Code Assessment
of the Basin Smart Contracts

PUBLIC

Date May 05, 2026

Produced for

By

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help GroveLabs with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Basin according to **Scope** to support you in forming an opinion on their security risks.

GroveLabs implements Basin, a liquidity pool that facilitates atomic swaps between a tokenized credit asset, its underlying collateral, and a stablecoin.

The most critical subjects covered in our audit are asset solvency, functional correctness, access control, precision of arithmetic operations and front-running. The general subjects covered are specification, gas efficiency and trustworthiness.

In summary, we find that the codebase provides a good level of security. The significant findings reported in `Version 1` were addressed in `Version 2`, which itself raised one additional medium severity correctness issue and one informational finding. The medium severity issue was addressed in `Version 3`; the informational finding was acknowledged.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,
ChainSecurity

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Basin repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Version	Date	Commit Hash	Note
1	14 April 2026	24da77561c117aa2cd2e57211faee9904b9e0568	Initial Review
2	23 April 2026	a0adf507aecd9b8d0dfa31ceb141e2f33572e3d7	After Intermediate Report
3	24 April 2026	6c0cb0b11ce569fc42de9417b3a9a3d570b8a2bb	Further Fixes
4	5 May 2026	a79269a3f5f0253110e9cbca15d79aa9ffb62c4d	Final Commit

For the Solidity smart contracts, the compiler version `0.8.24` was chosen.

```
src/
├── GroveBasin.sol
├── GroveBasinFactory.sol
├── interfaces/
│   ├── IAaveV3PoolLike.sol
│   ├── IAsyncVaultLike.sol
│   ├── IChronicleOracleLike.sol
│   ├── IERC4626VaultLike.sol
│   ├── IGroveBasin.sol
│   ├── IGroveBasinPocket.sol
│   ├── IGroveRateProvider.sol
│   ├── IPSMLike.sol
│   └── ITokenRedeemer.sol
├── pockets/
│   ├── AaveV3UsdtPocket.sol
│   ├── BasePocket.sol
│   ├── MorphoUsdtPocket.sol
│   └── UsdsUsdcPocket.sol
├── rate-providers/
│   ├── ChronicleRateProvider.sol
│   └── FixedRateProvider.sol
└── redeemers/
    ├── BUIDLTokenRedeemer.sol
    └── JTRSYTokenRedeemer.sol
```

In **Version 2**, `src/interfaces/IATokenLike.sol` was added to the scope.

2.1.1 Excluded from scope

Any files not listed above are out of scope. The following are explicitly out of scope:

- Deployment scripts in `script/` are out of scope. They are provided for reference only.
- Test files in `test/` are out of scope.
- All external protocols that the system integrates with are out of scope, including:
  - Aave V3 lending pool (used by `AaveV3UsdtPocket`)
  - Morpho ERC-4626 vaults (used by `MorphoUsdtPocket`)
  - Sky PSM (used by `UsdsUsdcPocket`)
  - Chronicle oracles (used by `ChronicleRateProvider`)
  - Centrifuge ERC-7540 async vaults (used by `JTRSYTokenRedeemer`)
  - BUIDL redemption infrastructure (used by `BUIDLTokenRedeemer`)
- Dependencies and libraries including OpenZeppelin contracts and erc20-helpers.
- The `grove-address-registry` and `xchain-ssr-oracle` library submodules.

3 System Overview

This system overview describes the latest version of the contracts as defined in the Assessment Overview.

GroveLabs offers Basin, a liquidity pool that facilitates atomic swaps between a tokenized credit asset, its underlying collateral, and a stablecoin. The protocol addresses the liquidity gap for tokenized treasury products (such as BUIDL or JTRSY) whose native redemption mechanisms are slow, gated, and require KYC. Separate basins are deployed per credit product. A single `liquidityProvider` account deposits tokens to earn yield from credit token appreciation, swap fees, and pocket yield strategies. End users can swap the credit token against either of the other two without direct issuer interaction; direct swaps between the collateral and the stablecoin are disallowed.

3.1 GroveBasin

`GroveBasin` is the core liquidity pool contract that manages three token types: a swap token (user facing stablecoin such as USDT or USDS), a collateral token (the asset credit tokens redeem to, typically USDC), and a credit token (yield bearing tokenized RWA such as JTRSY or BUIDL). The contract uses a share based accounting model where all assets are converted to a common USD value (1e18 precision) via rate providers (1e27 precision), and shares represent proportional ownership of the total pool value regardless of which asset was deposited. Shares are tracked in a custom mapping rather than as an ERC20 token; they are non transferable and only mint on deposit, burn on withdraw, or accrue to the fee claimer on swaps involving credit tokens.

Swaps are priced using three independent rate providers, one per token. The contract enforces staleness checks on rates before executing swaps, reverting if the rate age exceeds the configured threshold. A maximum swap size limit, enforced against the swap's 1e18 USD value (not the raw token amount), prevents large swaps that could cause excessive slippage. Fees are charged on swaps involving credit tokens: a purchase fee when buying credit tokens and a redemption fee when selling them. Fee revenue is minted as new shares to a designated fee claimer address, diluting existing shareholders proportionally. If the fee claimer is unset (`address(0)`), `_accrueFeeShares` returns silently without minting, and the fee value remains in the pool, accruing to existing shareholders via share price increase.

The contract delegates swap token custody to a configurable pocket contract, an external adapter that either deploys the swap token to yield (Aave, Morpho) or converts it to a different stablecoin (Sky PSM). Credit token redemptions are handled through registered redeemer contracts that abstract over different redemption mechanisms (direct transfer for BUIDL, ERC-7540 async vaults for JTRSY). The redemption flow has two phases driven by `REDEEMER_ROLE`. `initiateRedeem` stores a `RedeemRequest` (block number, redeemer, credit amount, expected collateral) keyed by `keccak256(abi.encode(request))`, increments a `pendingRedemptions` counter for the target redeemer, and forwards credit tokens to the redeemer. `completeRedeem` resolves a request by id, calls back into the redeemer to retrieve collateral, and clears the entry. A nonzero `pendingRedemptions` count blocks `removeTokenRedeemer` so that a redeemer cannot be unregistered while requests are in flight.

The contract implements granular pause functionality with three levels: global pause via `bytes4(0)`, function level pause via selector, and directional swap pauses via `PAUSED_SWAP_CREDIT_TO_COLLATERAL`, `PAUSED_SWAP_CREDIT_TO_SWAP`, `PAUSED_SWAP_COLLATERAL_TO_CREDIT`, and `PAUSED_SWAP_SWAP_TO_CREDIT` (the swap and collateral token direct pairs revert with `InvalidSwap` regardless of pause state). In addition, credit token deposits and withdrawals can be paused via the dedicated keys `PAUSED_DEPOSIT_CREDIT` and `PAUSED_WITHDRAW_CREDIT`. Withdrawals of the swap and collateral tokens are intentionally unpausable to ensure users can always exit. Pausing is split across two roles: `PAUSER_ROLE` can only set keys to paused via `setPaused`, while `MANAGER_ADMIN_ROLE` is the sole role that can clear a pause via `setUnpaused`.

The contract provides the following administrative functions. `setMaxSwapSizeBounds` and `setStalenessThresholdBounds` automatically clamp the currently configured value into the new range if it

falls outside. `setFeeBounds` instead reverts with `CurrentFeeOutOfNewBounds` if either the purchase or redemption fee is outside the new range, requiring `OWNER_ROLE` to move the fees into range first.

- `setRateProvider()`: Callable by `MANAGER_ADMIN_ROLE`. Replaces a token's rate provider. Validates the new provider returns a nonzero rate.
- `setPocket()`: Callable by `MANAGER_ADMIN_ROLE`. Withdraws all funds from the current pocket's yield strategy and transfers the entire swap token balance to the new pocket.
- `addTokenRedeemer()` / `removeTokenRedeemer()`: Callable by `MANAGER_ADMIN_ROLE`. Registers or removes redeemer contracts (granting or revoking `REDEEMER_CONTRACT_ROLE`). Removal fails if the redeemer has pending redemptions.
- `setFeeClaimer()`: Callable by `MANAGER_ADMIN_ROLE`. Sets the address that receives fee revenue minted as shares.
- `setMaxSwapSizeBounds()`: Callable by `MANAGER_ADMIN_ROLE`. Sets the lower and upper bounds within which `MANAGER_ROLE` can adjust `maxSwapSize`.
- `setStalenessThresholdBounds()`: Callable by `MANAGER_ADMIN_ROLE`. Sets the lower and upper bounds within which `MANAGER_ROLE` can adjust `stalenessThreshold`.
- `setFeeBounds()`: Callable by `MANAGER_ADMIN_ROLE`. Adjusts the allowable fee range. Reverts if the current purchase or redemption fee falls outside the new bounds.
- `setMaxSwapSize()`: Callable by `MANAGER_ROLE`. Adjusts the swap size limit within governance defined bounds.
- `setStalenessThreshold()`: Callable by `MANAGER_ROLE`. Adjusts rate staleness tolerance within bounds.
- `setPurchaseFee()` / `setRedemptionFee()`: Callable by `OWNER_ROLE`. Sets fees within the configured min/max bounds.
- `setPaused(key)`: Callable by `PAUSER_ROLE`. Sets a pause key (global `bytes4(0)`, function selector, or asset specific key) to paused.
- `setUnpaused(key)`: Callable by `MANAGER_ADMIN_ROLE`. Clears a pause key.
- `revokeRole(role, account)`: Standard `AccessControl` function, overridden so that `PAUSER_ROLE` holders can revoke `MANAGER_ROLE` or `REDEEMER_ROLE` from any account in addition to the usual admin path.

