Pashov Audit Group

Regnum Aurum Security Review

August 12th 2025 - August 29th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

• **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
• **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
• **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

• **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
• **Medium** - only a conditionally incentivized attack vector, but still relatively likely
• **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Regnum Aurum

Regnum Aurum (RAAC) is a fractionalization platform that tokenizes real estate into NFTs (RAACNFT) and fractional index tokens (iRAAC), enabling on-chain lending, borrowing, and liquidity against property value. By combining Chainlink-powered appraisals, a hybrid RWA Vault, and veRAAC governance the protocol enables programmable debt positions against real estate assets with on-chain liquidation mechanisms.

5. Executive Summary

A time-boxed security review of the **RegnumAurumAcquisitionCorp/contracts** repository was done by Pashov Audit Group, during which **DemoreXTess, pontifex, Hals, Oblivionis, TheWeb3Mechanic, smbv1923** engaged to review **Regnum Aurum.** A total of **15** issues were uncovered.

Protocol Summary

Project Name	Regnum Aurum
Protocol Type	RWA tokenization
Timeline	August 12th 2025 - August 29th 2025

Review commit hash:

- d4f76fb86886d4d73093711d406755493e592b3a (RegnumAurumAcquisitionCorp/contracts)

Fixes review commit hash:

- 824afcb349666d90da377c6e3f4748c6606a6bf4 (RegnumAurumAcquisitionCorp/contracts)

Scope

`ERC20AssetAdapter.sol` `ERC721AssetAdapter.sol` `LendingPool.sol` `LendingPoolStorage.sol` `LiquidationProxy.sol` `VaultProxy.sol` `LiquidationStrategyProxy.sol` `LiquidationSwap.sol` `StabilityPool.sol` `StabilityPoolStorage.sol` `ComplianceRegistry.sol` `RAACHousePrices.sol` `WithCompliance.sol` `DebtToken.sol` `DEToken.sol` `RAACNFT.sol` `RToken.sol` `RWAIndexToken.sol` `ERC20VaultAdapter.sol` `ERC721VaultAdapter.sol` `RAACNFTVaultAdapter.sol` `RAACNFTVaultAdapterV2.sol` `RWAVault.sol` `PercentageMath.sol` `TimeWeightedAverage.sol` `WadRayMath.sol` `ReserveLibrary.sol` `StringUtils.sol`

6. Findings

Findings count

Severity	Amount
High	2
Medium	3
Low	10
Total findings	**15**

Summary of findings

ID	Title	Severity	Status
[H-01]	Parameter change skews getters	High	**Resolved**
[H-02]	Borrowers can avoid paying interest for lenders	High	**Resolved**
[M-01]	Yield double counting from lowered share price baseline on withdrawals	Medium	**Resolved**
[M-02]	Same block deposit check blocks `StabilityPool` withdrawals during liquidation	Medium	**Resolved**
[M-03]	Protocol misses fees from borrowing positions	Medium	**Resolved**
[L-01]	Hardcoded pool coin indices risk wrong swaps/ reverts	Low	**Resolved**
[L-02]	Missing `_disableInitializers()` in Implementation Contract	Low	**Resolved**
[L-03]	`stringToUint("")` Returns `0` Instead of Reverting	Low	**Resolved**
[L-04]	`unregisterAdapter()` can be DoS'd by dust deposits	Low	**Resolved**
[L-05]	Read-only reentrancy in `redeemNFT` / `withdraw` flow	Low	**Resolved**
[L-06]	Borrow dust bypasses minimum borrow constraint	Low	**Resolved**
[L-07]	Hard-coding Curve `price_oracle` index across chains yields wrong prices	Low	**Resolved**

ID	Title	Severity	Status
[L-08]	Any user can redeem from deprecated ERC-721 adapters	Low	Resolved
[L-09]	Surplus `rToken` stuck in `StabilityPool` when assets exceed liabilities	Low	Resolved
[L-10]	Missing percentage based borrowing cap can DoS all the liquidation actions	Low	Resolved

High findings

[H-01] Parameter change skews getters

Severity

Impact: High

Likelihood: Medium

Description

Interest accrual is always calculated using the last calculated variables for updating reserve state. For instance, if interest was 5% lastly and after 1 hour we call `deposit` function, it will use this 5% interest for the last 1 hour. However, `getNormalizedIncome` and `getNormalizedDebt` functions calculate these rates again return value according to that.

