

Sprinter: Stash Contracts

Security Review

Cantina Managed review by:
Slowfi, Lead Security Researcher
Kankodu, Security Researcher

August 3, 2026

Contents

1 Introduction **2**
1.1 About Cantina . . . 2
1.2 Disclaimer . . . 2
1.3 Risk assessment . . . 2
1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**
2.1 Scope . . . 3

3 Findings **4**
3.1 High Risk . . . 4
3.1.1 Gnosis Omnibridge rebalances bypass destination pool accounting . . . 4
3.2 Medium Risk . . . 4
3.2.1 `StashDex` direct repayments can ignore accrued Aave interest . . . 4
3.2.2 Gnosis Omnibridge repayments can deliver the wrong USDC variant . . . 5
3.2.3 Aave `withdrawProfit` shortfall borrow skips token LTV checks . . . 5
3.2.4 Aave `repayDirect` reverts when direct debt exists but Aave debt is zero . . . 6
3.2.5 Gnosis AMB `executeSignatures` can skip rebalance accounting . . . 7
3.2.6 Attacker can force the operator requirement on a user's withdrawals via allowances . . . 8
3.2.7 `claimableRedeemRequest` over-reports causing `fulfilRedeem` to revert . . . 9
3.2.8 Gnosis Repayer fix is incomplete in both directions . . . 11
3.3 Low Risk . . . 12
3.3.1 `LiquidityHub` zero-asset adjustment advertises impossible withdrawals . . . 12
3.3.2 ERC-7540 standard violations in `LiquidityHub` . . . 13
3.3.3 Stablecoin pool direct borrows of non assets cannot be repaid . . . 13
3.3.4 `ERC4626Adapter` zero share deposits can consume withdrawable profit . . . 14
3.3.5 `StashDex` oracle sync omits input only route assets . . . 15
3.3.6 Conditional virtual shares re-opens the ERC4626 share-price reset attack . . . 16
3.3.7 `StashDex` upgrade can abandon legacy direct debt . . . 17
3.4 Informational . . . 19
3.4.1 Aave repay can desync direct debt . . . 19
3.4.2 Ordinary Aave repay does not clear direct debt . . . 19
3.4.3 Superchain `StandardBridge` USDC routes can temporarily retain repayments . . . 20
3.4.4 Code improvement suggestions . . . 21
3.4.5 Regular Processor deployments leave process unusable with a zero oracle . . . 21
3.4.6 Safe proposals are treated as executed by maintenance scripts . . . 22
3.4.7 Aave token LTV cannot be set to zero while a default LTV is configured . . . 22
3.4.8 Aave-backed Hub fix underreports withdrawable liquidity . . . 23

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Sprinter Stash is a credit-based liquity protocol. LPs deposit stablecoins and earn from two yield sources, while solvers can access short-term, uncollateralized liquidity across different chains to execute crosschain transactions seamlessly.

From Jul 6th to Jul 13th the Cantina team conducted a review of sprinter-stash-contracts on commit hash 917bb28a. The team identified a total of **24** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	8	8	0
Low Risk	7	7	0
Gas Optimizations	0	0	0
Informational	8	4	4
Total	**24**	**20**	**4**

2.1 Scope

The security review had the following components in scope for sprinter-stash-contracts on commit hash 917bb28a:

```
contracts
├── LiquidityHub.sol
├── LiquidityPool.sol
├── LiquidityPoolAave.sol
├── LiquidityPoolAaveLongTerm.sol
├── LiquidityPoolStablecoin.sol
├── PaxosOracle.sol
├── Processor.sol
├── Rebalancer.sol
├── Repayer.sol
├── StashDex.sol
├── StashDexProcessor.sol
├── SubProcessor.sol
└── utils
    ├── AcrossAdapter.sol
    ├── AdapterHelper.sol
    ├── ArbitrumGatewayAdapter.sol
    ├── CCTPV2Adapter.sol
    ├── GnosisOmnibridgeAdapter.sol
    ├── HelperLib.sol
    ├── LayerZeroHelper.sol
    ├── StargateAdapter.sol
    ├── SuperchainStandardBridgeAdapter.sol
    └── USDT0Adapter.sol
```

3 Findings

3.1 High Risk

3.1.1 Gnosis Omnibridge rebalances bypass destination pool accounting

Severity: High Risk.

Context: Rebalancer.sol#L197-L203.

Description: For CCTP and CCTP v2 rebalances, the destination side `processRebalance` path measures the received amount and calls `ILiquidityPoolBase(destinationPool).deposit(depositAmount)`, which updates the destination pool accounting. The Gnosis Omnibridge initiation path is different: `initiateRebalance` withdraws from the source pool and calls `initiateTransferGnosisOmnibridge`, which relays tokens directly to `destinationPool`.

For Ethereum to Gnosis Omnibridge transfers, there is no corresponding destination-side `processRebalance` call that can execute `deposit()` on the Gnosis pool. The bridge delivers tokens directly to the destination address, but `totalDeposited` /pool accounting is never increased. The result is raw token balance at the pool without matching deposit accounting. Depending on pool type, that balance can be ignored for LP withdrawals, treated as surplus/profit, or left unavailable to the intended accounting path.

A rebalancer can move liquidity out of the source pool while the destination pool does not account it as deposited liquidity. This can strand or misclassify rebalanced funds and can cause solvency/accounting mismatches between chains.

Recommendation: Consider avoiding sending Omnibridge rebalances directly to a liquidity pool unless the destination bridge path can atomically call `deposit()`. Bridge to a destination-side `Rebalancer` or receiver contract, validate the received token, and have that contract call `ILiquidityPoolBase.deposit(depositAmount)`. Alternatively, disable Ethereum to Gnosis Omnibridge rebalancing routes until a destination accounting step exists.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 `StashDex` direct repayments can ignore accrued Aave interest

Severity: Medium Risk.

Context: StashDex.sol#L187-L204.

Description: `StashDex` tracks the nominal `amountOut` borrowed from its configured pool as `totalBorrowed`. Its repayment path caps repayment to that nominal value, and the Aave-backed pool's `repayDirect()` path caps direct repayment to `directDebt[borrowToken]`.

For Aave-backed pools, variable debt accrues interest above the nominal direct-borrow principal. After `StashDex` repays only the tracked principal, both `StashDex` `totalBorrowed` and pool `directDebt` can become zero while the Aave position still has residual variable debt from accrued interest.

Once `StashDex`'s nominal debt is zero, its normal guards treat the token as fully repaid. `forward()` can send remaining balances away, and `_setPool()` can allow the configured pool to be replaced, even though the old Aave position still carries interest generated by the `StashDex` borrow.

The issue is specific to integrations that treat nominal direct-borrow principal as the complete repayment obligation while the backing pool accrues external debt interest.

Interest generated by `StashDex` direct borrows can be left on the liquidity pool/Aave position while `StashDex` considers its debt repaid. This reduces pool yield and can worsen the Aave health factor over time. It can also let `StashDex` route future repayments or borrows to a new pool while the old Aave pool still carries residual debt from the prior direct-borrow flow.

Recommendation: Do not let `StashDex` completion guards depend only on nominal principal when the backing pool can accrue Aave debt above that principal. `StashDex` repayments should route through a flow that uses all available repayment funds to close the relevant Aave debt, or the integration should explicitly require residual funds to be sent to the pool/Repayer until the Aave debt produced by the borrow is settled.

If future direct borrow integrations use Aave-backed pools, document that nominal `directDebt` does not include all accrued Aave interest and that integrations need a separate repayment or reconciliation step for residual variable debt.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.2.2 Gnosis Omnibridge repayments can deliver the wrong USDC variant

Severity: Medium Risk.

Context: GnosisOmnibridgeAdapter.sol#L70-L83.

Description: Gnosis Omnibridge does not let the initiator explicitly choose the output token in the same way as the other bridge adapters, so generic input/output-token validation is not available for this provider. The accepted issue is narrower: the route must account for the USDC representation that Omnibridge actually delivers on Gnosis.