The contract provides the following user and LP functions:

- `deposit()`: Callable only by the immutable `liquidityProvider` address. Deposits any of the three tokens and mints shares to a specified receiver.
- `withdraw()`: Callable by any shareholder. Burns shares and withdraws a specified asset, capped by the basin's available balance for that asset (for the swap token this delegates to the pocket's `availableBalance`). Withdrawals of the swap and collateral tokens cannot be paused; credit token withdrawals are subject to `PAUSED_WITHDRAW_CREDIT`.
- `swapExactIn()` / `swapExactOut()`: Callable by anyone. Swap between the credit token and either the swap or collateral token at rate provider prices; direct conversions between the swap and collateral tokens revert with `InvalidSwap`. Credit token legs incur a purchase or redemption fee. Subject to pause, staleness, and max swap size checks.
- `initiateRedeem()`: Callable by `REDEEMER_ROLE`. Sends credit tokens to a registered redeemer to begin async redemption.
- `completeRedeem()`: Callable by `REDEEMER_ROLE`. Finalizes a pending redemption and receives collateral tokens.

3.2 GroveBasinFactory

`GroveBasinFactory` is a deployment helper that creates new `GroveBasin` instances with mandatory first depositor protection. The factory pulls one unit of the swap token from the deployer, deploys the basin, and immediately calls `depositInitial` to seed the pool with shares credited to `address(0)`. This prevents the share inflation attack where an attacker could donate tokens directly to an empty pool, causing subsequent depositors to receive zero shares.

The factory performs minimal configuration. After deployment, the basin has default values (`pocket` set to the basin's own address so swap tokens are held in place with no external yield strategy, no redeemers, no fee claimer, zero fees) and requires additional setup via the owner or manager admin roles.

- `deploy()`: Callable by anyone with sufficient swap token balance. Deploys a new basin via `CREATE2` and performs the seed deposit atomically. An overload accepts an explicit salt; the no-salt variant derives the salt from the owner and the three token addresses.

3.3 Pockets

Each `GroveBasin` has exactly one active pocket at a time. The pocket implementations below are alternatives; `MANAGER_ADMIN_ROLE` switches between them via `setPocket`, which unwinds the old pocket's yield position and moves the swap token balance to the new pocket.

Pockets are external custody contracts that hold swap tokens and can deploy them to yield generating strategies. On the standard flow only the swap token routes through the pocket; collateral and credit tokens stay in the basin. `UsdsUsdcPocket` is the exception: it can also transiently hold the collateral token (USDC) as a result of its PSM conversions, see below. The `BasePocket` abstract contract provides the authorization framework, restricting calls to either the basin itself or addresses holding `MANAGER_ROLE` on the basin.

During migration, `setPocket` reads the old pocket's ERC-20 `balanceOf` after calling `withdrawLiquidity` on it, so any portion the yield venue fails to return synchronously is not carried over. Migrated funds then sit idle in the new pocket until the next `depositLiquidity` call redeploys them. External pockets are expected to keep a standing allowance for the Basin on the swap token: `setPocket` relies on it during migration, and ordinary swap or withdraw paths rely on it in `_pushAsset` to forward swap tokens to the receiver.

`AaveV3UsdtPocket` deposits USDT to Aave V3's lending pool via `supply()` and withdraws via `withdraw()`. Available balance is computed as wallet USDT plus the aToken balance converted at 1:1.

`MorphoUsdtPocket` deposits USDT to a Morpho ERC-4626 vault. Available balance includes wallet USDT plus vault shares converted via `convertToAssets()`.

`UsdsUsdcPocket` converts between USDC and USDS via Sky's PSM (Peg Stability Module). On deposit and withdraw it moves USDS to and from the pocket. On swaps, it allows basins to tap into the USDC liquidity of the PSM if the USDC balance in the basin is insufficient; USDC withdrawals revert with `NonZeroPsmTout` when the PSM charges a nonzero fee on buying USDC, and any USDC produced through the PSM is pushed directly to the basin at the end of the same call. This pocket allows offering additional USDC liquidity that is sourced from the held USDS liquidity. Optimally, the manager would ensure that no USDC ever lives within the contract as it is not accounted for; a `sweep()` function is provided for `MANAGER_ROLE` or the basin to flush any residual USDC balance to the basin. The constructor additionally validates that the pocket is wired to the expected basin by checking that the basin's swap and collateral tokens match the configured USDS and USDC addresses. If `groveProxy` is not `address(0)` at construction, the pocket additionally grants that address an unlimited USDS allowance so Sky governance spells can withdraw USDS directly from the pocket.

All pockets implement:

- `depositLiquidity()`: Callable by basin or manager. Deploys to strategy.
- `withdrawLiquidity()`: Callable by basin or manager. Withdraws from strategy.

- `availableBalance()`: View function returning currently accessible balance.

3.4 Rate Providers

Rate providers supply token to USD conversion rates in 1e27 precision with associated timestamps for staleness checking.

`ChronicleRateProvider` wraps a Chronicle oracle, calling `readWithAge()` to fetch the rate and its last update time. It scales the Chronicle 1e18 precision rate to 1e27.

`FixedRateProvider` returns an immutable rate set at construction and always reports the current block timestamp as the age, effectively never going stale. Used for stablecoins assumed to maintain perfect peg.

Both implement:

- `getConversionRate()`: Returns the current rate.
- `getConversionRateWithAge()`: Returns rate and last update timestamp.

3.5 Redeemers

Redeemers abstract credit token redemption mechanics, allowing the basin to support different tokenized RWA products with varying redemption flows.

`BUIDLTokenRedeemer` handles direct redemptions where credit tokens are transferred to a designated redemption address (the issuer's vault). Only one redemption may be in flight at a time: `initiateRedeem` sets a `redemptionActive` flag and reverts if it is already set. The issuer processes the redemption off chain and sends collateral back to the redeemer contract. `completeRedeem` transfers the redeemer's entire collateral token balance to the basin, reverting with `NoCollateralBalance` if the balance is zero, and then clears the `redemptionActive` flag. This model relies on the issuer honoring redemptions with no on chain guarantee.

`JTRSYTokenRedeemer` handles ERC-7540 async vault redemptions. Only one redemption may be in flight at a time: `initiateRedeem` sets a `redemptionActive` flag and reverts with `RedemptionAlreadyActive` if it is already set, then calls `requestRedeem()` on the Centrifuge vault to queue the redemption. `completeRedeem` calls `redeem()` to claim processed collateral and clears the flag. The constructor validates that the vault's share token matches the basin's credit token (`ShareMismatch`) and that the vault's asset matches the basin's collateral token.

Both implement:

- `initiateRedeem()`: Callable only by basin. Transfers credit tokens and begins redemption.
- `completeRedeem()`: Callable only by basin. Claims collateral and returns it to basin.
- `setUp()` / `tearDown()`: Lifecycle hooks called when redeemer is added to or removed from basin.
- `sweep(token, amount)`: Callable by `MANAGER_ADMIN_ROLE` on the basin. Transfers the specified amount of the credit or collateral token from the redeemer back to the basin. Intended to be called only when no redemption is in flight.

4 Trust Model

OWNER_ROLE: Fully trusted. Holds `DEFAULT_ADMIN_ROLE` and can grant or revoke all other roles. Can set purchase and redemption fees within bounds. Can steal value by setting near-maximum fees (capped strictly below `BPS` by `setFeeBounds`) or by granting malicious actors administrative roles. The owner is expected to be a governance multisig or timelock. Expected to be the issuer multisig behind a timelock where GroveLabs holds cancellation rights.

MANAGER_ADMIN_ROLE: Highly trusted. Can set rate providers, pocket, redeemers, fee bounds, staleness bounds, swap size bounds, and fee claimer. Also the sole role that can clear pause flags via `setUnpaused`, and the sole role that can sweep stranded balances from registered redeemers via their `sweep` function. Attack vectors include: (1) setting a malicious rate provider that returns manipulated prices, enabling arbitrage theft of pool value, (2) setting a malicious pocket that does not return funds on withdrawal, (3) adding a malicious redeemer that steals credit tokens, (4) setting fee bounds to allow near-100% fees, (5) setting an attacker-controlled fee claimer to capture all fee revenue. Expected to be the GroveLabs governance proxy.