This will cause incorrect value to be returned in these functions. It's very critical because `getNormalizedIncome` and `getNormalizedDebt` are used in many important points in the codebase.

```
// Reserve update

        reserve.liquidityIndex = calculateLiquidityIndex(
            rateData.currentLiquidityRate,
            timeDelta,
            reserve.liquidityIndex
        );

        // Update usage index (debt index) using compounded interest
        reserve.usageIndex = calculateUsageIndex(
            rateData.currentUsageRate,
            timeDelta,
            reserve.usageIndex
        );
```

```
// Normalized income

    function getLiquidityIndex(ReserveData storage reserve, ReserveRateData storage rateData)
internal view returns (uint256) {
        uint256 timeDelta = block.timestamp - uint256(reserve.lastUpdateTimestamp);
        if(timeDelta < 1) {
            return reserve.liquidityIndex;
        }

        return calculateLiquidityIndex(
            calculateLiquidityRate(rateData.currentUtilizationRate, rateData.currentUsageRate,
rateData.protocolFeeRate, reserve.totalUsage),
            timeDelta,
            reserve.liquidityIndex
        );
    }
```

As you can see, it calculates the liquidity rate again in here. Only `protocolFeeRate` can change and can have different actual value here. It means if protocol changes protocol fee, it will return an incorrect value for normalized income.

> Note: Same situation happens in debt for different parameter changes.

Recommendations

Do not calculate liquidity rate or usage rate in `getNormalizedIncome` and `getNormalizedDebt` instead, use the cached version.

[H-02] Borrowers can avoid paying interest for lenders

Severity

Impact: Medium

Likelihood: High

Description

When we check the borrowers total debt value, we use `_positionScaledDebt` internal function. It uses following formula:

`rawDebtBalance * usageIndex / positionIndex`

However, it doesn't increase rawDebtBalance correctly in `borrow` function while it updates positionIndex to a new value.

```
(, uint256 underlyingAmount, , ) = IDebtToken(reserve.reserveDebtTokenAddress).mint(msg.sender,
msg.sender, amount, reserve.usageIndex, abi.encode(adapter, data));

        // We need to update the position index of the user
        position.positionIndex = reserve.usageIndex;

        // Transfer borrowed amount to user
        IRToken(reserve.reserveRTokenAddress).transferAsset(msg.sender, amount);

@>      position.rawDebtBalance += underlyingAmount;
```

In here, underlying amount is equal to the borrowing amount. In `debtToken.mint()` call, it will mint borrowers interest to borrower, however, this increase is returned as 3rd parameter in mint function, and we don't use it while updating `rawDebtBalance`. Moreover, we update position index to the last updated value, therefore the following expression will be equal to 1:

`usageIndex / positionIndex`

In order to return correct amount of debt balance, `rawDebtBalance` should account for interest too.

In this situation, borrowers can borrow X amount of crvUSD and then they can borrow 20 crvUSD and repay back 20 crvUSD immediately. With this method, they can bypass 100% of the interest.

> Note: Debt token will mint the interest correctly, therefore there will be a difference between balance of debt token and `_positionScaledDebt`.

Recommendations

Add interest increase to `rawDebtBalance` too.

Medium findings

[M-01] Yield double counting from lowered share price baseline on withdrawals

Severity

Impact: Medium

Likelihood: Medium

Description

In the `VaultStrategy` contract, yield is calculated in `_harvestYield()` as follows, where the `lastSharePrice` is only updated if the `currentSharePrice` is greater than the last recorded `lastSharePrice` :

```
function _harvestYield() internal {
        uint256 currentSharePrice = _getPricePerShare();

        if (vaultRewards.lastSharePrice == 0 || currentSharePrice <
vaultRewards.lastSharePrice) {
            // Initialize on first deposit
            if (vaultRewards.lastSharePrice == 0) {
                vaultRewards.lastSharePrice = currentSharePrice;
            }
            return;
        }

        uint256 priceDifference = currentSharePrice - vaultRewards.lastSharePrice;
>>      uint256 totalYield = (vaultRewards.totalShares * priceDifference) / 1e18;

        vaultRewards.pendingRewards += totalYield;
>>      vaultRewards.lastSharePrice = currentSharePrice; // @note : will update with higher
price

        //...
    }
```