The adapter already recognizes one side of the representation mismatch. On Gnosis → Ethereum transfers, if the local token is `GNOSIS_USDCE`, it first converts to `GNOSIS_USDCXDAI` because Omnibridge uses the older USDCxDAI representation. The reverse Ethereum → Gnosis path did not perform the corresponding destination-side conversion before funds reached a pool that may account for USDCe.

As a result, a repayment can deliver the Omnibridge USDC representation while the intended destination pool's accounting/debt expects the configured Gnosis asset. The transfer succeeds at the bridge-token level, but it does not necessarily settle the intended pool asset.

Cross chain repayments can arrive as the wrong Gnosis USDC representation. Funds may sit as an unsupported or non-accounted token while the original pool debt remains outstanding until the asset is converted or the route is corrected.

Recommendation: Model the USDCe/USDCxDAI conversion in both directions. If the destination pool expects USDCe, bridge to a destination-side receiver/ `Rebalancer` that converts the Omnibridge representation before delivering or accounting the repayment. For routes where that normalization is not available, disallow the route for pools that only support the configured asset.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.2.3 Aave `withdrawProfit` shortfall borrow skips token LTV checks

Severity: Medium Risk.

Context: LiquidityPool.sol#L254-L258.

Description: Normal Aave-backed borrow paths enforce both the global health-factor check and the protocol's configured per-token LTV cap. `_afterBorrowLogic()` calls `_checkHealthFactor()` and then `_checkTokenLTV(totalCollateralBase, borrowToken)`. `_afterBorrowManyLogic()` does the same for each borrowed token.

`_withdrawProfitLogic()` contains another path that can create Aave variable debt. If the current token balance is lower than the profit amount to withdraw, it borrows the shortfall from Aave and then checks only `_checkHealthFactor()`:

```
if (int256(balance) < profit) {
    uint256 shortfall = uint256(profit) - balance;
    AAVE_POOL.borrow(address(token), shortfall, INTEREST_RATE_MODE_VARIABLE,
    ↪  NO_REFERRAL, address(this));
    _checkHealthFactor();
    currentDebt += shortfall;
}
```

This skips the per-token LTV cap that every ordinary borrow path enforces. The skipped check matters because token-specific LTV limits can be much stricter than the global health-factor constraint. In particular, `profit` can include surplus created from `directDebt`, so the shortfall can be material and not just dust.

`WITHDRAW_PROFIT_ROLE` can cause the pool to borrow a token above its configured token LTV while still passing the global health-factor check. This weakens the configured risk controls and can concentrate variable debt in tokens that the admin intentionally capped at a lower exposure.

Recommendation: After the shortfall borrow, enforce the same checks used by the ordinary borrow paths:

```
uint256 totalCollateralBase = _checkHealthFactor();
_checkTokenLTV(totalCollateralBase, address(token));
```

If profit-withdrawal borrows are intentionally exempt from token LTV limits, document that trust assumption explicitly and make the configuration reflect it.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.2.4 Aave `repayDirect` reverts when direct debt exists but Aave debt is zero

Severity: Medium Risk.

Context: LiquidityPoolAave.sol#L328-L357.

Description: `LiquidityPoolAave._repayTokenDirect()` calculates the repayment amount from the pool's `directDebt` mapping, decreases that mapping, pulls tokens from the caller, and unconditionally forwards the same amount into Aave through `_executeRepay()`:

```
uint256 outstandingDebt = $.directDebt[borrowToken];
uint256 repayAmount = Math.min(outstandingDebt, maxRepayAmount);
if (repayAmount == 0) return false;

unchecked { $.directDebt[borrowToken] = outstandingDebt - repayAmount; }
IERC20(borrowToken).safeTransferFrom(_msgSender(), address(this), repayAmount);
_executeRepay(borrowToken, repayAmount);
```

`_executeRepay()` then calls `AAVE_POOL.repay(borrowToken, repayAmount, 2, address(this))`. This call can revert when the pool has no Aave variable debt for that token, even if `directDebt[borrowToken] > 0`.

The state is reachable because direct debt and Aave debt are not always the same thing. A direct borrow can be served from local balance in an Aave long-term pool without creating matching Aave debt, or ordinary `repay()` can close Aave variable debt while leaving direct debt unchanged. In either case, the direct borrower is expected to use `repayDirect()` to clear `directDebt`, but `repayDirect()` can fail because there is no Aave debt left to repay.

This contradicts the intended recovery model where `repayDirect()` should still be able to reduce direct debt even if Aave debt was already closed.

Direct debt can become impossible to settle through the normal direct-repayment path. Integrations such as `StashDex` can be stuck with outstanding nominal debt, unable to call `forward()` or change pools cleanly, while the pool keeps stale direct debt accounting.

Recommendation: Cap the amount sent to Aave by the current variable debt balance and skip the Aave call when that balance is zero. `repayDirect()` should still reduce `directDebt` and pull repayment tokens even if there is no remaining Aave variable debt:

```
uint256 aaveDebt = HelperLib.balanceOfThis(vdToken);
uint256 aaveRepayAmount = Math.min(repayAmount, aaveDebt);
if (aaveRepayAmount > 0) {
    _executeRepay(borrowToken, aaveRepayAmount);
}
```

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.2.5 Gnosis AMB `executeSignatures` can skip rebalance accounting

Severity: Medium Risk.

Context: GnosisOmnibridgeAdapter.sol#L96-L116.

Description: For Gnosis → Ethereum Omnibridge transfers, `processTransferGnosisOmnibridge()` finalizes the bridge message by calling `ETHEREUM_AMB.executeSignatures(message, signatures)`, measures the destination pool balance delta, and returns the received amount to `Rebalancer.processRebalance()`. The Rebalancer then calls `ILiquidityPoolBase(destinationPool).deposit(depositAmount)` to credit the destination pool accounting.

The Gnosis AMB `executeSignatures` entry point is permissionless. Anyone who obtains the bridge message and validator signatures can execute the message directly on the AMB before the protocol's `processRebalance()` transaction. Once executed, the message is marked processed and a later call to `executeSignatures` for the same message reverts.

If an external caller finalizes the AMB message first, the tokens are delivered to `destinationPool`, but the protocol's later `processRebalance()` reverts inside `executeSignatures()` before it reaches `deposit()`. The destination pool receives raw token balance without the matching `totalDeposited` accounting update.

This is related to, but separate from, direct Ethereum → Gnosis Omnibridge rebalances that have no destination-side process step at all. Here the intended destination-side process step exists, but it can be front-run because the bridge finalization itself is not restricted to the `Rebalancer`.

Anyone can grief Gnosis → Ethereum Omnibridge rebalances so that bridged principal reaches the pool but is never credited as deposited liquidity. The uncredited balance can later be ignored by withdrawal accounting or misclassified as surplus/profit, depending on the pool type and follow-up operations.

Recommendation: Consider not requiring the `Rebalancer` to be the first executor of the AMB message. Make Gnosis Omnibridge processing idempotent: if the message was already executed, the `Rebalancer` should still be able to verify the expected transfer and call `deposit()` for the delivered amount. Alternatively, bridge to a destination-side receiver that records pending transfers and exposes a separate accounting-credit step that cannot be invalidated by permissionless AMB finalization.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.2.6 Attacker can force the operator requirement on a user's withdrawals via allowances

Severity: Medium Risk.

Context: LiquidityHub.sol#L371-L377.

Description: User A goes to the SHARES contract and approves User B for some tokens, so that User B can withdraw on behalf of User A. This is a perfectly valid use case, and it shouldn't require User A to set User B as an operator by calling `setOperator` on `LiquidityHub`.

