THREE SIGMA

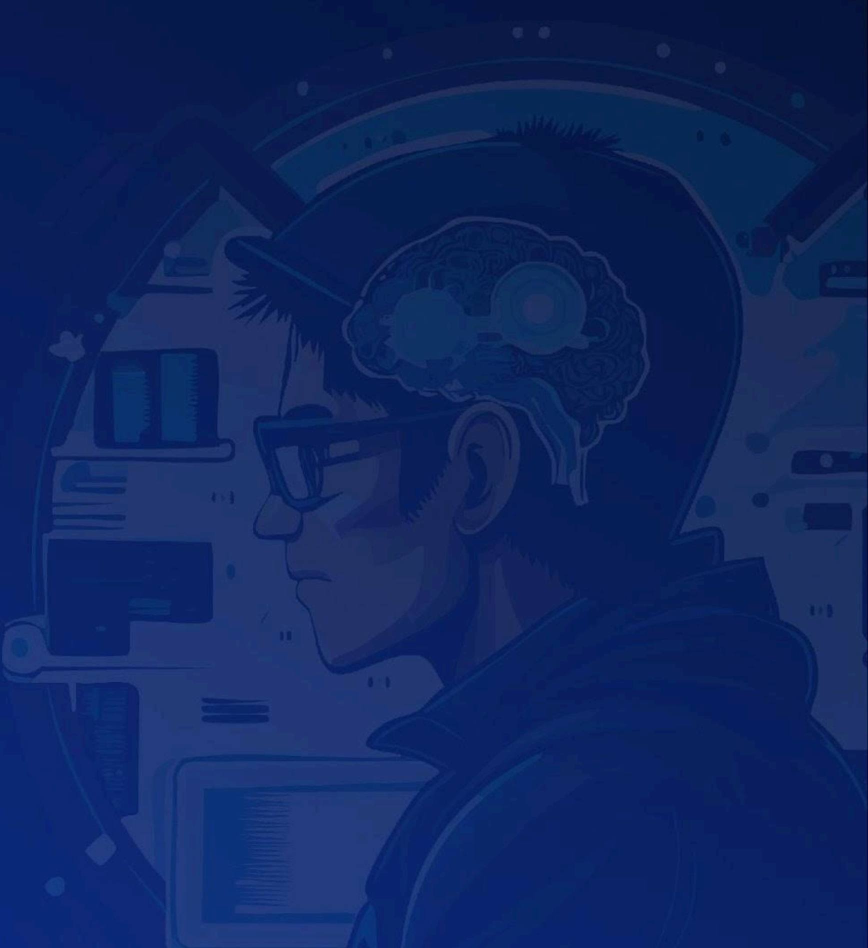
Synnax Yield bearing stablecoin

Security Review

Disclaimer

Security Review

Synnax Yield bearing stablecoin



Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code’s manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Security Review

Synnax Yield bearing stablecoin

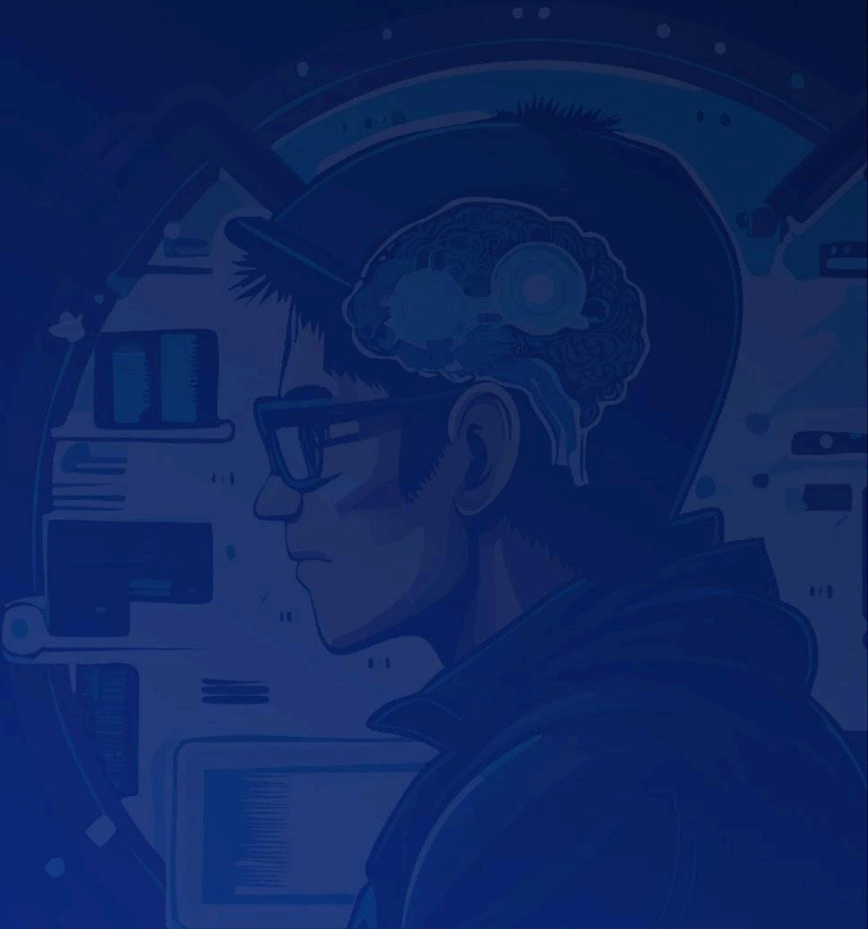
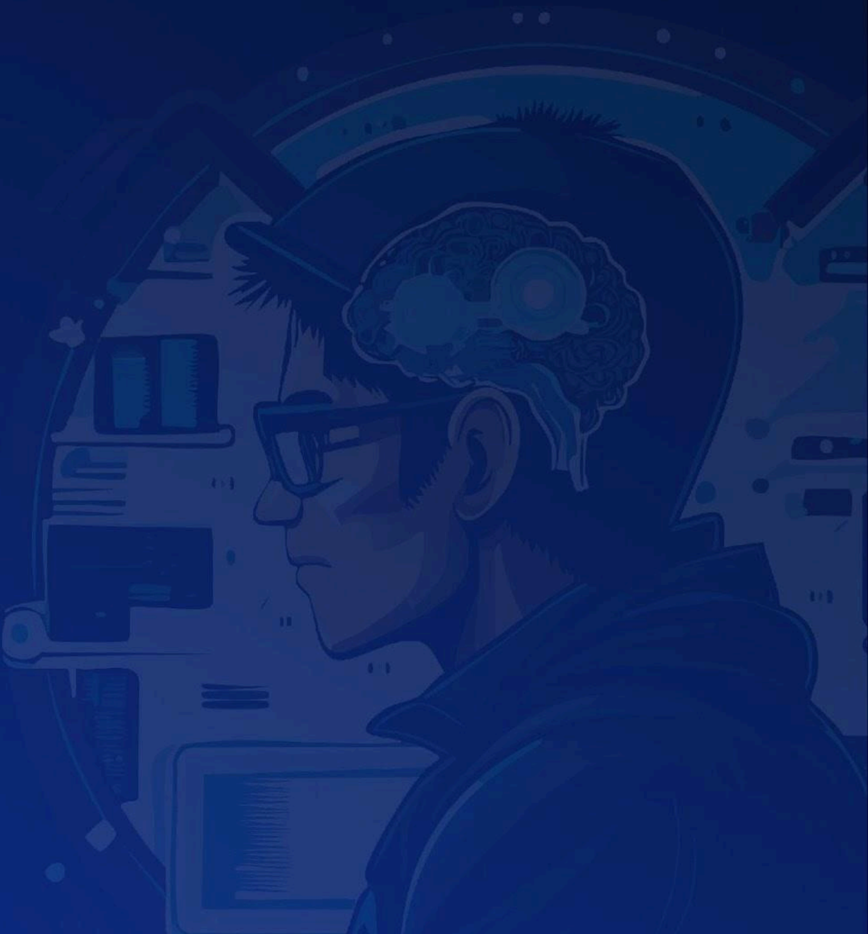


Table of Contents

Summary

Security Review

Synnax Yield bearing stablecoin



Summary

Three Sigma audited Synnax in a 4.8 person week engagement. The audit was conducted from 18/11/2025 to 03/12/2025.

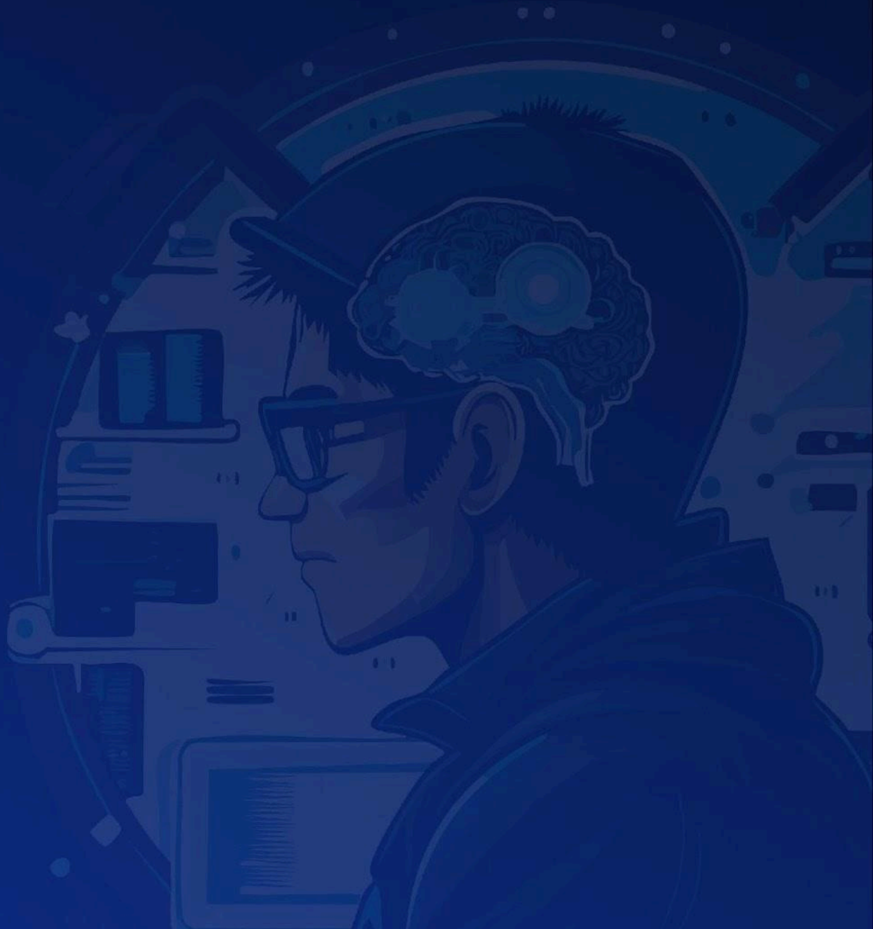
Protocol Description

Synnax is deployed on the Sei blockchain and implements syUSD, an ERC-20 USD-pegged stablecoin. syUSD is issued through a fork of the Abracadabra architecture: DegenBox acts as a BentoBox-style vault and protocol minter for whitelisted contracts, while CauldronV4 provides over-collateralized lending markets that accept volatile collateral and enforce oracle-based LTV and liquidation thresholds. The system also includes two Peg Stability Modules, SyUSDStableMinting for single-stable mint and redeem and SyUSDMultiCollateralManager for multi-collateral mint and redeem plus strategy debt accounting. Yield is pursued via strategy vaults (SyUSDStrategyVault) that deploy collateral managed by SyUSDMultiCollateralManager into strategies and realize profits back into syUSD.

Scope

Security Review

Synnax Yield bearing stablecoin



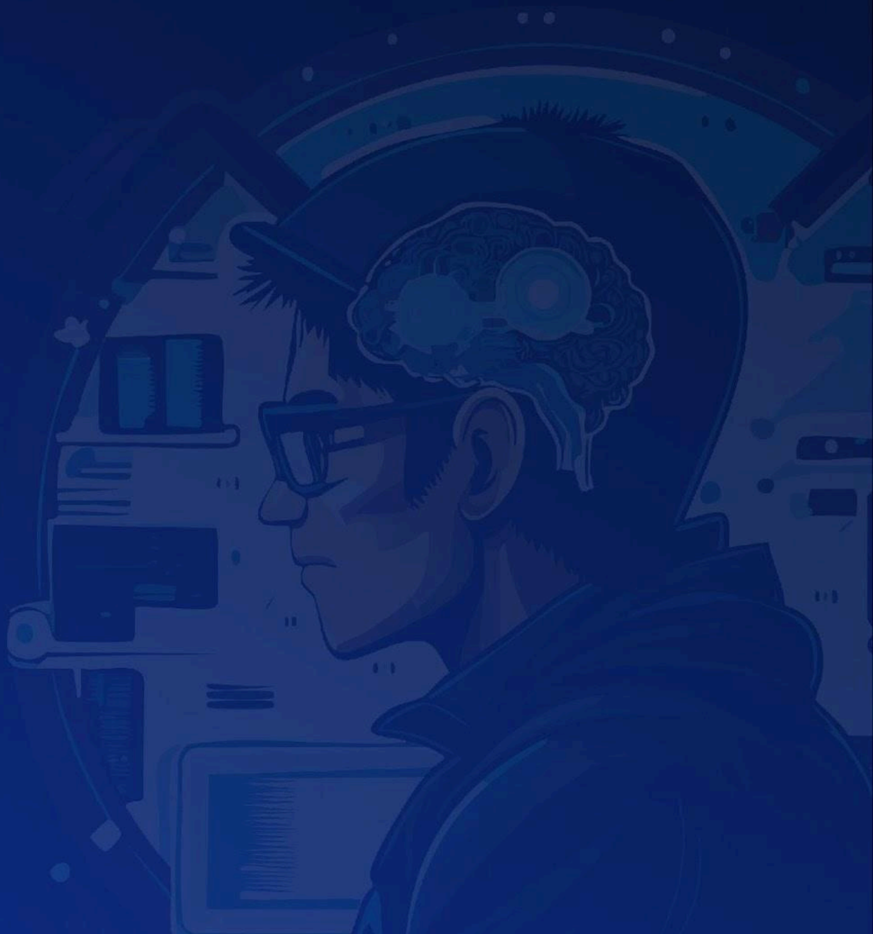
Scope

Filepath	nSLOC
src/cauldrons/CauldronV4.sol	518
src/syUSD/SyUSDMultiCollateralManager.sol	414
src/DegenBox.sol	337
src/cauldrons/SyUSDStableMinting.sol	232
src/strategies/syUSD/SyUSDStrategyVault.sol	221
src/cauldrons/CauldronV5Migratable.sol	185
src/cauldrons/CauldronV5.sol	100
src/mixins/MasterContractManager.sol	90
src/strategies/syUSD/SyUSDVaultShareRouter.sol	71
src/factory/MasterCauldronV5Factory.sol	52
src/oracles/Api3PriceFeed.sol	45
src/strategies/syUSD/YearnSyUSDStrategyVault.sol	42
src/strategies/syUSD/AaveSyUSDStrategyVault.sol	40
src/oracles/ChainlinkOracle.sol	35
src/registry/OracleRegistry.sol	20
src/cauldrons/StablePoolCauldron.sol	1
Total	**2403**