However, the `withdrawFromVault()` function updates `lastSharePrice` downward when the `currentSharePrice` falls below the previous recorded `lastSharePrice` . This introduces an issue because `lastSharePrice` should act as a **high-water mark**. Lowering it causes the vault to incorrectly recognize **recovery of losses** as **new yield**, effectively withdrawing principal under the label of yield:

```
function withdrawFromVault(uint256 amount) public onlyProxy {
        uint256 maxWithdrawable = _maxWithdraw(address(this));
        if (maxWithdrawable == 0) return;

        uint256 currentSharePrice = _getPricePerShare();
        uint256 totalYield = 0;
        uint256 priceDifference = 0;
        if (vaultRewards.lastSharePrice > 0 && currentSharePrice > vaultRewards.lastSharePrice)
{
            priceDifference = currentSharePrice - vaultRewards.lastSharePrice;
            totalYield = (vaultRewards.totalShares * priceDifference) / 1e18;
        }

        vaultRewards.pendingRewards += totalYield;
>>      vaultRewards.lastSharePrice = currentSharePrice; // @audit : will update with lower
price

       //...
    }
```

Example:

1. Vault deposits **1000 USDC** at share price **1.0** → receives **1000 shares**.
   `lastSharePrice = 1.0`
2. Vault grows to share price **1.2** → accrued yield = **200 USDC**.
   `_harvestYield()` correctly recognizes this yield.
3. Market dips → share price falls to **0.8**.
   `withdrawFromVault()` resets `lastSharePrice = 0.8`.
4. Market recovers → share price returns to **1.2**.
   `_harvestYield()` calculates yield as `(1.2 - 0.8) * 1000 = 400 USDC`.
   But the true yield is zero, not 400, as the previously accrued yield of 200 has already been harvested previously when the share price was at 1.2, where this extra 400 will be withdrawn/taken from the vault's principal (`totalDeposits`).

This double-counting occurs every time the price recovers after a drop, leading to withdrawal of principal asset as yield and unfair allocation of the excess yield to the fee collector. Additionally, it prevents the vault from being removed via `withdrawAll()` since `totalDeposits` no longer matches the principal represented by `totalShares`.

Recommendations

Modify `withdrawFromVault()` function to only update the `lastSharePrice` when `currentPrice > lastSharePrice` to ensure that `_harvestYield()` only accounts for **new yield** and prevents withdrawing from principal when harvesting yield.

[M-02] Same block deposit check blocks `StabilityPool` withdrawals during liquidation

Severity

Impact: Medium

Likelihood: Medium

Description

When the `StabilityPool` liquidates user positions, it may lack enough `crvUSDToken` to cover the liquidation. In such cases, it withdraws the shortfall from the `LendingPool`, which burns the corresponding `rToken` and transfers the underlying `crvUSD` to the `StabilityPool`:

```
// LiquidationStrategyProxy contract:
function liquidateBorrower(address poolAdapter, address vaultAdapter, address user, bytes
calldata data, uint256 minSharesOut) external onlyProxy {
        //...

        uint256 scaledPositionDebt = lendingPool.getPositionScaledDebt(poolAdapter, user, data);
        uint256 initialCRVUSDBalance = crvUSDToken.balanceOf(address(this));
        uint256 availableRTokens = rToken.balanceOf(address(this));

        // We need to get the amount of rToken that is needed to cover the debt, or 0 if the
debt is covered
        uint256 rTokenAmountRequired = initialCRVUSDBalance >= scaledPositionDebt ? 0 :
scaledPositionDebt - initialCRVUSDBalance;
        if (availableRTokens < rTokenAmountRequired) revert InsufficientBalance();

        // We unwind the position
        if (rTokenAmountRequired > 0) {
>>           lendingPool.withdraw(rTokenAmountRequired);
        }
        //...
}
```