However, an attacker can force User A into that requirement by calling `requestRedeem` with `shares = 1 wei`, `controller = User A`, `owner = attacker`. The attacker requests a redeem for 1 wei of their own shares with `controller = User A`. This means that when User B wants to withdraw on behalf of User A, User B must now also have been set as an operator by User A (because the `fromPending > 0` branch is entered). Tx fails otherwise.

The root cause is in `LiquidityHub._withdraw`. `requestRedeem` lets anyone credit `redeemRequests[controller]` for an arbitrary `controller`, and `_withdraw` unconditionally routes any available pending amount through a branch that requires operator authorization.

Since the injected 1 wei is never automatically cleared, the requirement persists: every allowance-based withdrawal on behalf of User A reverts with `Unauthorized` until User A additionally sets the spender as an operator (or the pending amount is redeemed away by an authorized party). The attack costs the attacker 1 wei of their own shares per targeted user and grants them nothing, but it degrades the SHARES-approval withdrawal flow for arbitrary victims.

Proof of Concept: The following test was added in `test/LiquidityHub.ts`:

```
it.only("Should allow an attacker to force operator requirement on withdraw from another
↪  user", async function () {
  const {
    lpToken, liquidityHub, usdc, deployer, user, user2, user3, USDC, LP,
  } = await loadFixture(deployAll);

  // User A (user) approves User B (user3) on the SHARES contract so User B can withdraw
  ↪  on their behalf.
  await usdc.connect(deployer).transfer(user, 10n * USDC);
  await usdc.connect(user).approve(liquidityHub, 10n * USDC);
  await liquidityHub.connect(user).deposit(10n * USDC, user);
  await lpToken.connect(user).approve(user3, 10n * LP);

  // Attacker requests a redeem for 1 wei of their own shares with controller = User A.
  await usdc.connect(deployer).transfer(user2, 1n);
  await usdc.connect(user2).approve(liquidityHub, 1n);
  await liquidityHub.connect(user2).deposit(1n, user2);
  await liquidityHub.connect(user2).requestRedeem(1n, user, user2);

  // User B can no longer withdraw on behalf of User A without being set as an operator
  ↪  by User A.
  await expect(liquidityHub.connect(user3).withdraw(1n * USDC, user2, user))
```

```
    .to.be.revertedWithCustomError(liquidityHub, "Unauthorized");
});
```

Recommendation: In `_withdraw`, only consume from the pending redeem request when the caller is the owner or an operator. When the caller is instead an approved spender (allowance on the SHARES contract), withdraw entirely from the owner's live share balance and leave the pending amount untouched, so an attacker-injected pending request cannot force the operator requirement.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.2.7 `claimableRedeemRequest` over-reports causing `fulfilRedeem` to revert

Severity: Medium Risk.

Context: LiquidityHub.sol#L266-L272.

Description: When calculating how much assets are redeemable, the hub calls `LIQUIDITY_POOL.balance(IERC20(asset()))` which simply returns the current token balance.

However, when actual withdraw is called on `LiquidityPool`, it only lets you withdraw no more than the internal storage `$.totalDeposited`.

In case of `LiquidityPoolAave`, `LIQUIDITY_POOL.balance(IERC20(asset()))` doesn't simply return the current balance but it still has a similar mismatch in some cases.

This means that `claimableRedeemRequest` sometimes over-reports. `maxRedeem` and `maxWithdraw` have the same bug as well.

`claimableRedeemRequest` over-reporting has the effect of `fulfilRedeem` failing for every receiver.

Concretely, whenever the pool holds funds above `totalDeposited` (e.g. yield/profit that is physically in the pool but only withdrawable via `withdrawProfit`, not `withdraw`), `claimableRedeemRequest` counts those funds as claimable. When such a claim is fulfilled, `LiquidityHub._withdraw` calls `LIQUIDITY_POOL.withdraw(receiver, assets)` with `assets > totalDeposited`, which reverts with `InsufficientLiquidity`. Because `fulfilRedeem` processes receivers in a single transaction using these over-reported amounts, the whole call reverts and every receiver fails.

Proof of Concept: Added below test in `test/LiquidityHub.ts`.

```
// Uses the real LiquidityPool (not the TestLiquidityPool mock) because the mock lets
↪  withdraw
// transfer freely, whereas the real pool caps withdraw at its internal $.totalDeposited.
describe("claimableRedeemRequest over-reports vs pool withdrawable", function () {
  const FULFIL_REDEEM_ROLE = toBytes32("FULFIL_REDEEM_ROLE");

  const deployWithRealPool = async () => {
    const [deployer, admin, user] = await hre.ethers.getSigners();
    const LIQUIDITY_ADMIN_ROLE = toBytes32("LIQUIDITY_ADMIN_ROLE");

    const usdc = (await deploy("TestUSDC", deployer)) as TestUSDC;
    const USDC = 10n ** (await usdc.decimals());

    const liquidityPoolImpl = (await deployX(
      "LiquidityPool", deployer, "LiquidityPoolReal", {}, usdc,
      ↪  networkConfig.BASE.WrappedNativeToken
    )) as LiquidityPool;
    const liquidityPoolInit = (await liquidityPoolImpl.initialize.populateTransaction(
      admin, admin, admin)
    ).data;
```

```
  const liquidityPoolProxy = (await deployX(
    "TransparentUpgradeableProxy", deployer,
    ↪  "TransparentUpgradeableProxyLiquidityPoolReal", {},
    liquidityPoolImpl, admin, liquidityPoolInit
  )) as TransparentUpgradeableProxy;
  const liquidityPool = (
    await getContractAt("LiquidityPool", liquidityPoolProxy, deployer)
  ) as LiquidityPool;

  const liquidityHubAddress = await getDeployXAddressBase(
    deployer, "TransparentUpgradeableProxyLiquidityHubReal", false);
  const lpToken = (
    await deployX("SprinterUSDCLPShare", deployer, "SprinterUSDCLPShareReal", {},
    ↪  liquidityHubAddress)
  ) as SprinterUSDCLPShare;
  const LP = 10n ** (await lpToken.decimals());

  const liquidityHubImpl = (
    await deployX("LiquidityHub", deployer, "LiquidityHubReal", {}, lpToken,
    ↪  liquidityPool)
  ) as LiquidityHub;
  const liquidityHubInit = (await liquidityHubImpl.initialize.populateTransaction(
    usdc, admin, admin, admin, admin, getBigInt(MaxUint256) * USDC / LP)
  ).data;
  await deployX(
    "TransparentUpgradeableProxy", deployer,
    ↪  "TransparentUpgradeableProxyLiquidityHubReal", {},
    liquidityHubImpl, admin, liquidityHubInit
  );
  const liquidityHub = (
    await getContractAt("LiquidityHub", liquidityHubAddress, deployer)
  ) as LiquidityHub;

  await liquidityPool.connect(admin).grantRole(LIQUIDITY_ADMIN_ROLE, liquidityHub);
  await liquidityHub.connect(admin).grantRole(FULFIL_REDEEM_ROLE, admin);

  return {deployer, admin, user, usdc, lpToken, liquidityHub, liquidityPool, USDC, LP};
};

it("claimableRedeemRequest over-reports, making fulfilRedeem fail for every receiver",
↪  async function () {
  const {deployer, admin, user, usdc, liquidityHub, liquidityPool, USDC, LP} =
    await loadFixture(deployWithRealPool);

  // User deposits 10 USDC. pool.totalDeposited = 10 USDC, pool balance = 10 USDC.
  await usdc.connect(deployer).transfer(user, 10n * USDC);
  await usdc.connect(user).approve(liquidityHub, 10n * USDC);
  await liquidityHub.connect(user).deposit(10n * USDC, user);

  // 1 USDC of yield lands in the pool (raw balance = 11 USDC) but is NOT part of
  ↪  totalDeposited,
  // and the adjuster credits it to the hub (totalAssets = 11 USDC).
  await usdc.connect(deployer).transfer(liquidityPool, 1n * USDC);
  await liquidityHub.connect(admin).adjustTotalAssets(1n * USDC, INCREASE);

  expect(await liquidityPool.balance(usdc)).to.equal(11n * USDC);
  expect(await liquidityPool.totalDeposited()).to.equal(10n * USDC);

  await liquidityHub.connect(user).requestRedeemWithFulfil(10n * LP);

  // claimableRedeemRequest reports all 10 LP because it uses balance() = 11 USDC:
  //   availableShares = convertToShares(11 USDC) = 11 * 10 / 11 = 10 LP.
  // But the pool can withdraw at most totalDeposited = 10 USDC, which is only 9 LP
  ↪  worth.
  expect(await liquidityHub.claimableRedeemRequest(0n, user)).to.equal(10n * LP);
```

```
    // Because claimableRedeemRequest over-reports, redeem tries to withdraw 11 USDC from
    ↪  the pool,
    // exceeding totalDeposited (10 USDC), so fulfilRedeem reverts for the receiver.
    await expect(liquidityHub.connect(admin).fulfilRedeem([user]))
      .to.be.revertedWithCustomError(liquidityPool, "InsufficientLiquidity");
  });
});
```