Methodology

Security Review

Synnax Yield bearing stablecoin



Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

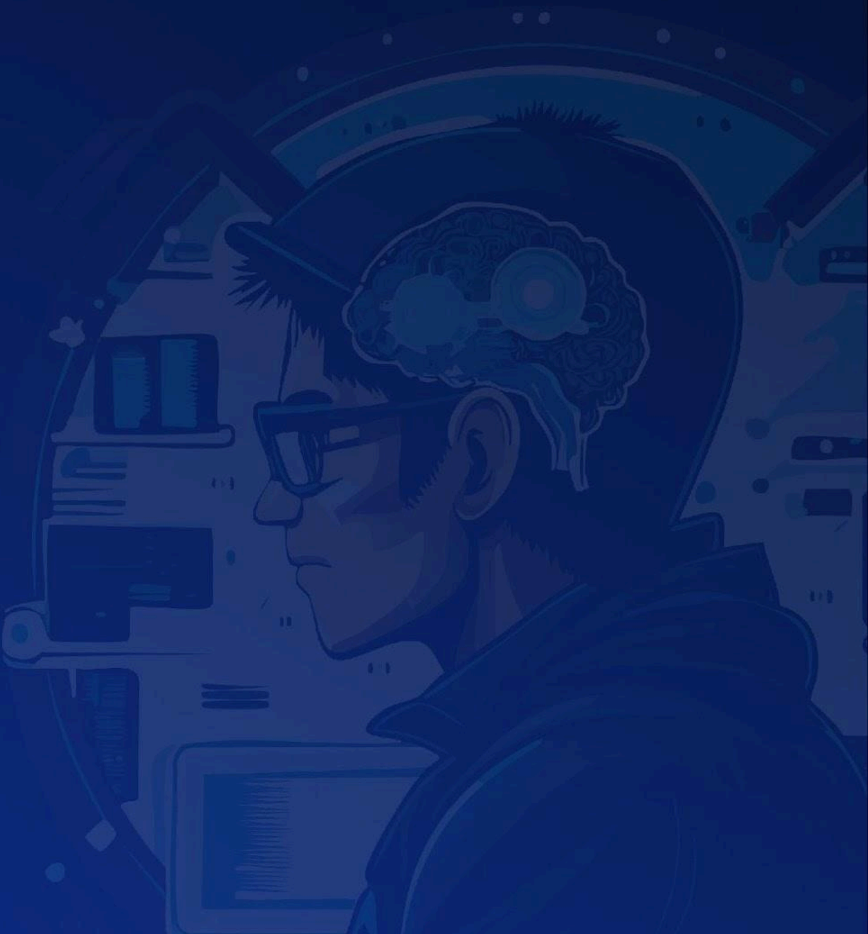
In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Security Review

Synnax Yield bearing stablecoin



Project Dashboard

Application Summary

Name	Synnax
Repository	https://github.com/Synnax-Protocol/contract-protocol
Commit	e51afa1
Language	Solidity
Platform	Sei

Engagement Summary

Timeline	18/11/2025 to 03/12/2025
№ of Auditors	2
Review Time	4.8 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	3	3	0
High	9	9	0
Medium	10	10	0
Low	8	8	0
None	8	7	1

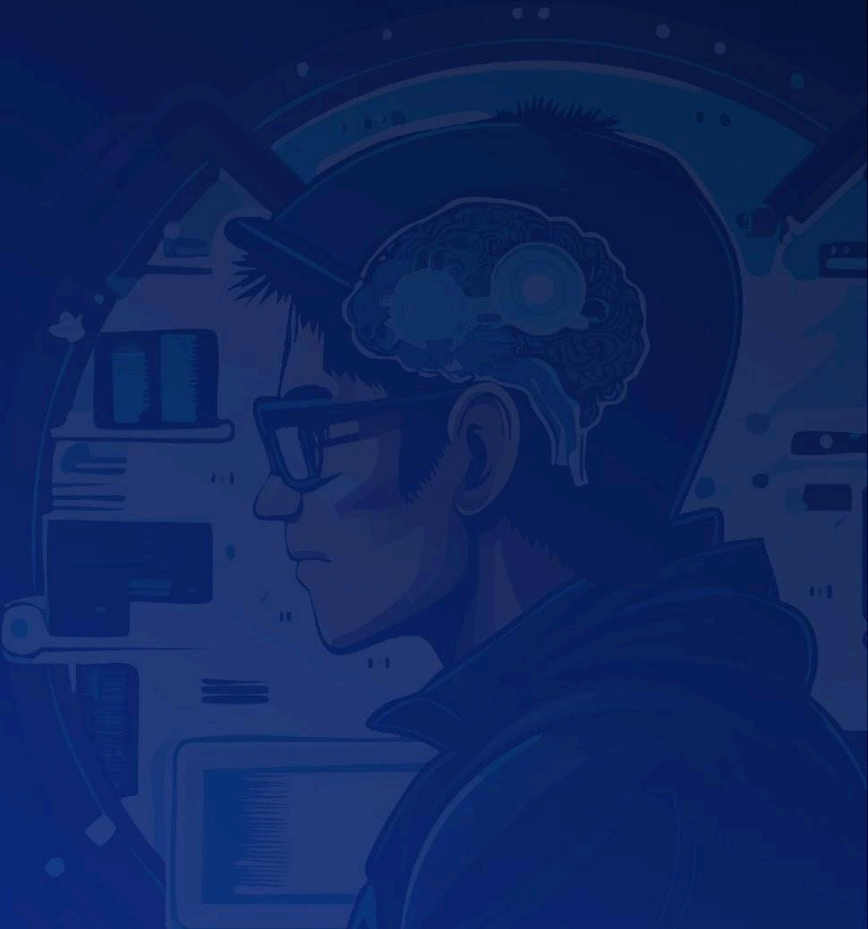
Category Breakdown

Suggestion	8
Documentation	0
Bug	30
Optimization	0
Good Code Practices	0

Risk Section

Security Review

Synnax Yield bearing stablecoin



Risk Section

- **Strategies out of scope**: The audited contract *SyUSDStrategyVault* is an abstract base implementation and is not intended to be deployed or used directly. Concrete strategy contracts inheriting from this base, which will be deployed and interacted with by users, were not included in the scope of this audit. Consequently, this assessment is limited to the base contract logic and does not cover the behavior or security properties of concrete implementations.

- **Oracle Risks - Oracle contracts were not included in the audit scope**, but show potential concerns:
  - **Mixed Oracle Price Conventions:** The oracle contracts are out of the audit scope. For the calculations in the contracts to be correct, the price format returned by the oracles must be as follows:
    - In SyUSDMultiCollateralManager: 1 token = price USD
    - In SyUSDStableMinting: 1 token = price USD
    - In CauldronV4: 1 USD = rate token
  - **Oracle Staleness Checks**: Oracles represent a single point of failure in the system. Therefore, we expect the oracle to have the following security measures in place:
    - There should be a staleness check to ensure the returned price is fresh.
    - There should be an L2 sequencer uptime check, as the protocol is deployed on Sei.
    - The oracle should incorporate multiple data sources in case one fails to report the price.

- **Tokens Compatibility**: The protocol is not compatible with fee-on-transfer (FOT) tokens and rebasing tokens.

- **PSM effectiveness**: *SyUSDStableMinting* acts as a PSM, so having uncapped fees can alter the price stability if set too high.

- **Misconfiguration Risks:** For the calculation in *CauldronV4::_isSolvent* to work, the oracle's decimals must equal to the collateral's decimals. For example, if the collateral is USDC (6 decimals), and the current exchange rate is 1 USD = 1.01 USDC, then the *_exchangeRate* must be 1.01e6.

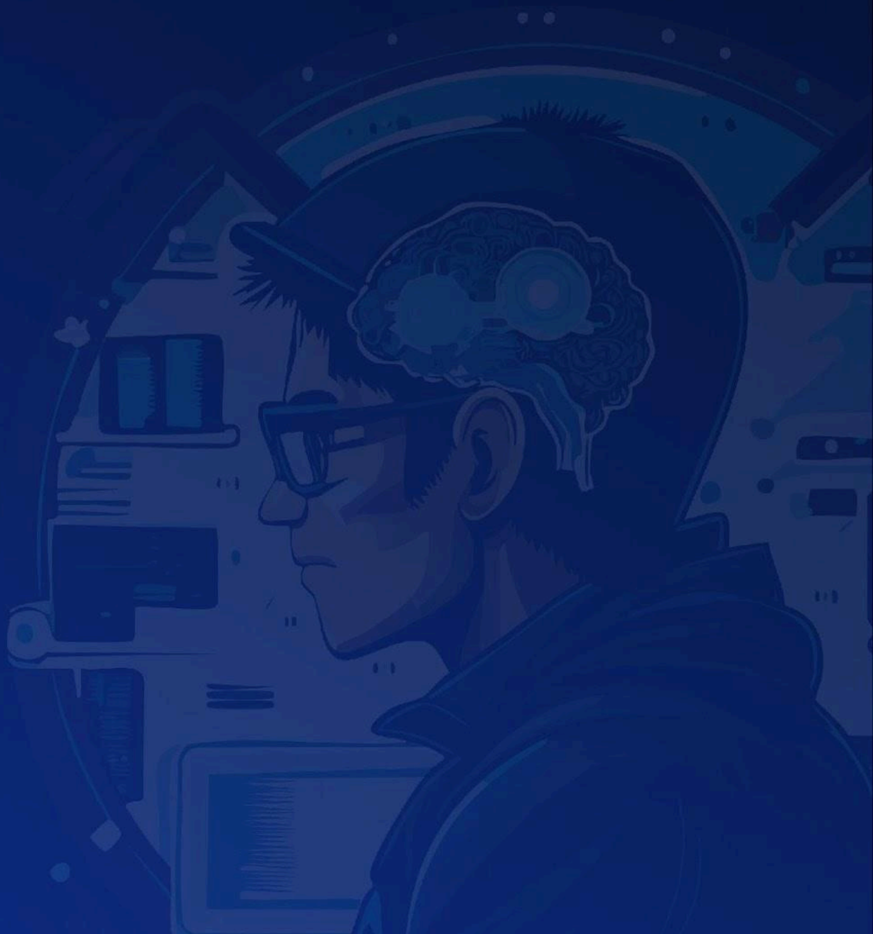
- **Partial Audit Scope**: The audit covers only specified contracts within the repository. Vulnerabilities in contracts outside this scope could introduce risks, potentially impacting overall system security.

- **Administrative Key Management Risks**: The system makes use of administrative keys to manage critical operations. Compromise or misuse of these keys could result in unauthorized actions and financial losses.
- **Upgradability Risk**: Administrators have the capability to update contract logic at any moment due to the project's upgradable design. Contract upgradability, while beneficial, also poses the risk of harmful modifications if administrative privileges or upgrade processes are compromised.

Findings

Security Review

Synnax Yield bearing stablecoin



Findings

3S-Synnax-C01

Missing collateral reservation in **SyUSDStrategyVault::initUnstake** leads to persistent DoS on queued unstakes

Id	3S-Synnax-C01
Classification	Critical
Impact	Critical
Likelihood	High
Category	Bug
Status	Addressed in #a74969b.

Description

SyUSDStrategyVault::initUnstake initiates delayed redemptions by pulling underlying collateral out of the strategy via **_collectForUnstake**, burning user shares, and recording the principal and profit that will later be converted back into **syUSD** in **SyUSDStrategyVault::unstake**. However, the implementation does not reserve the underlying that was unlocked for a specific user; instead it leaves the withdrawn collateral as generic idle balance. When multiple users call **initUnstake** while the vault has insufficient idle collateral, they can all rely on the same idle balance, causing the same underlying units to be “promised” to more than one pending unstake. After the cooldown, the first caller can successfully claim, but subsequent callers may revert in **unstake** because the idle collateral has been consumed, and there is no way for users to cancel or adjust the stale pending unstake. This creates a race condition where queued unstakes can repeatedly fail and the DoS is effectively passed from user to user. As a consequence, the collateral de-allocated and promised to multiple user will get permanently blocked in the strategy contract.

The likelihood of the vault not having enough collateral when **initUnstake** is called is high since all the deposits end up into being allocated to strategies, thus the locked collateral will keep growing at every concurrent **initUnstake** request.

A concrete scenario illustrates the issue. Assume the vault has 1000 units of underlying deployed in strategy, zero idle underlying, **unstakeCooldown** is one day, and both Alice and Bob hold 500 shares each. Alice first calls **initUnstake** for 500 shares: **_collectForUnstake** observes idle underlying equal to zero and **needed** equal to 500, triggers a withdrawal of 500 underlying from the strategy, records **pendingUnstakes[alice]** with **principal** equal to 500 (and associated fields), and burns Alice's 500 shares, leaving 500 underlying idle in the vault. Bob then calls **initUnstake** for his 500 shares in the same period: **_collectForUnstake** now sees **idle** equal to 500 and **needed** equal to 500, so it does not withdraw additional funds from the strategy, records **pendingUnstakes[bob]** with **principal** equal to 500, and burns Bob's 500 shares. At this point, there are 500 underlying units idle in the vault but a total of 1000 underlying units effectively promised across Alice and Bob's pending unstakes. After the cooldown, when Alice calls **unstake**, the vault can successfully transfer 500 underlying to the manager (via **returnPrincipalAndMint** and **realizeProfit**), mint **syUSD**, and pay Alice. When Bob later calls **unstake**, the vault has zero idle underlying, so any attempt to transfer 500 underlying into the manager will eventually revert, leaving Bob's position stuck.

The only way out is for another user to trigger additional **_collectForUnstake** withdrawals, which just moves the DoS to whoever is last in line.

Recommendation

Track and reserve the underlying collateral that has been unlocked for pending unstakes so it is excluded from future **_collectForUnstake** calls, and release the reservation only when **SyUSDStrategyVault::unstake** completes. A simple approach is to introduce a **reservedForUnstake** state variable in underlying units, increment it in **initUnstake** by **principal + profit** for each new request, subtract it from the idle balance when computing available funds in **_collectForUnstake**, and decrement it in **unstake** once the corresponding principal and profit are successfully converted to **syUSD** and transferred.

3S-Synnax-C02

Missing allowance check in withdraw allows unauthorized burning of shares

Id	3S-Synnax-C02
Classification	Critical
Impact	Critical
Likelihood	Medium
Category	Bug
Status	Addressed in #a490f65.

Description

The **SyUSDStrategyVault** contract implements an ERC4626-compliant vault that allows users to stake syUSD tokens and receive vault shares in return. When **unstakeCooldown** is set to zero, users can call **withdraw** or **redeem** to burn their shares and receive the underlying syUSD tokens back.

Both **withdraw** and **redeem** functions internally call **SyUSDStrategyVault::_withdraw** to process the withdrawal. However, unlike the standard ERC4626 implementation, this override does not verify that the **caller** has sufficient allowance to spend the **owner**'s shares when **caller != owner**.

The standard ERC4626 pattern includes a check using **_spendAllowance(owner, caller, shares)** to ensure that only authorized parties can burn shares on behalf of another user. Without this validation, any address can call **withdraw** or **redeem** and specify an arbitrary **owner** address, causing that owner's shares to be burned without permission while the withdrawn assets are sent to a **receiver** of the caller's choosing.

#### Steps to reproduce

1. Alice deposits syUSD into the **SyUSDStrategyVault** and receives vault shares

2. The contract has **unstakeCooldown == 0** (no cooldown period enforced)

3. Bob (a malicious actor) calls **withdraw(assets, bob, alice)** where **alice** is the owner and **bob** is the receiver

4. The function executes successfully, burning Alice's shares without checking if Bob has allowance to spend them

5. Bob receives the syUSD tokens while Alice's shares are reduced without her consent

Recommendation

Add the allowance check from the standard ERC4626 implementation to verify the **caller** has permission to spend the **owner**'s shares. The **_withdraw** function should validate allowance before burning shares:

```
 function _withdraw(address caller, address receiver, address owner, uint256 assets,
uint256 shares)
    internal override {
+    if (caller != owner) {
+        _spendAllowance(owner, caller, shares);
+    }
    uint256 minted = _beforeWithdraw(assets);
    _burn(owner, shares);
    IERC20(asset()).safeTransfer(receiver, minted);
    emit Withdraw(caller, receiver, owner, minted, shares);
 }
```

This ensures that when **caller** and **owner** are different addresses, the caller must have been granted sufficient allowance by the owner to perform the withdrawal operation on their behalf.

3S-Synnax-C03

Commented out oracle price fetch in **returnPrincipalAndMint** leads to protocol insolvency in case of collateral depeg

Id	3S-Synnax-C03
Classification	Critical
Impact	Critical
Likelihood	Medium
Category	Bug
Status	Addressed in #90ed49f.

Description

The **SyUSDMultiCollateralManager::returnPrincipalAndMint** function allows strategies with **STRATEGY_ROLE** to return borrowed collateral and receive newly minted syUSD tokens in a single operation. According to the inline comment, this function should "use oracle + fee like normal mint" and employ oracle-based pricing to calculate the syUSD amount, consistent with the **mint** and **realizeProfit** functions. However, the oracle price fetch is commented out, causing the function to use a hardcoded 1:1 conversion rate instead of the current market price reported by the oracle.

