yAudit

December 2, 2025

Prepared for
Centrifuge

Audited by
HHK
adriro
yAudit Block 7 fellows

Centrifuge V3 Review

Smart Contract Security Assessment

Contents

1 Review Summary 2

1.1 Protocol Overview . . . 2
1.2 Audit Scope . . . 2
1.3 Risk Assessment Framework . . . 2
1.3.1 Severity Classification . . . 2
1.4 Key Findings . . . 3
1.5 Overall Assessment . . . 3

2 Audit Overview 3

2.1 Project Information . . . 3
2.2 Audit Team . . . 3
2.3 Audit Timeline . . . 3
2.4 Audit Resources . . . 4
2.5 Critical Findings . . . 5
2.6 High Findings . . . 5
2.6.1 Shares are transferred twice during request redeem for legacy vaults . . . 5
2.7 Medium Findings . . . 6
2.7.1 Zero deposits into the `balanceSheet` will block future snapshots . . . 6
2.8 Low Findings . . . 8
2.8.1 The `AsyncRequestManager::max*` view functions will return incorrect values if the share token implements amount based restrictions . . . 8
2.8.2 Transfer restriction could cause losses when redemptions are fulfilled . . . 9
2.8.3 Inconsistent Vault Validation Between Router Functions . . . 10
2.9 Gas Savings Findings . . . 11
2.9.1 Redundant shareQueue.isPositive assignments in BalanceSheet operations 11
2.9.2 Duplicate limit checks for `maxMint` and `maxWithdraw` . . . 12
2.9.3 Simplify manager lookup in AsyncVault . . . 12
2.9.4 Avoid asset self-transfer in VaultRouter . . . 13
2.9.5 Cache storage variable . . . 13
2.10 Informational Findings . . . 14
2.10.1 Share and asset queue drift in BalanceSheet due to incorrect signed-emulation logic . . . 14
2.10.2 OnOfframpManagerFactory.newManager() allows creation of OnOfframpManager contracts with arbitrary pair of (poolId, shareClassId) . . . 16
2.10.3 Incorrect NatSpec on `isValid()` misrepresents validation logic . . . 18
2.10.4 ShareClassId Validation Bypass in OnOfframpManager Cross-Chain Updates 19
2.10.5 `onRedeemRequest()` is never called . . . 20
2.10.6 Events part of executions initiated in the LegacyVaultAdapter are emitted in the legacy vault . . . 21
2.10.7 Incorrect argument in CancelRedeemClaim event . . . 22
2.10.8 Incorrect argument in RedeemRequest event . . . 22
2.10.9 Apply CEI in BalanceSheet . . . 23
2.10.10 Validate entities are registered in Spoke contract . . . 23
2.10.11 OnOfframpManager should raise if the update kind is not supported . . . 24
2.11 Final Remarks . . . 24

1 Review Summary

1.1 Protocol Overview

Centrifuge V3 is an open, decentralized protocol for onchain asset management. Built on immutable smart contracts, it enables permissionless deployment of customizable tokenization products

1.2 Audit Scope

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Reduced transaction costs
Informational	Code quality and best practice recommendations	Improved maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	1
🟨 Medium	1
🟩 Low	3
⬛ Informational	11

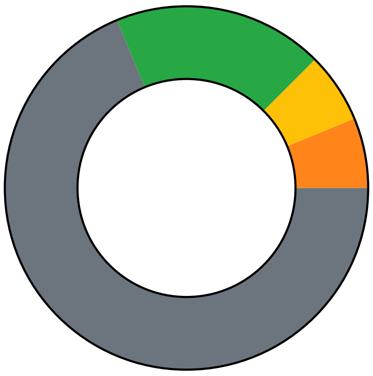

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

2 Audit Overview

2.1 Project Information

Protocol Name: Centrifuge

2.2 Audit Team

HHK, adriro, yAudit Block 7 fellows

2.3 Audit Timeline

The audit was conducted from June 16 to 27, 2025.

2.4 Audit Resources

Category	**Mark**	**Description**
Access Control	Average	Given the shallow authentication system, it is challenging to determine who has access to each system function.
Mathematics	Good	The reviewed contracts present correctly implemented mathematical relations.
Complexity	Average	Despite its modularity and good design, Centrifuge is a big protocol with complex asynchronous flows that can even span multiple chains.
Libraries	Good	There are no explicit external dependencies. Some libraries are derived from or inspired by other protocols, such as Maker DAO or Uniswap.
Decentralization	Low	As the protocol deals with real-world assets, most of its functionality is permissioned, and tokens have transfer restrictions.
Code Stability	Good	The codebase remained stable during the engagement.
Documentation	Good	The contracts are well-documented with clear comments and good NatSpec coverage. Detailed high-level documentation was provided to the auditors to help them understand the architecture and the general context surrounding the vaults.
Monitoring	Good	Monitoring mechanisms are in place to track key events and changes within the system.
Testing and verification	Average	The codebase features a rich testing suite. However, the legacy adapter wasn't covered. The protocol team stated that this functionality is still under discussion and will be released eventually after V3 is deployed.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

2.6.1 Shares are transferred twice during request redeem for legacy vaults

The legacy vault will transfer shares to the escrow upon request redeem, but this also happens as part of the execution of the new async manager.