Recommendation: The amount that can actually be pulled out of the pool via `withdraw` is bounded by `totalDeposited`, not by the raw balance. Bound the available assets by both quantities everywhere the hub reasons about redeemable assets (`claimableRedeemRequest`, `maxRedeem`, and `maxWithdraw`):

```
uint256 availableAssets = Math.min(
    LIQUIDITY_POOL.balance(IERC20(asset())),
    LIQUIDITY_POOL.totalDeposited()
);
```

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.2.8 Gnosis Repayer fix is incomplete in both directions

Severity: Medium Risk.

Context: Repayer.sol#L257-L258, Repayer.sol#L267-L284, GnosisOmnibridgeAdapter.sol#L67-L84.

Description: The remediation for finding #4 adds the destination-side `processTransferGnosisOmnibridge()` normalization, but `Repayer.initiateRepay()` still passes the logical `destinationPool` directly to Omnibridge. This bypasses the new processing path in one direction and conflicts with its adapter precondition in the other.

Remediation reviewed: PR 275.

For Ethereum → Gnosis, a caller supplies the configured Gnosis pool as `destinationPool`. The route check succeeds and Omnibridge delivers USDCxDAI directly to that pool. Because the Gnosis Repayer never receives the tokens, `processRepay()` cannot call the transmuter to convert USDCxDAI to USDCe. The original wrong-USDC-representation condition therefore remains: the transfer can complete while the intended USDCe repayment is not accounted by the pool.

For Gnosis → Ethereum, a caller supplying the configured Ethereum pool cannot initiate a USDCe repayment. The adapter converts USDCe to USDCxDAI, but first requires `destinationPool == address(this)`, so the normal call reverts with `InvalidDestinationPool`. The fork test works around this by passing the Repayer itself instead of the configured destination pool. That is not equivalent: `isRouteAllowed(address(this), ...)` always returns true, the initiation event no longer identifies the logical pool, and the public parameter has different semantics only for this route.

The existing Ethereum → Gnosis unit test reinforces the first bug by expecting a direct bridge to the liquidity pool. Its bridge mock ignores the receiver argument, so it does not exercise whether the destination Repayer actually receives and normalizes the bridged token.

Impact: Ethereum → Gnosis repayments can still deliver USDCxDAI to a pool that expects USDCe, leaving the intended debt outstanding. Gnosis → Ethereum USDCe repayments revert when called with the configured destination pool. Thus the accepted finding remains exploitable in one direction and the route is unusable with its documented arguments in the other.

Recommendation: Keep `destinationPool` as the logical, route-checked destination, but use `address(this)` as the Omnibridge receiver in both directions, as `Rebalancer` already does. The destination Repayer can then normalize the token in `processRepay()` and transfer the result to the original pool. Preserve the logical pool in events and route validation. Add end-to-end tests for both directions using the configured pool argument and a bridge mock that delivers tokens to the requested receiver.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.3 Low Risk

3.3.1 `LiquidityHub` zero-asset adjustment advertises impossible withdrawals

Severity: Low Risk.

Context: LiquidityHub.sol#L314-L324.

Description: `adjustTotalAssets(amount, false)` can reduce the Hub's accounting `totalAssets` to zero while live shares still exist. In that state, `_getTotalsForConversion()` treats zero accounting assets as a bootstrap case and substitutes `supplyAssets = 1`:

```
if (supplyAssets == 0) {
    supplyAssets = 1;
}
```

That fallback is safe for an empty vault bootstrap, but it is also applied when `totalSupply() > 0`. As a result, after an authorized adjustment from `totalAssets = 1` to `totalAssets = 0`, a holder with 1 live share can still see `maxWithdraw(owner) == 1` because conversions are performed against the virtual `supplyAssets = 1`.

The advertised withdrawal is impossible. A subsequent `withdraw(1, receiver, owner)` enters `_withdraw()` and reverts with a Solidity panic at:

```
$.totalAssets -= assets;
```

because real accounting `totalAssets` is zero.

The issue was reproduced by the ERC-7540 Hub Echidna harness with this minimized sequence:

1. Mint 1 live share backed by 1 pool asset.
2. Authorized `adjustTotalAssets(..., false)` reduces Hub accounting assets from 1 to 0.
3. `maxWithdraw(owner)` still returns 1.
4. `withdraw(1, ...)` panics when subtracting from zero `totalAssets`.

After an authorized loss adjustment to zero, ERC-4626 views can advertise withdrawable assets that cannot be withdrawn. Users, integrations, and fulfilment logic can rely on `maxWithdraw()` / `maxRedeem()` and still hit a low-level arithmetic panic. This is not permissionless theft because it requires `ASSETS_ADJUST_ROLE`, but it is an accounting/view consistency bug and can break exits after a full-loss adjustment.

Recommendation: Do not apply the bootstrap virtual-asset fallback when shares already exist. If `totalAssets() == 0 && totalSupply() > 0`, conversion and max-withdraw paths should treat shares as economically worthless and return zero withdrawable assets, or `adjustTotalAssets()` should reject decreasing accounting assets to zero while live or pending shares remain.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.3.2 ERC-7540 standard violations in `LiquidityHub`

Severity: Low Risk.

Context: LiquidityHub.sol#L33, LiquidityHub.sol#L232-L235.

Description: `LiquidityHub` implements the asynchronous redemption flow of ERC-7540 when `LiquidityPool` does not have enough balance, but it does not conform to the standard in three places:

1. The vault does not expose a `supportsInterface` (EIP-165) that returns `true` when the `0x620ee8e4` interfaceId is passed. `supportsInterface` is only inherited from `AccessControlUpgradeable`, so today it returns `true` for `IAccessControl` and `IERC165` and nothing else. Without the async-redeem interfaceId, offchain components have no way to know that this vault supports async redemptions as specified by EIP-7540, and will classify it as a plain synchronous ERC-4626 vault.
2. The vault does not expose a `shares()` function as defined by ERC-7575. Shares are minted on a separate token contract (`SHARES`), not on the vault itself, so the vault address is not the share token. The token is only reachable through the non-standard `SHARES()` getter. A `shares()` accessor allows offchain components to discover the ERC-20 share token that is minted when users deposit into this vault and without it, any integrator that assumes the ERC-4626 default (share token == vault address) reads balances and total supply from the wrong contract.
3. `setOperator` does not return a `bool` as specified by ERC-7540. The implementation returns nothing, which changes the function's ABI.