The function currently implements the following logic:

```
function returnPrincipalAndMint(address token, uint256 amount, address to)
    external
    onlyRole(STRATEGY_ROLE)
    nonReentrant
    validateCollateral(token)
    returns (uint256 syUSDOut)
  {
    StrategyDebt storage d = strategyDebt[msg.sender][token];
    if (amount > d.principal) revert ZeroAmount();
    d.principal -= amount;
    (uint256 deposited, ) = _depositToBox(token, amount);
    totalCollateralAmount[token] += deposited;
    // Use oracle + fee like normal mint
    Collateral storage c = collaterals[token];
    //uint256 price = _oraclePrice(c, true);
    uint256 gross = (amount * ONE) / (10 ** c.decimals);
```

```
        uint256 fee = gross * feeIn / BPS;
        syUSDOut = gross - fee;
        totalMinted += gross;
        c.psmMinted += gross;
        _mintSyUSD(syUSDOut, to);
        if (fee > 0) _mintSyUSD(fee, treasury);
        emit PrincipalReturned(msg.sender, token, amount, syUSDOut);
    }
```

This creates an accounting inconsistency where the protocol mints syUSD based on token quantity rather than token value. While the protocol intends to support only stablecoins as collateral, stablecoins are not immune to depeg events. Historical precedents include USDC depegging to approximately $0.88 during the March 2023 Silicon Valley Bank crisis and UST collapsing below $0.10 during the Terra Luna crash. When a stablecoin collateral depegs below one dollar, the 1:1 conversion enables strategies to mint more syUSD than the collateral value supports, leading to protocol undercollateralization.

The accounting error manifests in the **totalMinted** variable, which tracks the cumulative syUSD minted by the PSM to enforce global and per-collateral mint caps. When a strategy returns depegged collateral using **returnPrincipalAndMint**, the protocol increments **totalMinted** by the full token amount while only receiving fractional dollar value in collateral backing. This creates a systemic imbalance where the PSM's claimed minting obligations exceed the actual collateral value securing them. The discrepancy becomes particularly severe during extended depeg events, as strategies can repeatedly return principal and mint syUSD at the inflated 1:1 rate while direct users minting via the **mint** function receive appropriately discounted syUSD amounts based on oracle pricing.

Steps to Reproduce

1. Protocol uses stablecoin collateral. A stablecoin depeg event occurs where the oracle price drops below $1.00, as happened when USDC depegged during the March 2023 Silicon Valley Bank crisis.

2. A strategy with **STRATEGY_ROLE** calls **allocateCollateral** to borrow collateral from the protocol, establishing a **strategyDebt** entry.

3. The strategy calls **returnPrincipalAndMint** with the borrowed collateral amount while the collateral's oracle price is below one dollar.

4. The function executes with the commented-out oracle call, using the hardcoded formula **gross = (amount * ONE) / (10 ** c.decimals)** which treats one unit of collateral as one syUSD regardless of actual value.

5. The protocol mints syUSD based on quantity rather than value, causing **totalMinted** to exceed the actual collateral backing, resulting in protocol insolvency.

Recommendation

Uncomment the oracle price fetch and use the same pricing calculation as **mint** and **realizeProfit**:

```
-       // Use oracle + fee like normal mint
-       Collateral storage c = collaterals[token];
-       //uint256 price = _oraclePrice(c, true);
-       uint256 gross = (amount * ONE) / (10 ** c.decimals);
+        // Use oracle + fee like normal mint
+        Collateral storage c = collaterals[token];
+        uint256 price = _oraclePrice(c, true);
+        uint256 gross = (amount * price) / (10 ** c.decimals);
```

3S-Synnax-H01

Profit cap feature causes denial of service on vault withdrawal operations

Id	3S-Synnax-H01
Classification	High
Impact	High
Likelihood	Medium
Category	Bug
Status	Addressed in #c888242.

Description

The **SyUSDMultiCollateralManager::realizeProfit** function allows strategies to convert earned profit (underlying tokens) into syUSD. This function is invoked by **SyUSDStrategyVault** during three user-facing operations:

- **SyUSDStrategyVault::harvest** - realizes strategy profits

- **SyUSDStrategyVault::withdraw** / **SyUSDStrategyVault::redeem** - via **_beforeWithdraw**

- **SyUSDStrategyVault::unstake** - completes delayed withdrawals

The function implements a profit cap mechanism that reverts with **ProfitTooHigh()** when profit exceeds a threshold:

```
uint256 gross = (amount * price) / (10 ** c.decimals);
if (gross > maxAllowed) revert ProfitTooHigh();
```

This design is fundamentally flawed. When the profit cap is exceeded, **realizeProfit** reverts, which propagates up to all dependent vault functions. Users cannot withdraw their funds, complete pending unstakes, or harvest yields. The vault becomes effectively locked until profits decrease below the cap threshold. This is a condition outside user control.

Furthermore, the profit cap calculation itself is incorrect. The function uses **_oraclePrice(c, true)** which caps the oracle price at **ONE**. This causes **gross** to always be less than or equal to **fairValue**, making the condition **gross > maxAllowed** unreachable under normal circumstances. The cap mechanism is both broken and dangerous:

- When using the capped price, the check never triggers when it should

- If the implementation were corrected to use actual prices, users would face DoS when strategies perform well

Recommendation

Remove the profit cap feature entirely. A mechanism that blocks user withdrawals based on strategy performance is unsuitable for a vault design. Stability of syUSD should be managed through other mechanisms that do not impact user fund accessibility.

3S-Synnax-H02

Missing collateral transfer from strategy vault to SyUSDMultiCollateralManager leads to broken settlement and inconsistent collateral accounting

Id	3S-Synnax-H02
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #77a9887.

Description

SyUSDStrategyVault is designed to obtain underlying collateral from **SyUSDMultiCollateralManager** via **SyUSDMultiCollateralManager::allocateCollateral**, invest it, and later settle principal and profit back through **SyUSDMultiCollateralManager::returnPrincipalAndMint** and **SyUSDMultiCollateralManager::realizeProfit**. The constructor **SyUSDStrategyVault::constructor** grants an infinite approval **underlying_.safeApprove(address(manager_), type(uint256).max)** to the manager, indicating the intended pattern is for the manager to actively pull collateral from the vault when these settlement functions are called.

In the current implementation, **SyUSDStrategyVault** never transfers underlying back to the manager, and **SyUSDMultiCollateralManager::_depositToBox** always calls **IBentoBoxV2::deposit** with **from = address(this)**, assuming the manager already holds the **token** balance. As a consequence, **SyUSDMultiCollateralManager::returnPrincipalAndMint** and **SyUSDMultiCollateralManager::realizeProfit** will either revert when the manager lacks sufficient underlying, or, if the manager happens to hold its own **token** balance, they may mint **syUSD** against collateral that was not actually pulled back from the strategy vault, leaving the originally allocated collateral stranded at the strategy and breaking the intended accounting between **strategyDebt**, PSM collateral, and **syUSD** supply.

Recommendation

SyUSDMultiCollateralManager::returnPrincipalAndMint and **SyUSDMultiCollateralManager::realizeProfit** should explicitly pull the underlying **token** from the calling strategy using the existing infinite approval before depositing into **DegenBox**, so that the manager, not the vault, is the source of funds for **_depositToBox**.

3S-Synnax-H03

Immediate share burn without asset adjustment leads to inflated share value and accounting inconsistencies

Id	3S-Synnax-H03
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #efbea92.

Description

The **SyUSDStrategyVault::initUnstake** function implements the first step of a delayed unstake flow: it computes the expected **syUSD** to be returned, pulls or unwinds the corresponding underlying via **_collectForUnstake**, and records an **Unstake** struct in **pendingUnstakes**. However, it also burns the user's **shares** immediately via **_burn(msg.sender, shares)** while leaving the vault's underlying assets and **syUSD** liabilities unchanged until **SyUSDStrategyVault::unstake** is called.

This reduces **totalSupply** while **totalAssets** is unchanged, temporarily inflating the per-share value reported by ERC4626 views and used by **previewRedeem**, **previewWithdraw**, and the router's **burnSharesForSyUSD**. During this window, other users (or the router) interacting with the vault can receive more **syUSD** per share than economically justified, and multiple overlapping **initUnstake** calls worsen the distortion, leading to accounting inconsistencies between queued positions and active share holders.

Recommendation

Align share supply and asset/liability accounting during the unstake flow so that **totalAssets / totalSupply** remains a correct valuation for active shares. One approach is to defer burning the user's **shares** until **SyUSDStrategyVault::unstake**, keeping them non-transferable during the cooldown, or to track a **pendingUnstakeAssets** accounting variable and subtract this reserved amount from **totalAssets** in **SyUSDStrategyVault::totalAssets**, **previewRedeem**, and **previewWithdraw**.

3S-Synnax-H04

Assuming 1:1 parity between syusd and underlying collateral in SyUSDStrategyVault allows value extraction from the PSM and mispricing of strategy allocations

Id	3S-Synnax-H04
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #79d0b80.

Description

The **SyUSDStrategyVault** contract is an ERC4626 vault where the asset is **syUSD** and the underlying investment asset is a stablecoin (for example USDC) obtained from **SyUSDMultiCollateralManager**. In **SyUSDStrategyVault::_afterDeposit** and related helpers such as **_toUnderlying**, the vault deterministically converts **assets** units of **syUSD** into **underlyingAmt** using a fixed scalar that assumes **1 syUSD** always corresponds to **1 unit** of the underlying stable, independent of market prices or the oracle used by **SyUSDMultiCollateralManager**. This means the vault burns **assets syUSD** and then calls **SyUSDMultiCollateralManager::allocateCollateral** to pull **underlyingAmt** units of the stable from the PSM purely based on nominal amounts.

When **syUSD** trades below parity while the underlying stable remains closer to 1 USD (or trades differently), the vault can obtain more stablecoin value from the PSM than the burned **syUSD** was worth (e.g., burning 100 **syUSD** valued at $95 and pulling 100 **USDC** valued at $98), effectively over-consuming the PSM's collateral and breaking the intended value equivalence between burned **syUSD** and allocated stable. This mispricing is particularly problematic because **SyUSDMultiCollateralManager** itself already has oracle wiring and is valuing collateral in USD, but **SyUSDStrategyVault** bypasses this and assumes a hard 1:1 peg, enabling systematic value drift from the PSM toward the strategy layer whenever **syUSD** and the underlying depeg relative to each other.

Recommendation

Adjust the **SyUSDStrategyVault** conversion logic so that the amount of underlying requested from **SyUSDMultiCollateralManager** is based on the same oracle-driven valuation used by the PSM, rather than a fixed scalar that assumes **1 syUSD = 1 underlying**. Instead of deriving **underlyingAmt** solely from **_toUnderlying** and a static decimal scale, pull or derive a fair-value price and compute how many units of the underlying correspond to the burned **syUSD** at current oracle rates (or equivalently, how much **syUSD** should be burned for the requested underlying), ensuring that **SyUSDStrategyVault::_afterDeposit** and **SyUSDStrategyVault::_collectForUnstake** respect the PSM's pricing assumptions.

3S-Synnax-H05

Incorrect preview functions cause unstake operations to revert due to unaccounted fee deductions

Id	3S-Synnax-H05
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #a2de132.

Description

The **SyUSDStrategyVault** contract implements an ERC4626 vault that allows users to stake syUSD tokens and earn yield from underlying collateral strategies. Users can initiate an unstake operation via **initUnstake**, which burns their shares and stores expected syUSD amounts in a pending state. After a cooldown period, users call **unstake** to complete the withdrawal and receive their syUSD tokens.

The core issue is that **previewRedeem** and **previewWithdraw** use the default ERC4626 implementation, which assumes a simple share-to-asset ratio without accounting for two critical factors applied by **SyUSDMultiCollateralManager**:

1. **Fee deductions**: Both **returnPrincipalAndMint** and **realizeProfit** charge a **feeIn** fee, reducing the output by **(gross * feeIn) / BPS**

2. **Oracle-based pricing**: The **realizeProfit** function uses **_oraclePrice(c, true)** to calculate the syUSD amount minted from profit collateral, rather than a 1:1 conversion. The oracle price can be capped at 1.0 when the collateral trades above peg, or can be less than 1.0 when below peg.

When **unstake** executes, it calls **manager.returnPrincipalAndMint** (which uses 1:1 conversion after fees) and **manager.realizeProfit** (which applies oracle price and fees):

```
// returnPrincipalAndMint - 1:1 conversion with fee
uint256 gross = (amount * ONE) / (10 ** c.decimals);
uint256 fee = gross * feeIn / BPS;
syUSDOut = gross - fee;
// realizeProfit - oracle price with fee
uint256 price = _oraclePrice(c, true);
```

uint256 gross = (amount * price) / (10 ** c.decimals);
uint256 fee = gross * feeIn / BPS;
syUSDOut = gross - fee;

However, **previewRedeem** uses the standard ERC4626 calculation **shares * totalAssets() / totalSupply()**, which:

- Does not subtract any fees

- Does not account for oracle price variations

This results in **syUSD < p.syUSDExpected** in **unstake**, causing a revert with **NoLiquidity** error.

#### Steps to Reproduce

1. Deploy **SyUSDMultiCollateralManager** with **feeIn = 100** (1% fee)

2. Deploy **SyUSDStrategyVault** with total shares = 100 and total assets = 1000 syUSD

3. User calls **initUnstake(shares=10, minSyUSD=0, receiver=user)**

   - **previewRedeem(10)** returns **10 * 1000 / 100 = 100** syUSD

   - **p.syUSDExpected** is set to 100

   - Assuming all underlying is principal, **p.principal = 100**, **p.profit = 0**

4. After cooldown, user calls **unstake()**

   - **returnPrincipalAndMint** is called with **amount=100**

   - Fee calculation: **fee = 100 * 100 / 10000 = 1**

   - **syUSDOut = 100 - 1 = 99**

   - Check **99 < 100** triggers revert with **NoLiquidity**

#### Additional Impacts

Beyond the primary unstake revert issue, the incorrect preview functions affect multiple operations:

1. **redeem** and **withdraw** return less syUSD than the preview functions indicate

2. **burnSharesForSyUSD** slippage protection becomes ineffective due to inaccurate **previewRedeem** values

3. **SyUSDVaultShareRouter::quote** provides incorrect exchange rate quotes, leading to potential slippage when users call **swap**

4. Users relying on **previewWithdraw** to determine required shares will receive less syUSD than intended

Recommendation

Override **previewRedeem** and **previewWithdraw** to account for the fee structure and the oracle price in **SyUSDMultiCollateralManager**. Additionally, remove strict equality check in **unstake** (**syUSD < p.syUSDExpected**) to account for scenarios where oracle price decreases after the **initUnstake**.

3S-Synnax-H06

Incorrect asset balance source in **SyUSDStrategyVault::totalAssets** leads to share price undervaluation and dilution-based value extraction

Id	3S-Synnax-H06
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #0d4f669.

Description

The **SyUSDStrategyVault::totalAssets** function determines the total value of assets controlled by the vault and, via the ERC4626 formula **sharePrice = totalAssets() * 1e18 / totalSupply()**, directly defines the share price. The function currently reads the balance of **asset()** (which is the **syUSD** token) held by the vault instead of the balance of the underlying token (for example, USDC or USDT).

```
function totalAssets() public view override returns (uint256) {
    return IERC20(asset()).balanceOf(address(this)) +
_toSyUSD(_underlyingBalance());
}
```

This logic is inconsistent with the vault's actual flows. In **SyUSDStrategyVault::_afterDeposit**, **syUSD** received on deposit is immediately burned, so **IERC20(asset()).balanceOf(address(this))** is effectively always zero. However, the vault does hold idle underlying tokens during normal operations. In **SyUSDStrategyVault::_collectForUnstake**, the vault withdraws underlying from the strategy to the vault while processing unstakes, leaving an idle underlying balance until the unstake completes. During harvests, profit is also withdrawn as underlying to the vault before being processed. These idle underlying balances are not included in **totalAssets()**.

When idle underlying exists in the vault, **totalAssets()** under-reports the true asset value. As a result, new deposits are priced using an artificially low share price and receive more shares than economically justified, diluting existing shareholders. An attacker can monitor

underlying.balanceOf(address(this)) and time deposits when the idle underlying balance is high relative to the reported value to maximize value extraction.

Steps to Reproduce

1. Initialize the vault so that the strategy holds 100,000 USDC and **totalSupply()** is 100,000 shares (share price 1.0).