Technical Details

The implementation of the original `requestRedeem()` function transfers the shares from the user to the escrow after calling the manager.

```
131:        address escrow = manager.escrow();
132:        try ITranche(share).authTransferFrom(sender, owner, escrow, shares) returns (bool) {}
133:        catch {
134:            // Support tranche tokens that block authTransferFrom. In this case ERC20 approval needs to be set
135:            require(ITranche(share).transferFrom(owner, escrow, shares), "ERC7540Vault/transfer-from-failed");
136:        }
137:
```

The LegacyVaultAdapter contract, working as the manager of the legacy vault, will forward the call to the new AsyncRequestManager which will also attempt to transfer the shares.

```
144:        balanceSheet.transferSharesFrom(vault_.poolId(), vault_.scId(), sender_, owner, address(globalEscrow), shares_);
```

Impact

High. The issue could block redemption requests or cause a duplicate share transfer, leading to potential losses.

Recommendation

As the legacy functionality must be maintained, the adapter should implement a logic similar to the new manager implementation but without dealing with the share transfer.

Developer Response

Fixed in PR#478.

2.7 Medium Findings

2.7.1 Zero deposits into the `balanceSheet` will block future snapshots

The `OnOfframpManager` and `syncDepositVault` accepts deposits from any accounts and don't enforce minimum deposits allowing to increase the balanceSheet queue counter. The counter can't be reset when there is no deposits in queue which will block snapshots.

Technical Details

The `deposit()` function is called by sync and async vaults as well as the `OnOfframpManager`.
When depositing, it will call the internal function `_updateAssets()` inside which it will increment the `shareQueue.queuedAssetCounter` if the previous queued deposits and withdrawals are set to 0. Then it will increase the deposits queued by the deposited amount.
Later when the manager calls `submitQueuedAssets()` to sync the hub with the `balanceSheet` it will reset the queued deposits and withdrawals as well as decrement the `shareQueue.queuedAssetCounter`. The `assetCounter` variable is used inside the function to determine if a snapshot should happen, this is the case If `shareQueue.queuedAssetCounter == assetCounter`, it is also subtracted from it at the end of the function. `assetCounter` will always be either 0 or 1, depending if there is queued deposits and withdrawals telling the function to trigger snapshot only once the queue has been cleared.
However, when depositing there is no check on zero deposits which allows any user to increment the `shareQueue.queuedAssetCounter` variable infinitely. This is an issue has the `submitQueuedAssets()` function relies on it to trigger snapshots and expects it to be incremented only when there is queued deposits and withdrawals.
By making the variable out of sync, the `isSnapshot` parameter sent to the hub will always be false and there is no way to fix the `shareQueue.queuedAssetCounter`. This could lead the hub to be out of sync with the `balanceSheet`.
POC:

```
contract OnOfframpManagerDepositZeroSuccessTests is OnOfframpManagerBaseTest {
    using CastLib for *;
    using UpdateContractMessageLib for *;

    function testDeposit() public {
        //setup
        vm.prank(address(spoke));
        manager.update(
            POOL_A,
            defaultTypedShareClassId,
            UpdateContractMessageLib.UpdateContractUpdateAddress({
                kind: bytes32("onramp"),
                assetId: defaultAssetId,
                what: bytes32(""),
                isEnabled: true
            }).serialize()
        );

        balanceSheet.updateManager(POOL_A, address(manager), true);
        assertEq(erc20.balanceOf(address(manager)), 0);
        assertEq(balanceSheet.availableBalanceOf(manager.poolId(), manager.scId(), address(erc20), erc20TokenId), 0);

        //do 3 empty deposits
        manager.deposit(address(erc20), erc20TokenId, 0, address(manager));
        manager.deposit(address(erc20), erc20TokenId, 0, address(manager));
        manager.deposit(address(erc20), erc20TokenId, 0, address(manager));

        assertEq(erc20.balanceOf(address(manager)), 0);
        assertEq(
            balanceSheet.availableBalanceOf(manager.poolId(), manager.scId(), address(erc20), erc20TokenId), 0
        );
        //the counter gets incremented 3 times
        (,,uint32 queuedAssetCounter,) = balanceSheet.queuedShares(manager.poolId(), manager.scId());
        assertEq(queuedAssetCounter, 3);

        //add a >1 valid deposit
        erc20.mint(address(manager), 1e18);
        manager.deposit(address(erc20), erc20TokenId, 1e18, address(manager));

        //now we're at 4
        (,, queuedAssetCounter,) = balanceSheet.queuedShares(manager.poolId(), manager.scId());
        assertEq(queuedAssetCounter, 4);

        //let's try to create a snapshot
        balanceSheet.submitQueuedAssets(manager.poolId(), manager.scId(), balanceSheet.spoke().assetToId(address(erc20), 0), 0);

        //effectively reduces by 1 since balance > 0
        (,, queuedAssetCounter,) = balanceSheet.queuedShares(manager.poolId(), manager.scId());
        assertEq(queuedAssetCounter, 3);

        //doing it again will not reduce the counter though
        balanceSheet.submitQueuedAssets(manager.poolId(), manager.scId(), balanceSheet.spoke().assetToId(address(erc20), 0), 0);

        (,, queuedAssetCounter,) = balanceSheet.queuedShares(manager.poolId(), manager.scId());
        assertEq(queuedAssetCounter, 3);
    }
}
```