At the end of liquidation, any leftover `crvUSDToken` (e.g. swapped from `iRAAC`) will be deposited back into the `LendingPool`. This deposit sets `depositBlock[StabilityPool] = block.number`:

```
// LiquidationStrategyProxy contract
    function liquidateBorrower(address poolAdapter, address vaultAdapter, address user, bytes
calldata data, uint256 minSharesOut) external onlyProxy {
    //...

        _handleLiquidation(poolAdapter, vaultAdapter, data, minSharesOut);

        // Deposit crvUSD back to get rTokens (including the excess)
        // Get the final crvUSD balance after the exchange
        uint256 finalCRVUSDBalance = crvUSDToken.balanceOf(address(this));
        if (finalCRVUSDBalance > 0) {
            // Approve lending pool to take crvUSD for deposit
            bool approveCRVUSDDeposit = crvUSDToken.approve(address(lendingPool),
finalCRVUSDBalance);
            if (!approveCRVUSDDeposit) revert ApprovalFailed();
>>           lendingPool.deposit(finalCRVUSDBalance);
        }
        emit BorrowerLiquidated(user, IAssetAdapter(poolAdapter).getAssetToken(), data,
scaledPositionDebt);
    }
```

```
// LendingPool contract:
 function deposit(uint256 amount) external nonReentrant whenNotPaused onlyValidAmount(amount)
notBlacklisted(msg.sender) {
        //...

>>       depositBlock[msg.sender] = block.number;

        //...
    }
```

If another liquidation occurs within the same block and again requires a withdrawal from the `LendingPool`, the `LendingPool.withdraw()` call will revert due to the following check in the `RToken._update()` when trying to transfer `rToken` from the `StabilityPool` to the `LendingPool` :

```
 function _update(address from, address to, uint256 amount) internal override {
        //...
        if (ILendingPool(_lendingPool).isUserDepositInSameBlock(from)) revert
CannotDepositAndTransferInSameBlock();
        //...
    }
```

This prevents liquidation from completing other liquidations in the same block, leading to failed liquidations and increased costs for liquidators due to accrued interest.

Recommendations

Exclude the `StabilityPool` address from the block-based same-block deposit check:

```
// LendingPool contract:
 function deposit(uint256 amount) external nonReentrant whenNotPaused onlyValidAmount(amount)
notBlacklisted(msg.sender) {
        //...

-       depositBlock[msg.sender] = block.number;
+       if (msg.sender != stabilityPool) depositBlock[msg.sender] = block.number;

        //...
    }
```

[M-03] Protocol misses fees from borrowing positions

Severity

Impact: Medium

Likelihood: Medium

Description

Protocol always subtracts protocol fee rate while calculation of interest rate for liquidity providers, but it never mints or allocate these funds. It means borrowers pay for these fees too, but nobody receives these funds, including lenders.

It means even if all the borrowers pay for their positions and then lenders withdraw 100% of the funds, there will be funds in the system.

```
    function calculateLiquidityRate(uint256 utilizationRate, uint256 usageRate, uint256
protocolFeeRate, uint256 totalDebt) internal pure returns (uint256) {
        if (totalDebt < 1) {
            return 0;
        }

        uint256 grossLiquidityRate = utilizationRate.rayMul(usageRate);
        uint256 protocolFeeAmount = grossLiquidityRate.rayMul(protocolFeeRate);
@>      uint256 netLiquidityRate = grossLiquidityRate - protocolFeeAmount;

        return netLiquidityRate;
    }
```

Recommendations

Allocate/mint these funds to the fee recipient in order to gather fees from borrowing positions.

Low findings

[L-01] Hardcoded pool coin indices risk wrong swaps/reverts

`_swap` calls the Curve pool with hardcoded indices `(1, 0)`:

```
// Exchange index token (1) for crvUSD (0) since we swapped the order in the pool
liquidityPool.exchange(1, 0, amount, minDy, address(this)); // hardcoded indices
```

This assumes a single pool and a fixed coin order. If the pool is replaced or its coin order differs, approvals and `exchange()` indices won't match the actual tokens, causing **reverts or unintended swaps**. Additionally, `minDy` is derived from `pricePerShare()` rather than the pool's own quote, increasing slippage/mispricing risk.

[L-02] Missing `_disableInitializers()` in Implementation Contract

The `StabilityPool` implementation contract inherits from `Initializable`, but its constructor does not call `_disableInitializers()`. This means the logic contract can still be initialized directly with arbitrary parameters if it's ever deployed without a proxy, which deviates from the recommended OpenZeppelin upgradeable contract pattern. While this does not compromise the proxy's storage/ownership, it can cause confusion or misuse if the implementation contract is interacted with directly.