2. Have User A call **SyUSDStrategyVault::initUnstake** with 50,000 shares; **SyUSDStrategyVault::_collectForUnstake** withdraws 50,000 USDC from the strategy to the vault, and 50,000 shares are burned, leaving 50,000 shares outstanding and 50,000 USDC idle in the vault.

3. Observe that **underlying.balanceOf(address(this))** is 50,000 USDC while **_underlyingBalance()** (strategy side) is also 50,000 USDC, but **IERC20(asset()).balanceOf(address(this))** is zero because **syUSD** was burned in **SyUSDStrategyVault::_afterDeposit**.

4. Have an attacker (or any depositor) call **SyUSDStrategyVault::deposit** with 50,000 **syUSD** during the unstake cooldown period. **SyUSDStrategyVault::totalAssets** returns **0 + _toSyUSD(_underlyingBalance()) = 50,000e18**, ignoring the idle 50,000 USDC in the vault.

5. Confirm that the attacker receives 50,000 shares at the reported price of 1.0 instead of the correct 25,000 shares at a true price of 2.0, and can later redeem at the correct higher price, extracting value from existing shareholders.

Recommendation

The **totalAssets()** implementation should account for the vault's idle underlying balance rather than the **syUSD** balance, which is burned on deposit:

```
function totalAssets() public view override returns (uint256) {
-   return IERC20(asset()).balanceOf(address(this)) +
_toSyUSD(_underlyingBalance());
+   return _toSyUSD(underlying.balanceOf(address(this))) +
_toSyUSD(_underlyingBalance());
}
```

3S-Synnax-H07

First depositor share price manipulation through flashloan fee accumulation

Id	3S-Synnax-H07
Classification	High
Impact	High
Likelihood	Medium
Category	Bug
Status	Addressed in #40d2c7a.

Description

The **DegenBox** contract implements a vault system that manages token deposits using a share-based accounting mechanism. The **DegenBox::deposit** function allows users to deposit tokens and receive shares proportional to their contribution, where the share-to-token ratio is maintained through a **Rebase** struct containing **base** (total shares) and **elastic** (total token amount).

The root cause of this vulnerability is that the contract does not mint an initial amount of shares to a dead address when a new token is first deposited. This absence makes it economically feasible for an attacker to artificially inflate the price of a share by manipulating the ratio between **totals[token].elastic** and **totals[token].base**. The attacker can leverage the flashloan fee mechanism, which increases **elastic** without correspondingly increasing **base**, to establish an unfavorable share price before any legitimate users deposit. Once the share price is inflated, the attacker can exploit the integer division in **BoringRebase::toBase** to either steal funds from subsequent depositors or create precision loss that renders the vault impractical for normal use.

#### Steps to reproduce

Prerequisites:

- The **DegenBox** contract has no existing deposits for a specific token (e.g., WSEI)

- The attacker has access to external flashloan providers or sufficient capital

Attack scenario:

1. The attacker obtains 2,000,000e18 WSEI tokens from an external flashloan provider (not using **DegenBox::flashLoan**)

2. The attacker transfers all 2,000,000e18 WSEI directly to the **DegenBox** contract address

3. The attacker calls **DegenBox::flashLoan** to borrow 2,000,000e18 WSEI from the contract and repays it with the 0.05% fee

- Fee amount: **2_000_000e18 * 50 / 100_000 = 1_000e18** WSEI

- After flashloan: **totals[WSEI].elastic = 1_000e18** and **totals[WSEI].base = 0**

4. The attacker calls **DegenBox::deposit** with **share = 1000** (the **MINIMUM_SHARE_BALANCE**)

- The function calculates: **amount = total.toElastic(1000, true) = 1000** tokens (since **base == 0**, returns **elastic = base**)

- After deposit: **totals[WSEI].elastic = 1_000e18 + 1000** and **totals[WSEI].base = 1000**

- Share price: **(1_000e18 + 1000) / 1000 = 1e18 + 1** tokens per share

Impact analysis:

Impact 1: Fund theft through front-running

When a legitimate user attempts to deposit after the attacker has inflated the share price, the attacker can front-run and further manipulate **elastic** to cause share calculation rounding to zero:

1. Victim calls **deposit(token, victim, victim, 1000e18, 0)** attempting to deposit 1000e18 tokens

2. Attacker front-runs and repeats the flashloan manipulation multiple times to increase **totals[token].elastic** to an extremely high value (e.g., by executing multiple flashloans to accumulate fees and inflate **elastic** to 1e25 or higher)

3. Victim's transaction executes: **share = total.toBase(1000e18, false) = (1000e18 * 1000) / 1e25 = 0** (rounds down due to integer division)

4. The deposit function's check evaluates **total.base.add(0) < MINIMUM_SHARE_BALANCE**, which is **1000 < 1000 = false**, so the check passes

5. The victim receives 0 shares: **balanceOf[token][victim] = balanceOf[token][victim] + 0 = 0**

6. The victim's 1000e18 tokens are transferred to the contract and added to **totals[token].elastic**, but no corresponding shares are minted

7. The attacker, holding the only existing shares (1000 shares), can withdraw and claim the entire token balance including the victim's deposit

The root cause is that when **share == 0** is passed to **deposit**, the function calculates shares via **total.toBase(amount, false)**, which performs integer division that rounds down. With an inflated **elastic** and minimal **base**, even substantial deposit amounts result in zero shares, effectively donating the tokens to existing share holders.

Impact 2: Denial of service for small depositors and precision loss

With an inflated share price of **1e18 + 1** tokens per share, the protocol experiences severe usability and precision issues:

1. **Minimum deposit barrier:** Any deposit must result in at least 1 share to be meaningful. With a share price of **1e18 + 1** tokens per share, users must deposit more than **1e18** tokens. For an 18-decimal token, this represents at least 1 full token unit. Depending on token value, this could represent hundreds or thousands of dollars, effectively excluding small depositors and preventing incremental deposits.

2. **Precision loss:** The granularity of shares becomes extremely coarse. A 1 share difference represents approximately **1e18** tokens. This means:

   - Users cannot make precise deposits or withdrawals smaller than **1e18** token units

   - Rounding errors of up to **1e18** tokens (1 full token unit) can occur on each operation

---

Recommendation

Implement a virtual share offset mechanism to prevent share price manipulation. When the first deposit for a token occurs (when **total.base == 0**), mint an initial amount of shares to a dead address to establish a baseline share price that cannot be manipulated. Moreover, the function should early return if the **share** is zero to prevent any unintended donation to the contract.

```
function deposit(IERC20 token_, address from, address to, uint256 amount, uint256
share)
   public
   payable
   allowed(from)
   returns (uint256 amountOut, uint256 shareOut)
{
   // Checks
   require(to != address(0), "BentoBox: to not set");
   // Effects
   IERC20 token = token_ == USE_ETHEREUM ? wethToken : token_;
   _onBeforeDeposit(token, from, to, amount, share);
   Rebase memory total = totals[token];
   // If a new token gets added, the tokenSupply call checks that this is a deployed
contract
   require(total.elastic != 0 || token.totalSupply() > 0, "BentoBox: No tokens");

+   // First deposit: mint MINIMUM_SHARE_BALANCE to dead address to prevent
share price manipulation
+   bool isFirstDeposit = (total.base == 0);
   if (share == 0) {
        // value of the share may be lower than the amount due to rounding, that's ok
        share = total.toBase(amount, false);
+
+       // Handle first deposit case
+       if (isFirstDeposit) {
+           require(share > MINIMUM_SHARE_BALANCE, "BentoBox: First deposit too
small");
+           // Burn MINIMUM_SHARE_BALANCE shares to dead address
+           balanceOf[token][address(0xdead)] = MINIMUM_SHARE_BALANCE;
+           total.base = MINIMUM_SHARE_BALANCE.to128();
+           share = share - MINIMUM_SHARE_BALANCE;
+       }
+
        // Any deposit should lead to at least the minimum share balance
-       if (total.base.add(share.to128()) < MINIMUM_SHARE_BALANCE) {
+       if (share == 0 || total.base.add(share.to128()) < MINIMUM_SHARE_BALANCE) {
            return (0, 0);
        }
    } else {
        // amount may be lower than the value of share due to rounding, in that case,
add 1 to amount
        amount = total.toElastic(share, true);
+
+       // Handle first deposit case
+       if (isFirstDeposit) {
+           require(share > MINIMUM_SHARE_BALANCE, "BentoBox: First deposit too
small");
+           // Burn MINIMUM_SHARE_BALANCE shares to dead address
+           balanceOf[token][address(0xdead)] = MINIMUM_SHARE_BALANCE;
+           total.base = MINIMUM_SHARE_BALANCE.to128();
+           share = share - MINIMUM_SHARE_BALANCE;
+       }
    }
    ...
}
```

3S-Synnax-H08

Yield generated in DegenBox becomes inaccessible due to amount-based accounting

Id	3S-Synnax-H08
Classification	High
Impact	High
Likelihood	Medium
Category	Bug
Status	Addressed in #17433c2.

Description

The **SyUSDMultiCollateralManager** contract functions as a Peg Stability Module (PSM) that allows users to mint **syUSD** by depositing approved collateral tokens and redeem collateral by burning **syUSD**. All collateral is deposited into the **bentoBox** (DegenBox), which uses a share-based accounting system where shares can appreciate in value as the box generates yield from strategies.

The contract maintains two accounting variables for each collateral token: **totalCollateralShares** (tracking DegenBox shares) and **totalCollateralAmount** (tracking the absolute token amounts deposited). The issue arises because **totalCollateralAmount** records only the initial deposit amounts and does not account for yield accrued within DegenBox. When DegenBox shares appreciate in value, the actual token balance controlled by the contract exceeds **totalCollateralAmount**, causing the excess yield to become permanently inaccessible.

In **SyUSDMultiCollateralManager::completeRedeem**, the contract decreases **totalCollateralAmount** by the withdrawn amount. Similarly, in **SyUSDMultiCollateralManager::allocateCollateral**, it reduces **totalCollateralAmount** when allocating to strategies. When users attempt to withdraw amounts exceeding the tracked **totalCollateralAmount**, these operations revert due to arithmetic underflow, effectively locking the yield.

Steps to Reproduce:

1. Alice deposits 100 USDC via **mint()**. The contract receives 100 USDC, deposits it to DegenBox receiving 100 shares, and sets **totalCollateralAmount[USDC] = 100e6**.

2. DegenBox generates 10% yield on USDC through its strategy. The share-to-token ratio changes: 100 shares are now worth 110 USDC. The contract's actual USDC balance is 110 USDC, but **totalCollateralAmount[USDC]** remains at 100e6.

3. Bob attempts to redeem 100 USDC worth of **syUSD** via **initRedeem()** and **completeRedeem()**. When **completeRedeem()** withdraws 110 USDC from DegenBox (burning the corresponding shares), it attempts to execute:

```
totalCollateralAmount[token] -= 110e6;
```

4. This operation reverts with an underflow error because **totalCollateralAmount[USDC]** is only 100e6, making the 10 USDC yield permanently inaccessible to both users and strategies.

Recommendation

Remove the **totalCollateralAmount** state mapping and replace it with an external view function that dynamically calculates the collateral amount from shares using **DegenBox::toAmount**. This ensures the accounting always reflects the actual token balance including accrued yield.

```
    /* ========== STORAGE ========== */
    IBentoBoxV2 public bentoBox;
    IsyUSD public syUSD;

    mapping(address => Collateral) public collaterals;
-    mapping(address => uint256) public totalCollateralAmount;
    mapping(address => uint256) public totalCollateralShares;
    mapping(address => mapping(address => StrategyDebt)) public strategyDebt;
    mapping(address => mapping(address => PendingRedeem)) public
pendingRedeems;

+    /// @notice Returns the total collateral amount for a given token including yield
+    /// @param token The collateral token address
+    /// @return amount The total token amount corresponding to shares held
+    function totalCollateralAmount(address token) external view returns (uint256
amount) {
+        return bentoBox.toAmount(IERC20(token), totalCollateralShares[token], false);
+    }
+
    // ... existing code ...
```

Remove all direct assignments and arithmetic operations on **totalCollateralAmount**. This approach eliminates the synchronization issue by calculating collateral amounts on-demand from the authoritative share balance, ensuring yield is always accessible.

3S-Synnax-H09

Overly broad global pause on DegenBox transfers leads to blocked liquidations and potential bad debt

Id	3S-Synnax-H09
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #af5b76c.

Description

DegenBox is the shared token vault for the protocol, and **DegenBox::transfer** is the core primitive used by Cauldrons to move collateral and **syUSD** shares during normal operations and during liquidations. The global pause is implemented via the **paused** state variable and enforced in the **allowed** modifier, which is applied to **DegenBox::transfer** (and other entry points). When **DegenBox::setPause(true)** is called, the **allowed** modifier immediately reverts all calls to **DegenBox::transfer**, regardless of caller or purpose. Cauldrons such as **CauldronV4::liquidate** rely on **IBentoBoxV2::transfer** to move collateral from insolvent borrowers and repay their debt; these calls resolve to **DegenBox::transfer**, so liquidations are fully disabled while the vault is paused. This global behavior means that an owner-triggered pause intended to mitigate one issue (for example deposits or arbitrary user transfers) will also prevent liquidations from executing, allowing under-collateralized positions to persist and potentially accumulate bad debt for the system while the pause is active.

Recommendation

Refactor the pausability mechanism in **DegenBox** to distinguish between user-facing transfers and protocol-critical operations such as liquidations and repayments, instead of guarding all **DegenBox::transfer** calls behind a single **paused** flag.

3S-Synnax-M01

Incorrect principal/profit split in **_collectForUnstake** leads to delayed profit realization for unstaking users

Id	3S-Synnax-M01
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #79d0b80.

Description

SyUSDStrategyVault::_collectForUnstake is responsible for determining how much of the underlying that backs a given unstake request should be treated as returned principal versus realized profit, before calling **SyUSDMultiCollateralManager::returnPrincipalAndMint** and **SyUSDMultiCollateralManager::realizeProfit**. As of now, the function computes **needed** as the underlying amount corresponding to the user's **syUSD** to be unstaked, then splits it using the vault-wide debt: it fetches **debt = manager.strategyDebt(address(this), address(underlying))** and sets **principal = needed.min(debt)** and **profit = needed - principal**. Since **debt** represents the entire strategy principal of the vault and not just the portion attributable to the unstaking user, the first **needed** units returned by users are always classified as principal until the global **debt** has been fully repaid; user-level profit only starts to be realized after the aggregate vault debt is significantly reduced. This leads to profit realization being materially delayed in time for early unstakers, even if, economically, their position already contains a substantial profit.

Recommendation

The principal/profit split in **SyUSDStrategyVault::_collectForUnstake** should be computed proportionally based on the vault's aggregate unrealized PnL rather than by comparing the user's **needed** amount to the full vault-wide **manager.strategyDebt** balance. Use **_underlyingBalance()** together with the idle **underlying** balance and **manager.strategyDebt** to derive the total unrealized profit, then allocate the user's share of profit proportionally to **needed**, and treat the remainder as principal. A possible fix is:

```
function _collectForUnstake(uint256 syUSD) internal returns (uint256 principal,
uint256 profit) {
    uint256 needed = _toUnderlying(syUSD);
    if (needed == 0) return (0, 0);

-    uint256 idle = underlying.balanceOf(address(this));
-    if (idle < needed) _withdrawFromStrategy(needed - idle);
-
-    uint256 debt = manager.strategyDebt(address(this), address(underlying));
-    principal = needed.min(debt);
-    profit = needed - principal;
+    uint256 idle = underlying.balanceOf(address(this));
+    if (idle < needed) _withdrawFromStrategy(needed - idle);
+
+    uint256 totalDebt = manager.strategyDebt(address(this), address(underlying));
+    uint256 totalUnderlying = _underlyingBalance() +
underlying.balanceOf(address(this));
+    if (totalUnderlying <= totalDebt || totalDebt == 0) {
+        principal = needed;
+        profit = 0;
+    } else {
+        uint256 totalProfit = totalUnderlying - totalDebt;
+        profit = needed * totalProfit / totalUnderlying;
+        principal = needed - profit;
+    }
 }
```

3S-Synnax-M02

Share redemption and withdrawal operations revert due to precision requirements in asset conversion

Id	3S-Synnax-M02
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #347ab94.