MANAGER_ROLE: Partially trusted. Can adjust max swap size and staleness threshold within governance-defined bounds. Limited attack surface as all changes are bounded. Could temporarily set very high staleness threshold to allow swaps with outdated prices, or very low max swap size to grief users. Expected to be a GroveLabs multisig.

PAUSER_ROLE: Limited trust. Can only set pause flags; clearing a pause requires `MANAGER_ADMIN_ROLE` via `setUnpaused`. Cannot pause withdrawals of the swap or collateral tokens by design. Credit-token withdrawals and deposits are pausable via `PAUSED_WITHDRAW_CREDIT` and `PAUSED_DEPOSIT_CREDIT`, and `completeRedeem` is also pausable. Worst case is griefing via pausing swaps, deposits, credit-token withdrawals, and redemption completion, but users can always exit via the swap or collateral tokens.

REDEEMER_ROLE: Trusted for redemption operations. Can initiate and complete redemptions. Could initiate redemptions at unfavorable times or fail to complete them, but cannot extract value beyond normal redemption flows.

liquidityProvider: Immutable address set at construction. Only account permitted to deposit. Fully trusted to manage LP capital. Cannot be changed after deployment.

End Users: Untrusted. Can swap and withdraw their own shares. Protected by slippage parameters (`minAmountOut`, `maxAmountIn`) and cannot affect other users beyond normal pool interactions.

External Dependencies:

The system relies on Chronicle oracles for credit token and potentially swap token pricing. Oracle manipulation or failure would allow swaps at incorrect prices. The staleness check provides limited protection (default 1 week) but a sophisticated attack during the valid window could drain value. Fixed rate providers for stablecoins assume perfect peg, any depeg would create arbitrage opportunities. Due to Proof-of-Asset oracles not implementing the `decimals` function, we assume that any Chronicle oracle uses 18 decimals for their answers (see their FAQ).

Pockets delegate funds to external protocols (Aave V3, Morpho vaults, Sky PSM). Failures, exploits, or pauses in these protocols could lock or lose swap token funds. The basin has no direct visibility into pocket health beyond `availableBalance()`.

Redeemers depend on credit token issuers honoring redemptions. For BUIDL, this is entirely off-chain trust. For JTRSY, the Centrifuge vault provides on-chain tracking but ultimate settlement depends on the issuer.

Upgradeability: The core `GroveBasin` contract is not upgradeable. However, rate providers, pockets, and redeemers can be swapped by `MANAGER_ADMIN_ROLE`, providing indirect mutability of critical components.

5 System Considerations

We leverage this section to highlight findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require a modification inside the project. Instead, they should raise awareness in order to improve the overall understanding for users and developers.

SC1 `maxSwapSize` Caps Individual Swaps, Not Cumulative Flow

System Consideration | Version 1

Each swap path (`swapExactIn`, `swapExactOut`, and their previews) checks `_getAssetValue(assetIn, amountIn, false) > maxSwapSize` in isolation. There is no bookkeeping of swap volume across calls, so `maxSwapSize` acts as a per-call upper bound only, not as a rate limit over any time window.

A caller can therefore submit N back-to-back swaps of size `maxSwapSize` each, within the same block or transaction, and move `N * maxSwapSize` of value through the pool without ever tripping the check. Any reasoning that treats `maxSwapSize` as a flow limit does not hold.

SC2 `PAUSED_WITHDRAW_CREDIT` Does Not Halt Share Redemption, Only the Credit Token Exit Lane

System Consideration | Version 1

A share represents a proportional claim on the pool's USD value, redeemable via `withdraw` against any of `creditToken`, `collateralToken`, or `swapToken`. Only withdrawals of `creditToken` are gated: `previewWithdraw` calls `_checkPaused(PAUSED_WITHDRAW_CREDIT)` when `asset == creditToken`, which also covers the global pause, since `_checkPaused` reverts on either the global key `paused[bytes4(0)]` or the supplied key. Withdrawals of `collateralToken` or `swapToken` take no pause path on either the preview or the execute side, and always execute.

With multiple shareholders present, `PAUSED_WITHDRAW_CREDIT` or the global pause creates a race: shareholders willing to exit through `collateralToken` or `swapToken` drain those balances first, so while the pause is active not all shares can be redeemed.

SC4 Chronicle Toll Access Control Bypass

System Consideration | Version 1

Chronicle oracles gate `read` and `readWithAge` behind a toll whitelist, restricting on-chain consumers to authorized tollers. `ChronicleRateProvider.getConversionRate` and `getConversionRateWithAge` are `external view` with no access control of their own, so once the rate provider is tolled by Chronicle, any address can read the oracle's value through it. This defeats Chronicle's on-chain access model for every feed wrapped by this contract.

SC5 External Venues May Block Pocket Deposits and Withdrawals

System Consideration | Version 1

Each pocket forwards to an external venue (Aave, a Morpho vault, or the PSM), and any of them can revert in either direction: withdrawals fail when the venue is out of liquidity (Aave/Morpho at 100% utilization, PSM

USDC reserve drained), and deposits fail when the venue is at capacity (Aave supply cap, Morpho `maxDeposit`, PSM out of USDS for `sellGem`). Any Basin flow that touches the pocket then stalls until the venue recovers.

As of **Version 2**, the deposits are wrapped in a `try` / `catch` block so that deposits are not DoSed by external system limitations.

SC6 Liquidity Provider Can Bypass Swap Fees via Deposit + Withdraw

System Consideration | Version 1

`deposit` is gated to the `liquidityProvider`, while `withdraw` is open to any shareholder. In combination, this path allows the liquidity provider (or any shareholder it seeds) to move value between the three configured assets without paying `purchaseFee` or `redemptionFee`: deposit asset A to mint shares, then withdraw asset B by burning those shares. The pool's rate-provider-priced share math performs the conversion.

SC7 Output-Asset Liquidity Can Be Drained to DoS Subsequent Swaps

System Consideration | Version 1

Any caller can swap out the entire pool balance of an output asset. Subsequent swaps into that asset then fail until the balance is replenished through liquidity-provider deposits or opposite-direction swaps. `maxSwapSize` bounds only individual calls and not cumulative flow, so the drain can be assembled from multiple back-to-back swaps within a single transaction.

The attack is bounded by cost: each swap pays `purchaseFee` or `redemptionFee`, which is accrued as fee shares for the fee claimer. A griefer therefore pays the configured swap fee (plus gas) on every unit of liquidity they remove, so this DoS lasts only as long as the griefer is willing to keep burning value on a pool that collects fees in return.

SC8 Pending Redemption Value Can Be Under- or Overestimated

System Consideration | Version 1

The estimated value of requested redemptions may be under- or overestimated, leading to unfair pricing for depositors, withdrawers and the `feeClaimer`. Depending on the scenario, the result can be profitable or loss-making for respective parties.

`GroveBasin` supports both rebasing and yield-accruing credit tokens. The balance physically held by Basin is valued at its live `balanceOf`, so any rebase accrues naturally into `totalAssets()`. Similarly, any rate change is applied. Credit tokens that have been handed to an `ITokenRedeemer` for an in-flight redemption are accounted differently: `_initiateRedeem` snapshots the nominal `creditTokenAmount` transferred out and records it in `pendingCreditTokenBalance`:

```
pendingCreditTokenBalance       += creditTokenAmount;

IERC20(creditToken).approve(redeemer, creditTokenAmount);
ITokenRedeemer(redeemer).initiateRedeem(creditTokenAmount);
```

`totalAssets()` then values the pending portion at that snapshotted nominal:

```
return _getAssetValue(
    // ...
```

```
    +  _getAssetValue(
           creditToken,
           _getAvailableBalance(address(creditToken)) + pendingCreditTokenBalance,
           false
       );
```

The snapshotted value is what Basin expects the redeemer to settle against on `completeRedeem`, and it is what the position is worth to Basin between `initiateRedeem` and `completeRedeem`. This is the intended accounting, but it also means that any rebase or rate change on the credit token that occurs while a redemption is pending is not reflected in `totalAssets()`: depending on how the underlying credit-token value moves, the pending portion can be over- or underestimated.

Note that once `completeRedeem` executes, profits and losses are realized accordingly:

- If the received amount was higher than expected: shares could have been bought cheaply before completion, and the fee claimer would have been overpaid.
- If the received amount was lower than expected: shares could have been preemptively withdrawn to escape the loss, concentrating the penalty on the remaining shareholders.

SC9 Redemption Completion Silently Accepts Partial Collateral

System Consideration | Version 1

While Pending Redemption Value Can Be Under- or Overestimated covers how the snapshotted value of an in-flight redemption can drift, the completion path itself can introduce additional mispricing.