Impact

Medium. The `isSnapshot` parameter will always be false which may impact the HUB accounting.

Recommendation

Block zero deposits or do not increment the queue counter on zero deposits.

Developer Response

Fixed in 168b35f.

2.8 Low Findings

2.8.1 The `AsyncRequestManager::max*` view functions will return incorrect values if the share token implements amount based restrictions

The `AsyncRequestManager` contract's `maxDeposit()`, `maxMint()`, `maxWithdraw()` and `maxRedeem()` functions will return an incorrect value if the share token implements a hook with amount based transfer restrictions, causing them to return non-zero maximum values even when no actual actions can be performed.

Technical Details

The root cause of this issue lies in how the above mentioned functions validate transfer restrictions:

```
if (!_canTransfer(vault_, ESCROW_HOOK_ID, user, 0))
```

Unlike the rest of the contract, where `_canTransfer()` is always called with the actual share amount being transferred, the view functions deviate from this pattern by hardcoding the share amount to zero. When a hook implements amount based transfer restrictions (e.g., maximum investment limits per user, global caps, or per-transaction limits), passing zero to `_canTransfer()` will likely return `true` since zero doesn't violate any amount based restrictions. However, when users attempt to perform the actual operation with the returned maximum values the hook will correctly enforce its restrictions and revert the transaction.

Impact

Low. This issue causes the `maxDeposit()`, `maxMint()`, `maxWithdraw()` and `maxRedeem()` functions to return inaccurate maximum values when amount based transfer restrictions are implemented, although impact is limited since the actual operations enforce this restrictions.

Recommendation

Modify the `maxDeposit()`, `maxMint()`, `maxWithdraw()` and `maxRedeem()` functions to use the actual share amounts when calling `_canTransfer()` instead of hardcoding it to zero. This approach maintains the existing interface and ensures consistency between view functions and actual operations by providing accurate information about whether the intended operation is possible to execute.

```diff
    function maxDeposit(IBaseVault vault_, address user) public view returns (uint256 assets) {
-       if (!_canTransfer(vault_, ESCROW_HOOK_ID, user, 0)) {
-           return 0;
-       }
+       if (!_canTransfer(vault_, ESCROW_HOOK_ID, user, investments[vault_][user].maxMint)) {
+           return 0;
+       }
```

```diff
        assets = uint256(_maxDeposit(vault_, user));
    }

    function maxMint(IBaseVault vault_, address user) public view returns (
uint256 shares) {
-       if (!_canTransfer(vault_, ESCROW_HOOK_ID, user, 0)) {
-           return 0;
-       }
        shares = uint256(investments[vault_][user].maxMint);
+       if (!_canTransfer(vault_, ESCROW_HOOK_ID, user, shares)) {
+           return 0;
+       }
    }

    function maxWithdraw(IBaseVault vault_, address user) public view returns (
uint256 assets) {
-       if (!_canTransfer(vault_, user, address(0), 0)) return 0;
+       AsyncInvestmentState memory state = investments[vault_][user];
+       shares = uint256(_assetToShareAmount(vault_, state.maxWithdraw, state.
redeemPrice, MathLib.Rounding.Down));
+       if (!_canTransfer(vault_, user, address(0), shares)) return 0;
        assets = uint256(investments[vault_][user].maxWithdraw);
    }

    function maxRedeem(IBaseVault vault_, address user) public view returns (
uint256 shares) {
-       if (!_canTransfer(vault_, user, address(0), 0)) return 0;
-       AsyncInvestmentState memory state = investments[vault_][user];

-       shares = uint256(_assetToShareAmount(vault_, state.maxWithdraw, state.
redeemPrice, MathLib.Rounding.Down));
+       AsyncInvestmentState memory state = investments[vault_][user];

+       shares = uint256(_assetToShareAmount(vault_, state.maxWithdraw, state.
redeemPrice, MathLib.Rounding.Down));
+       if (!_canTransfer(vault_, user, address(0), shares)) return 0;
    }
```

Developer Response

Fixed in PR#482.