Description

The **SyUSDStrategyVault** contract implements an ERC4626 vault that accepts **syUSD** (18 decimals) and converts it to underlying tokens with potentially different decimal precision. To ensure proper conversion, the contract uses a **_scalar** value calculated as **10 ** (18 - underlyingDecimals)** and enforces that all asset amounts must be perfectly divisible by this scalar through the **_checkConvertible** function.

The issue arises when users attempt to redeem or withdraw using share-based operations. The ERC4626 share-to-asset conversion involves division operations that can produce results with remainder values due to rounding. When these converted asset amounts are not perfectly divisible by **_scalar**, the **_checkConvertible** function reverts with a **Slippage()** error, preventing legitimate withdrawal operations.

This affects multiple critical user flows:

1. **SyUSDStrategyVault::mint** - When minting a specific number of shares, the converted asset amount may not satisfy the divisibility requirement

2. **SyUSDStrategyVault::redeem** - Direct share redemption converts shares to assets before checking divisibility

3. **SyUSDStrategyVault::burnSharesForSyUSD** - Router function that burns shares and checks the resulting asset amount

4. **SyUSDStrategyVault::mintSharesForSyUSD** - Router function that receives assets from share conversions in cross-vault operations

5. **SyUSDStrategyVault::initUnstake** - Cooldown-based withdrawal that converts shares to assets

6. **SyUSDVaultShareRouter::swap** - Cross-vault swap that relies on **burnSharesForSyUSD** and **mintSharesForSyUSD**

#### Steps to Reproduce

1. Deploy **SyUSDStrategyVault** with USDC (6 decimals) as the underlying token, resulting in **_scalar = 10^12**

2. User A deposits **1000e18** syUSD and receives shares

3. Due to the strategy's activity, the share price changes slightly

4. User A attempts to redeem their shares using **redeem(shares, receiver, owner)**

5. The **previewRedeem** function calculates assets, which may be a value like **99999999999999999999** (not divisible by **10^12**)

6. The **_beforeWithdraw** function calls **_checkConvertible(assets)**, which checks **assets % _scalar != 0**

7. Transaction reverts with **Slippage()** error despite being a legitimate redemption

The same issue occurs when using **mint** with a share amount that converts to non-divisible assets, or when initiating unstaking through **initUnstake**.

Additionally, for cross-vault swaps via **SyUSDVaultShareRouter::swap**:

1. Deploy two vaults: VaultA (USDC underlying) and VaultB (USDT underlying)

2. User holds shares in VaultA and calls **swap** to move to VaultB

3. VaultA's **burnSharesForSyUSD** converts shares to **syUSD** amount

4. This **syUSD** amount is passed to VaultB's **mintSharesForSyUSD**

5. Even if both vaults have the same **_scalar**, the intermediate conversions may produce amounts not perfectly divisible by VaultB's scalar

6. Transaction reverts with **Slippage()** error, preventing legitimate vault-to-vault migrations

---

Recommendation

Remove **_checkConvertible** from internal functions and apply validation at the public function level. For share-based operations (**mint**, **redeem**, **burnSharesForSyUSD**), adjust the shares downward to the nearest value that produces convertible assets. For asset-based operations (**deposit**, **withdraw**), validate that user-provided assets are convertible. For router operations, round assets down and refund unused shares.

```
// Add new helper to round down shares to produce convertible assets
function _roundSharesToConvertible(uint256 shares) internal view returns (uint256) {
    uint256 assets = previewRedeem(shares);
    uint256 convertibleAssets = _roundToConvertible(assets);
    return convertibleAssets > 0 ? previewWithdraw(convertibleAssets) : 0;
}
function _roundToConvertible(uint256 syUSD) internal view returns (uint256) {
    return (syUSD / _scalar) * _scalar;
}
function deposit(uint256 assets, address receiver)
    public override whenNotPaused nonReentrant returns (uint256) {
+   _checkConvertible(assets);
    return super.deposit(assets, receiver);
}
function mint(uint256 shares, address receiver)
    public override whenNotPaused nonReentrant returns (uint256) {
-   return super.mint(shares, receiver);
+   uint256 adjustedShares = _roundSharesToConvertible(shares);
+   if (adjustedShares == 0) revert ZeroAmount();
+   if (adjustedShares < shares) {
+       // User asked for more shares than convertible amount allows
+       uint256 assets = previewMint(adjustedShares);
+       if (assets == 0) revert ZeroAmount();
+   }
+   return super.mint(adjustedShares, receiver);
}
function withdraw(uint256 assets, address receiver, address owner)
    public override nonReentrant noCooldown returns (uint256 shares) {
+   _checkConvertible(assets);
    shares = previewWithdraw(assets);
    _withdraw(msg.sender, receiver, owner, assets, shares);
}
function redeem(uint256 shares, address receiver, address owner)
    public override nonReentrant noCooldown returns (uint256 assets) {
-   assets = previewRedeem(shares);
+   uint256 adjustedShares = _roundSharesToConvertible(shares);
+   if (adjustedShares == 0) revert ZeroAmount();
+   assets = previewRedeem(adjustedShares);
-   _withdraw(msg.sender, receiver, owner, assets, shares);
+   _withdraw(msg.sender, receiver, owner, assets, adjustedShares);
}
function mintSharesForSyUSD(address payer, address receiver, uint256 assets,
uint256 minShares)
    external onlyRole(ROUTER_ROLE) returns (uint256 shares) {
-   _checkConvertible(assets);
+   // Round down assets from router operations (no revert, just adjust)
+   uint256 convertibleAssets = _roundToConvertible(assets);
+   if (convertibleAssets == 0) revert ZeroAmount();

-   shares = previewDeposit(assets);
+   shares = previewDeposit(convertibleAssets);
    if (shares < minShares) revert Slippage();
-   _deposit(payer, receiver, assets, shares);
+   _deposit(payer, receiver, convertibleAssets, shares);
}
function burnSharesForSyUSD(address owner, address receiver, uint256 shares,
uint256 minAssets)
    external onlyRole(ROUTER_ROLE) returns (uint256 assets) {
-   assets = previewRedeem(shares);
-   _checkConvertible(assets);
+   // Adjust shares to nearest convertible amount
+   uint256 adjustedShares = _roundSharesToConvertible(shares);
+   if (adjustedShares == 0) revert ZeroAmount();
+   assets = previewRedeem(adjustedShares);
    if (assets < minAssets) revert Slippage();
-   _withdraw(msg.sender, receiver, owner, assets, shares);
+   _withdraw(msg.sender, receiver, owner, assets, adjustedShares);
}
function initUnstake(uint256 shares, uint256 minSyUSD, address receiver)
    external nonReentrant whenNotPaused returns (uint256 syUSD) {
    if (unstakeCooldown == 0 || shares == 0 || receiver == address(0)) revert
InvalidZero();
    if (pendingUnstakes[msg.sender].shares > 0) revert AlreadyPending();
-   syUSD = previewRedeem(shares);
-   _checkConvertible(syUSD);
+   uint256 adjustedShares = _roundSharesToConvertible(shares);
+   if (adjustedShares == 0) revert ZeroAmount();
+   syUSD = previewRedeem(adjustedShares);
    if (syUSD < minSyUSD) revert Slippage();
    uint64 day = uint64(block.timestamp / 1 days);
    if (maxUnstakePerDay < type(uint256).max) {
        uint256 total = unstakedPerDay[day] + syUSD;
        if (total > maxUnstakePerDay) revert LimitExceeded();
        unstakedPerDay[day] = total;
    }
    (uint256 principal, uint256 profit) = _collectForUnstake(syUSD);
    if (principal + profit == 0) revert NoLiquidity();
-   _burn(msg.sender, shares);
+   _burn(msg.sender, adjustedShares);
    pendingUnstakes[msg.sender] = Unstake({
-       shares: shares,
+      shares: adjustedShares,
       principal: principal,
       profit: profit,
       syUSDExpected: syUSD,
       receiver: receiver,
       readyAt: uint64(block.timestamp + unstakeCooldown),
       dayIndex: day
   });
-  emit UnstakeInitiated(msg.sender, shares, syUSD, receiver,
uint64(block.timestamp + unstakeCooldown));
+   emit UnstakeInitiated(msg.sender, adjustedShares, syUSD, receiver,
uint64(block.timestamp + unstakeCooldown));
}
function _afterDeposit(uint256 assets) internal {
-  _checkConvertible(assets);
+   // No check needed - validation happens at public function level
   IsyUSD(address(asset())).burn(assets);
   uint256 underlyingAmt = _toUnderlying(assets);
   if (underlyingAmt == 0) return;
   uint256 borrowed = manager.allocateCollateral(address(underlying),
underlyingAmt, address(this));
   if (borrowed > 0) _deployToStrategy(borrowed);
}
function _beforeWithdraw(uint256 assets) internal returns (uint256 minted) {
-  _checkConvertible(assets);
+   // No check needed - validation happens at public function level
   (uint256 principal, uint256 profit) = _collectForUnstake(assets);
   if (principal > 0)
       minted += manager.returnPrincipalAndMint(address(underlying), principal,
address(this));
   if (profit > 0)
       minted += manager.realizeProfit(address(underlying), profit, address(this));
}
```

For the **SyUSDVaultShareRouter::swap** function, refund unused shares:

```
function swap(
   SyUSDStrategyVault vaultIn,
   SyUSDStrategyVault vaultOut,
   uint256 sharesIn,
   uint256 minSharesOut,
   address recipient
) external nonReentrant returns (uint256 sharesOut) {
   if (sharesIn == 0) revert ZeroAmount();
   if (recipient == address(0)) revert ZeroRecipient();
   if (vaultIn == vaultOut) revert SameVault();
   if (vaultIn.asset() != address(syUSD) || vaultOut.asset() != address(syUSD)) revert
InvalidVault();
   // 1. Pull vaultIn shares
   IERC20(address(vaultIn)).safeTransferFrom(msg.sender, address(this), sharesIn);
   // 2. Burn → get syUSD (may use fewer shares due to rounding)
   uint256 syUSDBefore = syUSD.balanceOf(address(this));
+   uint256 sharesBefore = IERC20(address(vaultIn)).balanceOf(address(this));
   vaultIn.burnSharesForSyUSD(address(this), address(this), sharesIn, 0);
   uint256 syUSDAmount = syUSD.balanceOf(address(this)) - syUSDBefore;
+   uint256 sharesUsed = sharesBefore -
IERC20(address(vaultIn)).balanceOf(address(this));
   if (syUSDAmount == 0) revert ZeroAmount();
   // 3. Mint new vault shares
   syUSD.safeApprove(address(vaultOut), syUSDAmount);
   sharesOut = vaultOut.mintSharesForSyUSD(address(this), recipient,
syUSDAmount, minSharesOut);
-   // 4. Refund dust (extremely rare, but safe)
+   // 4. Refund unused shares and dust syUSD
+   uint256 unusedShares = sharesIn - sharesUsed;
+   if (unusedShares > 0) {
+       IERC20(address(vaultIn)).safeTransfer(msg.sender, unusedShares);
+   }
   uint256 leftover = syUSD.balanceOf(address(this));
   if (leftover > 0) syUSD.safeTransfer(msg.sender, leftover);
   emit Swap(msg.sender, vaultIn, vaultOut, sharesUsed, syUSDAmount, sharesOut,
recipient);
}
```

This approach ensures that all operations use convertible amounts while maintaining user protections through slippage parameters. Share-based operations automatically adjust to the nearest valid amount, and any unused shares are properly refunded to users.

3S-Synnax-M03

Insufficient vault validation allows bypassing unstake flow through malicious vault contracts

Id	3S-Synnax-M03
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #f3ce160.

Description

The **SyUSDVaultShareRouter::swap** function enables users to swap vault shares between different strategy vaults in a single transaction. The function burns shares from **vaultIn** to retrieve **syUSD**, then mints shares in **vaultOut** using the retrieved **syUSD**. Any leftover **syUSD** is refunded to the caller.

The current implementation only validates that both **vaultIn** and **vaultOut** return the **syUSD** address when calling their **asset()** function. However, there is no verification that these vault addresses correspond to legitimate protocol-deployed contracts. This allows an attacker to provide a malicious **vaultOut** contract under their control to extract **syUSD** without following the intended unstake flow.

When a user swaps from a legitimate **vaultIn** to a malicious **vaultOut**, the legitimate vault shares are burned and converted to **syUSD**. The malicious **vaultOut** contract can implement a no-op **mintSharesForSyUSD** function that accepts the **syUSD** approval but does not transfer the tokens. Consequently, the **syUSD** remains in the router contract and is refunded to **msg.sender** via the dust refund mechanism, effectively allowing the attacker to instantly redeem their vault shares without adhering to any unstake delay or restrictions that may exist in the protocol.

#### Steps to reproduce

1. Deploy a malicious vault contract with the following characteristics:

   - Implements an **asset()** function that returns the **syUSD** token address

   - Implements a **mintSharesForSyUSD()** function that does nothing and returns 0

2. Acquire legitimate vault shares from a protocol-deployed **vaultIn** contract

3. Call **SyUSDVaultShareRouter::swap** with:

- **vaultIn**: legitimate protocol vault address
- **vaultOut**: malicious vault address
- **sharesIn**: amount of shares to redeem
- **minSharesOut**: 0
- **recipient**: attacker's address

4. The router burns the legitimate **vaultIn** shares and receives **syUSD**
5. The router approves **syUSD** to the malicious **vaultOut** but receives 0 shares back
6. The entire **syUSD** amount remains in the router and is refunded to the attacker
7. The attacker successfully extracts **syUSD** without following the proper unstake flow

Recommendation

Implement a whitelist mechanism to ensure only protocol-approved vault contracts can be used in the **swap** function. This can be achieved by maintaining a mapping of authorized vaults or requiring vaults to be registered through a controlled registry.

```
 contract SyUSDVaultShareRouter is ReentrancyGuard {
    using SafeERC20 for IERC20;
    IERC20 public immutable syUSD;
+    mapping(address => bool) public isAuthorizedVault;
+    address public owner;
+    event VaultAuthorized(address indexed vault, bool authorized);
+
+    error UnauthorizedVault();
+    error Unauthorized();
+
+    modifier onlyOwner() {
+        if (msg.sender != owner) revert Unauthorized();
+        _;
+    }
    constructor(IERC20 syUSD_) {
        syUSD = syUSD_;
+        owner = msg.sender;
    }
+    function setVaultAuthorization(address vault, bool authorized) external
onlyOwner {
+        isAuthorizedVault[vault] = authorized;
+        emit VaultAuthorized(vault, authorized);
+    }
```

```
    function swap(
        SyUSDStrategyVault vaultIn,
        SyUSDStrategyVault vaultOut,
        uint256 sharesIn,
        uint256 minSharesOut,
        address recipient
    ) external nonReentrant returns (uint256 sharesOut) {
        if (sharesIn == 0) revert ZeroAmount();
        if (recipient == address(0)) revert ZeroRecipient();
        if (vaultIn == vaultOut) revert SameVault();
        if (vaultIn.asset() != address(syUSD) || vaultOut.asset() != address(syUSD))
revert InvalidVault();
+        if (!isAuthorizedVault[address(vaultIn)] ||
!isAuthorizedVault[address(vaultOut)]) revert UnauthorizedVault();
        // ... existing code ...
    }
 }
```

3S-Synnax-M04

Missing slippage protection in swap functions

Id	3S-Synnax-M04
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #40d25bf.

Description

The **SyUSDStableMinting** contract implements a Peg Stability Module (PSM) that allows users to swap between **syUSD** and stable tokens at a rate determined by oracle prices and configurable fees. Two core functions enable this bidirectional swap:

- **SyUSDStableMinting::swapSYUSDForStable** - Allows users to exchange **syUSD** for stable tokens

- **SyUSDStableMinting::swapStableForSYUSD** - Allows users to exchange stable tokens for **syUSD**

The functions calculate output amounts based on the current oracle price (via **_previewTokenUSDAmount**) and apply dynamic fees (via **_calculateFee**). However, neither function accepts a minimum output amount parameter (**minAmountOut**), leaving users vulnerable to unfavorable price movements between transaction submission and execution.

Between the time a user submits their transaction and when it gets executed on-chain, the following can occur:

1. The contract owner updates the fee structure via **setFeeRate**

2. The oracle price updates, changing the conversion rate via **_previewTokenUSDAmount**

This results in users receiving significantly less output tokens than anticipated, with no ability to revert the transaction if the slippage exceeds their tolerance.