Recommendation: Bring the vault into conformance with ERC-7540:

```
// 1. Advertise the async-redeem (and operator / ERC-7575) interfaces via ERC-165.
function supportsInterface(bytes4 interfaceId) public view virtual override returns
↪  (bool) {
    return interfaceId == 0x620ee8e4    // ERC-7540 async redeem
        || interfaceId == 0xe3bc4e65    // ERC-7540 operator
        || interfaceId == 0x2f0a18c5    // ERC-7575 vault
        || super.supportsInterface(interfaceId);
}

// 2. Expose the ERC-20 share token per ERC-7575.
function shares() public view returns (address) {
    return address(SHARES);
}

// 3. Return bool from setOperator per ERC-7540.
function setOperator(address operator, bool approved) public returns (bool) {
    _getStorage().operators[_msgSender()][operator] = approved;
    emit OperatorSet(_msgSender(), operator, approved);
    return true;
}
```

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.3.3 Stablecoin pool direct borrows of non assets cannot be repaid

Severity: Low Risk.

Context: LiquidityPoolStablecoin.sol#L28-L32.

Description: `LiquidityPoolStablecoin` allows borrowing any token held by the pool, but it inherits the base direct-repayment implementation, which only accepts the pool `ASSETS` token. If `DIRECT_BORROW_ROLE` is ever granted on a stablecoin pool and the borrower uses `borrowDirect()` for another stablecoin, the resulting `directDebt[token]` cannot be cleared through `repayDirect()`.

The stablecoin pool removes the base borrow token restriction.

`borrowDirect()` is inherited from `LiquidityPoolBase` and records direct debt for the actual borrowed token on `LiquidityPool.sol` contract.

However, the inherited `_repayDirect()` only accepts a single repayment token and requires it to equal `ASSETS`.

So a direct borrow of, for example, PYUSD from a USDC stablecoin pool increments `directDebt[PYUSD]`, but `repayDirect([PYUSD], ...)` reverts with `InvalidAsset`, and `repayDirect([USDC], ...)` does not reduce the PYUSD debt.

This also breaks the `StashDex` repayment integration if a stablecoin pool is configured as a token-out pool: `StashDex.swap()` can call `borrowDirect(tokenOut, amountOut)`, but `StashDex.repay(tokenOut)` later calls `pool.repayDirect([tokenOut], ...)` and reverts for any `tokenOut != ASSETS`.

Even a manual token transfer back to the pool does not fix the accounting. The stablecoin pool's profit logic gates surplus withdrawals only on the USDC `ASSETS` balance plus `directDebt[ASSETS]`; once that condition is true, non-asset token balances are returned as withdrawable surplus without subtracting `directDebt[token]`. As a result, a manually returned PYUSD balance can still be withdrawn as profit while `directDebt[PYUSD]` remains outstanding and uncleared.

Current production `StashDex` config points its pooled tokens at the Aave pool, so this is not active in the shipped config. The contract-level integration is still inconsistent: granting `DIRECT_BORROW_ROLE` on a stablecoin pool or routing `StashDex` through it can create direct debt that cannot be repaid through the intended direct-repayment path. Operators would need manual accounting or a contract change to clear the debt.

Recommendation: Consider either overriding `_repayDirect()` in `LiquidityPoolStablecoin` to accept and reduce direct debt for any borrowed stablecoin, or restrict `borrowDirect()` on this pool to `ASSETS` only. If multi-token direct borrowing is intentionally unsupported, make that explicit with an override that reverts before debt is recorded.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.3.4 `ERC4626Adapter` zero share deposits can consume withdrawable profit

Severity: Low Risk.

Context: ERC4626Adapter.sol#L132-L152.

Description: `ERC4626Adapter.depositWithPull()` is permissionless and forwards the supplied assets into the target ERC4626 vault:

```
function depositWithPull(uint256 amount) external override whenNotPaused() {
    ASSETS.safeTransferFrom(_msgSender(), address(this), amount);
    _deposit(_msgSender(), amount);
}
```

`_deposit()` then calls `TARGET_VAULT.deposit(amount, address(this))` and increases adapter `totalDeposited` by the raw asset amount:

```
TARGET_VAULT.deposit(amount, address(this));
_getStorage().totalDeposited += amount;
```

The adapter does not check how many vault shares were minted. If the ERC4626 vault rounds a very small deposit down to zero shares, `totalDeposited` still increases even though the adapter's vault share balance does not.

This can shift previously withdrawable vault profit into deposited-principal accounting. `withdrawProfit()` computes deposited backing through `TARGET_VAULT.previewWithdraw(totalDeposited)` and preserves that many shares before redeeming the remainder as profit. If a zero-share deposit increases `totalDeposited`, the adapter must reserve more shares for principal, reducing the amount that can be withdrawn as profit.

The Echidna issue mode harness reproduced this with a target vault that had 20 ether of withdrawable profit. A permissionless `depositWithPull(1)` minted zero vault shares but increased `totalDeposited` by 1. A later `withdrawProfit([ASSETS])` paid `19.999999999999999999 ether`, consuming 1 wei of previously withdrawable profit through the accounting shift.

Any caller can make tiny deposits that increase adapter `totalDeposited` without increasing the adapter's vault-share balance. This can grief profit withdrawal and distort adapter accounting by converting previously withdrawable profit into accounted principal.

The direct demonstrated loss is bounded by the caller's deposited dust amount, so this is not direct theft of pool assets. The issue is still relevant because the function is permissionless and the adapter accounting should not treat a deposit as principal-backed if no vault shares were minted.

Recommendation: Use the share amount returned by `TARGET_VAULT.deposit()` and reject zero share deposits:

```
uint256 shares = TARGET_VAULT.deposit(amount, address(this));
require(shares > 0, ZeroShares());
_getStorage().totalDeposited += amount;
```

Alternatively, check `TARGET_VAULT.previewDeposit(amount) > 0` before pulling funds. If the adapter is only meant to receive operational rebalances, consider restricting `depositWithPull()` or adding a minimum deposit threshold that prevents permissionless dust from changing principal accounting.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.3.5 `StashDex` oracle sync omits input only route assets

Severity: Low Risk.

Context: deployPaxosOracle.ts#L26-L43.

Description: `deployPaxosOracle.ts` and the `update-paxos-oracle-assets` task register only tokens listed in `config.StashDex.Pools`. `deployStashDex.ts`, however, only requires each route's `TokenOut` to have a configured pool. It accepts any configured `TokenIn` token:

```
for (const {TokenOut} of stashDexConfig.Routes) {
  assert(stashDexConfig.Pools[TokenOut], `Route tokenOut ${TokenOut} has no pool
  ↪  configured in StashDex.Pools`);
}
```

At runtime, `StashDex.swap()` prices both the input and output token through the immutable oracle before it transfers funds:

```
uint256 valueIn = ORACLE.getAssetValue(_tokenToAssetId(tokenIn), amountIn * precision);
uint256 valueOut = ORACLE.getAssetValue(_tokenToAssetId(tokenOut), amountOut *
↪  precision);
```

If a future route uses a token only as `TokenIn`, the `StashDex` deployment accepts the route but the Paxos oracle deploy/sync flow omits that asset. The route is then configured on-chain but unusable because the input-token oracle lookup reverts.

A valid-looking `StashDex` route can be deployed or retained while swaps through it always revert. The failure happens before the input transfer, so this is an availability/configuration issue rather than a direct fund-loss path.

Current visible route configs use symmetric pool tokens, so this is primarily a guardrail issue for future route changes.

Recommendation: Build the Paxos oracle desired asset set from every token used by `StashDex` routes and pools, not only `StashDex.Pools`. Deployment and sync scripts should also assert that the configured oracle supports both `TokenIn` and `TokenOut` for every allowed route before installing the route.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.3.6 Conditional virtual shares re-opens the ERC4626 share-price reset attack

Severity: Low Risk.

Context: LiquidityHub.sol#L314-L324.

Description: OpenZeppelin's ERC4626 defends against share-price manipulation by adding virtual shares and virtual assets to every conversion:

```
convertToAssets = shares * (totalAssets + 1)          / (totalSupply + 10**offset)
convertToShares = assets * (totalSupply + 10**offset) / (totalAssets + 1)
```

These virtual amounts are always present, so a ”phantom” position permanently owns a slice of the vault and the effective exchange rate can never be reset to the base 1:1 rate, even after every real share is burned.