`GroveBasin._completeRedeem` is final: it deletes the `RedeemRequest`, decrements `pendingCreditTokenBalance` by the full `creditTokenAmount`, and delegates to `ITokenRedeemer.completeRedeem(request)`, accepting whatever the redeemer returns:

```
delete redeemRequests[redeemRequestId];
pendingCreditTokenBalance -= request.creditTokenAmount;
pendingRedemptions[request.redeemer]--;

uint256 collateralTokenReturned = ITokenRedeemer(request.redeemer).completeRedeem(request);
```

Redeemer contracts are authorized via `REDEEMER_CONTRACT_ROLE` and implement arbitrary redemption logic, so the actual collateral amount returned on completion is fully at the redeemer's discretion. The redemption path silently accepts partial or zero collateral:

- `RedeemCompleted` is emitted with only the amount actually returned. It never compares that amount to the request's recorded `collateralTokenAmount`, so the event stream does not directly expose the loss; observers must reconstruct it from the original request.
- The external `completeRedeem` is gated only by `REDEEMER_ROLE`; there is no on-chain check that the off-chain or vault-side settlement has delivered the full quoted `collateralTokenAmount` before the call proceeds.

Whether calling `completeRedeem` is safe therefore depends entirely on the per-redeemer implementation.

`JTRSYTokenRedeemer` routes through the ERC-7540 vault:

```
collateralTokenReturned = IAsyncVaultLike(vault).redeem(request.creditTokenAmount,
    address(this), address(this));
```

If the full `creditTokenAmount` is not yet claimable, `vault.redeem` reverts and the transaction fails. The vault itself acts as the guard against premature completion.

`BUIDLTokenRedeemer` has no equivalent guard:

```
collateralTokenReturned = Math.min(request.collateralTokenAmount,
    IERC20(collateralToken).balanceOf(address(this)));
IERC20(collateralToken).safeTransfer(address(basin), collateralTokenReturned);
```

If `REDEEMER_ROLE` calls `completeRedeem` before the off-chain redemption has been fully paid out, the returned amount can be significantly below the expected value. The redeemer transfers only the available balance to Basin, and `_completeRedeem` then deletes the `RedeemRequest` and clears `pendingCreditTokenBalance` by the full `creditTokenAmount`. From Basin's accounting the redemption is now settled. Any collateral that arrives at `BUIDLTokenRedeemer` afterwards has no matching on-chain request to settle against and sits at the redeemer contract. The correctness of the BUIDL leg therefore depends on `REDEEMER_ROLE` waiting for the off-chain settlement to complete before calling `completeRedeem`; the contract enforces no such ordering.

This is intentional design flexibility and outside the scope of the core contract's invariants, but governance must be aware that a misconfigured or hostile redeemer can systematically return less collateral than the credit-token value warrants, producing a one-way leak of value from existing LPs.

As of **Version 2**, the BUIDL redeemer exposes a `sweep` that is unguarded against `redemptionActive`; misusing it during an active redemption introduces its own accounting and completion hazards. See Sweeping the BUIDL Redeemer During an Active Redemption Corrupts Share Price and Blocks Completion.

SC10 Rate-Provider Updates Can Be Sandwiched on Swaps

System Consideration **Version 1**

Asset values are computed from external rate-provider contracts, so swap pricing moves in discrete steps whenever a rate provider updates. A caller who sees or anticipates a pending rate update can sandwich it by swapping into the soon-to-be-repriced asset using the old rate, waiting for the update to land, and swapping back out at the new rate, pocketing the delta at the expense of existing LPs. The `stalenessThreshold` check caps how old an accepted rate can be and therefore bounds the window in which a stale-but-still-accepted rate can be exploited, but it does not protect against sandwiching a rate update that lands within the freshness window.

The same sandwich pattern applies to deposits and withdrawals, which are priced through the same rate providers.

SC11 Swap Previews Do Not Account for Available Output Liquidity

System Consideration **Version 1**

`previewSwapExactIn` and `previewSwapExactOut` compute the output (or required input) purely from the configured rate providers and fees. Neither compares the resulting `amountOut` against `_getAvailableBalance(assetOut)` or the pocket's withdrawable balance. A preview can therefore return a nonzero, valid-looking quote even when the pool does not hold enough of `assetOut` to actually settle the swap, in which case the underlying `swapExactIn` / `swapExactOut` call reverts during the pocket withdrawal or the final `_pushAsset`.

Integrators that rely on a successful preview as a gate on their transaction need an independent liquidity check, since the preview itself does not guarantee executability.

SC12 A Partially-Fulfilled or Cancelled JTRSY Redemption Cannot Be Settled on-Chain

System Consideration | Version 2

`JTRSYTokenRedeemer.completeRedeem` calls `vault.redeem(request.creditTokenAmount, ...)` with the full nominal amount recorded at `initiateRedeem` time. If `creditTokenAmount` is not redeemable due to unexpected reasons such as forced cancellations or similar, the vault's `maxRedeem()` never rises to `creditTokenAmount` and `vault.redeem` always reverts. `GroveBasin` exposes no other path that can decrement `pendingCreditTokenBalance` or `pendingRedemptions[redeemer]`, so the request stays on the books indefinitely, `totalAssets()` keeps valuing the pending portion at the full snapshotted nominal, and `removeTokenRedeemer` is permanently blocked by the pending-request counter.

Operational recovery depends entirely on off-contract mechanisms at Centrifuge. Governance must rely on the endorsed-operator path to route the cancelled or partially-settled proceeds (along with any unredeemed shares) back to the `JTRSYTokenRedeemer` address, then use `sweep` to move the recovered balances into Basin. `sweep` moves value but does not reconcile the `RedeemRequest`: the swept collateral lands in `_getAvailableBalance(collateralToken)` while `pendingCreditTokenBalance` still carries the full request nominal, so the same position is counted in both places until the stale request is cleared, and the contract exposes no on-chain path to clear it.

This scenario is considered unlikely in practice, but when it does occur, any partial or cancelled fulfillment on the Centrifuge side should be treated as an operational incident requiring coordinated off-chain remediation rather than an in-protocol recovery. The redeemer cannot be decommissioned through `removeTokenRedeemer` while such a request is outstanding, and a full on-chain resolution (restoring Basin's accounting and freeing the redeemer slot) would ultimately require migrating to a new Basin instance.

SC13 Sweeping the BUIDL Redeemer During an Active Redemption Corrupts Share Price and Blocks Completion

System Consideration | Version 2

`BUIDLTokenRedeemer.sweep` is callable by `MANAGER_ADMIN_ROLE` and does not check `redemptionActive`. Once the off-chain settlement delivers the collateral to the redeemer, sweeping before `completeRedeem` transfers that collateral straight to Basin while the live `RedeemRequest` is still on the books. `totalAssets()` then counts the swept amount twice (once through `_getAvailableBalance(collateralToken)`, once through `pendingCreditTokenBalance`), and the share price is inflated for as long as the request remains open.

Completing the request afterwards is also no longer possible. `BUIDLTokenRedeemer.completeRedeem` reads `balanceOf(address(this))` and reverts with `NoCollateralBalance` when the balance is zero, so the request is stuck until governance (or anyone) sends a non-zero amount of collateral back to the redeemer to reset it. In the meantime, `pendingRedemptions[redeemer]` remains positive, which in turn blocks `removeTokenRedeemer`.

Governance should never sweep the redeemer unnecessarily while a redemption is active. When a malicious, partial, or otherwise abnormal settlement is detected (see Redemption Completion Silently Accepts Partial Collateral), the following recovery order should be followed:

1. Pause the relevant functionality.
2. Wait for full payment to reach the redeemer.
3. Sweep the recovered funds to Basin.

4. Unpause.

6 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- **Likelihood** represents the likelihood of a finding to be triggered or exploited in practice
- **Impact** specifies the technical and business-related consequences of a finding
- **Severity** is derived based on the likelihood and the impact

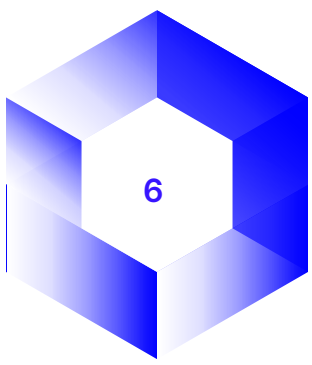
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	**Impact**		
	High	Medium	Low
High	**Critical**	**High**	**Medium**
Medium	**High**	**Medium**	**Low**
Low	**Medium**	**Low**	**Low**

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

7 Findings

In this section, we describe the findings identified during the course of the engagement. The findings are categorized by severity as explained in the Terminology section. Below we provide an overview of the total number of findings per severity, as well as the number of open issues.



Open Issues

Total number of findings

Critical | High | Medium | Low | Informational

7.1 Overview

The following table lists all identified findings along with their severity and current status. A finding is considered resolved once it has been fully corrected or the specification has been changed accordingly. Findings with any other status, including partial fixes, acknowledgements, and accepted risks, remain open.