Recommendation

Add a **minAmountOut** parameter to both swap functions to allow users to specify the minimum acceptable output amount. The transaction should revert if the calculated output amount is less than the user's specified minimum.

```
-    function swapSYUSDForStable(address receiver, uint256 syUSDAmount) external isActive nonReentrant {
+    function swapSYUSDForStable(address receiver, uint256 syUSDAmount, uint256 minStableAmountOut) external isActive nonReentrant {
         _ensureNonzeroAddress(receiver);
         _ensureNonzeroAmount(syUSDAmount);
         // Calculate stable amount với fees
         uint256 stableTokenAmountUSD = _previewTokenUSDAmount(syUSDAmount, FeeDirection.OUT);
         uint256 fee = _calculateFee(stableTokenAmountUSD, FeeDirection.OUT);
         uint256 stableAmountReceived = stableTokenAmountUSD - fee;
+
+        require(stableAmountReceived >= minStableAmountOut, "PSM: Insufficient output amount");
         // ... rest of function
     }
-    function swapStableForSYUSD(address receiver, uint256 stableAmount) external isActive nonReentrant {
+    function swapStableForSYUSD(address receiver, uint256 stableAmount, uint256 minSyUSDAmountOut) external isActive nonReentrant {
         _ensureNonzeroAddress(receiver);
         _ensureNonzeroAmount(stableAmount);
         // Transfer stable token từ user → PSM
         IERC20(STABLE_TOKEN).safeTransferFrom(msg.sender, address(this), stableAmount);
         // Deposit stable vào DegenBox (PSM account)
         IERC20(STABLE_TOKEN).approve(address(bentoBox), stableAmount);
         bentoBox.deposit(IERC20(STABLE_TOKEN), address(this), address(this), stableAmount, 0);
         IERC20(STABLE_TOKEN).approve(address(bentoBox), 0);
         // Calculate syUSD amount với fees
         uint256 stableAmountInUSD = _previewTokenUSDAmount(stableAmount, FeeDirection.IN);
         uint256 syUSDFee = _calculateFee(stableAmountInUSD, FeeDirection.IN);
         uint256 syUSDAmountReceived = stableAmountInUSD - syUSDFee;
+
+        require(syUSDAmountReceived >= minSyUSDAmountOut, "PSM: Insufficient output amount");
         // ... rest of function
}
```

3S-Synnax-M05

syUSD tokens not burned during liquidation process

Id	3S-Synnax-M05
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Addressed in #0d5f112.

Description

The **CauldronV4::liquidate** function manages the liquidation of undercollateralized positions by allowing liquidators to repay users' debt in exchange for their collateral. When a liquidation occurs, the function calculates the total amount of **syUSD** debt to be repaid (including a distribution fee), receives the **syUSD** tokens from the liquidator via the **bentoBox**, but does not burn the repaid debt amount as it should.

In contrast, the **_repay** function properly handles debt repayment by withdrawing the **syUSD** tokens from the **bentoBox** and calling the **burn** function on the **syUSD** token contract. The **liquidate** function does not follow this pattern, leaving the received **syUSD** tokens in the **bentoBox** under the cauldron's balance.

This discrepancy results in two issues:

- The total supply of **syUSD** becomes inflated since repaid debt is not removed from circulation

- If the **bentoBox** implements yield farming strategies for **syUSD**, the cauldron contract accumulates unintended profits on these unburned tokens, which dilutes the earnings of legitimate **syUSD** depositors

Recommendation

Add a pending burn mechanism to handle cases where **syUSD** liquidity might be allocated to strategies in the **bentoBox**. This approach uses a try-catch block to attempt immediate burning, and defers to a later burn if liquidity is insufficient:

```
+   // Track syUSD that needs to be burned but couldn't due to insufficient liquidity
+   uint256 public pendingBurnShare;
+
+   event LogPendingBurn(uint256 share);
+   event LogBurnProcessed(uint256 amount);
        ...
        totalBorrow.elastic = totalBorrow.elastic.sub(allBorrowAmount.to128());
        totalBorrow.base = totalBorrow.base.sub(allBorrowPart.to128());
        totalCollateralShare = totalCollateralShare.sub(allCollateralShare);
        // Apply a percentual fee share to sSpell holders
+        uint256 distributionAmount;
        {
-           uint256 distributionAmount = (
+            distributionAmount = (
                allBorrowAmount.mul(LIQUIDATION_MULTIPLIER) /
LIQUIDATION_MULTIPLIER_PRECISION
            ).sub(allBorrowAmount).mul(DISTRIBUTION_PART) /
DISTRIBUTION_PRECISION; // Distribution Amount
            allBorrowAmount = allBorrowAmount.add(distributionAmount);
            accrueInfo.feesEarned =
accrueInfo.feesEarned.add(distributionAmount.to128());
        }
        uint256 allBorrowShare = bentoBox.toShare(syUSD, allBorrowAmount, true);
        // Swap using a swapper freely chosen by the caller
        // Open (flash) liquidation: get proceeds first and provide the borrow after
        if (swapper != ISwapperV2(address(0))) {
            bentoBox.transfer(collateral, address(this), address(swapper),
allCollateralShare);
            swapper.swap(
                address(collateral), address(syUSD), msg.sender, allBorrowShare,
allCollateralShare, swapperData
            );
        } else {
            bentoBox.transfer(collateral, address(this), to, allCollateralShare);
        }
        // @audit remove redundant calculation
-       allBorrowShare = bentoBox.toShare(syUSD, allBorrowAmount, true);
        bentoBox.transfer(syUSD, msg.sender, address(this), allBorrowShare);
+        // Burn the debt portion (excluding distribution fee which remains as protocol
fees)
+        uint256 burnAmount = allBorrowAmount.sub(distributionAmount);
+        uint256 burnShare = bentoBox.toShare(syUSD, burnAmount, false);
+
+       // Try to withdraw and burn. If it fails due to insufficient liquidity, track it for later
+       try bentoBox.withdraw(syUSD, address(this), address(this), 0, burnShare) {
+           IsyUSD(address(syUSD)).burn(burnAmount);
+           emit LogBurnProcessed(burnAmount);
+       } catch {
+           // If withdraw fails (e.g., syUSD is in strategy), add to pending burns
+           pendingBurnShare = pendingBurnShare.add(burnShare);
+           emit LogPendingBurn(burnShare);
+       }
+   }
+
+   /// @notice Processes pending burn amounts when sufficient liquidity is available
+   /// @dev Can be called by anyone to burn accumulated syUSD from liquidations
+   function processPendingBurns() external {
+       uint256 pending = pendingBurnShare;
+       require(pending > 0, "Cauldron: no pending burns");
+
+       (uint256 burnAmount,) = bentoBox.withdraw(syUSD, address(this),
address(this), 0, pending);
+       IsyUSD(address(syUSD)).burn(burnAmount);
+
+       pendingBurnShare = 0;
+       emit LogBurnProcessed(burnAmount);
+   }
```

This solution ensures that:

- The repaid debt is burned immediately when liquidity is available

- If **bentoBox** has allocated **syUSD** to strategies, the burn is deferred and tracked in **pendingBurnShare**

- Anyone can call **processPendingBurns** when sufficient liquidity becomes available

3S-Synnax-M06

Single collateral **enabled** flag gating both mint and redeem leads to blocked redemptions and potential peg instability

Id	3S-Synnax-M06
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Addressed in #463167f.

Single collateral **enabled** flag gating both mint and redeem leads to blocked redemptions and potential peg instability

Description

SyUSDMultiCollateralManager manages which collateral assets can be used to mint and redeem **syUSD** via the **collaterals[token].enabled** flag, controlled by **SyUSDMultiCollateralManager::updateCollateral**. This flag is consumed through the **validateCollateral** modifier, which is applied uniformly to both the mint path (**mint**) and the redemption path (**initRedeem** and **completeRedeem**).

When a collateral is disabled via **updateCollateral**, new mints are correctly blocked, but redemptions for that collateral are also blocked because both **initRedeem** and **completeRedeem** revert on **validateCollateral**. As a result, any portion of **syUSD** backed by that collateral cannot be redeemed through this PSM, which can exacerbate a depeg and effectively lock part of the backing. While **cancelRedeem** remains callable and re-mints **syUSD**, users that want to exit into the disabled collateral cannot do so, and protocol operators are forced to choose between allowing new exposure (keeping **enabled=true**) or halting redemptions along with mints.

Recommendation

Introduce separate control flags for minting and redemption on a per-collateral basis, so that disabling new exposure does not prevent users and strategies from redeeming and burning **syUSD**. Concretely, extend the **Collateral** struct to include distinct **mintEnabled** and **redeemEnabled** booleans (or equivalent), add separate modifiers such as **validateMintCollateral** and **validateRedeemCollateral**, and apply them only to the respective flows. Additionally, **SyUSDMultiCollateralManager::addCollateral** and

SyUSDMultiCollateralManager::updateCollateral should be updated to set and manage the two flags explicitly rather than a single **enabled** boolean.

3S-Synnax-M07

Incorrect accounting in cancelRedeem inflates totalMinted and permanently reduces minting capacity

Id	3S-Synnax-M07
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #66f2c2b.

Description

The **SyUSDMultiCollateralManager** contract manages a multi-collateral PSM (Peg Stability Module) where users can mint and redeem syUSD tokens. The redemption process is split into two phases: **initRedeem** initiates the redemption by burning syUSD and queuing the collateral withdrawal, while **completeRedeem** finalizes it after a cooldown period. Users can also cancel pending redemptions via **cancelRedeem**.

The **totalMinted** variable tracks the total amount of syUSD minted through the PSM and is used to enforce the **globalMintCap** in the **mint** function. The accounting logic contains a flaw in how **totalMinted** is updated across the redemption lifecycle:

- When **SyUSDMultiCollateralManager::initRedeem** is called, the syUSD tokens are burned immediately, but **totalMinted** remains unchanged

- When **SyUSDMultiCollateralManager::completeRedeem** is called, **totalMinted** is decreased by the redeemed amount

- When **SyUSDMultiCollateralManager::cancelRedeem** is called, syUSD tokens are re-minted to the user and **totalMinted** is increased by the returned amount

This creates an accounting mismatch: canceling a redemption increases **totalMinted** even though the original **initRedeem** never decreased it. Each cancelled redemption permanently inflates **totalMinted** by the redemption amount, artificially consuming the available minting capacity.

Steps to Reproduce:

1. Assume **totalMinted** is initially 1000 syUSD and **globalMintCap** is 10000 syUSD

2. User calls **SyUSDMultiCollateralManager::initRedeem** with 500 syUSD

- 500 syUSD is burned from the user

- **totalMinted** remains 1000 (unchanged)

3. User calls **SyUSDMultiCollateralManager::cancelRedeem** for the same token

- 500 syUSD is minted back to the user

- **totalMinted** increases to 1500

4. Result: **totalMinted** is now 1500 instead of 1000, permanently reducing the remaining mint capacity by 500 syUSD

5. If repeated multiple times, **totalMinted** can reach **globalMintCap**, preventing all future **mint** and **cancelRedeem** operations even though the actual circulating supply hasn't increased

---

Recommendation

Decrease **totalMinted** when syUSD is burned in **SyUSDMultiCollateralManager::initRedeem**, rather than when the redemption is completed. This ensures that **cancelRedeem** correctly restores the state without inflating the counter.

```
function initRedeem(
    address token,
    uint256 syUSDIn,
    uint256 minOut,
    address to
) external whenNotPaused nonReentrant validateCollateral(token) updateBlock {
    if (syUSDIn == 0 || to == address(0)) revert ZeroAmount();
    Collateral storage c = collaterals[token];
    uint256 price = _oraclePrice(c, false);
    uint256 grossCollateral = (syUSDIn * 10 ** c.decimals) / price;
    uint256 fee = grossCollateral * feeOut / BPS;
    uint256 netOut = grossCollateral - fee;
    if (netOut < minOut) revert Slippage();
    if (redeemedThisBlock + syUSDIn > maxRedeemPerBlock) revert LimitExceeded();
    IERC20(address(syUSD)).safeTransferFrom(msg.sender, address(this), syUSDIn);
    syUSD.burn(syUSDIn);
+    totalMinted -= syUSDIn;
+    collaterals[token].psmMinted = collaterals[token].psmMinted >= syUSDIn
+        ? collaterals[token].psmMinted - syUSDIn
+        : 0;
    PendingRedeem storage p = pendingRedeems[msg.sender][token];
    // Ensure clean initialization
    if (p.receiver == address(0)) {
        p.receiver = to;
    }
    p.syUSD += syUSDIn;
    p.collateral += netOut; // net collateral after fee
    p.fee += fee; // fee amount
    lastAction[msg.sender][token] = block.timestamp;
    redeemedThisBlock += syUSDIn;
    emit RedeemInitiated(msg.sender, token, syUSDIn, grossCollateral);
}
function completeRedeem(address token) external whenNotPaused nonReentrant
validateCollateral(token) returns (uint256 out) {
    if (block.timestamp < lastAction[msg.sender][token] + cooldown) revert
CooldownActive();
    PendingRedeem memory p = pendingRedeems[msg.sender][token];
    if (p.collateral == 0) revert ZeroAmount();
    out = p.collateral; // net collateral (fee already deducted in initRedeem)
    delete pendingRedeems[msg.sender][token];
    delete lastAction[msg.sender][token];
    // Withdraw net collateral to user
    (uint256 withdrawnUser, uint256 shareUser) = bentoBox.withdraw(IERC20(token),
address(this), p.receiver, out, 0);
    // Withdraw fee to treasury if any
    uint256 shareFee = 0;
    uint256 withdrawnFee = 0;
    if (p.fee > 0) {
        (withdrawnFee, shareFee) = bentoBox.withdraw(IERC20(token), address(this),
treasury, p.fee, 0);
    }
    // Update accounting
    totalCollateralShares[token] -= (shareUser + shareFee);
    totalCollateralAmount[token] -= (withdrawnUser + withdrawnFee);
-   totalMinted -= p.syUSD;
-   collaterals[token].psmMinted = collaterals[token].psmMinted >= p.syUSD
-       ? collaterals[token].psmMinted - p.syUSD
-       : 0;
    emit RedeemCompleted(msg.sender, token, out, p.fee);
}
```

3S-Synnax-M08

Missing migration-origin check in **rollbackUserMigration** allows master contract owner to erase arbitrary positions and confiscate collateral

Id	3S-Synnax-M08
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Addressed in #dec0257.

Description

CauldronV5Migratable::rollbackUserMigration is intended as an emergency tool to undo incorrectly migrated positions from a V4 cauldron into the V5 migratable cauldron. The function is gated only by **onlyMasterContractOwner**, which in **CauldronV4** checks **msg.sender == masterContract.owner()**, i.e. the global cauldron master owner defined by the **Owned** base contract, and by a simple **migrationCompleted** flag. However, **rollbackUserMigration** never checks that the target account was actually created by the migration process, even though the migration path explicitly sets **userMigrated[user] = true** in **CauldronV5Migratable::_applyMigratedPosition** and exposes this via **isUserMigrated**. As a result, the master contract owner can call **rollbackUserMigration** for any user, regardless of whether their position originated from **batchMigrateFromV4** or was opened natively in V5.

Internally, **rollbackUserMigration** reads **userCollateralShare[user]** and **userBorrowPart[user]** from the inherited **CauldronV4** storage, subtracts these from **totalCollateralShare** and **totalBorrow** respectively, and then zeroes the user-level mappings. No collateral is transferred out of BentoBox and no syUSD is burned or repaid in this function, so the actual collateral held at **bentoBox.balanceOf(collateral, address(this))** remains under the cauldron's control while the user's accounting entries are erased, and the aggregate debt tracked in **totalBorrow** is reduced as if it had been repaid. This is ok if performed over a migrated user, since the collateral still has to be added to the new DegenBox; if performed on user positions not created through migration anyway, this effectively allows the master contract owner to arbitrarily wipe any user's position, confiscate their claim on collateral, and forgive their outstanding debt at the accounting level, which is a strong centralization and integrity risk that exceeds the stated purpose of "fix incorrect migration".

Recommendation

Restrict **CauldronV5Migratable::rollbackUserMigration** so that it can only be used on accounts that were actually migrated from V4, by enforcing **userMigrated[user] == true** before modifying **userCollateralShare** and **userBorrowPart**, and revert otherwise using a dedicated error. This aligns the function's effective scope with its documented purpose and prevents the master contract owner from using it as a generic position wipe mechanism, shielding from human errors.

