ETH and receive vault shares at par value**, even if the vault has accrued yield from a strategy, effectively enabling them to benefit from the strategy's gains without having contributed to them.
This design flaw allows **value siphoning** and results in **yield dilution** for earlier depositors.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Introduce dynamic share issuance based on the vault's current total balance (underlying tokens plus those in strategy) and total supply. Additionally, implement a `getPricePerShare()` view function to track and expose the current share value for front-end integration and user transparency.
These changes are critical to protecting existing depositors from dilution, accurately tracking profit, and aligning the vault with industry-standard yield vault patterns (e.g., Yearn, Beefy).

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.