2.8.2 Transfer restriction could cause losses when redemptions are fulfilled

Using `maxRedeem()` inside `fulfillRedeemRequest()` could return zero pending claims if the user is affected by transfer restrictions.

Technical Details

The implemention of `fulfillRedeemRequest()` relies on `maxRedeem()` to recalculate the `redeemPrice`.

```
317:         // Calculate new weighted average redeem price and update order book values
318:         state.redeemPrice = _calculatePriceAssetPerShare(
319:             vault_,
320:             ((maxRedeem(vault_, user)) + fulfilledShares).toUint128(),
321:             state.maxWithdraw + fulfilledAssets,
322:             MathLib.Rounding.Down
323:         );
```

The intention here is to use `maxRedeem(vault_, user)` along with `state.maxWithdraw` to update the price given the additions of `fulfilledShares` and `fulfilledAssets`.
However, `maxRedeem()` returns zero if the user is currently affected by transfer restrictions, in which case the redemption price will ignore existing assets pending to be claimed.

Impact

Low.

Recommendation

Refactor `maxRedeem()` into a new variant without the transfer checks, like `_maxDeposit()`, and use this logic in `fulfillRedeemRequest()`.

Developer Response

Fixed in PR#462.

2.8.3 Inconsistent Vault Validation Between Router Functions

VaultRouter applies inconsistent vault validation patterns across similar functions.

Technical Details

The VaultRouter contract shows inconsistent vault validation between similar operations:

- `claimDeposit()` performs no vault validation
- `claimRedeem()` calls `spoke.vaultDetails(vault)` which validates the vault exists

Impact

Informational. This creates potential confusion about when vault validation is required.

Recommendation

Standardize vault validation across router functions or document the design rationale if the difference is intentional.

Developer Response

Fixed in PR#479.

2.9 Gas Savings Findings

2.9.1 Redundant shareQueue.isPositive assignments in BalanceSheet operations

Redundant SSTORE operations wastes gas.

Technical Details

There are redundant SSTORE operations when `isPositive` is already in the correct state in the `BalanceSheet.sol` contract's `revoke()` function.

```
    function revoke(PoolId poolId, ShareClassId scId, uint128 shares) external authOrManager(poolId) {
        ...
        if (!shareQueue.isPositive) { // escaping the if block means shareQueue is positive
            shareQueue.delta += shares;
        } else if (shareQueue.delta > shares) {
            shareQueue.delta -= shares;
            shareQueue.isPositive = true; // @audit-info already positive, can remove
    }
```

Impact

Gas savings.

Recommendation

Remove the redundant `isPositive` assignment in `revoke()`:

```
    function revoke(PoolId poolId, ShareClassId scId, uint128 shares) external authOrManager(poolId) {
        ...
        if (!shareQueue.isPositive) { // escaping the if block means shareQueue is positive
            shareQueue.delta += shares;
        } else if (shareQueue.delta > shares) {
            shareQueue.delta -= shares;
-           shareQueue.isPositive = true;
    }
```

Developer Response

Fixed in PR#461.

2.9.2 Duplicate limit checks for `maxMint` and `maxWithdraw`

Technical Details

The implementation of `_processDeposit()` checks twice that `sharesUp <= state.maxMint`. Given the check in line 375, the conditional in line 376 should not be needed.

```
375:         require(sharesUp <= state.maxMint, ExceedsDepositLimits());
376:         state.maxMint = state.maxMint > sharesUp ? state.maxMint - sharesUp : 0;
```

The same happens in `_processRedeem()` when updating `maxWithdraw`.

```
428:         require(assetsUp <= state.maxWithdraw, ExceedsRedeemLimits());
429:         state.maxWithdraw = state.maxWithdraw > assetsUp ? state.maxWithdraw -
assetsUp : 0;
```

Impact

Gas savings.

Recommendation

Remove the conditional in lines 376 and 429. The subtractions can also be wrapped in an `unchecked` math block.

Developer Response

Fixed in PR#479.

2.9.3 Simplify manager lookup in AsyncVault

Technical Details

The AsyncVault contract fetches its manager using an external call to itself instead of just referencing the storage variable.

```
148:     function asyncManager() public view returns (IAsyncRequestManager) {
149:         return IAsyncRequestManager(address(IAsyncRedeemVault(this).
asyncRedeemManager()));
150:     }
```

Impact

Gas savings.