Medium	#001	`GroveBasinFactory.deploy` Can Be Griefed by Donating Dust to the Predicted Address	Code Corrected
Medium	#020	A JTRSY Redemption Request Can Be Left Permanently Unredeemable After a Batched Centrifuge Fulfillment	Code Corrected
Low	#002	`_accrueFeeShares` Rounds Against LPs yet May Still Underpay the Fee Claimer	Code Corrected
Low	#003	`BUIDLTokenRedeemer` Has No Path to Recover Stranded Collateral	Code Corrected
Low	#004	`setFeeBounds` Lets the Manager Admin Override Owner-Only Fee Settings	Code Corrected
Low	#005	`UsdsUsdcPocket` Accounting Breaks if the PSM Ever Charges a Fee	Code Corrected
Low	#006	Pocket/Basin Integration Lacks Sanity Checks on Expected Tokens	Code Corrected
Low	#007	README Does Not Reflect the Current Codebase	Specification Changed
Low	#008	USDC Potentially Unaccounted at `UsdsUsdcPocket`	Code Corrected
Low	#009	Zero-Amount Redeem Request Permanently Blocks Redeemer Removal	Code Corrected
Info	#010	100% Fee Disables `swapExactOut`	Code Corrected
Info	#011	`maxSwapSize` Check Rounds the Swap Value Down, Letting Swaps Marginally Exceed the Cap	Code Corrected
Info	#012	`redeemRequestId` Derivation Collides on Identical Same-Block Redemptions	Acknowledged
Info	#013	Approval Calls Bypass or Duplicate `SafeERC20`'s Logic	Code Corrected
Info	#014	Constructor Does Not Emit Events for Initialized State	Acknowledged
Info	#015	Gas Optimizations	Partially Corrected / Acknowledged
Info	#016	Inconsistencies Across `GroveBasin`	Partially Corrected / Acknowledged
Info	#017	Lack of Sanity Checks	Code Corrected
Info	#018	Pause Design	Code Corrected

Info	#019	Wei-Scale Rounding Dust in Pocket Accounting	Risk Accepted	
Info	#021	`GroveBasinFactory.deploy` Can Still Be Griefed via Mempool Frontrunning	Acknowledged	
Info	#022	Confusing Revert Reasons in User Facing Flows	Code Corrected	

7.2 Descriptions

#001 `GroveBasinFactory.deploy` Can Be Griefed by Donating Dust to the Predicted Address

Security	Medium	Version 1	Code Corrected

Factory deployments can be DoSed through donations to pending contract addresses.

The `GroveBasin` contract addresses created are predictable from `(factoryAddress, factoryNonce)` due to `CREATE` being used. Donating 1 wei of any of the three tokens (swap, collateral, or credit) to the predicted address leads to reverts within `depositInitial`. Consider the following scenario:

1. Alice reads the factory's current nonce and computes the next predicted address.
2. Alice sends 1 wei of one or more candidate tokens to that address. `totalAssets` becomes nonzero as long as any of them is the swap, collateral, or credit token passed to `deploy`.
3. Every call to `deploy` now deploys the contract and invokes `depositInitial`.
4. `depositInitial` reverts due to `convertToShares` and thus `previewDeposit` returning 0 when `totalAssets > 0 && totalShares == 0` holds.

Hence, deployments can be DoSed. The deployer must keep deploying with token trios that exclude every donated token until the next predicted address has none.

Ultimately, deployments can be DoSed and the deployer can be griefed.

Code Corrected

`GroveBasinFactory` now deploys via `CREATE2`. The default `deploy(...)` overload derives its salt from `keccak256(abi.encode(owner, swapToken, collateralToken, creditToken))`, and a second overload accepts an arbitrary `salt`. Since the basin address no longer depends on the factory's nonce, an attacker can no longer precompute a single address and grief every subsequent deployment by funding it.

A residual mempool frontrunning grief vector still exists and is reported separately as a v2 informational finding.

#020 A JTRSY Redemption Request Can Be Left Permanently Unredeemable After a Batched Centrifuge Fulfillment

Correctness	Medium	Version 2	Code Corrected

`JTRSYTokenRedeemer.initiateRedeem` passes `address(this)` as the controller on every `vault.requestRedeem`, so multiple concurrent basin requests accumulate under a single controller slot at Centrifuge's `AsyncRequestManager`. When the hub fulfills the aggregate batch at a single weighted-average `redeemPrice`, integer quantization on `(fulfilledAssets, fulfilledShares)` shaves a few wei off the true NAV, and each subsequent `vault.redeem(X_i)` consumes `maxWithdraw` through a ceiling division, leaving slightly less than a proportional share for the remaining requests. The last basin request in the batch therefore finds `maxRedeem() < creditTokenAmount` (typically by one wei) and `completeRedeem` reverts with `AsyncRequestManager.ExceedsMaxRedeem`.

Observed on a mainnet fork against the live JTRSY vault:

```
initiateRedeem(x1 = 100_000_000_000)      // 100k JTRSY (6 dec)
initiateRedeem(x2 = 250_000_000_000)      // 250k JTRSY
// hub fulfills 350_000_000_000 shares for 383_354_406_327 USDC
```

```
completeRedeem(request1)                    // ok; Basin receives 109_529_830_379 USDC
completeRedeem(request2)                    // reverts: maxRedeem() == 249_999_999_999 <
    250_000_000_000
```

Funds are not at risk (they remain in the Centrifuge pool escrow), but the tail request is stuck until a later hub fulfillment lifts the weighted-average price enough to cover the dust. `sweep` does not help here, since the shortfall is a missing `maxRedeem` allowance at Centrifuge rather than collateral at the redeemer. Submitting a follow-up `initiateRedeem` for the shortfall amount can unstick earlier requests but shifts the same dust onto whichever request becomes the new tail.

> **Code Corrected**
>
> `JTRSYTokenRedeemer` now enforces a single active redemption at a time: `initiateRedeem` reverts with `RedemptionAlreadyActive` while a prior request is in flight, and the flag is only cleared on `completeRedeem`. Concurrent basin requests can therefore no longer accumulate under a single Centrifuge controller slot, removing the precondition for the weighted-average rounding that caused the tail request to fall one wei short.

#002 `_accrueFeeShares` Rounds Against LPs yet May Still Underpay the Fee Claimer

Correctness	Low	Version 1	Code Corrected

`_accrueFeeShares` rounds up both when converting the fee asset to USD value and when converting that value to shares. Both roundings favor the fee claimer, so the minted share count is slightly larger than the fair value of the accrued fee.

As a result, every fee accrual marginally dilutes existing LPs in favor of the fee claimer. In contrast, the fee claimer is not guaranteed to receive the full fees as part of the withdrawal.

Consider the following example regarding LP share dilution:

1. Assume that the fee is set to 1 bps.
2. Assume that `totalAssets() == 10e18` and that `totalShares == 1e18`.
3. A `swapExactIn` is called with an `amountIn` small enough that `grossOut == 1`.
4. The fee paid will be 1 due to rounding up of fees. Thus, `amountOut == 0`.
5. `_accrueFeeShares` will give 1 wei of shares to the fee claimer due to rounding up.
6. The fee claimer calls withdraw for infinite `maxAssetsToWithdraw` and redeems the share via `convertToAssetValue(1) == (10e18 + 1) / (1e18 + 1) == 9` (assuming a 1:1 asset-to-value conversion), receiving 9 wei for a 1-wei fee. The remaining shareholders have paid 8 to the fee claimer.

Note that the above is the case if the withdraw token has 18 decimals. If the withdraw token, for example, had only 6 decimals, the return would be 0.

Thus, to summarize:

- Rounding the fee up is fair for all participants and favors the system over the swapper.
- Rounding the minted shares up is unfair to existing LPs.
- The rounding when withdrawing fee shares might still lead to the fee claimer being underpaid.

Ultimately, the fee payments are inconsistent, unfair, and do not mint shares symmetrically to how `deposit` mints them.

Code Corrected

The fee share minting now rounds down to protect existing shareholders from dilution.

#003 `BUIDLTokenRedeemer` Has No Path to Recover Stranded Collateral

Design	Low	Version 1	Code Corrected

`BUIDLTokenRedeemer.completeRedeem` returns at most the quoted `request.collateralTokenAmount` to Basin:

```
collateralTokenReturned = Math.min(request.collateralTokenAmount,
    IERC20(collateralToken).balanceOf(address(this)));
IERC20(collateralToken).safeTransfer(address(basin), collateralTokenReturned);
```

Any collateral held by the redeemer beyond that cap stays at the contract. The rest of the interface is `initiateRedeem`, `completeRedeem`, and no-op `setUp` / `tearDown`; the contract has no sweep, rescue, or skim function, and neither does Basin, so there is no way to move stranded collateral out of the redeemer once it has landed there.

If `completeRedeem` is called before the off-chain redemption has paid out the full quoted `collateralTokenAmount`, the redeemer transfers the available balance and Basin deletes the `RedeemRequest`. Any USDC that arrives afterwards has no matching request to settle against and is permanently stranded at the redeemer.

Code Corrected

Both `BUIDLTokenRedeemer` and `JTRSYTokenRedeemer` now expose a `sweep(token, amount)` function, callable by `MANAGER_ROLE` holders, that transfers `creditToken` or `collateralToken` from the redeemer back to the basin. Any amount stranded by `completeRedeem`'s `Math.min` cap or arriving after the `RedeemRequest` was deleted is recoverable through this path.