3S-Synnax-M09

Missing enforcement of collateral migration verification allows CauldronV5Migratable to operate with underfunded DegenBox collateral

Id	3S-Synnax-M09
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Addressed in #dec0257.

Description

CauldronV5Migratable extends **CauldronV5** to migrate user positions from an old **CauldronV4** into a new cauldron instance, while the associated collateral tokens in **DegenBox** are expected to be manually moved by protocol administrators from the old vault to the new one. The contract provides **CauldronV5Migratable::verifyCollateralBalance**, which compares **v4.totalCollateralShare()** against **bentoBox.balanceOf(collateral, address(this))** to detect whether the new cauldron's backing collateral in **DegenBox** is sufficient, and also has an internal **_verifyCollateralBalance** helper that reverts if the new **DegenBox** balance is too low.

However, this verification is never enforced in the migration flow or in the activation of the new cauldron: **_verifyCollateralBalance** is commented out in **CauldronV5Migratable::batchMigrateFromV4**, **verifyCollateralBalance** is a pure view helper not wired into any state-changing path, and **migrationCompleted** is only used as a flag to stop further migrations rather than as a prerequisite for enabling user operations. As a result, it is possible for administrators to fully migrate accounting data (users' **userCollateralShare**, **userBorrowPart**, and **totalBorrow**) and start using the new cauldron (e.g., via **CauldronV5::borrow**, **CauldronV4::cook** actions, and liquidations) even if the corresponding collateral tokens have not yet been transferred, or have been incompletely transferred, into the new **DegenBox** vault. This leads to a situation where on-chain accounting records in **CauldronV5Migratable** assume more collateral than is actually held in **DegenBox**, breaking the implicit invariant that **bentoBox.balanceOf(collateral, address(this))** should cover **totalCollateralShare**, and enabling undercollateralized borrowing or failed withdrawals and liquidations once users begin interacting with the migrated cauldron.

Recommendation

Enforce collateral-balance verification as part of the migration lifecycle and as a prerequisite for enabling user-facing operations on the new cauldron, instead of leaving **CauldronV5Migratable::verifyCollateralBalance** as an informational view-only helper. One robust approach is to introduce a dedicated **bool collateralVerified** flag in **CauldronV5Migratable**, set only after **_verifyCollateralBalance** succeeds for the chosen **v4Cauldron** (for example in a new **finalizeMigration(address v4Cauldron)** function restricted to **onlyMasterContractOwner**), and then gate all user operations that rely on correct collateral backing with a modifier such as **whenCollateralVerified** that requires **collateralVerified == true** before allowing **CauldronV4::borrow**, **CauldronV4::cook** actions that change borrow or collateral state, and liquidation functions to execute. In addition, the commented **_verifyCollateralBalance** call in **CauldronV5Migratable::batchMigrateFromV4** should either be re-enabled (to prevent starting a batch migration when the collateral has not yet been moved into the new **DegenBox**) or replaced by a clear two-phase process where accounting migration is performed first, manual collateral transfer is executed off-chain, and then **finalizeMigration** calls **_verifyCollateralBalance** and flips **collateralVerified** to unblock the cauldron. This ensures that the new cauldron cannot be used in production unless **bentoBox.balanceOf(collateral, address(this))** and the migrated **totalCollateralShare** are consistent with the old **CauldronV4**'s state.

Finally, once the migration has been finalized and **collateralVerified** has been set to **true**, the migration process itself should be irreversibly blocked: state-changing migration functions such as **batchMigrateFromV4** and **rollbackUserMigration** should explicitly revert when **collateralVerified == true** (e.g. **require(!collateralVerified, "migration finalized");**), so that the system cannot accidentally or maliciously re-open or mutate migrated positions after the new cauldron is live.

3S-Synnax-M10

Missing syUSD exclusion in DegenBox flash loan functions allows syUSD to be flash-loaned despite documentation

Id	3S-Synnax-M10
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #cb4c547.

Description

DegenBox::flashLoan and **DegenBox::batchFlashLoan** implement the generic flash loan functionality for any **IERC20** held in the vault: they compute a fee, transfer the requested **token** to the **receiver**, invoke the borrower callback, and finally require that **_tokenBalanceOf(token)** is at least the original elastic total plus the fee. The NatSpec comments for both functions explicitly state that the **syUSD** token is not supported, but the implementation never checks whether the **token** argument (or any element in the **tokens** array) corresponds to **syUSD**. In practice, once **syUSD** has been minted into and deposited in **DegenBox** (via **DegenBox::mintForCauldron** and subsequent deposits), it can be flash-loaned like any other asset. This discrepancy between documentation and behavior can lead integrators, auditors, or higher-level protocol code to incorrectly assume that **syUSD** is non-flash-loanable and design invariants or safeguards under that assumption, potentially enabling unexpected behaviors if those assumptions are exploited.

Recommendation

Clarify whether **syUSD** is intended to be flash-loanable and align the implementation and documentation accordingly. If **syUSD** should not be flash-loanable, introduce an explicit way for **DegenBox** to know the **syUSD** token address (for example, by adding a dedicated **syUSD** state variable or a small registry of disallowed tokens) and add **require** checks in both **DegenBox::flashLoan** and **DegenBox::batchFlashLoan** that reject requests where the **token** (or any **tokens[i]**) equals **syUSD**. If instead **syUSD** flash loans are acceptable under the protocol's threat model, update the NatSpec comments and any external documentation that currently claims **syUSD** is not supported so that downstream code and risk analysis do not rely on an inaccurate non-flash-loanability guarantee.

3S-Synnax-L01

Harvest front-running allows depositors to extract value or evade losses

Id	3S-Synnax-L01
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #5f8381c.

Description

The **DegenBox** contract implements a vault system where deposited tokens can be deployed to yield-generating strategies. The **harvest** function is responsible for executing strategy operations and updating the share-to-amount ratio based on realized profits or losses. When **harvest** is called, any profit increases **totals[token].elastic**, benefiting all existing shareholders proportionally. Conversely, any loss decreases **totals[token].elastic**, distributing the loss across all shareholders.

Since **harvest** is publicly callable and its effects on the share price are predictable by observing pending transactions in the mempool, malicious actors can exploit this timing:

Steps to Reproduce:

1. A strategy accumulates unrealized profit (or loss) that will be realized upon the next **harvest** call

2. An attacker monitors the mempool for pending **harvest** transactions

3. **Profit extraction scenario:**

- The attacker observes a pending **harvest** call that will realize profit

- The attacker front-runs with a **deposit** transaction using higher gas

- The attacker's deposit executes first, acquiring shares at the pre-profit price

- The **harvest** executes, increasing the value per share

- The attacker back-runs with a **withdraw** transaction

- The attacker captures a portion of the profit without having capital at risk during the strategy's operation

4. **Loss evasion scenario:**

- The attacker observes a pending **harvest** call that will realize a loss
- The attacker front-runs with a **withdraw** transaction using higher gas
- The attacker exits at the pre-loss share price
- The **harvest** executes, reducing the value per share
- The remaining depositors absorb the full loss
- The attacker can re-enter after the loss is socialized

This vulnerability extracts value from legitimate long-term depositors who bear the strategy risk without capturing proportional rewards.

Note:

- This issue also affects the forked version of Abracadabra's DegenBox, as the vulnerable code originates from the Abracadabra protocol.

- Sei uses Federated Byzantine Agreement (FBA) consensus, which heavily mitigates front-running by design. However, this issue remains relevant if the protocol is deployed on other EVM-compatible chains or if Sei's architecture changes in the future.

Recommendation

Implement a time-delayed deposit/withdraw mechanism that prevents immediate entry or exit around **harvest** events. Deposits and withdrawals should follow a two-phase with an enforced delay period.

3S-Synnax-L02

Global cooldown configuration in **SyUSDMultiCollateralManager::setConfig** allows altering waiting period for ongoing redemptions

Id	3S-Synnax-L02
Classification	Low
Impact	Low
Likelihood	Medium
Category	Bug
Status	Addressed in #012ddfe.

Description

In **SyUSDMultiCollateralManager**, the **setConfig** function updates the global **cooldown** parameter used by **SyUSDMultiCollateralManager::completeRedeem** to enforce the delay between **initRedeem** and redemption completion. The contract tracks for each user and collateral only a **lastAction[user][token]** timestamp and, at redemption time, checks **block.timestamp < lastAction[msg.sender][token] + cooldown**.

Because **cooldown** is read from storage at **completeRedeem** time rather than being snapshotted when **initRedeem** is called, an administrator can increase or decrease the cooldown after a user has initiated a redeem, effectively changing the waiting period for already pending redemptions. This behavior weakens user guarantees about the redemption delay and can lead to unexpected or perceived unfair changes in settlement timing.

Recommendation

Snapshot the effective cooldown for each redemption request at **SyUSDMultiCollateralManager::initRedeem** and enforce that fixed value in **SyUSDMultiCollateralManager::completeRedeem**, instead of reading the mutable global **cooldown**. A straightforward approach is to extend the **PendingRedeem** struct with a **readyAt** timestamp computed once at initialization and to gate **completeRedeem** on that field, making future configuration changes apply only to new redemptions.

3S-Synnax-L03

Unrestricted **to** parameter in **SyUSDStrategyVault::harvest** allows diversion of yield away from vault depositors

Id	3S-Synnax-L03
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #a37a63e.

Description

SyUSDStrategyVault::harvest is intended to realize strategy profits by calling **SyUSDMultiCollateralManager::realizeProfit**, minting **syUSD** against excess underlying and crediting that yield to vault depositors via increased vault assets. The function accepts a **to** parameter, and internally sets **recipient** to **to** (or **address(this)** when **to** is zero), then forwards this **recipient** to **SyUSDMultiCollateralManager::realizeProfit**. Because calls to **SyUSDStrategyVault::harvest** are controlled by the **KEEPER_ROLE**, a keeper can choose any arbitrary **to** address, causing the newly minted **syUSD** to be sent outside the vault and not reflected in idle vault balance nor **_underlyingBalance**, breaking the intended yield flow and allowing profits that economically belong to depositors to be redirected elsewhere.

Recommendation

Restrict the profit recipient in **SyUSDStrategyVault::harvest** so that the **syUSD** minted by **SyUSDMultiCollateralManager::realizeProfit** is always accounted as vault assets, for example by removing the **to** parameter and always using **address(this)** (or by strictly requiring **to == address(this)** if the signature must be preserved). If profits need to be redistributed elsewhere, that redistribution should be done in a separate, explicit step that transfers from the vault and remains consistent with **totalAssets** and **_underlyingBalance** accounting.

3S-Synnax-L04

Single-slot pending redemption design with shared cooldown leads to confusing UX and potential misdirected collateral withdrawals

Id	3S-Synnax-L04
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #012ddfe.

Description

SyUSDMultiCollateralManager::initRedeem is responsible for initializing a syUSD to collateral redemption for a given **msg.sender** and collateral **token**. It aggregates all pending redemptions for the same **(user, token)** pair into a single **PendingRedeem** struct stored in **pendingRedeems[msg.sender][token]**, and it also records the last interaction time in **lastAction[msg.sender][token]**.

When **initRedeem** is called a second time before the previous redemption is completed or cancelled, the function refreshes **lastAction[msg.sender][token]** to the current **block.timestamp** and accumulates **syUSD**, **collateral** and **fee** into the existing **PendingRedeem**, but it only sets **p.receiver = to** when **p.receiver** is zero, effectively ignoring any new **to** value once the first redemption has been initialized. As a result, users cannot maintain multiple independent pending redemptions for the same collateral with distinct parameters: all redemptions are merged into a single position with a single **receiver** and a single cooldown timestamp. This design is not obvious from the interface and can lead to unexpected behavior, such as a user believing they have updated the **to** address for a later redemption when in reality **SyUSDMultiCollateralManager::completeRedeem** will still send the aggregated collateral to the original **receiver**. It also forces users who want truly parallel redemptions with different destinations or timings to use multiple addresses, which is a poor UX pattern for a core redemption primitive.

Recommendation

Refactor the pending redemption tracking so that each **initRedeem** call creates an independent redemption record with its own parameters and cooldown, rather than aggregating all redemptions for a given **(user, token)** pair into a single **PendingRedeem**. A straightforward approach is to introduce a per-user, per-token redemption identifier (for example a monotonically increasing **redeemId** counter stored alongside the mapping) and change **pendingRedeems** to map **(user, token, redeemId)** to a **PendingRedeem** struct that includes its own timestamp; **SyUSDMultiCollateralManager::initRedeem** would create a new entry, return the **redeemId**, and **SyUSDMultiCollateralManager::completeRedeem** and **SyUSDMultiCollateralManager::cancelRedeem** would take a **redeemId** argument to operate on a specific pending redemption. This preserves the semantics of the **to** parameter for every call, avoids unintuitive cooldown resets across aggregated amounts, and lets users manage multiple redemptions in parallel without relying on multiple externally owned accounts.

3S-Synnax-L05

Stale clone tracking in setMasterContract allows outdated clone references and off-chain state confusion

Id	3S-Synnax-L05
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #d648f9a.

Description

The **MasterContractManager::setMasterContract** function is designed to update the association between a clone address and its master contract. It updates the **masterContractOf** mapping and adds the clone address to the **clonesOf** array of the newly assigned master contract. However, when a clone is reassigned from one master contract to another, the contract fails to remove the clone address from the **clonesOf** array of the old master contract. As a result, the clone appears in both the old and new master's clone arrays, while only the mapping reflects the current association. This can lead to off-chain data consumers or integrations being misled about the true state, increasing the risk of operational confusion or misreporting. The relevant logic is shown below:

```
function setMasterContract(address cloneAddress, address masterContract) public
onlyOwner {
   masterContractOf[cloneAddress] = masterContract;
   clonesOf[masterContract].push(cloneAddress);
   emit LogSetMasterContract(cloneAddress, masterContract);
}
```

Recommendation

Explicitly remove the clone address from the **clonesOf** array of the previous master contract before appending it to the new one.

3S-Synnax-L06

Resetting collateral minted tracking in **SyUSDMultiCollateralManager:addCollateral** allows per-collateral mint cap bypass and misaccounting

Id	3S-Synnax-L06
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #a3d2e8f.

Description

SyUSDMultiCollateralManager::addCollateral is intended to onboard a new collateral type by initializing its **Collateral** struct, including its **decimals**, **mintCap**, **psmMinted** counter, and associated **oracle**. The function writes **collaterals[token] = Collateral({ ... psmMinted: 0, ... })** without checking whether the collateral was already configured, which means that calling **addCollateral** again for an existing **token** overwrites all previous configuration and unconditionally resets **psmMinted** to zero.

Since **SyUSDMultiCollateralManager::mint** relies on **c.psmMinted** to enforce the per-collateral cap via **c.psmMinted + gross <= c.mintCap**, resetting **psmMinted** discards the historical minted amount for that collateral while the corresponding **syUSD** supply remains outstanding. As a result, an admin (or a compromised admin key, or a mistaken deployment script) can call **addCollateral** again for an already-used **token** and effectively “refresh” its headroom under **mintCap**, enabling additional **syUSD** mints backed by the same collateral beyond the intended per-asset limit, while **totalMinted** and actual **syUSD** circulation are unaffected. This breaks the intended per-collateral risk limits and makes on-chain accounting (**psmMinted** vs real outstanding exposure) inconsistent, which can materially affect risk monitoring and automated controls that use **psmMinted** and **mintCap** as safety bounds.

Recommendation

Guard **SyUSDMultiCollateralManager::addCollateral** so that it can only be used for truly new collaterals, and force configuration changes for existing collaterals through

SyUSDMultiCollateralManager::updateCollateral (which preserves **psmMinted**) or through a dedicated reconfiguration function that does not reset **psmMinted**.

3S-Synnax-L07

Misleading natspec for **SyUSDStableMinting::getLiquidity** leads to incorrect interpretation of BentoBox shares as raw token balances

Id	3S-Synnax-L07
Classification	Low
Impact	Low
Likelihood	High
Category	Bug
Status	Addressed in #d343851.