Recommendation

The manager can be referenced by using the `asyncRedeemManager` variable. Note that the `asyncManager()` function is called in every interaction with the manager, present in most functions.

Developer Response

Fixed in PR#479.

2.9.4 Avoid asset self-transfer in VaultRouter

The `deposit()` implementation executes an ERC20 transfer from the contract to itself.

Technical Details

- VaultRouter.sol#L135

Impact

Gas savings.

Recommendation

Avoid the transfer if `owner == address(this)`. This should help to save gas and also avoid conflicts with non-standard ERC20 implementations.

Developer Response

Fixed in df6c58b.

2.9.5 Cache storage variable

Multiple part of the codes could use some caching of storage variables to save gas.

Technical Details

In `spoke.sol`:

- In `updatePricePoolPerShare()` the variable `shareClass.pricePoolPerShare.computedAt` is read twice.
- In `shareToken()` the variable `shareClass.shareToken` is read twice.
- In `pricePoolPerShare()` the variable `shareClass.pricePoolPerShare` is read twice.

In `AsyncRequestManager.sol`:

- In `approvedDeposits()`, `issuedShares()`, `revokedShares()`, `_withdraw()` the variable `balanceSheet` is read multiple times.
- In `fulfillDepositRequest()` and `fulfillRedeemRequest()` the variables `state.maxMint`, `state.pendingDepositRequest` and `state.maxWithdraw`, `state.pendingRedeemRequest` are read multiple times.

In: `SyncManager.sol`:

- In `_issueShares()` the variable `balanceSheet` is read multiple times.
- In `_shareToAssetAmount()` the variable `spoke` is read twice.

In `BalanceSheet`:

- In `multicall()` the variable `gateway` is read multiple times.
- In `issue()`, `revoke()` and `submitQueuedShares()` the variable `shareQueue` is read multiple times.
- In `submitQueuedAssets()` and `_updateAssets()` the variable `assetQueue` is read multiple times.

Impact

Gas.

Recommendation

Cache storage variables.

Developer Response

Acknowledged. We consider readability more valuable here, and gas cost seems minimal.

2.10 Informational Findings

2.10.1 Share and asset queue drift in BalanceSheet due to incorrect signed-emulation logic

The `BalanceSheet` contract attempts to track net share issuance vs. revocation between snapshots by storing an unsigned `delta` plus a boolean `isPositive` flag. However, when the absolute amount of issuance equals the absolute amount of revocation (or vice-versa), the code's branch conditions yield `delta == 0` with `isPositive == false` in one sequence, but `delta == 0` with `isPositive == true` in another.

Technical Details

`BalanceSheet::issue()` and `BalanceSheet::revoke()` are supposed to keep a running signed total of share changes until the next cross-chain snapshot.
Instead of using a true signed integer, the code tracks

```solidity
struct ShareQueueAmount {
    uint128 delta;     // absolute magnitude
    bool    isPositive;
}
```

and then in `issue()` and `revoke()` it updates `(delta, isPositive)` via conditional branches.
However, when the absolute amounts are equal (e.g. you issue 50 shares then revoke 50 shares, or revoke 50 then issue 50), you end up with `delta == 0` but the sign flips depending on which function ran last.

```solidity
/// @inheritdoc IBalanceSheet
    function issue(PoolId poolId, ShareClassId scId, address to, uint128 shares)
external authOrManager(poolId) {
        emit Issue(poolId, scId, to, _pricePoolPerShare(poolId, scId), shares);
        ShareQueueAmount storage shareQueue = queuedShares[poolId][scId];
        if (shareQueue.isPositive || shareQueue.delta == 0) {
            shareQueue.delta += shares;
            shareQueue.isPositive = true;
        } else if (shareQueue.delta > shares) {
            shareQueue.delta -= shares;
            shareQueue.isPositive = false;
        } else {
            shareQueue.delta = shares - shareQueue.delta;
            shareQueue.isPositive = true;
        }
        IShareToken token = spoke.shareToken(poolId, scId);
        token.mint(to, shares);
    }

    /// @inheritdoc IBalanceSheet
    function revoke(PoolId poolId, ShareClassId scId, uint128 shares) external
authOrManager(poolId) {
        emit Revoke(poolId, scId, msg.sender, _pricePoolPerShare(poolId, scId), shares);
        ShareQueueAmount storage shareQueue = queuedShares[poolId][scId];
        if (!shareQueue.isPositive) {
            shareQueue.delta += shares;
        } else if (shareQueue.delta > shares) {
            shareQueue.delta -= shares;
            shareQueue.isPositive = true;
        } else {
            shareQueue.delta = shares - shareQueue.delta;
            shareQueue.isPositive = false;
        }
        IShareToken token = spoke.shareToken(poolId, scId);
        token.authTransferFrom(msg.sender, msg.sender, address(this), shares);
        token.burn(address(this), shares);
    }
```