#004 `setFeeBounds` Lets the Manager Admin Override Owner-Only Fee Settings

Correctness	Low	Version 1	Code Corrected

`setPurchaseFee` and `setRedemptionFee` are restricted to `OWNER_ROLE`, but `setFeeBounds` is restricted to `MANAGER_ADMIN_ROLE` and silently clamps the current fees into the new bounds: if `purchaseFee < newMinFee` it is raised to `newMinFee`, and if `purchaseFee > newMaxFee` it is lowered to `newMaxFee` (same for `redemptionFee`). By choosing the bounds appropriately, the manager admin can therefore pick any fee value in `[0, BPS]` without the owner's approval.

Consider the following scenario:

1. `purchaseFee == 10` and `redemptionFee == 10`, set previously by `OWNER_ROLE`.
2. `MANAGER_ADMIN_ROLE` calls `setFeeBounds(0, 0)`. The bounds `(0, 0)` satisfy `newMinFee <= newMaxFee` and `newMaxFee <= BPS`, so the call succeeds.
3. Because `purchaseFee > newMaxFee` and `redemptionFee > newMaxFee`, the clamp logic invokes `_setPurchaseFee(0)` and `_setRedemptionFee(0)`.
4. Both fees are now 0 without `OWNER_ROLE` ever being consulted.

The same trick drives the fees to any target value by choosing `newMinFee == newMaxFee == target`, so the role separation between `MANAGER_ADMIN_ROLE` and `OWNER_ROLE` on fee configuration is effectively erased for any target in `[0, BPS]`.

Code Corrected

`setFeeBounds` now reverts if one of the fees is out-of-bounds for the new boundaries.

Note that this change has the following consequence: setting a nonzero `minFee` on a fresh basin now takes a bootstrap sequence. `MANAGER_ADMIN_ROLE` first opens the bounds with `setFeeBounds(0, max)`, `OWNER_ROLE` sets `purchaseFee` and `redemptionFee` to their desired values, then `MANAGER_ADMIN_ROLE` tightens the lower bound. Basins that keep `minFee == 0` are unaffected.

#005 `UsdsUsdcPocket` Accounting Breaks if the PSM Ever Charges a Fee

Correctness	Low	Version 1	Code Corrected

`UsdsUsdcPocket`'s deposit, withdraw, and `availableBalance` paths assume the Sky PSM executes `sellGem` and `buyGem` at a strict 1:1 rate modulo precision. Note that the below focusses on the expected setup where USDS is the `swapToken` and USDC is the `collateralToken`.

Deposits and liquidity withdrawals are unaffected, since no PSM swap is triggered. However, swaps might be affected when the output token is USDC. More specifically, `buyGem` could lead to more USDS than expected being consumed, leading to a loss in share value.

Code Corrected

`UsdsUsdcPocket.withdrawLiquidity` now reverts when `IPSMLike(psm).tout() != 0`, preventing the basin from consuming USDS at a loss whenever the PSM charges a fee on `buyGem`.

`depositLiquidity` is intentionally not symmetric: under the new USDC custody design (see USDC Potentially Unaccounted at `UsdsUsdcPocket`), the USDC branch only ever handles donations, so a non-zero `psm.tin()` is acceptable on that path.

#006 Pocket/Basin Integration Lacks Sanity Checks on Expected Tokens

Design	Low	Version 1	Code Corrected

`IGroveBasinPocket` exposes a generic `asset`-parameterized interface (`depositLiquidity`, `withdrawLiquidity`, `availableBalance`), but every concrete pocket is written against a specific, hardcoded token set and reverts, silently no-ops, or misaccounts when asked about anything else. The two integration points that could catch a mismatch both skip the check: `GroveBasin.setPocket` accepts any address without verifying that `pocket.basin() == address(this)`, and the pockets' constructors take a `basin_` address without verifying that the Basin's `swapToken` (and, where relevant, `collateralToken`) match the tokens the pocket was built for. A misconfigured pairing therefore compiles through on both sides and only manifests once value flows.

Several failure modes follow from these missing checks.

For the USDT pockets (`AaveV3UsdtPocket` , `MorphoUsdtPocket`), the expected `basin.swapToken` is USDT:

- If `basin.swapToken != USDT` , swaps move funds to the pocket if `assetIn == basin.swapToken` . However, the funds would be unwithdrawable due to a lack of approval and due to `withdrawLiquidity` reverting with `InvalidAsset` . However, `availableBalance` would nonetheless account for the token.
- If `basin.collateralToken == USDT` , delayed donations could occur. More specifically, swaps could try to tap into extra USDT liquidity (unaccounted for in `totalAssets` , potentially donated to pocket) leading to a sudden increase in share price during a swap.

For `UsdsUsdcPocket` , the expected `basin.swapToken` is USDS:

- If `basin.swapToken == USDC` , `basin.deposit` may lead to direct swaps. `availableBalance` would nonetheless return correct amounts. However, `basin.collateralToken == USDS` would not be able to tap into the extra liquidity on swaps.
- Other uncommon setups exist and would fail similarly to what was described above.

Across all pockets, `setPocket` would similarly create problems during migration.

To summarize, sanity checks are missing throughout the infrastructure:

- No guarantee that the `GroveBasin` instance is the `pocket.basin` .
- No guarantee that the `swapToken` and `collateralToken` match the pocket expectation.

Code Corrected

Two binding checks have been added to prevent the misconfiguration paths described above:

- Each pocket constructor (`AaveV3UsdtPocket` , `MorphoUsdtPocket` , `UsdsUsdcPocket`) now reverts unless `basin.swapToken()` matches its hardcoded swap token. `UsdsUsdcPocket` additionally requires `basin.collateralToken() == USDC` .
- `GroveBasin.setPocket` now reverts unless `IGroveBasinPocket(newPocket).basin() == address(this)` (the trivial self-pocket case is exempted).

Together, these make a basin/pocket pair mis-binding fail at deployment or migration rather than during value flow.

#007 README Does Not Reflect the Current Codebase

Correctness	Low	Version 1	Specification Changed

The `README.md` describes an earlier version of the contracts and does not match the current code:

- **Access control model.** The `README.md` claims `setPocket` uses OpenZeppelin `Ownable` and `onlyOwner` . `GroveBasin` actually inherits `AccessControl` ; `setPocket` is gated by `MANAGER_ADMIN_ROLE` , distinct from `OWNER_ROLE` .
- **`initiateRedeem` flow.** The `README.md` describes it as restricted to the owner and as moving the credit token "from the pocket to Grove Basin". In the code it is gated by `REDEEMER_ROLE` , never touches the pocket, and forwards the credit token to an `ITokenRedeemer` registered for that asset.
- **`deposit` is permissioned.** The `README.md` presents `deposit` as openly callable with a referral code. The current signature takes no referral parameter and reverts `NotLiquidityProvider` for any caller other than the immutable `liquidityProvider` .
- **`withdraw` referral code.** The `README.md` likewise documents a referral parameter that does not exist on `withdraw` . Only `swapExactIn` and `swapExactOut` accept a `referralCode` .

- **Deployment path.** The `README.md` references `deploy/GroveBasinDeploy.sol`, which does not exist. Bootstrapping is performed by `GroveBasinFactory` together with `depositInitial`, which can only be called once.

In addition, several parts of the code are not mentioned at all: the full role hierarchy and the custom `revokeRole` override, the pause system with its multiple keys, the purchase and redemption fees together with the `feeClaimer` accrual, the rate provider abstraction together with its staleness checks, the `maxSwapSize` limit, and the pocket and redeemer implementations under `src/pockets/` and `src/redeemers/`.

In **Version 2**, the following mismatches were introduced:

- `setUnpaused` is undocumented.
- `README.md` still describes `PAUSER_ROLE` as able to unpause, which is no longer true in **Version 2**. The `MANAGER_ADMIN_ROLE` description does not mention that it can unpause.

> **Specification Changed**
>
> The specification has been adjusted to match the codebase more accurately.

#008 USDC Potentially Unaccounted at `UsdsUsdcPocket`

Design	Low	Version 1	Code Corrected

`UsdsUsdcPocket.withdrawLiquidity` is gated by `onlyBasinOrManager`, so a holder of `MANAGER_ROLE` can invoke it directly. The USDC branch buys USDC against the pocket's USDS via the PSM and leaves the resulting USDC at the pocket (the function has no recipient parameter and does not transfer out to the caller). When `basin.swapToken == USDS`, `availableBalance(USDS)` reads only the raw USDS balance and ignores any USDC the pocket holds, so a manager-triggered USDC conversion silently turns accounted USDS into unaccounted USDC at the pocket. `totalAssets()` drops by the full converted amount (plus any PSM fee), reducing the value backing every LP share.

The situation is reversible: a manager can call `depositLiquidity(USDC, ...)` to sell the USDC back to USDS via the PSM, or a swap whose `assetOut == basin.collateralToken == USDC` will consume the pocket's local USDC balance. `setPocket` migration is the one path where the unaccounted USDC is lost: it only sweeps the `swapToken` (USDS) balance from the old pocket and leaves any USDC behind.