Call `_disableInitializers();` in the implementation contract's constructor to explicitly lock the logic contract and prevent direct initialization with custom parameters. This ensures only the proxy can be properly initialized.

[L-03] `stringToUint("")` Returns `0` Instead of Reverting

The current implementation of `StringUtils.stringToUint` returns `0` when given an empty string (`""`). This can mask malformed or missing inputs — e.g., in `_processRequest`, `args[2]` could be an empty string, causing `houseId` to become `0`. If `0` is not a valid `houseId`, this may silently pass incorrect data and lead to logic errors.

```
function stringToUint(string memory s) internal pure returns (uint256) {
    bytes memory b = bytes(s);
    uint256 result;
    for (uint256 i; i < b.length; i++) {
        uint8 digit = uint8(b[i]) - 48;
        if (digit > 9) revert NonNumericCharacter();
        result = result * 10 + digit;
    }
    return result; // "" -> 0
}
```

Usage in `LendingPool.sol`:

```
uint256 houseId = args[2].stringToUint(); // "" becomes 0
```

Recommendation

Make `stringToUint` revert on empty strings to avoid silent coercion.

[L-04] `unregisterAdapter()` can be DoS'd by dust deposits

The `RWAVault.unregisterAdapter()` function is intended for the contract owner to remove an existing adapter from the supported list. Before removal, it checks that `IVaultAssetAdapter(adapter).totalValue() == 0`, ensuring no assets remain in the adapter. However, this logic introduces a potential DoS vector: an attacker (or even a benign user) can deposit a dust amount into the adapter, preventing the owner from ever unregistering it. Since even a minimal non-zero balance blocks the process, the adapter could remain stuck in the system indefinitely.

```
    function unregisterAdapter(address adapter) external onlyOwner {
        require(supportedAdapters[adapter], "RWAVAult: not supported");
>>      if (IVaultAssetAdapter(adapter).totalValue() > 0) revert("RWAVault: adapter has value");
        //...
    }
```

Recommendation:
Introduce a forced removal mechanism to handle dust deposits.

[L-05] Read-only reentrancy in `redeemNFT` / `withdraw` flow

The `redeemNFT` process relies on `RAACNFTVaultAdapterV2.withdraw`, which internally calls `super.withdraw`. Since `super.withdraw` performs an ERC721 `safeTransfer`, this triggers an **external call** to the receiver's `onERC721Received` before the adapter updates its internal accounting (`nftsValue`).

```
value = super.withdraw(data, to); // external call → onERC721Received
(uint256 usdValue,) = IRAACHousePrices(address(priceOracle)).getRawPrice(tokenId);
if (nftsValue < usdValue) {
    nftsValue = 0;
} else {
    nftsValue -= usdValue;
}
```

During the `onERC721Received` callback, a malicious receiver can invoke view functions such as `totalAssets()`, `convertToShares()`, or `pricePerShare()`. Because `nftsValue` has not yet been reduced, these functions will return **inflated asset values**, enabling *read-only reentrancy*. An attacker could use this within the same transaction to manipulate valuations in other protocols that trust these views (e.g., over-minting, unfair swaps, or borrowing against inflated collateral).

Recommendations

1. **Apply the Checks-Effects-Interactions (CEI) pattern**: Update `nftsValue` **before** calling `super.withdraw` to ensure accounting is consistent even if reentrancy occurs.
2. **Protect read functions**: Add a lightweight `nonReentrantView` modifier to sensitive view functions (`totalAssets` , `pricePerShare` , `convertToShares`) to block calls during external transfers/mints.

[L-06] Borrow dust bypasses minimum borrow constraint

While the protocol enforces a minimum borrow amount (`MIN_BORROW_AMOUNT`) in `_validateBorrow`, borrowers can bypass this constraint by partially repaying their debt and leaving behind a **dust-sized leftover** smaller than the minimum borrow threshold.