Case (A): `issue(50)` → `revoke(50)` (a) `issue(50)` sees `delta==0 || isPositive==true` → sets `delta=50, isPositive=true` (b) `revoke(50)` sees `!isPositive==false` and `delta>shares==false` → else-branch → sets `delta=0, isPositive=false`

Case (B): `revoke(50)` → `issue(50)` (a) `revoke(50)` sees `!isPositive==true` → first-branch → sets `delta=50, isPositive=false` (b) `issue(50)` sees `delta>0 || isPositive==true` false, and `delta>shares==false` → else-branch → sets `delta=0, isPositive=true`
Because zero in Solidity is neither positive nor negative, there's no meaningful distinction—but the Hub will receive a "zero with a negative sign" vs. "zero with a positive sign," and potentially handle them differently.

Impact

Informational.

Recommendation

Enforce "zero is positive" invariant. Immediately after each branch in both `issue()` and `revoke()`, add:

```
if (shareQueue.delta == 0) {
    shareQueue.isPositive = true;
}
```

This guarantees `(0, true)` is the canonical neutral state.
OR, use native signed arithmetic. Replace `(uint128 delta, bool isPositive)` with a single `int256 deltaSigned;`:

```
int256 deltaSigned;
// In issue():
deltaSigned += int256(shares);
// In revoke():
deltaSigned -= int256(shares);
```

This eliminates future manual emulation and leverages built-in sign handling.

Developer Response

Fixed in PR#462 and PR#488.

2.10.2 OnOfframpManagerFactory.newManager() allows creation of OnOfframpManager contracts with arbitrary pair of (poolId, shareClassId)

Missing input validation in `OnOfframpManagerFactory.newManager()` allows creation of OnOfframpManager contracts with inconsistent poolId/ShareClassId relationships, potentially leading to operational failures and funds being locked.

Technical Details

The `OnOfframpManagerFactory.newManager()` function lacks critical input validation to ensure that the provided `ShareClassId` actually belongs to the specified `PoolId`. This breaks a fundamental invariant in the system where ShareClassIds are designed to embed their parent PoolId.

```
function newManager(PoolId poolId, ShareClassId scId) external returns (
IOnOfframpManager) {
    // @audit-issue No validation that scId belongs to poolId
    OnOfframpManager manager = new OnOfframpManager{salt: keccak256(abi.encode(poolId.
raw(), scId.raw()))}(
        poolId, scId, spoke, balanceSheet
    );

    emit DeployOnOfframpManager(poolId, scId, address(manager));
    return IOnOfframpManager(manager);
}
```

The ShareClassId type is structured to embed the PoolId in its upper 64 bits:

```
// ShareClassId.newShareClassId()
function newShareClassId(PoolId poolId, uint32 index) pure returns (ShareClassId scId) {
    return ShareClassId.wrap(bytes16((uint128(PoolId.unwrap(poolId)) << 64) + index));
}
```

However, `newManager()` accepts any arbitrary combination of `poolId` and `scId` parameters without verifying this relationship. This allows creation of managers where:

- The constructor receives `poolId = X` and `scId = Y`
- But `scId` was actually created for `poolId = Z` (where `Z != X`)

Impact

Informational. Managers can be deployed with inconsistent poolId/ShareClassId relationships. These managers can then be updated as long as the poolId matches, regardless of ShareClassId validity.

Recommendation

Add validation to ensure the ShareClassId belongs to the specified PoolId:

```
function newManager(PoolId poolId, ShareClassId scId) external returns (
IOnOfframpManager) {
    // Extract embedded poolId from ShareClassId
    uint64 embeddedPoolId = uint64(uint128(scId.raw()) >> 64);
    require(embeddedPoolId == poolId.raw(), InvalidShareClassForPool());

    OnOfframpManager manager = new OnOfframpManager{salt: keccak256(abi.encode(poolId.
raw(), scId.raw()))}(
        poolId, scId, spoke, balanceSheet
    );

    emit DeployOnOfframpManager(poolId, scId, address(manager));
    return IOnOfframpManager(manager);
}
```

Add the corresponding error definition:

```
error InvalidShareClassForPool();
```

This ensures that OnOfframpManager contracts are only created with valid, consistent pool/share class relationships, preventing operational failures and maintaining system invariants.

Developer Response

Fixed in PR#461.

2.10.3 Incorrect NatSpec on `isValid()` misrepresents validation logic

The NatSpec (@dev) on the `isValid()` function inaccurately describes the validation behavior. The documentation states that the function returns `false` if the price is zero. However, the actual implementation does not check whether `price == 0`. This mismatch between the spec and implementation can mislead developers and auditors, especially in edge cases such as zero-price deposits.