> **Code Corrected**
>
> USDC is no longer custodied at `UsdsUsdcPocket`:
>
> - `UsdsUsdcPocket.withdrawLiquidity(USDC, ...)` now `safeTransfer`s the converted USDC directly to the basin, and `GroveBasin._withdrawLiquidityInPocket` no longer follows up with a `safeTransferFrom` (the pocket-side `safeApprove(basin, max)` for USDC has been removed accordingly). A manager-triggered USDC conversion can therefore no longer leave value stranded at the pocket.
> - A new `sweep()` (callable by basin or manager) transfers any residual USDC at the pocket back to the basin, providing an explicit recovery path for donations.
> - Consequently, `availableBalance(USDC)` no longer adds the USDS-converted balance and simply returns `usdc.balanceOf(address(this))`, which is correct under the new invariant that no USDC should ordinarily sit at the pocket.

#009 Zero-Amount Redeem Request Permanently Blocks Redeemer Removal

Correctness	Low	Version 1	Code Corrected

A `REDEEMER_ROLE` holder can permanently prevent removal of a specific redeemer by calling `GroveBasin.initiateRedeem(redeemer, 0)`. The function lacks a zero-amount check, creating a `RedeemRequest` with `creditTokenAmount = 0` and incrementing `pendingRedemptions[redeemer]`.

This request can never be completed: `_completeRedeem()` explicitly reverts when `request.creditTokenAmount == 0`. Since no code path decrements the counter for stuck requests, `removeTokenRedeemer()` is permanently blocked by the `pendingRedemptions[redeemer] > 0` check.

The `MANAGER_ADMIN_ROLE` loses the ability to remove that redeemer contract.

> **Code Corrected**
>
> GroveLabs disallows passing `creditTokenAmount == 0` to `initiateRedeem` making the above impossible to reach.

#010 100% Fee Disables `swapExactOut`

Informational	Version 1	Code Corrected

The purchase and the redemption fees can be set to 100% due to `setFeeBounds` allowing the actuall value to be within the bounds. However, `_getGrossAmountFromNet` reverts due to a division-by-zero in such cases.

As a result, a 100% fee disables `swapExactOut`.

> **Code Corrected**
>
> GroveLabs now disallows setting 100% fees.

#011 `maxSwapSize` Check Rounds the Swap Value Down, Letting Swaps Marginally Exceed the Cap

Informational	Version 1	Code Corrected

The four swap paths check `_getAssetValue(assetIn, amountIn, false) > maxSwapSize`. The `false` flag selects `Math.Rounding.Floor`, so an `amountIn` whose true USD value sits marginally above `maxSwapSize` can round down to `maxSwapSize` and pass the check.

The shortfall is bounded by the floor-rounding error of `Math.mulDiv(amount * rate, 1e18, ratePrecision * tokenPrecision, Floor)`, i.e. at most ~1 wei of value in 1e18 precision per swap, so the cap is not enforced strictly, though only by a dust amount.

> **Code Corrected**
>
> The code has been adjusted to defensively round up.

#012 `redeemRequestId` Derivation Collides on Identical Same-Block Redemptions

Informational | **Version 1** | **Acknowledged**

`_initiateRedeem` derives the request ID as `keccak256(abi.encode(blockNumber, redeemer, creditTokenAmount, collateralTokenAmount))`. Within a single block, `blockNumber` is constant and `collateralTokenAmount` is deterministic given `creditTokenAmount` (the swap quote is pure with respect to intra-block state), so any two `_initiateRedeem` calls with the same `redeemer` and the same `creditTokenAmount` in the same block produce the same ID. The second call hits the `RequestAlreadyExists` revert even though it is a legitimate, independent redemption.

To summarize, the request ID is not necessarily unique.

Acknowledged

GroveLabs acknowledges the finding and argues that the uniqueness property is not required.

#013 Approval Calls Bypass or Duplicate `SafeERC20`'s Logic

Informational | **Version 1** | **Code Corrected**

Two approval-handling patterns in the codebase are off-spec with `SafeERC20` and inconsistent with how `safeApprove` is used elsewhere:

1. `_initiateRedeem` calls `IERC20(creditToken).approve(redeemer, creditTokenAmount)` and `IERC20(creditToken).approve(redeemer, 0)` directly, instead of `safeApprove`. `GroveBasin` already declares `using SafeERC20 for IERC20;` and uses `safeApprove` on every other approval path, so this is the single call site that does not revert on a `false`-returning token and does not rotate through `approve(0) -> approve(amount)` for USDT-style tokens that reject non-zero-to-non-zero transitions. Arguably, the `creditToken` is not expected to have such weird behaviour.
2. `MorphoUsdtPocket.depositLiquidity` and `AaveV3UsdtPocket.depositLiquidity` both do `safeApprove(spender, 0)` immediately followed by `safeApprove(spender, amount)`. This is redundant: `SafeERC20.safeApprove` first tries the direct `approve(spender, amount)` and, only if it fails, internally falls back to `approve(spender, 0)` + `approve(spender, amount)`.

Code Corrected

The code has adjusted accordingly.

#014 Constructor Does Not Emit Events for Initialized State

Informational | **Version 1** | **Acknowledged**

`GroveBasin`'s constructor writes several state variables that have dedicated "Set" events elsewhere in the contract, but emits none of them. Off-chain indexers that reconstruct state from event logs therefore miss the initial values and must special-case the deployment transaction, or read the contract directly.

The following constructor writes have corresponding events that are not emitted:

- `swapTokenRateProvider`, `collateralTokenRateProvider`, `creditTokenRateProvider`: would map to `RateProviderSet` (one per token).
- `pocket = address(this)`: would map to `PocketSet`.
- `maxSwapSize = 50_000_000e18`: would map to `MaxSwapSizeSet`.
- `maxSwapSizeLowerBound`, `maxSwapSizeUpperBound`: would map to `MaxSwapSizeBoundsSet`.
- `minStalenessThreshold`, `maxStalenessThreshold`: would map to `StalenessThresholdBoundsSet`.
- `stalenessThreshold = minStalenessThreshold`: would map to `StalenessThresholdSet`.

All subsequent changes to these variables through the normal setters emit the corresponding event, so the constructor is the only point at which the state transition is unobservable from logs.

Acknowledged

GroveLabs is aware and has acknowledged the finding.

#015 Gas Optimizations

Informational	Version 1	Partially Corrected / Acknowledged

Below is a non-exhaustive list of potential gas optimizations:

1. `ChronicleRateProvider.getConversionRate` and `FixedRateProvider.getConversionRate` call `this.getConversionRateWithAge()`, which forces an external staticcall back into the same contract instead of a cheap internal jump. Making the age-returning variant internal (or duplicating its short body) would remove the staticcall and calldata/returndata overhead on every rate read.
2. `setMaxSwapSizeBounds` and `setStalenessThresholdBounds` emit their derived "Set" event using an SLOAD of the state variable they may have just written (`maxSwapSize`, `stalenessThreshold`), even though the new value is already held in a local. Emitting the local avoids the redundant storage read.
3. `_hasPocket()` reads pocket from storage, and every caller then reads pocket a second time in the same function body. This occurs in `_withdrawLiquidityInPocket`, `_getAvailableBalance`, `_pushAsset`, and `setPocket` (which already caches pocket_ but then calls `_hasPocket()` anyway). Caching address `pocket_ = pocket` once per function and comparing against `address(this)` directly collapses two `SLOADs` into one.
4. `convertToAssets(address, uint256)` fetches the asset's rate twice: once indirectly through `convertToAssetValue` and `totalAssets()` (which sums all three assets and therefore reads the target asset's rate), and once directly via `_getTokenRateAndPrecision(asset)` at the end. The same double-read pattern appears in `previewDeposit` (via `convertToShares` and `totalAssets()` while `_getAssetValue` already touched the asset's rate). On Chronicle-backed providers this is an extra `readWithAge()` staticcall per conversion; caching the rate once per call eliminates it.

Partially Corrected

(1) and (2) have been addressed. (3) and (4) were acknowledged since they would introduce unnecessary complexity.