`LiquidityHub` has different calculation. In `_getTotalsForConversion` the virtual amounts are only applied when a total is exactly zero.

Whenever both totals are non-zero (the normal operating state) the conversion is the naive `assets * totalSupply / totalAssets`, with no virtual protection. As a result the share price can be driven back to the base rate. An attacker who momentarily controls all shares (Unlikely but possible if a flash loan provider has all the totalSupply) can redeem every share to bring `totalSupply` and `totalAssets` to `0`, then re-mint the same shares at the reset 1:1 rate and pocket the accumulated yield.

The rate becomes elevated whenever yield is added via `depositProfit` or `adjustTotalAssets`, which is the protocol's normal behaviour.

The likelyhood is very low because it requires a flashloan provider to control all of the total shares AND all of the underlying assets are available to be withdrawn.

Proof of Concept: Add the following test to `test/LiquidityHub.ts`:

```
it.only("PoC: price-reset attack drains accrued yield", async function () {
  const {
    lpToken, liquidityHub, usdc, deployer, user, user2, admin, USDC, LP, liquidityPool,
  } = await loadFixture(deployAll);
```

```
  // Victim (user) and attacker (user2) each deposit 10 USDC -> 10 LP. Price = 1 USDC/LP.
  await usdc.connect(deployer).transfer(user, 10n * USDC);
  await usdc.connect(deployer).transfer(user2, 10n * USDC);
  await usdc.connect(user).approve(liquidityHub, 10n * USDC);
  await usdc.connect(user2).approve(liquidityHub, 10n * USDC);
  await liquidityHub.connect(user).deposit(10n * USDC, user);
  await liquidityHub.connect(user2).deposit(10n * USDC, user2);

  // 20 USDC of real yield is added to the pool. Share price doubles to 2 USDC/LP.
  await usdc.connect(deployer).transfer(admin, 20n * USDC);
  await usdc.connect(admin).approve(liquidityHub, 20n * USDC);
  await liquidityHub.connect(admin).depositProfit(20n * USDC);
  expect(await liquidityHub.totalAssets()).to.equal(40n * USDC);
  expect(await liquidityHub.totalSupply()).to.equal(20n * LP);
  // Victim's 10 LP are worth 20 USDC.
  expect(await liquidityHub.convertToAssets(10n * LP)).to.equal(20n * USDC);

  // Attacker flash-borrows all of the victim's shares ( from some flashloan provider).
  await lpToken.connect(user).transfer(user2, 10n * LP);

  // 1) Redeem EVERY share -> totalSupply and totalAssets both hit 0, draining the vault.
  await liquidityHub.connect(user2).redeem(20n * LP, user2, user2);
  expect(await liquidityHub.totalSupply()).to.equal(0n);
  expect(await liquidityHub.totalAssets()).to.equal(0n);
  expect(await usdc.balanceOf(user2)).to.equal(40n * USDC);

  // 2) Price is reset to the base 1:1 rate. Re-mint the same 20 LP for only 20 USDC.
  await usdc.connect(user2).approve(liquidityHub, 20n * USDC);
  await liquidityHub.connect(user2).deposit(20n * USDC, user2);
  expect(await liquidityHub.totalSupply()).to.equal(20n * LP);

  // 3) Return the borrowed 10 LP to the victim (repay the flashloan).
  await lpToken.connect(user2).transfer(user, 10n * LP);

  // Attacker profit: deposited 10 USDC, walks away with 20 USDC cash + 10 LP.
  expect(await usdc.balanceOf(user2)).to.equal(20n * USDC);
  expect(await lpToken.balanceOf(user2)).to.equal(10n * LP);
  // The victim's shares were devalued from 20 USDC to 10 USDC: their yield was stolen.
  expect(await liquidityHub.convertToAssets(10n * LP)).to.equal(10n * USDC);
});
```

Recommendation: Apply the virtual shares and virtual assets unconditionally, exactly as OpenZeppelin does, instead of only when a total is zero:

```
function _getTotalsForConversion() internal view returns (uint256, uint256) {
    return (totalSupply() + 10 ** _decimalsOffset(), totalAssets() + 1);
}
```

With the offset always present, a full redemption leaves the virtual position owning the residual assets, so the effective rate never resets to base and re-minting costs at least what was redeemed, making the round-trip unprofitable.

Sprinter Tech: See transaction 0x738bbc37ee272b5601e9cc18250827cf07911e622f8b9bbe348eeae50489f613.

Cantina Managed: Fix verified.

3.3.7 `StashDex` upgrade can abandon legacy direct debt

Severity: Low Risk.

Context: StashDex.sol#L38-L41, StashDex.sol#L156-L160, StashDex.sol#L186-L203, StashDex.sol#L259-L266.

Description: The remediation for the StashDex/Aave interest issue changes StashDex from direct-debt accounting to the new `LiquidityPool.borrowWithRole()` flow:

Commit 12a46251.

That is directionally correct for future swaps, because new StashDex borrows no longer increase pool `directDebt` and repayments are expected to route through the Repayer. However, the upgrade does not migrate or guard existing deployed state.

Before the remediation, StashDex stored each token's `totalBorrowed`, incremented it in `swap()`, repaid it through `repay()` using `config.pool.repayDirect()`, called `repay()` before `forward()`, exposed `getTotalBorrowed()`, and blocked `_setPool()` while `totalBorrowed != 0`.

After the remediation, `TokenConfig.totalBorrowed` is left in storage only as deprecated layout, but no production function reads or reduces it. The `repay()` and `getTotalBorrowed()` functions are removed, `forward()` transfers the full balance to `RECEIVER` without first repaying legacy debt, and `_setPool()` can replace the configured pool without checking outstanding legacy debt.

This creates an unsafe upgrade path for any already-deployed StashDex instance with nonzero `tokenConfig[token].totalBorrowed` and matching pool `directDebt[token]`:

1. The old implementation borrows through `borrowDirect()` and records StashDex legacy debt in both StashDex `totalBorrowed` and the pool's `directDebt`.
2. The proxy is upgraded to the remediation implementation.
3. The old `totalBorrowed` value still exists in ERC-7201 storage, but the new implementation no longer exposes `getTotalBorrowed()` or a `repay()` path that uses it.
4. If repayment tokens later reach StashDex, `forward()` now sends them to `RECEIVER` instead of calling `repayDirect()` first.
5. `setPool()` can move the token to a new pool even while the old pool still accounts direct debt to StashDex.
6. If StashDex is the only configured direct borrower for the old pool, the normal integration path can no longer reduce the old `directDebt` without another upgrade, role change, or manual rescue action.

The issue is about upgrade safety, not the future borrow model. The new model avoids creating new StashDex direct debt, but it can permanently abandon direct debt that already existed before the upgrade.

Legacy direct debt can remain outstanding in the old liquidity pool while StashDex continues operating against the new implementation or even a new configured pool. That stale debt can distort pool accounting, interfere with profit withdrawal and health-factor management, and leave LP collateral backing debt that the upgraded integration no longer knows how to repay.

On live Ethereum state reviewed during this remediation check, StashDex still had nonzero legacy debt for USDC and USDG before this migration concern was raised. Those values matched the corresponding pool `directDebt` values, which confirms this is not only a theoretical storage-layout edge case.

Recovery would require an additional privileged action, such as a temporary legacy repayment function, a role grant to a replacement repayment helper, or another upgrade. The submitted remediation does not include that migration or pre-upgrade guard.

Recommendation: Make the upgrade explicitly handle legacy StashDex debt before removing the old repayment path. Acceptable approaches include:

1. Before upgrading, enumerate every configured token and require `tokenConfig[token].totalBorrowed == 0` and the corresponding pool `directDebt[token] == 0`.
2. Keep a temporary legacy repayment function after the upgrade that reads the deprecated `totalBorrowed` field and calls `repayDirect()` until all legacy debts are zero.
3. Preserve the old `_setPool()` outstanding-debt guard for tokens with nonzero deprecated `totalBorrowed`.