Technical Details

The current NatSpec and implementation for `isValid()` in `Spoke` contract shows complete divergence:

```solidity
/// @dev Price struct that contains a price, the timestamp at which it was computed and the max age of the price.
struct Price {
    uint128 price;
    uint64 computedAt;
    uint64 maxAge;
}

/// @dev Checks if a price is valid. Returns false if price is 0 or computedAt is 0. Otherwise checks for block
/// timestamp <= computedAt + maxAge
function isValid(Price memory price) view returns (bool) {
    if (price.computedAt != 0) {  // Initialization check
        return block.timestamp <= price.validUntil();
    } else {
        return false; // Uninitialized state
    }
}
```

This shows that the function does not reject `price == 0`, contrary to the comment. A zero price `0.0` is intentional and should be treated as valid as per the terms outlined by the project.

Impact

Informational. This is a documentation inconsistency. It does not directly impact functionality but may cause confusion or faulty assumptions.

Recommendation

Fix the NatSpec to reflect the actual behavior:

```diff
- /// @dev Checks if a price is valid. Returns false if price is 0 or computedAt is 0. Otherwise checks for block
- /// timestamp <= computedAt + maxAge
+ /// @dev Checks if a price is valid. Returns false if computedAt is 0. Otherwise checks for block
+ /// timestamp <= computedAt + maxAge
+ /// @dev A price of 0 may still be valid if within its validity window.
function isValid(Price memory price) view returns (bool) {
    if (price.computedAt != 0) {  // Initialization check
        return block.timestamp <= price.validUntil();
    } else {
        return false; // Uninitialized state
    }
}
```

Developer Response

Fixed in PR#462.

2.10.4 ShareClassId Validation Bypass in OnOfframpManager Cross-Chain Updates

The `OnOfframpManager` contract is intended to manage on-/off-ramp parameters per share-class. `OnOfframpManager.update()` method validates the `poolId` and caller (`spoke`) but silently discards the `ShareClassId` (`scId`).
Any cross-chain `UpdateContract` message that is authorised for `Share-Class-A` can therefore be redirected to the `OnOfframpManager` of `Share-Class-B` simply by choosing that manager's address as the target.

Technical Details

The vulnerability exists in the `OnOfframpManager.update()` function which implements the `IUpdateContract` interface for cross-chain configuration updates. While the function correctly validates the poolId and caller authorization, it completely ignores the ShareClassId parameter, unlike other managers in the system.

Root Cause Analysis

```
// OnOfframpManager.sol:50-53 - VULNERABLE
function update(PoolId poolId_, ShareClassId, /* scId */ bytes calldata payload) external {
    require(poolId == poolId_, InvalidPoolId());     // □ Pool validation
    require(msg.sender == spoke, NotSpoke());        // □ Caller validation
    // □ ShareClassId completely ignored!
```

Compare this to the properly implemented `SyncManager.update()`:

```
// SyncManager.sol:57-63 - SECURE
function update(PoolId poolId, ShareClassId scId, bytes memory payload) external auth {
    // ...
    require(address(spoke.shareToken(poolId, scId)) != address(0), ShareTokenDoesNotExist());
    // □ Properly validates ShareClassId exists
```

Call chain analysis
(a) A Pool manager submits `Hub.updateContract` crafting an update for Share-Class-A. (b) Sets the target address to `OnOfframpManager_B`. (c) Message arrives on the spoke: `poolId` matches, `scId = A` (mismatched), but `update()` still executes on manager B. (d) The pool manager enables `onramp[asset]`, grants `relayer[attacker]`, or rewires `offramp[asset]` to their account. (e) Subsequent deposits / withdrawals in Share-Class-B follow the their-controlled rules, enabling undisclosed assets or siphoning funds.
Project require that ”a balance-sheet manager of one pool should never control another”; in V3 each share-class has its own manager, so the same principle applies at share-class scope. Docs emphasise multiple investment assets per share-class; that modularity only holds if config messages can't leak across classes.
The issue is not about whether or not managers can be trusted, but more about their extended capacity beyond the intended scope of their capacity initialization.

Impact

Informational.

Recommendation

Implement proper ShareClassId validation in `OnOfframpManager.update()` consistent with other managers:

```
function update(PoolId poolId_, ShareClassId scId_, bytes calldata payload) external {
    require(poolId == poolId_, InvalidPoolId());
    require(msg.sender == spoke, NotSpoke());

    // NOTE: ADD THIS CRITICAL VALIDATION:
    require(scId == scId_, InvalidShareClassId());

    // Alternative validation approach (like SyncManager):
    // require(address(ISpoke(spoke).shareToken(poolId_, scId_)) != address(0),
ShareTokenDoesNotExist());

    uint8 kind = uint8(UpdateContractMessageLib.updateContractType(payload));
    // ... rest of function unchanged
}
```

Developer Response

Fixed in PR#462.