Specifically:

- `repay(...)` (called by the borrower) has no minimum repayment requirement — any `amount > 0` is accepted.
- `repayOnBehalf(...)` enforces a 1% minimum repayment, but this applies only when a third party is repaying.
- As a result, a borrower can take out a valid loan above `MIN_BORROW_AMOUNT` , then immediately repay most of it and leave a small residual debt (`debt < MIN_BORROW_AMOUNT`).

Although it does not result in direct financial loss, it undermines the intent of the minimum borrow rule and may negatively affect protocol operations.

Recommendations

Enforce a rule that **no residual debt smaller than** `MIN_BORROW_AMOUNT` may remain after repayment.

[L-07] Hard-coding Curve `price_oracle` index across chains yields wrong prices

The oracle fallback currently assumes a fixed Curve `price_oracle` index for ETH/crvUSD (e.g., calling `price_oracle(0)` and expecting it to be "WETH in crvUSD"). Across chains and pools, **token ordering differs**, so the same index can reference a different asset, producing **wrong or inverted prices**. This propagates into `crvUSDPriceInUSD` , affecting LTV checks, liquidations, and circuit-breaker logic.

Concrete differences you'll encounter:

- **TriCrypto/crypto v2** pools: `price_oracle(k)` returns the EMA price of `coin[k+1]` **in units of** `coin[0]`. If `coin[0] = crvUSD`, then:
- If `coin[1] = WETH` → use `k = 0`.
- If `coin[2] = WETH` → use `k = 1`.

Examples of cross-chain differences:

- Ethereum mainnet TriCRV ordered as `crvUSD, WETH, CRV` → **K = 0** (https://www.curve.finance/dex/ethereum/pools/factory-tricrypto-4/deposit).
- Optimism TriCrypto-crvUSD ordered as `crvUSD, WBTC, WETH` → **K = 1** (https://www.curve.finance/dex/optimism/pools/factory-tricrypto-0/deposit).
- Base `crvUSD/tBTC/WETH` ordered as `crvUSD, tBTC, WETH` → **K = 1** (https://www.curve.finance/dex/base/pools/factory-tricrypto-1/deposit).

Because "ETH" means chain-specific **WETH** with a different address per chain, relying on a fixed index is unsafe.

Recommendations

Adopt a deploy-time, immutable index K set by the owner. Add `uint8 public immutable K;` and set it **once at deployment**.

```
    uint8 public immutable K;  // 0 or 1 for TriCrypto (coin[K+1] vs coin[0])

    constructor(address initialOwner, uint8 _k) Ownable(initialOwner) {
        require(_k <= 1, "invalid K"); // TriCrypto supports k in {0,1}
        K = _k;
        fallbackPriceInUSD = 1e18;
        lastFallbackUpdateTimestamp = block.timestamp;
    }

    function _crvUSDPriceInUSDFromCurve() internal view returns (uint256 crvUSDPriceInUSD,
uint256 lastUpdateTimestamp, bool success) {
<...>
        uint256 ETHPriceInCrvUSD = curveOracle.price_oracle(K);
```

[L-08] Any user can redeem from deprecated ERC-721 adapters

The vault maintains a whitelist for redeemable ERC-721 adapters through `supportedRedeemableERC721Adapters` and the `redeemableERC721Adapters` array, with registration controlled by `registerRedeemableERC721Adapter` and removal by `unregisterRedeemableERC721Adapter`. Unlike deposits, which are explicitly gated by `onlySupportedDepositableAdapter`, the redemption path has no equivalent gate. The user-facing `redeemNFT(address _adapter, uint256 _tokenId, uint256 maxSharesBurn)` function does not check `supportedRedeemableERC721Adapters[_adapter]` and relies solely on `getNextRandomNFT()`, which selects an adapter from the `redeemableERC721Adapters` array without verifying that it is still supported at call time.

Separately, `unregisterAdapter(address adapter)` can mark an adapter as unsupported for the vault (supportedAdapters[adapter] = false) without removing it from the redeemable array. This creates a desynchronization risk where an adapter can be deprecated at the vault level yet remain selectable for redemption because the redemption flow neither consults the `supportedRedeemableERC721Adapters` mapping nor enforces that the adapter is currently supported. In such a state, calls to `redeemNFT` can continue to withdraw through an adapter that governance intended to disable, or they may revert unpredictably if the deprecated adapter changes behavior, creating a denial-of-service hazard.