Description

SyUSDStableMinting::getLiquidity is intended to expose the current liquidity held by the PSM for the stable token and **syUSD**, and its NatSpec explicitly states that it returns “raw USDC (stable token) balance (token decimals)” and “raw syUSD balance (18 decimals)”. In practice, the function calls **bentoBox.balanceOf(IERC20(STABLE_TOKEN), address(this))** and **bentoBox.balanceOf(IERC20(address(syUSD)), address(this))**, which in the DegenBox/BentoBox model return internal share units, not the underlying token amounts. The correct way to obtain raw token balances from shares is to use **IBentoBoxV2::toAmount**, and treating shares as raw balances can lead integrators, risk dashboards, or other off-chain tooling to misjudge the actual liquidity available to honor redemptions or swaps, especially when the share-to-amount ratio has drifted over time due to strategy activity.

Recommendation

Align the function behavior with its NatSpec by converting BentoBox shares into underlying token amounts using **IBentoBoxV2::toAmount**, or alternatively adjust the documentation to clearly state that the function returns shares rather than raw token balances.

3S-Synnax-L08

Incorrect redeem fee upper bound in cauldronv5 allows configuring up to 10% fee instead of documented 1%

Id	3S-Synnax-L08
Classification	Low
Impact	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #636191f.

Description

CauldronV5::setRedeemFee is intended to let the master contract owner configure the collateral redeem fee, stored in **REDEEM_FEE** and applied in **CauldronV5::_removeCollateral** as **feeShare = share * REDEEM_FEE / REDEEM_FEE_PRECISION**. With **REDEEM_FEE_PRECISION = 1e5**, the effective fee rate is **REDEEM_FEE / 1e5**.

The function currently enforces **require(newRedeemFee <= 10000, "Fee too high"); // Max 1%**, but **10000 / 1e5 = 0.1 = 10%**, so the code actually allows configuring up to a 10% redeem fee while the inline comment and the stated precision ("basis points × 10") suggest the intended maximum is 1%. This discrepancy between the enforced bound and the documented behavior can mislead governance and integrators into believing the fee is capped at 1% when a much higher fee is technically allowed, exposing users to substantially larger-than-expected collateral haircuts when calling **CauldronV5::removeCollateral**.

Recommendation

Align the maximum allowed **REDEEM_FEE** with the documented 1% cap by tightening the check in **CauldronV5::setRedeemFee** so that the largest permitted value corresponds to **0.01 * REDEEM_FEE_PRECISION** (i.e. **1000** when **REDEEM_FEE_PRECISION = 1e5**), and update or verify all fee documentation and examples to match the actual precision. For example:

```
 function setRedeemFee(uint256 newRedeemFee) external onlyMasterContractOwner
{
```

```
-        require(newRedeemFee <= 10000, "Fee too high"); // Max 1%
+         require(newRedeemFee <= 1000, "Fee too high"); // Max 1%
         emit LogRedeemFeeChanged(REDEEM_FEE, newRedeemFee);
         REDEEM_FEE = newRedeemFee;
     }
```

3S-Synnax-N01

Unused **oracleDecimals** field in **Collateral** struct

Id	3S-Synnax-N01
Classification	None
Category	Suggestion
Status	Addressed in #fb5bcc9.

Description

In **SyUSDMultiCollateralManager**, the **Collateral** struct includes an **oracleDecimals** field which is written in **SyUSDMultiCollateralManager::addCollateral** using **oracle.decimals()**, but never read anywhere in the contract. The effective pricing logic is fully delegated to the oracle implementation (out of the audit scope): **SyUSDMultiCollateralManager::_oraclePrice** assumes that **Collateral.oracle** returns the asset's USD price already scaled to 18 decimals and does not consult **oracleDecimals** at all. As a result, **oracleDecimals** adds an extra storage slot per collateral without influencing behavior, and its presence incorrectly suggests that the manager handles varying oracle scales, which increases the risk that integrators or future maintainers rely on a behavior that does not exist.

Recommendation

Remove **oracleDecimals** from the **Collateral** struct and from **SyUSDMultiCollateralManager::addCollateral**, and clearly document that **SyUSDMultiCollateralManager::_oraclePrice** expects **Collateral.oracle** to return a price scaled to 18 decimals. If, in the future, support for oracles with different scales is needed, it should be implemented explicitly in **_oraclePrice** rather than via an unused field.

3S-Synnax-N02

Unused keeper role and Unauthorized error definitions lead to dead code and maintenance overhead

Id	3S-Synnax-N02
Classification	None
Category	Suggestion
Status	Addressed in #14f4366.

Description

SyUSDMultiCollateralManager is an upgradable multi-collateral PSM and strategy debt manager built on **AccessControlUpgradeable**, with operational permissions enforced via roles such as **DEFAULT_ADMIN_ROLE**, **STRATEGY_ROLE** and **PAUSER_ROLE**. The contract declares a **KEEPER_ROLE** constant and a custom **Unauthorized** error, but neither is referenced anywhere in the implementation: no function is guarded by **onlyRole(KEEPER_ROLE)** and no code path ever reverts with **Unauthorized()**. Because **AccessControlUpgradeable** already uses its own internal role checks and revert patterns based on **_msgSender()** and **_checkRole**, these unused declarations do not participate in any actual authorization logic and instead behave as dead code. This creates unnecessary noise for reviewers, and can mislead integrators into assuming there is an additional keeper permission layer in place.

Recommendation

Remove the unused **KEEPER_ROLE** constant and **Unauthorized** error from **SyUSDMultiCollateralManager** to reflect the actual access-control surface and reduce confusion.

3S-Synnax-N03

Missing treasury address validation when enabling fees leads to denial of service on fee-bearing operations

Id	3S-Synnax-N03
Classification	None
Category	Suggestion
Status	Addressed in #72cbb53.

Description

In **SyUSDMultiCollateralManager**, the **treasury** address is left unset during **SyUSDMultiCollateralManager::initialize** and can remain **address(0)** until **SyUSDMultiCollateralManager::setTreasury** is called by an admin. The contract allows **SyUSDMultiCollateralManager::setFees** to set **feeIn** and **feeOut** to non-zero values without enforcing that **treasury** has been configured.

When fees are positive, multiple flows attempt to send fee proceeds to **treasury**: **SyUSDMultiCollateralManager::mint**, **SyUSDMultiCollateralManager::returnPrincipalAndMint**, and **SyUSDMultiCollateralManager::realizeProfit** all invoke **_mintSyUSD(fee, treasury)**, which internally calls **bentoBox.withdraw(IERC20(address(syUSD)), address(this), to, amount, 0)** with **to = treasury**, while **SyUSDMultiCollateralManager::completeRedeem** calls **bentoBox.withdraw(IERC20(token), address(this), treasury, p.fee, 0)** for collateral fees.

In **DegenBox::withdraw** the **to** parameter is required to be non-zero (**require(to != address(0), "BentoBox: to not set");**), so if **treasury** is still **address(0)** when any of these fee paths execute, the entire operation reverts.

As a result, once non-zero fees are configured without a valid **treasury**, user minting, certain strategy interactions, and redeem completion will consistently fail, effectively disabling fee-bearing functionality until configuration is corrected.

Recommendation

Guard the configuration so that non-zero **feeIn** or **feeOut** cannot be set while **treasury** is still **address(0)**. Because **SyUSDMultiCollateralManager::setTreasury** already enforces a non-zero address via **ZeroAddress()**, the most robust fix is to extend **SyUSDMultiCollateralManager::setFees** with an additional check that **treasury != address(0)** whenever any fee is enabled. This prevents the protocol from entering a state where fee-bearing operations revert due to **DegenBox::withdraw** rejecting a zero recipient, while still allowing fees to be set to zero independently of **treasury**.

3S-Synnax-N04

Lack of pause enforcement in strategy functions allows asset movements and minting when protocol is paused

Id	3S-Synnax-N04
Classification	None
Category	Suggestion
Status	Addressed in #df20983.

Description

The **SyUSDMultiCollateralManager** contract uses a pause mechanism, intended to stop user operations such as minting and redeeming syUSD during maintenance or emergency states. While user-facing functions are correctly protected by the **whenNotPaused** modifier, core strategy functions responsible for collateral allocation and syUSD supply management (**allocateCollateral**, **returnPrincipalAndMint**, and **realizeProfit**) do not use this modifier. As a result, protocol actors with **STRATEGY_ROLE** can continue to move assets and mint tokens even when all other operations are suspended. This discrepancy allows protocol state changes that may cause confusion and introduces inconsistent behavior during paused periods.

Recommendation

Add the **whenNotPaused** modifier to all functions responsible for modifying protocol state, including privileged strategy functions

3S-Synnax-N05

Two-step bentobox withdrawals with rounded share accounting leads to unclear syusd-to-stable swap reverts

Id	3S-Synnax-N05
Classification	None
Category	Suggestion
Status	Addressed in #a961b2f.

Description

SyUSDStableMinting::swapSYUSDForStable is the PSM redemption function that burns **syUSD** and returns the corresponding amount of the configured stablecoin from **DegenBox**. Before burning **syUSD**, it checks that the PSM has enough shares in **DegenBox** by comparing **bentoBox.balanceOf(STABLE_TOKEN, address(this))** against **bentoBox.toShare(STABLE_TOKEN, totalStableNeeded, true)**, where **totalStableNeeded** includes both the user payout and the fee. However, **DegenBox::withdraw** internally converts each requested **amount** to shares via **totals[token].toBase(amount, true)**, rounding up per call, and in this function the contract performs two separate withdrawals: one for **fee** and one for **stableAmountReceived**. Because **toBase(a + b, true) <= toBase(a, true) + toBase(b, true)** due to per-call rounding, the single check using **bentoBox.toShare(STABLE_TOKEN, totalStableNeeded, true)** can pass while the sum of shares actually required by the two **withdraw** calls exceeds the PSM's share balance, causing the second withdrawal to revert despite the pre-check succeeding. This creates an edge-case where users experience failed **swapSYUSDForStable** executions under specific **DegenBox** rebase and rounding conditions, even though the pre-flight reserve check indicated sufficient liquidity.

Recommendation

Change **SyUSDStableMinting::swapSYUSDForStable** to withdraw the full **totalStableNeeded** from **DegenBox** in a single **withdraw** call (using **amount = totalStableNeeded** and **share = 0**), send the tokens to the PSM contract itself, and then split the resulting stable balance into the **fee** portion for **treasury** and the **stableAmountReceived** portion for **receiver** using plain ERC-20 transfers. This aligns the pre-check based on **bentoBox.toShare** with the actual **DegenBox::withdraw** share consumption (one rounded computation instead of two) and removes the rounding gap between the reserve check and the subsequent withdrawals.

3S-Synnax-N06

Misleading natspec on SyUSDStableMinting:_getPriceInUSD leads to incorrect oracle scaling assumptions

Id	3S-Synnax-N06
Classification	None
Category	Suggestion
Status	Addressed in #ebc25bf.

Description

The **SyUSDStableMinting::_getPriceInUSD** function is used by **SyUSDStableMinting::_previewTokenUSDAmount** to convert between the stable token amount and its USD value, which then drives how much **syUSD** is minted or how much stable is returned in **swapStableForSYUSD**, **swapSYUSDForStable**, and the associated **preview** functions. The NatSpec comment on **SyUSDStableMinting::_getPriceInUSD** states that it returns either **min(1$, oraclePrice)** or **max(1$, oraclePrice)** using a decimal scale of **(36 - asset_decimals)** (for example, **1e28** for an 8-decimal asset), but the implementation actually compares the oracle output against the constant **MANTISSA_ONE** and returns a flat 18-decimals value. Together with **SyUSDStableMinting::_previewTokenUSDAmount**, which explicitly documents its return as “USD value scaled by 1e18” and divides/multiplies by **STABLE_PRECISION**, this shows that the effective design is for **oracle.getPrice(STABLE_TOKEN)** to already use 18 decimals, making the NatSpec comment on **_getPriceInUSD** misleading.

If an integrator or future maintainer relies on the comment and wires an oracle that follows the described **(36 - asset_decimals)** scaling instead of 18 decimals, all swap and preview calculations would be off by several orders of magnitude, affecting pricing and fees despite the core logic being otherwise correct.

Recommendation

Align the documentation of **SyUSDStableMinting::_getPriceInUSD** with the actual 18-decimals behavior and clearly state that **oracle.getPrice(STABLE_TOKEN)** is expected to return a 1e18-scaled price, consistent with **SyUSDStableMinting::_previewTokenUSDAmount**.

3S-Synnax-N07

Missing burn-aware accounting in per-block mint limit allows denial of service of syUSD minting

Id	3S-Synnax-N07
Classification	None
Category	Suggestion
Status	Acknowledged

Description

The **DegenBox** contract uses the **mintedPerBlock** mapping and the **belowMaxMintPerBlock** modifier to enforce a global per-block cap on newly minted **syUSD** across all protocols that call **DegenBox::mintForCauldron**. In **DegenBox**, **mintedPerBlock[block.number]** is incremented in **DegenBox::mintForCauldron** and checked in **belowMaxMintPerBlock**, but it is never decreased when **syUSD** is burned later in the same block (for example in **CauldronV4::_repay**, **SyUSDStableMinting::swapSYUSDForStable**, or similar flows that call **IsyUSD::burn**). As a result, an attacker can, within a single block, repeatedly call **CauldronV4::borrow** (which internally calls **DegenBox::mintForCauldron**) up to the configured **maxMintPerBlock**, immediately repay in **CauldronV4::_repay** to end the block with no or very low net debt and supply change, yet leave **mintedPerBlock[block.number]** saturated for the remainder of that block. Any subsequent **syUSD** mint in the same block via **Cauldron** or the PSM (**SyUSDStableMinting::swapStableForSYUSD**, **SyUSDMultiCollateralManager::_mintSyUSD**) will then revert on the **belowMaxMintPerBlock** check, effectively allowing a well-funded adversary to deny minting capacity to other users or protocols one block at a time by paying only the configured opening and PSM fees but no interest, and repeating this pattern across blocks if desired.

Recommendation

Consider adjusting the per-block mint throttling so that it reflects net minted **syUSD** rather than gross mints, or at least so that one actor cannot monopolize the global **maxMintPerBlock** by mint-and-burn cycles within the same block. One straightforward approach is to expose a controlled function on **DegenBox** that allows whitelisted master contracts (Cauldrons and the PSM) to decrement **mintedPerBlock[block.number]** by the amount of **syUSD** they burn, floored at zero, and to invoke it from the relevant burn paths such as **CauldronV4::_repay** and the PSM redeem logic.

3S-Synnax-N08

Incorrect order of operations in collateral share calculation during liquidation

Id	3S-Synnax-N08
Classification	None
Category	Suggestion
Status	Addressed in #48bbf9e.

Description

The **CauldronV4::liquidate** function handles the liquidation of undercollateralized user positions. When a user's collateral value drops below the required threshold, liquidators can repay the user's debt and seize their collateral proportionally. The function calculates the amount of collateral shares to seize based on the borrow amount being repaid, the exchange rate, and the liquidation multiplier.

The issue occurs in the calculation of **collateralShare**. The current implementation converts **borrowAmount** to a share value first using **bentoBoxTotals.toBase()**, then multiplies by the liquidation multiplier and exchange rate. However, **bentoBoxTotals.toBase()** is designed to convert collateral amounts (elastic) to collateral shares (base), not borrow amounts. The correct approach is to first calculate the collateral amount by applying the liquidation multiplier and exchange rate to the borrow amount, then convert this collateral amount to shares.

While both formulas are mathematically equivalent in exact arithmetic, the order of operations matters in integer arithmetic due to rounding. The current implementation introduces unnecessary rounding errors by performing the share conversion on the wrong intermediate value.

Recommendation

Restructure the calculation to first compute the collateral amount, then convert it to shares:

```
-               uint256 collateralShare = bentoBoxTotals.toBase(borrowAmount,
false).mul(LIQUIDATION_MULTIPLIER).mul(
-                   _exchangeRate
-               ) / (LIQUIDATION_MULTIPLIER_PRECISION *
EXCHANGE_RATE_PRECISION);
+               uint256 collateralShare = bentoBoxTotals.toBase(
+                borrowAmount.mul(LIQUIDATION_MULTIPLIER).mul(_exchangeRate) /
(LIQUIDATION_MULTIPLIER_PRECISION * EXCHANGE_RATE_PRECISION),
+                false
+            );
```

This ensures that **bentoBoxTotals.toBase()** receives the actual collateral amount as its argument, minimizing rounding errors in the share conversion.