4. Make `forward()` repay legacy direct debt before forwarding any remaining balance until the migration is complete.
5. Add a forked migration test against deployed StashDex state that proves all legacy `totalBorrowed` and pool `directDebt` values are cleared before the new implementation is considered live.

Only after the legacy debt is proven zero should StashDex rely solely on the new `borrowWithRole()`/Repayer model.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Aave repay can desync direct debt

Severity: Informational.

Context: LiquidityPoolAave.sol#L285-L314.

Description: `LiquidityPoolAave.repay()` repays Aave variable debt but never reduces the pool's `directDebt` mapping. If debt created by `borrowDirect()` is later settled through the public `repay()` path instead of `repayDirect()`, Aave debt decreases while `directDebt` remains inflated. The profit withdrawal logic then treats that stale direct debt as surplus backing and can borrow from Aave again to pay it out as profit.

Using the wrong repayment entry point for direct-borrowed Aave debt can permanently decouple internal direct debt accounting from the actual Aave debt. Once decoupled, profit withdrawal can treat stale direct debt as withdrawable surplus, borrow new Aave debt to fund the withdrawal, and transfer repayment or donated balances out as protocol profit. This can leave LP collateral backing newly-created Aave debt even though the original direct borrow was already repaid through `repay()`.

Recommendation: Prevent the ordinary `repay()` path from settling direct borrow debt without updating direct-debt accounting. Options include tracking which Aave debt was opened by direct borrows and reducing `directDebt` when `repay()` settles that debt, blocking `repay()` for tokens with outstanding direct debt, or making `repayDirect()` the only repayment path that can touch Aave debt attributable to direct borrows. Profit withdrawal should not count `directDebt` as surplus backing unless the corresponding direct debt is still actually outstanding.

Sprinter Tech: This is expected behavior covered by tests. Direct debt must be repaid by direct borrowers through repayDirect(). Even if all the Aave debt is already closed (which could be done even by some arbitrary wallet talking directly to Aave) then repayDirect() will still reduce the direct debt, without sending any funds to Aave.

Cantina Managed: Acknowledged.

3.4.2 Ordinary Aave repay does not clear direct debt

Severity: Informational.

Context: Repayer.sol#L267-L288.

Description: `LiquidityPoolAave.repay()` can reduce Aave variable debt without reducing the pool's `directDebt` mapping. That means Aave debt and direct-debt accounting can diverge if debt originally associated with a direct borrower is paid through the ordinary repayment path or directly against Aave by some other actor.

This divergence is part of the direct-debt accounting model. `directDebt` is treated as the direct borrower's obligation to the pool, not as a strict mirror of the current Aave variable-debt balance. Ordinary `repay()` does not clear that obligation; direct borrowers must call `repayDirect()`.

Under this model, the important documentation point is that "Aave debt is repaid" and "direct debt is settled" are separate states. Integrations should not infer that clearing variable debt also clears direct-borrow accounting.

If an integration uses `repay()` or direct Aave repayment when it meant to settle direct debt, `directDebt` remains outstanding until the borrower calls `repayDirect()`. Profit and virtual-balance calculations will continue to include that direct debt according to the accepted model.

The separate edge where `repayDirect()` could fail after Aave debt was already zero was accepted as a real bug and is tracked in `medium-aave-repaydirect-reverts-when-aave-debt-is-zero.md`.

Recommendation: Document the distinction between ordinary Aave repayment and direct-debt settlement. Direct borrowers and integrations should be required to use `repayDirect()` for direct debt, even if Aave variable debt has already been reduced by another path.

Sprinter Tech: Acknowledged.

Cantina Managed: Acknowledged.

3.4.3 Superchain `StandardBridge` USDC routes can temporarily retain repayments

Severity: Informational.

Context: SuperchainStandardBridgeAdapter.sol#L48-L66.

Description: `SuperchainStandardBridgeAdapter.initiateTransferSuperchainStandardBridge()` accepts an `outputToken` from route `extraData`, checks that the token is allowlisted for the destination domain, and then calls `bridgeERC20To(localToken, outputToken, destinationPool, amount, ...)`.

For Superchain StandardBridge routes, allowlisting the destination token is not enough. The chosen `localToken` and `outputToken` must also be a valid StandardBridge token pair. In the reviewed configuration, some native USDC repayment routes could select `Provider.SUPERCHAIN_STANDARD_BRIDGE` for OP Mainnet or Base destinations even though the configured destination asset was Circle native USDC rather than the StandardBridge mintable representation.

If such a route is used, funds can enter the bridge flow while the destination repayment does not complete normally. This is a temporary stranding/delay scenario rather than permanent loss when the bridge message can be made processable by a same-amount reverse bridge that provides the missing destination-side liquidity.

The intended destination pool remains unpaid until operators recover the route or process the delayed message. This can delay debt settlement and require manual bridge recovery, but the funds are not expected to be permanently lost.

Recommendation: Do not route native USDC/`ASSET` repayments through `SUPERCHAIN_STANDARD_BRIDGE`; use CCTP/CCTP v2 for native USDC instead. If StandardBridge support remains enabled for other assets, route sync should validate provider-specific token pairs, not only whether the destination token is generally allowlisted.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.4.4 Code improvement suggestions

Severity: Informational.

Context: *(No context files were provided by the reviewer)*.

Description:

1. In `StashDex.swap`, the fee check `require(valueIn * (BPS - route.feeBps) >= valueOut * BPS, InsufficientOutput());` reverts with `InsufficientOutput` when the input value net of fees does not cover the requested output value. The error name is misleading. When this check fails it should fail with `InsufficientInput` or `ExcessiveOutput` instead of `InsufficientOutput`. If the user specifies an insufficient output the swap doesn't fail, it just accepts it and gives the user less output tokens than they deserve.

Make sure there is no confusion that `amountOut` specifies the minimum amount the user wants out. It can be ANY amount, and `StashDex` will return exactly that much amount as long as it is less than the `tokenIn` value minus fees.

2. `Processor.process4626` takes an `IERC4626 tokenIn` and values the shares with `tokenIn.convertToAssets(sharesIn)` before passing the result to `_assertOutputAmount`. That valuation is only meaningful if the vault's underlying asset is the target asset, but nothing enforces it. Add a sanity check to make sure the underlying asset for the `tokenIn` vault is `TARGET_ASSET`, by requiring that `tokenIn.asset()` equals `address(TARGET_ASSET)`.
3. `uint256 precision = 10**12;` is declared as a local variable inside the function body in two places, `StashDex.sol:149` and `Netter.sol:51`. Make it a global constant.

Recommendation: Make the changes as suggested.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.4.5 Regular Processor deployments leave process unusable with a zero oracle

Severity: Informational.

Context: upgradeTokenProcessor.ts#L52-L62.

Description: `Processor.process` always calls `ORACLE.getAssetValue` to compare the input and minimum output values. The base `Processor` constructor does not reject a zero oracle, and the deploy and upgrade scripts for regular token processors pass `ZERO_ADDRESS` as the oracle constructor argument.

The test suite confirms the resulting behavior: a processor deployed with a zero oracle reverts when `process()` is called because the external oracle call goes to address zero. That means regular processors deployed through the provided scripts can forward tokens and use helper-specific flows, but their generic conversion path is unusable.

Assets that require the generic `process()` path for an unwind or swap cannot be processed by regular token processors deployed from these scripts. Operators must either route around the processor, forward assets without the intended slippage check, or upgrade/redeploy with a real oracle before completing the unwind.

Recommendation: Require a nonzero oracle for processors that expose `process()`, and pass the intended oracle in `deployTokenProcessor.ts` and `upgradeTokenProcessor.ts`. If some deployments are intentionally forward-only, split that mode into a separate contract or disable `process()` explicitly so the deployed interface does not advertise an unusable path.