Recommendations

In `redeemNFT`, validate that the selected adapter is currently allowed for redemption by checking `supportedRedeemableERC721Adapters[adapter]` before proceeding with share burning and withdrawal.

[L-09] Surplus `rToken` stuck in `StabilityPool` when assets exceed liabilities

During liquidation, the strategy pays the borrower's debt in `crvUSD` and then **re-deposits any leftover** `crvUSD` **back into the LendingPool**, which mints additional `rToken` **to the** `StabilityPool`:

```
// LiquidationStrategyProxy (simplified)
uint256 finalCRVUSDBalance = crvUSDToken.balanceOf(address(this));
if (finalCRVUSDBalance > 0) {
    lendingPool.deposit(finalCRVUSDBalance); // mints rToken to StabilityPool
}
```

User withdrawals from the StabilityPool always send **exactly the user's scaled amount** and do **not** pro-rata any surplus:

```
// StabilityPool.withdraw(...)
deToken.burn(msg.sender, scaledAmount);
rToken.safeTransfer(msg.sender, scaledAmount); // exactly "scaledAmount", no share of surplus
```

`DEToken` is **index-denominated**, not share-of-pool. Its balances (and total supply) are derived from raw deposits × index, **independent of** `rToken.balanceOf(StabilityPool)`. Therefore, if liquidation leaves extra `crvUSD` that is re-deposited, the pool's **assets** can exceed the **liabilities** to DEToken holders:

- **Assets (scaled)**: `A = rToken.balanceOf(StabilityPool)`
- **Liabilities (scaled)**: `L = deToken.getRawTotalDeposits() * getNormalizedIncome()`

If `A > L`, the difference `A - L` is a **surplus** that:

- is **not distributable** to DEToken holders (no PPS/vault math, no surplus index);
- cannot be withdrawn by users (no matching DEToken to burn);
- has **no admin sweep** in the current code (so it effectively sits on the contract indefinitely),
- and will only be consumed opportunistically by future liquidations.

This creates **capital lock** and mis-accounting risk (assets parked without an explicit policy), and can confuse accounting/metrics (TVL vs. claimable).

Recommendations

A. Cap the post-liquidation re-deposit to avoid growing surplus. Before depositing the leftover `crvUSD`, compute how much (in scaled terms) the pool can accept without exceeding a target (liabilities or liabilities + small buffer).

B. Add an explicit `sweepSurplus()` admin function (with events & policy). Allow moving only the *excess* safely.

[L-10] Missing percentage based borrowing cap can DoS all the liquidation actions

Currently, there is no percentage based borrowing cap in the system. It's not a regular cap type, and it's not required for many protocols, but it's required for RAAC because of it's own unique liquidation design.

In RAAC liquidation design, debt is covered by lenders, which is a unique way. Stability pool holds RToken, and in liquidations, these RTokens are converted to crvUSD by calling `withdraw` function in Lending Pool:

```
        uint256 rTokenAmountRequired = initialCRVUSDBalance >= scaledPositionDebt ? 0 :
scaledPositionDebt - initialCRVUSDBalance;
        if (availableRTokens < rTokenAmountRequired) revert InsufficientBalance();

        // We unwind the position
        if (rTokenAmountRequired > 0) {
@>            lendingPool.withdraw(rTokenAmountRequired);
        }
```

This design can't be alive if we reach 100% borrowing utilization rate, because while calling `withdraw` it will revert because of insufficient liquidity. 100% of the funds borrowed by borrowers, and there is no way to liquidate any position in this case.

Borrowers can avoid liquidation by constantly borrowing 100% of the funds in the Lending Pool, and they don't have to pay back that because liquidation is almost impossible.

> Actually, protocol can liquidate positions by transferring crvUSD tokens directly to Stability Pool, and then they can liquidate it. However, these crvUSDs will be lost on protocol side, and it will cause loss of funds anyway.

Due to these reasons, percentage based borrowing cap is crucial for the current RAAC system design.

Recommendations

Consider applying percentage based borrowing cap, such as 80%.

> Note: This cap has a trade-off. If we leave too much gap in the pool, interest rates will be lower in the Lending Pool. However, if we choose a really tiny value for this gap, big positions cannot be liquidated.