2.10.5 `onRedeemRequest()` is never called

Technical Details

The functions `onRedeemRequest()` from the `BaseVault` and the `LegacyAdapter` are never called.

Impact

Informational.

Recommendation

Remove the functions or document why they aren't being used at the moment.

Developer Response

Acknowledged, leaving this for legacy reasons.

2.10.6 Events part of executions initiated in the LegacyVaultAdapter are emitted in the legacy vault

The events that happen during flows which are part of the vault functionality of the adapter will be emitted in the legacy vault.

Technical Details

The implementation of the LegacyVaultAdapter contract overrides the callbacks used to emit events to forward them to the legacy vault.
This will work fine for flows that are initiated in the legacy vault, but will also mean that executions initiated as part of the inherited new vault functionality in the adapter will also be emitted in the legacy vault.

- `onDepositClaimable()`
- `onCancelDepositClaimable()`
- `onRedeemClaimable()`
- `onCancelRedeemClaimable()`

Impact

Informational.

Recommendation

The adjustment would require changes to determine where flows were originally initiated to later log these in the proper place.

Developer Response

Acknowledged.

2.10.7 Incorrect argument in CancelRedeemClaim event

The `CancelRedeemClaim` event is emitted with `receiver` as the first argument and `controller` as the second, but in the definition of the event, these parameters are in the opposite order.

Technical Details

BaseVaults.sol#L290

Impact

Informational.

Recommendation

Switch the order of the `receiver` and `controller` arguments.

Developer Response

Fixed in PR#479.

2.10.8 Incorrect argument in RedeemRequest event

The `sender` argument is wired to `msg.sender`, but this is the caller to `onRedeemRequest()` and not the original caller for the request.

Technical Details

BaseVaults.sol#L319-L321

Impact

Informational.

Recommendation

Forward the original caller to `onRedeemRequest()`.

Developer Response

Acknowledged, left as is for legacy reasons.

2.10.9 Apply CEI in BalanceSheet

Technical Details

In `submitQueuedAssets()` and `submitQueuedShares()` the sender, along with the cross-chain functionality, is invoked before clearing the state, enabling potential reentrancy issues.

Impact

Informational.

Recommendation

Reset the state before calling the `sender` contract.

Developer Response

Fixed in 92ed22e.

2.10.10 Validate entities are registered in Spoke contract

There are multiple occurrences in the Spoke contract in which the asset or the vault are fetched from storage without validating if these have been registered.

Technical Details

`assetId`:

- `deployVault()`
- `linkVault()`
- `unlinkVault()`

`vault`:

- `linkVault()`
- `unlinkVault()`

Impact

Informational.

Recommendation

For the asset id, use the `idToAsset()` accessor which checks if the asset is not null. For the vault, use `vaultDetails()`. `registerVault()` could also check that `asset != address(0)` to provide consistency.

Developer Response

Vault checks were added in ca9f5cb.
Asset id checks were added in df6c58b.

Auditors Response

Further discussion related to the vault checks originally recommended in this finding revealed a severe issue in which managers could link or unlink vaults from other pools. This vulnerability was mitigated as part of the fixes in changeset `ca9f5cb`.

2.10.11 OnOfframpManager should raise if the update kind is not supported

The switch present in `update()` fails silently if `m.kind` is not between the supported options.

Technical Details

OnOfframpManager.sol#L59-L82

Impact

Informational.

Recommendation

Revert if the update kind is not supported.

Developer Response

Fixed in df6c58b.

2.11 Final Remarks

The Centrifuge V3 protocol features an innovative design that allows on-chain tokenization of real-world assets using EIP-7540 asynchronous vaults and a hub-and-spoke model, in which pools can be deployed on a main chain (hub) that replicate to other peripheral chains (spoke). The codebase and its architecture are well-designed and structured, demonstrating solid mathematical foundations and good documentation practices. However, the multi-chain and asynchronous nature of the protocol creates intricate interaction patterns that can be difficult to reason about comprehensively, introducing complexity challenges that require careful consideration. As part of these complex interactions, one high-severity issue was identified related to incorrect share transfer logic in the legacy adapter flows. Additionally, a medium-severity finding was discovered affecting the synchronization of shares between hub and spoke that could eventually impact cross-chain accountability. The Centrifuge team demonstrated exceptional responsiveness in addressing identified issues and engaging with the audit process. While the codebase features an excellent testing suite, the legacy adapter functionality remains uncovered, though following this report, The Centrifuge team decided to remove the adapter from the planned migration to V3, so this code is not in use anymore.