#016 Inconsistencies Across `GroveBasin`

Informational	Version 1	Partially Corrected / Acknowledged

Several behavioral patterns are applied inconsistently across the contract:

1. `setStalenessThreshold` is the only setter that uses a dedicated `SameThreshold` error to reject a write of the same value (`new == old`). `setPocket` also rejects same value writes, but folds the check into a compound precondition that shares the generic `InvalidPocket` error with the zero address branch. All remaining setters silently accept same value writes and emit their setter event regardless.
2. `setFeeBounds` does not emit `PurchaseFeeSet` / `RedemptionFeeSet` on no-op: the derived "Set" event is emitted only when the clamp actually fires, so a call that leaves the fees untouched produces no fee-change event. The other bound-setters emit their derived "Set" event (`MaxSwapSizeSet` / `StalenessThresholdSet`) unconditionally at the end of the function, even with `old == new`. Observers relying on the derived event therefore get an inconsistent signal depending on which bound was changed.
3. As a consequence of (1) and (2), `setStalenessThresholdBounds` can emit `StalenessThresholdSet(t, t)`, while `setStalenessThreshold` rejects that exact transition outright. The two entry points disagree on whether a no-op is a valid state.
4. The zero-address rate-provider precondition reverts with different errors depending on the entry point: the constructor reverts `ZeroRateProviderAddress`, while `setRateProvider` reverts `InvalidRateProvider` for the same condition.
5. `swapExactIn` and `swapExactOut` revert `ZeroReceiver` when `receiver == address(0)`, but `deposit` and `withdraw` accept a `receiver` parameter without validating it. A zero receiver on `deposit` credits shares to `address(0)` and on `withdraw` attempts a transfer to `address(0)` whose outcome is token dependent. The precondition exists on swaps and is simply missing on the other two user facing entry points.
6. Preview functions are inconsistent with their paired execute paths. `_checkPaused(key)` reverts on either the global pause (`paused[bytes4(0)]`) or the supplied key, so a preview that skips `_checkPaused` also skips the global pause. A caller using a preview as a gate can therefore receive a clean quote for an input that the execute call would reject:
   - `previewDeposit` calls `_checkPaused(PAUSED_DEPOSIT_CREDIT)` only when `asset == creditToken`, while `deposit` additionally calls `_checkPaused(msg.sig)` at entry. For credit deposits the preview misses the selector pause; for non-credit deposits it misses both the global pause and the selector pause.
   - `previewSwapExactIn` and `previewSwapExactOut` never call `_checkPaused`, so they do not enforce the global pause, the swap selector pause, or the direction-specific swap pause (`_getSwapPauseKey(assetIn, assetOut)`) that `swapExactIn` / `swapExactOut` both enforce. The swap previews do replicate `SwapSizeExceeded` when the executed swap would exceed `maxSwapSize`, so a caller sees an oversized swap rejected at preview time but a paused swap quoted cleanly.
   - The swap previews also omit the zero-amount reverts: `swapExactIn` reverts `ZeroAmountIn` when `amountIn == 0` and `swapExactOut` reverts `ZeroAmountOut` when `amountOut == 0`, while the previews can return zero.
7. `addTokenRedeemer` calls `ITokenRedeemer(redeemer).setUp(address(this))` directly, so a reverting `setUp` aborts registration. `removeTokenRedeemer` calls `tearDown` inside a bare `try/catch {}` that discards any revert. The asymmetry is defensible on the grounds that a misbehaving redeemer should not be able to brick its own removal, but it is undocumented, and a silently failing `tearDown` leaves no on chain signal that the redeemer was not cleanly deregistered.

Partially Corrected

Item (4) is addressed in code: `setRateProvider` at `GroveBasin.sol:168` now reverts `ZeroRateProviderAddress`, matching the constructor. Item (6) has been addressed.

Items (1), (3), (5), and (7) are unchanged. Item (2): `setFeeBounds` at `GroveBasin.sol:247` no longer clamps the existing fees into the new bounds; it now reverts `CurrentFeeOutOfNewBounds` when the current fees lie outside the new range.

#017 Lack of Sanity Checks

Informational | **Version 1** | **Code Corrected**

A few places accept inputs without validation that would catch wrong usage up-front:

1. `previewSwapExactInFee` and `previewSwapExactOutFee` do not validate `assetOut`. Both branches only check whether `assetOut == creditToken` and otherwise apply `redemptionFee`, so a caller passing an unsupported address silently gets a quote instead of a revert.
2. `_getTokenRateAndPrecision` has no default branch: for any `token` that is not `swapToken` or `collateralToken` it returns the credit-token rate, precision, and token-precision, even if `token` is not the credit token. Adding a `_requireValidAsset(token)` (or an explicit revert) here would make the helper self-validating and allow removing the redundant `_requireValidAsset` checks at the call sites that currently guard it.

Code Corrected

1. Corrected. The fee preview functions now validate the asset.
2. Not adjusted. The check is not strictly required as all paths leading to the code validate accordingly.

#018 Pause Design

Informational | **Version 1** | **Code Corrected**

The pauser role can both pause and unpause. In Sky-style designs the pauser is typically limited to pausing, with governance performing the unpause; the split is also inconsistent internally, since the pauser can undo a pause but cannot re-grant revoked roles, so only half of the post-incident recovery is within the pauser's reach.

Separately, `completeRedeem` has no pause check, so a pending redemption may always be executed.

Code Corrected

`completeRedeem` is now pausable and the pauser can now only pause while `MANAGER_ADMIN_ROLE` can unpause.

#019 Wei-Scale Rounding Dust in Pocket Accounting

Informational | Version 1 | Risk Accepted

Several rounding interactions between the Basin and the pockets produce wei-scale accounting drift that is absorbed by all shareholders collectively:

- `MorphoUsdtPocket`: `vault.deposit(X)` mints floored shares, and `availableBalance(USDT)` reads `convertToAssets(shares)` (also floored), typically returning `X - 1`. `totalShares` grows by the weight of `X` while `totalAssets` grows by `X - 1`, diluting existing LPs by a wei of value per deposit. On withdrawals, `vault.withdraw(X)` burns shares via `previewWithdraw` (ceil), so the pocket loses slightly more underlying than `X` and the extra loss spreads to remaining shareholders. On `setPocket` migration, `_withdrawLiquidityInPocket(availableBalance, USDT)` consumes `previewWithdraw(availableBalance)` shares from the vault, which is always at most the pocket's share count and sometimes strictly less, so a residual vault share can remain at the old pocket and the subsequent raw-`balanceOf(USDT)` sweep does not transfer vault shares.
- `AaveV3UsdtPocket`: the same pattern appears through Aave's ray-precision index math. `supply(amount).rayDiv(liquidityIndex)` is stored as scaled balance, and `aUSDT.balanceOf = scaledBalance.rayMul(liquidityIndex)` can come back as `amount - 1`. Aave V3 uses half-up rounding in both directions, so the effect is smaller and less systematic than the Morpho case.
- Basin-side `totalAssets()` sums three `_getAssetValue(..., false)` floor-rounded values, so reported `TA` is up to a few wei below its true value. Inside `convertToShares` (floor) this gives slightly more shares to a depositor; inside `_convertToSharesRoundUp` (ceil) this burns slightly more shares from a withdrawer. The two directions partially self-cancel on a balanced workload.

In all three cases the magnitudes are ~1 wei of value per transaction.

Risk Accepted
GroveLabs is aware and accepts the risk.

#021 `GroveBasinFactory.deploy` Can Still Be Griefed via Mempool Frontrunning

Informational | Version 2 | Acknowledged

The switch to `CREATE2` in **Version 2** resolves the mass grief vector described in `GroveBasinFactory.deploy` can be griefed by donating dust to the predicted address: the basin address now derives from the salt rather than from the factory's nonce, so an attacker can no longer precompute one address and fund it once to grief every subsequent deployment.

The underlying cause is however not addressed: `depositInitial` still reverts whenever `totalAssets > 0` while `totalShares == 0`, because `convertToShares` returns `0` in that state. An individual `deploy(...)` transaction can therefore still be griefed by any observer of the public mempool: the salt inputs (or the raw `salt` passed to the custom overload) are visible in the pending transaction, so the predicted basin address can be recomputed and 1 wei of `swapToken`, `collateralToken`, or `creditToken` donated to it ahead of inclusion. Retrying `deploy(...)` with the same salt inputs keeps hitting the same donated address, so the deployer has to vary the salt (e.g. via the custom overload) to get past the donation.

Acknowledged

GroveLabs is aware and has acknowledged the finding.

#022 Confusing Revert Reasons in User Facing Flows

Informational **Version 3** **Code Corrected**

Two paths surface revert messages that do not reflect the actual reason of the failure: Swap or withdraw of collateral token reverts with `InvalidAsset` when basin is short of collateral

A user calling `swapExactIn` or `swapExactOut` with `creditToken` as `assetIn` and `collateralToken` as `assetOut`, or anyone calling `withdraw(collateralToken, ...)`, receives `InvalidAsset()` when the basin's collateral token balance is below the requested output amount. This applies to basins paired with `AaveV3UsdtPocket` or `MorphoUsdtPocket`.

`GroveBasin._withdrawLiquidityInPocket` observes that the basin's collateral balance is below the amount and attempts to withdraw from the pocket: `pocket.withdrawLiquidity(deficit, collateralToken)`. Both pocket implementations only recognise their swap token (USDT) and revert `InvalidAsset()` for any other asset.

Both arguments passed by the caller are valid supported tokens of the basin. The true cause of the failure is the basin being short on collateral token, but the surfaced error suggests the asset itself is unsupported.

Code Corrected

`_withdrawLiquidityInPocket` will revert with `InsufficientFunds` if insufficient funds were freed. `AaveV3UsdtPocket` and `MorphoUsdtPocket` will additionally return 0 for any token other than USDT for the respective functions. As a consequence, `InsufficientFunds` will now be a consistent revert reason.

8 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.