Sprinter Tech: Acknowledged.

Cantina Managed: Acknowledged.

3.4.6 Safe proposals are treated as executed by maintenance scripts

Severity: Informational.

Context: safe.ts#L140-L169.

Description: When `SAFE` is configured, `createSender` returns a `SafeSigner`. For non-deployment transactions, `SafeSigner.sendTransaction` proposes a Safe transaction and only executes it if the current confirmation count already meets the Safe threshold. If the threshold is not met, it returns a fake `TransactionResponse` whose `wait()` resolves to `null`.

`upgradeProxyX` treats the signer as able to execute immediately whenever `ProxyAdmin.owner()` equals the signer address. With a `SafeSigner`, that address is the Safe itself, so `upgradeProxyX` calls `upgradeAndCall`, prints that the upgrade was sent and completed, and returns `txRequired: false` even when only a Safe proposal was created and no on-chain upgrade happened.

All scripts that call `upgradeProxyX` can therefore report an upgrade as complete while it is only pending in the Safe UI. This is especially risky for scripts with post-upgrade actions. For example, `upgradeTokenProcessor.ts` checks `txRequired` and, if it is false, proceeds as though the new implementation is live before initializing the new `SubProcessor` and granting `CONFIG_ROLE`. Under a multi-signature Safe, those follow-up steps can run against the old implementation or fail after the script has already reported the upgrade path as complete.

The same pattern affects the new Safe-enabled Hardhat maintenance tasks. Tasks such as `grant-role`, LTV updates, route updates, `update-tokens-repayer`, `StashDex` route/pool sync, Paxos oracle asset sync, and Netter processor-role grants call contracts through `createSender()` and then print success after awaiting the returned transaction or fake `wait()`. With a Safe below threshold, these tasks have only submitted a proposal, but their output says the role was granted or the configuration was updated.

Operators can believe critical upgrades, role grants, route changes, oracle asset changes, or post-upgrade initialization steps have executed when they are only pending Safe proposals. This can leave stale implementations, missing initialization, incomplete access-control changes, or unsynchronized routes in production.

Recommendation: Make Safe proposal state explicit. `SafeSigner.sendTransaction` should either return only after a real on-chain receipt exists or expose that the transaction was merely proposed. `upgradeProxyX`, Hardhat tasks, and dependent scripts should return `txRequired: true` or a separate `proposalPending` state unless execution is confirmed on-chain. Any post-upgrade initialization, role grant, or route/config update should be gated on a confirmed receipt.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.

3.4.7 Aave token LTV cannot be set to zero while a default LTV is configured

Severity: Informational.

Context: LiquidityPoolAave.sol#L104-L117.

Description: `LiquidityPoolAave.setBorrowTokenLTVs()` stores the configured per-token LTV value directly. The enforcement paths then treat a stored value of `0` as "unset" and fall back to `defaultLTV`:

```
uint256 ltv = $.borrowTokenLTV[borrowToken];
if (ltv == 0) ltv = $.defaultLTV;
```

This means an admin cannot disable borrowing for a specific token by setting its LTV to zero while the pool has a nonzero default LTV. The emitted `BorrowTokenLTVSet(token, oldLTV, 0)` event suggests that the token's configured cap was set to zero, but `_checkTokenLTV()` and `balance()` both continue to use the default cap.

Risk maintenance can leave a token borrowable when operators intended to fully disable exposure to that token. Borrowing still requires the normal authorized borrowing paths and global health-factor checks, so this is not a permissionless loss path. It is an operational risk-control gap: a token-specific emergency reduction to zero does not take effect unless the global default LTV is also reduced.

Recommendation: Use a separate sentinel for "unset" instead of `0`, or store an explicit `isConfigured` flag per token. A configured LTV of zero should be enforced as zero and should make `balance(token)` return no token-specific borrow capacity.

Sprinter Tech: Acknowledged.

Cantina Managed: Acknowledged.

3.4.8 Aave-backed Hub fix underreports withdrawable liquidity

Severity: Informational.

Context: LiquidityHub.sol#L266-L301, LiquidityPoolAave.sol#L210-L215, LiquidityPoolAave.sol#L417-L434.

Description: The remediation for redeemable assets report (#13) fixes one mismatch but leaves the Aave-backed Hub using the wrong availability primitive.

Original report relationship: report was correct for an over-reporting case. The Hub used `LIQUIDITY_POOL.balance(asset)` in `claimableRedeemRequest`, `maxRedeem`, and `maxWithdraw`. For the base pool, that value could exceed the amount the pool would allow through `withdraw()`, because `LiquidityPool.withdraw()` is capped by `totalDeposited`. The remediation at commit 6d19ad81 introduced `_availableAssets()` and changed the Hub to use `min(LIQUIDITY_POOL.balance(asset), LIQUIDITY_POOL.totalDeposited())`. That addresses the original over-reporting direction.

This issue is the opposite direction and is specific to Aave-backed pools. In `LiquidityPoolAave`, `balance(token)` does not mean "amount that can be withdrawn from Aave". It is a borrow-capacity view: `_balance()` computes the amount that could still be borrowed, constrained by reserve liquidity, Aave LTV, the configured minimum health factor, token-specific LTV, and current debt.

Actual withdrawals use a different path. `LiquidityPoolAave._withdrawLogic()` checks that the pool has enough aTokens, calls `AAVE_POOL.withdraw(asset, amount, to)`, and then checks health factor. With no debt, or with enough health-factor headroom, the pool can withdraw its deposited collateral even when the Aave borrow capacity is much lower than the deposited/aToken balance.

Concrete example:

1. An Aave-backed Hub has `totalDeposited = 100 USDC` and the pool holds `100 aUSDC`.
2. The Aave reserve LTV/min-health-factor calculation makes `LiquidityPoolAave.balance(USDC)` return `80 USDC` of borrow capacity.
3. The fixed Hub computes `_availableAssets() = min(80, 100) = 80`.
4. `claimableRedeemRequest`, `maxRedeem`, and `maxWithdraw` all report only `80 USDC` as available.
5. A direct `LiquidityPool.withdraw(..., 100 USDC)` can still succeed because the withdrawal path is based on aToken collateral and health factor, not borrow capacity.

Thus the fix prevents the original over-reporting bug but still causes under-reporting for Aave-backed Hubs. The Hub can advertise less liquidity than is actually withdrawable and `fulfilRedeem()` can skip or partially fulfill redeem requests even when the underlying pool can withdraw the assets.

Users with pending redeem requests can remain unfulfilled even though enough Aave collateral is withdrawable. `maxWithdraw`, `maxRedeem`, and `claimableRedeemRequest` become artificially low, causing operators and integrations to treat available liquidity as unavailable. This can delay withdrawals and leave deployable liquidity idle until Aave borrow capacity increases, even though the pool's withdrawal path could satisfy the request.

This is not a duplicate of #13. #13 was an over-reporting issue where Hub views could exceed the amount `withdraw()` accepted. This report is an incomplete-fix/regression in the other direction: after the #13 fix, the Hub can under-report because the Aave implementation's `balance()` is borrow capacity, not withdrawable liquidity.

Recommendation: Do not use `LiquidityPool.balance()` as a generic withdrawability primitive. Add a separate pool view such as `withdrawableAssets(IERC20 token)` or `availableForWithdrawal(IERC20 token)` and have the Hub use that for ERC-7540 availability views.

For the base pool, the withdrawable amount should be bounded by actual token balance and `totalDeposited`. For `LiquidityPoolAave`, it should be based on the aToken collateral that can be withdrawn while preserving the configured health-factor constraints, not on remaining borrow capacity. Add tests that cover both directions: the original #13 over-reporting case and an Aave case where borrow capacity is lower than the amount that `withdraw()` can redeem.

Sprinter Tech: Fixed in PR 275.

Cantina Managed: Fix verified.